Flax

The Flax authors

May 02, 2024

CONTENTS

Neural networks with JAX

Flax delivers an **end-to-end and flexible user experience for researchers who use JAX with neural networks**. Flax exposes the full power of JAX. It is made up of loosely coupled libraries, which are showcased with end-to-end integrated guides and examples.

Flax is used by hundreds of projects (and growing), both in the open source community (like Hugging Face) and at Google (like PaLM, Imagen, Scenic, and Big Vision).

CHAPTER

ONE

FEATURES

Safety Flax is designed for correctness and safety. Thanks to its immutable Modules and Functional API, Flax helps mitigate bugs that arise when handling state in JAX.

Control Flax grants more fine-grained control and expressivity than most Neural Network frameworks via its Variable Collections, RNG Collections and Mutability conditions.

Functional API Flax's functional API radically redefines what Modules can do via lifted transformations like vmap, scan, etc, while also enabling seamless integration with other JAX libraries like Optax and Chex.

Terse code Flax's `compact` Modules enables submodules to be defined directly at their callsite, leading to code that is easier to read and avoids repetition.

CHAPTER

TWO

INSTALLATION

```
pip install flax
# or to install the latest version of Flax:
pip install --upgrade git+https://github.com/google/flax.git
```

Flax installs the vanilla CPU version of JAX, if you need a custom version please check out JAX's installation page.

CHAPTER

THREE

BASIC USAGE

```python
class MLP(nn.Module):                    # create a Flax Module dataclass
  out_dims: int

  @nn.compact
  def __call__(self, x):
    x = x.reshape((x.shape[0], -1))
    x = nn.Dense(128)(x)                 # create inline Flax Module submodules
    x = nn.relu(x)
    x = nn.Dense(self.out_dims)(x)       # shape inference
    return x

model = MLP(out_dims=10)                 # instantiate the MLP model

x = jnp.empty((4, 28, 28, 1))            # generate random data
variables = model.init(PRNGKey(42), x)   # initialize the weights
y = model.apply(variables, x)            # make forward pass
```

CHAPTER

FOUR

LEARN MORE

Getting started	Guides	Examples	Glossary
Developer notes	The Flax philosophy	API reference	

CHAPTER

FIVE

ECOSYSTEM

Notable examples in Flax include:

Hugging Face	NLP and computer vision models
DALLE Mini	Model for text-to-image generation
PaLM	540-billion parameter model for text generation
Imagen	Text-to-image diffusion models
Scenic	Libraries for large-scale computer vision
Big Vision	Large-scale computer vision models
T5x	Large Language Models
Brax	On-device differentiable reinforcement learning environments

5.1 Quickstart

Welcome to Flax!

Flax is an open source Python neural network library built on top of JAX. This tutorial demonstrates how to construct a simple convolutional neural network (CNN) using the Flax Linen API and train the network for image classification on the MNIST dataset.

5.1.1 1. Install Flax

```
!pip install -q flax
```

5.1.2 2. Loading data

Flax can use any data-loading pipeline and this example demonstrates how to utilize TFDS. Define a function that loads and prepares the MNIST dataset and converts the samples to floating-point numbers.

```
import tensorflow_datasets as tfds  # TFDS for MNIST
import tensorflow as tf             # TensorFlow operations

def get_datasets(num_epochs, batch_size):
```

(continues on next page)

(continued from previous page)

```
  """Load MNIST train and test datasets into memory."""
  train_ds = tfds.load('mnist', split='train')
  test_ds = tfds.load('mnist', split='test')

  train_ds = train_ds.map(lambda sample: {'image': tf.cast(sample['image'],
                                                           tf.float32) / 255.,
                                          'label': sample['label']}) # normalize train␣
↪set
  test_ds = test_ds.map(lambda sample: {'image': tf.cast(sample['image'],
                                                         tf.float32) / 255.,
                                        'label': sample['label']}) # normalize test set

  train_ds = train_ds.repeat(num_epochs).shuffle(1024) # create shuffled dataset by␣
↪allocating a buffer size of 1024 to randomly draw elements from
  train_ds = train_ds.batch(batch_size, drop_remainder=True).prefetch(1) # group into␣
↪batches of batch_size and skip incomplete batch, prefetch the next sample to improve␣
↪latency
  test_ds = test_ds.shuffle(1024) # create shuffled dataset by allocating a buffer size␣
↪of 1024 to randomly draw elements from
  test_ds = test_ds.batch(batch_size, drop_remainder=True).prefetch(1) # group into␣
↪batches of batch_size and skip incomplete batch, prefetch the next sample to improve␣
↪latency

  return train_ds, test_ds
```

5.1.3 3. Define network

Create a convolutional neural network with the Linen API by subclassing Flax Module. Because the architecture in this example is relatively simple—you're just stacking layers—you can define the inlined submodules directly within the `__call__` method and wrap it with the `@compact` decorator. To learn more about the Flax Linen `@compact` decorator, refer to the `setup` vs `compact` guide.

```
from flax import linen as nn  # Linen API

class CNN(nn.Module):
  """A simple CNN model."""

  @nn.compact
  def __call__(self, x):
    x = nn.Conv(features=32, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = nn.Conv(features=64, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = x.reshape((x.shape[0], -1))  # flatten
    x = nn.Dense(features=256)(x)
    x = nn.relu(x)
    x = nn.Dense(features=10)(x)
    return x
```

View model layers

Create an instance of the Flax Module and use the `Module.tabulate` method to visualize a table of the model layers by passing an RNG key and template image input.

```
import jax
import jax.numpy as jnp  # JAX NumPy

cnn = CNN()
print(cnn.tabulate(jax.random.PRNGKey(0), jnp.ones((1, 28, 28, 1))))
```

```
                                CNN Summary

 path     module  inputs             outputs            params

│         │ CNN     │ float32[1,28,28,... │ float32[1,10]      │                  │
├─────────┼─────────┼─────────────────────┼────────────────────┼──────────────────┤
│ Conv_0  │ Conv    │ float32[1,28,28,... │ float32[1,28,28,... │ bias: float32[32] │
│         │         │                     │                    │ kernel:          │
│         │         │                     │                    │ float32[3,3,1,32] │
│         │         │                     │                    │                  │
│         │         │                     │                    │ 320 (1.3 KB)     │
├─────────┼─────────┼─────────────────────┼────────────────────┼──────────────────┤
│ Conv_1  │ Conv    │ float32[1,14,14,... │ float32[1,14,14,... │ bias: float32[64] │
│         │         │                     │                    │ kernel:          │
│         │         │                     │                    │ float32[3,3,32,6... │
│         │         │                     │                    │                  │
│         │         │                     │                    │ 18,496 (74.0 KB) │
├─────────┼─────────┼─────────────────────┼────────────────────┼──────────────────┤
│ Dense_0 │ Dense   │ float32[1,3136]     │ float32[1,256]     │ bias:            │
│         │         │                     │                    │ float32[256]     │
│         │         │                     │                    │ kernel:          │
│         │         │                     │                    │ float32[3136,256] │
│         │         │                     │                    │                  │
│         │         │                     │                    │ 803,072 (3.2 MB) │
├─────────┼─────────┼─────────────────────┼────────────────────┼──────────────────┤
│ Dense_1 │ Dense   │ float32[1,256]      │ float32[1,10]      │ bias: float32[10] │
│         │         │                     │                    │ kernel:          │
│         │         │                     │                    │ float32[256,10]  │
│         │         │                     │                    │                  │
│         │         │                     │                    │ 2,570 (10.3 KB)  │
├─────────┼─────────┼─────────────────────┼────────────────────┼──────────────────┤
│         │         │                     │             Total  │ 824,458 (3.3 MB) │
└─────────┴─────────┴─────────────────────┴────────────────────┴──────────────────┘

                    Total Parameters: 824,458 (3.3 MB)
```

5.1.4 4. Create a `TrainState`

A common pattern in Flax is to create a single dataclass that represents the entire training state, including step number, parameters, and optimizer state.

Because this is such a common pattern, Flax provides the class `flax.training.train_state.TrainState` that serves most basic usecases.

```
!pip install -q clu
```

```
from clu import metrics
from flax.training import train_state  # Useful dataclass to keep train state
from flax import struct                # Flax dataclasses
import optax                           # Common loss functions and optimizers
```

We will be using the `clu` library for computing metrics. For more information on `clu`, refer to the repo and notebook.

```
@struct.dataclass
class Metrics(metrics.Collection):
  accuracy: metrics.Accuracy
  loss: metrics.Average.from_output('loss')
```

You can then subclass `train_state.TrainState` so that it also contains metrics. This has the advantage that we only need to pass around a single argument to functions like `train_step()` (see below) to calculate the loss, update the parameters and compute the metrics all at once.

```
class TrainState(train_state.TrainState):
  metrics: Metrics

def create_train_state(module, rng, learning_rate, momentum):
  """Creates an initial `TrainState`."""
  params = module.init(rng, jnp.ones([1, 28, 28, 1]))['params'] # initialize parameters␣
↪by passing a template image
  tx = optax.sgd(learning_rate, momentum)
  return TrainState.create(
      apply_fn=module.apply, params=params, tx=tx,
      metrics=Metrics.empty())
```

5.1.5 5. Training step

A function that:

- Evaluates the neural network given the parameters and a batch of input images with `TrainState.apply_fn` (which contains the `Module.apply` method (forward pass)).
- Computes the cross entropy loss, using the predefined `optax.softmax_cross_entropy_with_integer_labels()`. Note that this function expects integer labels, so there is no need to convert labels to onehot encoding.
- Evaluates the gradient of the loss function using `jax.grad`.
- Applies a pytree of gradients to the optimizer to update the model's parameters.

Use JAX's @jit decorator to trace the entire `train_step` function and just-in-time compile it with XLA into fused device operations that run faster and more efficiently on hardware accelerators.

```
@jax.jit
def train_step(state, batch):
  """Train for a single step."""
  def loss_fn(params):
    logits = state.apply_fn({'params': params}, batch['image'])
    loss = optax.softmax_cross_entropy_with_integer_labels(
        logits=logits, labels=batch['label']).mean()
    return loss
  grad_fn = jax.grad(loss_fn)
  grads = grad_fn(state.params)
  state = state.apply_gradients(grads=grads)
  return state
```

5.1.6 6. Metric computation

Create a separate function for loss and accuracy metrics. Loss is calculated using the `optax.softmax_cross_entropy_with_integer_labels` function, while accuracy is calculated using `clu.metrics`.

```
@jax.jit
def compute_metrics(*, state, batch):
  logits = state.apply_fn({'params': state.params}, batch['image'])
  loss = optax.softmax_cross_entropy_with_integer_labels(
        logits=logits, labels=batch['label']).mean()
  metric_updates = state.metrics.single_from_model_output(
    logits=logits, labels=batch['label'], loss=loss)
  metrics = state.metrics.merge(metric_updates)
  state = state.replace(metrics=metrics)
  return state
```

5.1.7 7. Download data

```
num_epochs = 10
batch_size = 32

train_ds, test_ds = get_datasets(num_epochs, batch_size)
```

5.1.8 8. Seed randomness

- Set the TF random seed to ensure dataset shuffling (with `tf.data.Dataset.shuffle`) is reproducible.
- Get one PRNGKey and use it for parameter initialization. (Learn more about JAX PRNG design and PRNG chains.)

```
tf.random.set_seed(0)
```

```
init_rng = jax.random.PRNGKey(0)
```

5.1.9 9. Initialize the `TrainState`

Remember that the function `create_train_state` initializes the model parameters, optimizer and metrics and puts them into the training state dataclass that is returned.

```
learning_rate = 0.01
momentum = 0.9
```

```
state = create_train_state(cnn, init_rng, learning_rate, momentum)
del init_rng  # Must not be used anymore.
```

5.1.10 10. Train and evaluate

Create a "shuffled" dataset by:

- Repeating the dataset equal to the number of training epochs
- Allocating a buffer of size 1024 (containing the first 1024 samples in the dataset) of which to randomly sample batches from
  - Everytime a sample is randomly drawn from the buffer, the next sample in the dataset is loaded into the buffer

Define a training loop that:

- Randomly samples batches from the dataset.
- Runs an optimization step for each training batch.
- Computes the mean training metrics across each batch in an epoch.
- Computes the metrics for the test set using the updated parameters.
- Records the train and test metrics for visualization.

Once the training and testing is done after 10 epochs, the output should show that your model was able to achieve approximately 99% accuracy.

```
# since train_ds is replicated num_epochs times in get_datasets(), we divide by num_
↪epochs
num_steps_per_epoch = train_ds.cardinality().numpy() // num_epochs
```

```
metrics_history = {'train_loss': [],
                   'train_accuracy': [],
                   'test_loss': [],
                   'test_accuracy': []}
```

```
for step,batch in enumerate(train_ds.as_numpy_iterator()):

  # Run optimization steps over training batches and compute batch metrics
  state = train_step(state, batch) # get updated train state (which contains the updated␣
↪parameters)
  state = compute_metrics(state=state, batch=batch) # aggregate batch metrics

  if (step+1) % num_steps_per_epoch == 0: # one training epoch has passed
    for metric,value in state.metrics.compute().items(): # compute metrics
```

(continues on next page)

(continued from previous page)

```
      metrics_history[f'train_{metric}'].append(value) # record metrics
    state = state.replace(metrics=state.metrics.empty()) # reset train_metrics for next␣
↪training epoch

    # Compute metrics on the test set after each training epoch
    test_state = state
    for test_batch in test_ds.as_numpy_iterator():
      test_state = compute_metrics(state=test_state, batch=test_batch)

    for metric,value in test_state.metrics.compute().items():
      metrics_history[f'test_{metric}'].append(value)

    print(f"train epoch: {(step+1) // num_steps_per_epoch}, "
          f"loss: {metrics_history['train_loss'][-1]}, "
          f"accuracy: {metrics_history['train_accuracy'][-1] * 100}")
    print(f"test epoch: {(step+1) // num_steps_per_epoch}, "
          f"loss: {metrics_history['test_loss'][-1]}, "
          f"accuracy: {metrics_history['test_accuracy'][-1] * 100}")
```

```
train epoch: 1, loss: 0.20290373265743256, accuracy: 93.87000274658203
test epoch: 1, loss: 0.07591685652732849, accuracy: 97.60617065429688
train epoch: 2, loss: 0.05760224163532257, accuracy: 98.28500366210938
test epoch: 2, loss: 0.050395529717206955, accuracy: 98.3974380493164
train epoch: 3, loss: 0.03897436335682869, accuracy: 98.83000183105469
test epoch: 3, loss: 0.04574578255414963, accuracy: 98.54767608642578
train epoch: 4, loss: 0.028721099719405174, accuracy: 99.15166473388672
test epoch: 4, loss: 0.035722777247428894, accuracy: 98.91827392578125
train epoch: 5, loss: 0.021948494017124176, accuracy: 99.37999725341797
test epoch: 5, loss: 0.035723842680454254, accuracy: 98.87820434570312
train epoch: 6, loss: 0.01705147698521614, accuracy: 99.54833221435547
test epoch: 6, loss: 0.03456473350524902, accuracy: 98.96835327148438
train epoch: 7, loss: 0.014007646590471268, accuracy: 99.6116714477539
test epoch: 7, loss: 0.04089202359318733, accuracy: 98.7880630493164
train epoch: 8, loss: 0.011265480890870094, accuracy: 99.73333740234375
test epoch: 8, loss: 0.03337760642170906, accuracy: 98.93830108642578
train epoch: 9, loss: 0.00918484665453434, accuracy: 99.78334045410156
test epoch: 9, loss: 0.034478139132261276, accuracy: 98.96835327148438
train epoch: 10, loss: 0.007260234095156193, accuracy: 99.84166717529297
test epoch: 10, loss: 0.032822880893945694, accuracy: 99.07852172851562
```

5.1.11 11. Visualize metrics

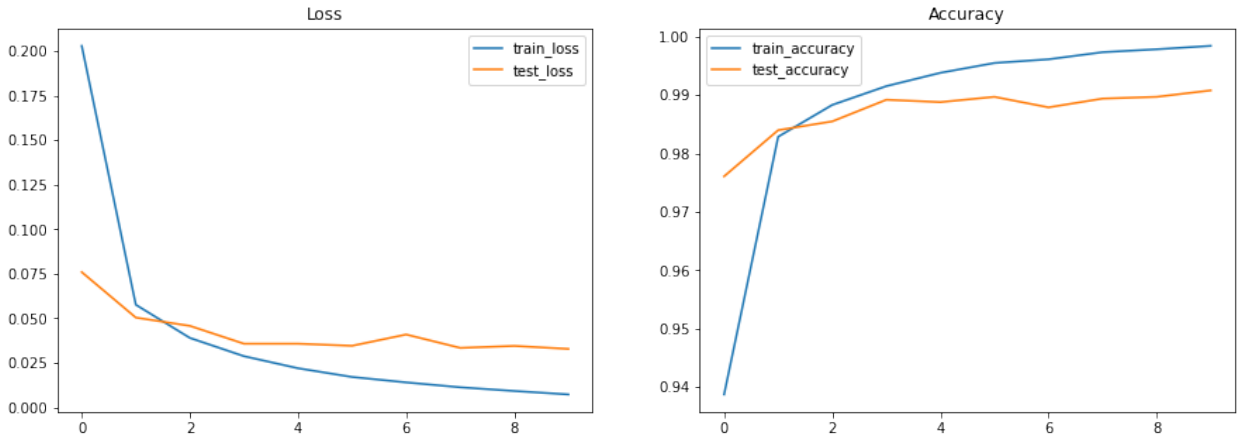
```
import matplotlib.pyplot as plt  # Visualization

# Plot loss and accuracy in subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
ax1.set_title('Loss')
ax2.set_title('Accuracy')
for dataset in ('train','test'):
  ax1.plot(metrics_history[f'{dataset}_loss'], label=f'{dataset}_loss')
```

(continues on next page)

(continued from previous page)

```
  ax2.plot(metrics_history[f'{dataset}_accuracy'], label=f'{dataset}_accuracy')
ax1.legend()
ax2.legend()
plt.show()
plt.clf()
```




```
<Figure size 600x400 with 0 Axes>
```

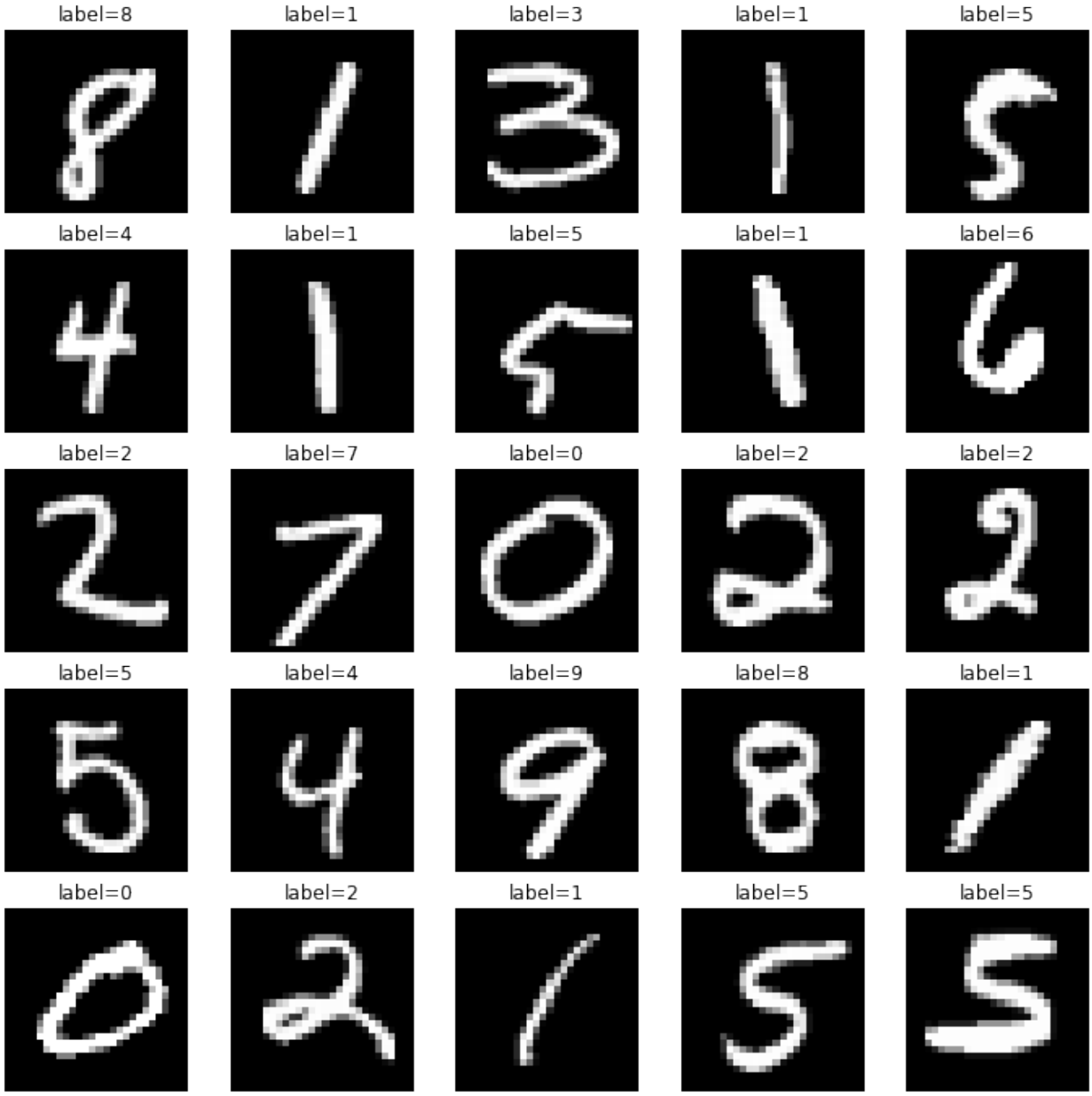
5.1.12 12. Perform inference on test set

Define a jitted inference function `pred_step`. Use the learned parameters to do model inference on the test set and visualize the images and their corresponding predicted labels.

```
@jax.jit
def pred_step(state, batch):
  logits = state.apply_fn({'params': state.params}, test_batch['image'])
  return logits.argmax(axis=1)

test_batch = test_ds.as_numpy_iterator().next()
pred = pred_step(state, test_batch)
```

```
fig, axs = plt.subplots(5, 5, figsize=(12, 12))
for i, ax in enumerate(axs.flatten()):
    ax.imshow(test_batch['image'][i, ..., 0], cmap='gray')
    ax.set_title(f"label={pred[i]}")
    ax.axis('off')
```



Congratulations! You made it to the end of the annotated MNIST example. You can revisit the same example, but structured differently as a couple of Python modules, test modules, config files, another Colab, and documentation in Flax's Git repo:

https://github.com/google/flax/tree/main/examples/mnist

5.2 Guides

5.2.1 Flax fundamentals

Flax Basics

This notebook will walk you through the following workflow:

- Instantiating a model from Flax built-in layers or third-party models.
- Initializing parameters of the model and manually written training.
- Using optimizers provided by Flax to ease training.
- Serialization of parameters and other objects.
- Creating your own models and managing state.

Setting up our environment

Here we provide the code needed to set up the environment for our notebook.

```
# Install the latest JAXlib version.
!pip install --upgrade -q pip jax jaxlib
# Install Flax at head:
!pip install --upgrade -q git+https://github.com/google/flax.git
```

```
WARNING: Running pip as root will break packages and permissions. You should install␣
↪packages reliably by using venv: https://pip.pypa.io/warnings/venv
WARNING: Running pip as root will break packages and permissions. You should install␣
↪packages reliably by using venv: https://pip.pypa.io/warnings/venv
```

```
import jax
from typing import Any, Callable, Sequence
from jax import lax, random, numpy as jnp
from flax.core import freeze, unfreeze
from flax import linen as nn
```

Linear regression with Flax

In the previous *JAX for the impatient* notebook, we finished up with a linear regression example. As we know, linear regression can also be written as a single dense neural network layer, which we will show in the following so that we can compare how it's done.

A dense layer is a layer that has a kernel parameter $W \in \mathcal{M}_{m,n}(\mathbb{R})$ where m is the number of features as an output of the model, and n the dimensionality of the input, and a bias parameter $b \in \mathbb{R}^m$. The dense layers returns $Wx + b$ from an input $x \in \mathbb{R}^n$.

This dense layer is already provided by Flax in the `flax.linen` module (here imported as `nn`).

```
# We create one dense layer instance (taking 'features' parameter as input)
model = nn.Dense(features=5)
```

Layers (and models in general, we'll use that word from now on) are subclasses of the `linen.Module` class.

Model parameters & initialization

Parameters are not stored with the models themselves. You need to initialize parameters by calling the `init` function, using a PRNGKey and dummy input data.

```
key1, key2 = random.split(random.PRNGKey(0))
x = random.normal(key1, (10,)) # Dummy input data
params = model.init(key2, x) # Initialization call
jax.tree_util.tree_map(lambda x: x.shape, params) # Checking output shapes
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more␣
↪info.)
```

```
FrozenDict({
    params: {
        bias: (5,),
        kernel: (10, 5),
    },
})
```

Note: JAX and Flax, like NumPy, are row-based systems, meaning that vectors are represented as row vectors and not column vectors. This can be seen in the shape of the kernel here.

The result is what we expect: bias and kernel parameters of the correct size. Under the hood:

- The dummy input data `x` is used to trigger shape inference: we only declared the number of features we wanted in the output of the model, not the size of the input. Flax finds out by itself the correct size of the kernel.
- The random PRNG key is used to trigger the initialization functions (those have default values provided by the module here).
- Initialization functions are called to generate the initial set of parameters that the model will use. Those are functions that take as arguments `(PRNG Key, shape, dtype)` and return an Array of shape `shape`.
- The init function returns the initialized set of parameters (you can also get the output of the forward pass on the dummy input with the same syntax by using the `init_with_output` method instead of `init`.

The output shows that the parameters are stored in a `FrozenDict` instance, which helps deal with the functional nature of JAX by preventing any mutation of the underlying dict and making the user aware of it. Read more about it in the `flax.core.frozen_dict.FrozenDict` API docs.

As a consequence, the following doesn't work:

```
try:
    params['new_key'] = jnp.ones((2,2))
except ValueError as e:
    print("Error: ", e)
```

```
Error:  FrozenDict is immutable.
```

To conduct a forward pass with the model with a given set of parameters (which are never stored with the model), we just use the `apply` method by providing it the parameters to use as well as the input:

```
model.apply(params, x)
```

```
Array([-1.3721193 ,  0.61131495,  0.6442836 ,  2.2192965 , -1.1271116 ],      ␣
↪dtype=float32)
```

Gradient descent

If you jumped here directly without going through the JAX part, here is the linear regression formulation we're going to use: from a set of data points $\{(x_i, y_i), i \in \{1, \ldots, k\}, x_i \in \mathbb{R}^n, y_i \in \mathbb{R}^m\}$, we try to find a set of parameters $W \in \mathcal{M}_{m,n}(\mathbb{R}), b \in \mathbb{R}^m$ such that the function $f_{W,b}(x) = Wx + b$ minimizes the mean squared error:

$$\mathcal{L}(W, b) \rightarrow \frac{1}{k} \sum_{i=1}^{k} \frac{1}{2} \|y_i - f_{W,b}(x_i)\|_2^2$$

Here, we see that the tuple (W, b) matches the parameters of the Dense layer. We'll perform gradient descent using those. Let's first generate the fake data we'll use. The data is exactly the same as in the JAX part's linear regression pytree example.

```
# Set problem dimensions.
n_samples = 20
x_dim = 10
y_dim = 5

# Generate random ground truth W and b.
key = random.PRNGKey(0)
k1, k2 = random.split(key)
W = random.normal(k1, (x_dim, y_dim))
b = random.normal(k2, (y_dim,))
# Store the parameters in a FrozenDict pytree.
true_params = freeze({'params': {'bias': b, 'kernel': W}})

# Generate samples with additional noise.
key_sample, key_noise = random.split(k1)
x_samples = random.normal(key_sample, (n_samples, x_dim))
y_samples = jnp.dot(x_samples, W) + b + 0.1 * random.normal(key_noise,(n_samples, y_dim))
print('x shape:', x_samples.shape, '; y shape:', y_samples.shape)
```

```
x shape: (20, 10) ; y shape: (20, 5)
```

We copy the same training loop that we used in the JAX pytree linear regression example with `jax.value_and_grad()`, but here we can use `model.apply()` instead of having to define our own feed-forward function (`predict_pytree()` in the JAX example).

```
# Same as JAX version but using model.apply().
@jax.jit
def mse(params, x_batched, y_batched):
  # Define the squared loss for a single pair (x,y)
  def squared_error(x, y):
    pred = model.apply(params, x)
```

(continues on next page)

(continued from previous page)

```
    return jnp.inner(y-pred, y-pred) / 2.0
  # Vectorize the previous to compute the average of the loss on all samples.
  return jnp.mean(jax.vmap(squared_error)(x_batched,y_batched), axis=0)
```

And finally perform the gradient descent.

```
learning_rate = 0.3  # Gradient step size.
print('Loss for "true" W,b: ', mse(true_params, x_samples, y_samples))
loss_grad_fn = jax.value_and_grad(mse)

@jax.jit
def update_params(params, learning_rate, grads):
  params = jax.tree_util.tree_map(
      lambda p, g: p - learning_rate * g, params, grads)
  return params

for i in range(101):
  # Perform one gradient update.
  loss_val, grads = loss_grad_fn(params, x_samples, y_samples)
  params = update_params(params, learning_rate, grads)
  if i % 10 == 0:
    print(f'Loss step {i}: ', loss_val)
```

```
Loss for "true" W,b:  0.023639796
```

```
Loss step 0:  35.343872
Loss step 10:  0.5143469
Loss step 20:  0.11384161
```

```
Loss step 30:  0.039326727
Loss step 40:  0.019916186
Loss step 50:  0.014209134
```

```
Loss step 60:  0.012425641
Loss step 70:  0.011850389
Loss step 80:  0.011661778
```

```
Loss step 90:  0.011599404
Loss step 100:  0.011578696
```

Optimizing with Optax

Flax used to use its own `flax.optim` package for optimization, but with FLIP #1009 this was deprecated in favor of Optax.

Basic usage of Optax is straightforward:

1. Choose an optimization method (e.g. `optax.adam`).
2. Create optimizer state from parameters (for the Adam optimizer, this state will contain the momentum values).
3. Compute the gradients of your loss with `jax.value_and_grad()`.

4. At every iteration, call the Optax `update` function to update the internal optimizer state and create an update to the parameters. Then add the update to the parameters with Optax's `apply_updates` method.

Note that Optax can do a lot more: it's designed for composing simple gradient transformations into more complex transformations that allows to implement a wide range of optimizers. There is also support for changing optimizer hyperparameters over time ("schedules"), applying different updates to different parts of the parameter tree ("masking") and much more. For details please refer to the official documentation.

```
import optax
tx = optax.adam(learning_rate=learning_rate)
opt_state = tx.init(params)
loss_grad_fn = jax.value_and_grad(mse)
```

```
for i in range(101):
  loss_val, grads = loss_grad_fn(params, x_samples, y_samples)
  updates, opt_state = tx.update(grads, opt_state)
  params = optax.apply_updates(params, updates)
  if i % 10 == 0:
    print('Loss step {}: '.format(i), loss_val)
```

```
Loss step 0:  0.011577619
Loss step 10:  0.26143155
Loss step 20:  0.07675066
Loss step 30:  0.036440928
```

```
Loss step 40:  0.022013078
Loss step 50:  0.01617866
Loss step 60:  0.013002755
Loss step 70:  0.012026124
```

```
Loss step 80:  0.011764515
Loss step 90:  0.011646035
Loss step 100:  0.011585519
```

Serializing the result

Now that we're happy with the result of our training, we might want to save the model parameters to load them back later. Flax provides a serialization package to enable you to do that.

```
from flax import serialization
bytes_output = serialization.to_bytes(params)
dict_output = serialization.to_state_dict(params)
print('Dict output')
print(dict_output)
print('Bytes output')
print(bytes_output)
```

```
Dict output
{'params': {'bias': Array([-1.4555765, -2.027799 ,  2.0790975,  1.2186143, -0.9980974], ␣
↪    dtype=float32), 'kernel': Array([[ 1.0098808 ,  0.18934363,  0.0445502 , -0.
↪92802227,  0.34784004],
```

(continues on next page)

(continued from previous page)

```
       [ 1.7298454 ,  0.98793703,  1.1640465 ,  1.1006079 , -0.10653931],
       [-1.2029461 ,  0.28635228,  1.4155982 ,  0.11870936, -1.3141482 ],
       [-1.1941488 , -0.18958506,  0.0341386 ,  1.3169425 ,  0.08060396],
       [ 0.13852414,  1.3713043 , -1.3187186 ,  0.5315267 , -2.2404995 ],
       [ 0.5629401 ,  0.8122313 ,  0.31752002,  0.5345511 ,  0.90500367],
       [-0.37926018,  1.7410394 ,  1.0790286 , -0.5039833 ,  0.92830586],
       [ 0.9706487 , -1.3153405 ,  0.33681527,  0.80993414, -1.2018455 ],
       [ 1.0194311 , -0.6202478 ,  1.081883  , -1.8389739 , -0.4580503 ],
       [-0.6436537 ,  0.456667  , -1.1329138 , -0.68538654,  0.16829044]],
↪dtype=float32)}}
Bytes output
b'\x81\xa6params\x82\xa4bias\xc7!\x01\x93\x91\x05\xa7float32\xc4\x14UP\xba\xbfu\xc7\x01\
↪xc0\xef\x0f\x05@\x8e\xfb\x9b?P\x83\x7f\xbf\xa6kernel\xc7\xd6\x01\x93\x92\n\x05\
↪xa7float32\xc4\xc8\xc6C\x81?L\xe3A>Dz6=\xde\x92m\xbf\x17\x18\xb2>\x93k\xdd?q\xe9|?z\
↪xff\x94?\xb8\xe0\x8c?H1\xda\xbd#\xfa\x99\xbf\xc4\x9c\x92>R2\xb5?\xe4\x1d\xf3=\x026\xa8\
↪xbf\xde\xd9\x98\xbf\x96"B\xbe\xeb\xd4\x0b=\x92\x91\xa8?\xb0\x13\xa5=F\xd9\r>\xe6\x86\
↪xaf?\xc5\xcb\xa8\xbf"\x12\x08?Xd\x0f\xc0\xd8\x1c\x10?d\xeeO?\xfc\x91\xa2>W\xd8\x08?R\
↪xaeg?d.\xc2\xbea\xda\xde?\x9c\x1d\x8a?\r\x05\x01\xbft\xa5m?o|x?\x14]\xa8\xbf\rs\xac>\
↪xd8WO?\x13\xd6\x99\xbf\xb8|\x82?\x8f\xc8\x1e\xbf${\x8a?\x7fc\xeb\xbf\x92\x85\xea\xbe}\
↪xc6$\xbfB\xd0\xe9>R\x03\x91\xbf~u/\xbfTT,>'
```

To load the model back, you'll need to use a template of the model parameter structure, like the one you would get from the model initialization. Here, we use the previously generated `params` as a template. Note that this will produce a new variable structure, and not mutate in-place.

The point of enforcing structure through template is to avoid users issues downstream, so you need to first have the right model that generates the parameters structure.

```
serialization.from_bytes(params, bytes_output)
```

```
FrozenDict({
    params: {
        bias: array([-1.4555765, -2.027799 ,  2.0790975,  1.2186143, -0.9980974],
              dtype=float32),
        kernel: array([[ 1.0098808 ,  0.18934363,  0.0445502 , -0.92802227,  0.34784004],
               [ 1.7298454 ,  0.98793703,  1.1640465 ,  1.1006079 , -0.10653931],
               [-1.2029461 ,  0.28635228,  1.4155982 ,  0.11870936, -1.3141482 ],
               [-1.1941488 , -0.18958506,  0.0341386 ,  1.3169425 ,  0.08060396],
               [ 0.13852414,  1.3713043 , -1.3187186 ,  0.5315267 , -2.2404995 ],
               [ 0.5629401 ,  0.8122313 ,  0.31752002,  0.5345511 ,  0.90500367],
               [-0.37926018,  1.7410394 ,  1.0790286 , -0.5039833 ,  0.92830586],
               [ 0.9706487 , -1.3153405 ,  0.33681527,  0.80993414, -1.2018455 ],
               [ 1.0194311 , -0.6202478 ,  1.081883  , -1.8389739 , -0.4580503 ],
               [-0.6436537 ,  0.456667  , -1.1329138 , -0.68538654,  0.16829044]],
              dtype=float32),
    },
})
```

Defining your own models

Flax allows you to define your own models, which should be a bit more complicated than a linear regression. In this section, we'll show you how to build simple models. To do so, you'll need to create subclasses of the base `nn.Module` class.

Keep in mind that we imported `linen as nn` *and this only works with the new linen API*

Module basics

The base abstraction for models is the `nn.Module` class, and every type of predefined layers in Flax (like the previous `Dense`) is a subclass of `nn.Module`. Let's take a look and start by defining a simple but custom multi-layer perceptron i.e. a sequence of Dense layers interleaved with calls to a non-linear activation function.

```
class ExplicitMLP(nn.Module):
  features: Sequence[int]

  def setup(self):
    # we automatically know what to do with lists, dicts of submodules
    self.layers = [nn.Dense(feat) for feat in self.features]
    # for single submodules, we would just write:
    # self.layer1 = nn.Dense(feat1)

  def __call__(self, inputs):
    x = inputs
    for i, lyr in enumerate(self.layers):
      x = lyr(x)
      if i != len(self.layers) - 1:
        x = nn.relu(x)
    return x

key1, key2 = random.split(random.PRNGKey(0), 2)
x = random.uniform(key1, (4,4))

model = ExplicitMLP(features=[3,4,5])
params = model.init(key2, x)
y = model.apply(params, x)

print('initialized parameter shapes:\n', jax.tree_util.tree_map(jnp.shape,␣
↪unfreeze(params)))
print('output:\n', y)
```

```
initialized parameter shapes:
 {'params': {'layers_0': {'bias': (3,), 'kernel': (4, 3)}, 'layers_1': {'bias': (4,),
↪'kernel': (3, 4)}, 'layers_2': {'bias': (5,), 'kernel': (4, 5)}}}
output:
 [[ 0.          0.          0.          0.          0.        ]
 [ 0.0072379  -0.00810347 -0.02550939  0.02151716 -0.01261241]
 [ 0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.        ]]
```

As we can see, a `nn.Module` subclass is made of:

- A collection of data fields (`nn.Module` are Python dataclasses) - here we only have the `features` field of type `Sequence[int]`.
- A `setup()` method that is being called at the end of the `__postinit__` where you can register submodules, variables, parameters you will need in your model.
- A `__call__` function that returns the output of the model from a given input.
- The model structure defines a pytree of parameters following the same tree structure as the model: the params tree contains one `layers_n` sub dict per layer, and each of those contain the parameters of the associated Dense layer. The layout is very explicit.

Note: lists are mostly managed as you would expect (WIP), there are corner cases you should be aware of as pointed out here

Since the module structure and its parameters are not tied to each other, you can't directly call `model(x)` on a given input as it will return an error. The `__call__` function is being wrapped up in the `apply` one, which is the one to call on an input:

```
try:
    y = model(x) # Returns an error
except AttributeError as e:
    print(e)
```

```
"ExplicitMLP" object has no attribute "layers". If "layers" is defined in '.setup()',␣
↪remember these fields are only accessible from inside 'init' or 'apply'.
```

Since here we have a very simple model, we could have used an alternative (but equivalent) way of declaring the submodules inline in the `__call__` using the `@nn.compact` annotation like so:

```
class SimpleMLP(nn.Module):
  features: Sequence[int]

  @nn.compact
  def __call__(self, inputs):
    x = inputs
    for i, feat in enumerate(self.features):
      x = nn.Dense(feat, name=f'layers_{i}')(x)
      if i != len(self.features) - 1:
        x = nn.relu(x)
      # providing a name is optional though!
      # the default autonames would be "Dense_0", "Dense_1", ...
    return x

key1, key2 = random.split(random.PRNGKey(0), 2)
x = random.uniform(key1, (4,4))

model = SimpleMLP(features=[3,4,5])
params = model.init(key2, x)
y = model.apply(params, x)

print('initialized parameter shapes:\n', jax.tree_util.tree_map(jnp.shape,␣
↪unfreeze(params)))
print('output:\n', y)
```

```
initialized parameter shapes:
 {'params': {'layers_0': {'bias': (3,), 'kernel': (4, 3)}, 'layers_1': {'bias': (4,),
↪'kernel': (3, 4)}, 'layers_2': {'bias': (5,), 'kernel': (4, 5)}}}
output:
 [[ 0.          0.          0.          0.          0.        ]
 [ 0.0072379  -0.00810347 -0.02550939  0.02151716 -0.01261241]
 [ 0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.        ]]
```

There are, however, a few differences you should be aware of between the two declaration modes:

- In `setup`, you are able to name some sublayers and keep them around for further use (e.g. encoder/decoder methods in autoencoders).
- If you want to have multiple methods, then you **need** to declare the module using `setup`, as the `@nn.compact` annotation only allows one method to be annotated.
- The last initialization will be handled differently. See these notes for more details (TODO: add notes link).

Module parameters

In the previous MLP example, we relied only on predefined layers and operators (`Dense`, `relu`). Let's imagine that you didn't have a Dense layer provided by Flax and you wanted to write it on your own. Here is what it would look like using the `@nn.compact` way to declare a new modules:

```
class SimpleDense(nn.Module):
  features: int
  kernel_init: Callable = nn.initializers.lecun_normal()
  bias_init: Callable = nn.initializers.zeros_init()

  @nn.compact
  def __call__(self, inputs):
    kernel = self.param('kernel',
                        self.kernel_init, # Initialization function
                        (inputs.shape[-1], self.features))  # shape info.
    y = lax.dot_general(inputs, kernel,
                        (((inputs.ndim - 1,), (0,)), ((), ())),) # TODO Why not jnp.dot?
    bias = self.param('bias', self.bias_init, (self.features,))
    y = y + bias
    return y

key1, key2 = random.split(random.PRNGKey(0), 2)
x = random.uniform(key1, (4,4))

model = SimpleDense(features=3)
params = model.init(key2, x)
y = model.apply(params, x)

print('initialized parameters:\n', params)
print('output:\n', y)
```

```
initialized parameters:
 FrozenDict({
```

(continues on next page)

(continued from previous page)

```
    params: {
        kernel: Array([[ 0.61506   , -0.22728713,  0.6054702 ],
               [-0.29617992,  1.1232013 , -0.879759  ],
               [-0.35162622,  0.3806491 ,  0.6893246 ],
               [-0.1151355 ,  0.04567898, -1.091212  ]], dtype=float32),
        bias: Array([0., 0., 0.], dtype=float32),
    },
})
output:
 [[-0.02996204  1.102088   -0.6660265 ]
 [-0.31092793  0.6323942  -0.53678817]
 [ 0.01424007  0.9424717  -0.6356147 ]
 [ 0.36818963  0.3586519  -0.00459214]]
```

Here, we see how to both declare and assign a parameter to the model using the `self.param` method. It takes as input `(name, init_fn, *init_args)` :

- `name` is simply the name of the parameter that will end up in the parameter structure.
- `init_fn` is a function with input `(PRNGKey, *init_args)` returning an Array, with `init_args` being the arguments needed to call the initialisation function.
- `init_args` are the arguments to provide to the initialization function.

Such params can also be declared in the `setup` method; it won't be able to use shape inference because Flax is using lazy initialization at the first call site.

Variables and collections of variables

As we've seen so far, working with models means working with:

- A subclass of `nn.Module`;
- A pytree of parameters for the model (typically from `model.init()`);

However this is not enough to cover everything that we would need for machine learning, especially neural networks. In some cases, you might want your neural network to keep track of some internal state while it runs (e.g. batch normalization layers). There is a way to declare variables beyond the parameters of the model with the `variable` method.

For demonstration purposes, we'll implement a simplified but similar mechanism to batch normalization: we'll store running averages and subtract those to the input at training time. For proper batchnorm, you should use (and look at) the implementation here.

```
class BiasAdderWithRunningMean(nn.Module):
  decay: float = 0.99

  @nn.compact
  def __call__(self, x):
    # easy pattern to detect if we're initializing via empty variable tree
    is_initialized = self.has_variable('batch_stats', 'mean')
    ra_mean = self.variable('batch_stats', 'mean',
                            lambda s: jnp.zeros(s),
                            x.shape[1:])
    mean = ra_mean.value # This will either get the value or trigger init
```

(continues on next page)

(continued from previous page)

```
    bias = self.param('bias', lambda rng, shape: jnp.zeros(shape), x.shape[1:])
    if is_initialized:
      ra_mean.value = self.decay * ra_mean.value + (1.0 - self.decay) * jnp.mean(x,␣
↪axis=0, keepdims=True)

    return x - ra_mean.value + bias


key1, key2 = random.split(random.PRNGKey(0), 2)
x = jnp.ones((10,5))
model = BiasAdderWithRunningMean()
variables = model.init(key1, x)
print('initialized variables:\n', variables)
y, updated_state = model.apply(variables, x, mutable=['batch_stats'])
print('updated state:\n', updated_state)
```

```
initialized variables:
 FrozenDict({
    batch_stats: {
        mean: Array([0., 0., 0., 0., 0.], dtype=float32),
    },
    params: {
        bias: Array([0., 0., 0., 0., 0.], dtype=float32),
    },
})
```

```
updated state:
 FrozenDict({
    batch_stats: {
        mean: Array([[0.01, 0.01, 0.01, 0.01, 0.01]], dtype=float32),
    },
})
```

Here, `updated_state` returns only the state variables that are being mutated by the model while applying it on data. To update the variables and get the new parameters of the model, we can use the following pattern:

```
for val in [1.0, 2.0, 3.0]:
  x = val * jnp.ones((10,5))
  y, updated_state = model.apply(variables, x, mutable=['batch_stats'])
  old_state, params = variables.pop('params')
  variables = freeze({'params': params, **updated_state})
  print('updated state:\n', updated_state) # Shows only the mutable part
```

```
updated state:
 FrozenDict({
    batch_stats: {
        mean: Array([[0.01, 0.01, 0.01, 0.01, 0.01]], dtype=float32),
    },
})
updated state:
 FrozenDict({
```

(continues on next page)

(continued from previous page)

```
    batch_stats: {
        mean: Array([[0.0299, 0.0299, 0.0299, 0.0299, 0.0299]], dtype=float32),
    },
})
```

```
updated state:
 FrozenDict({
    batch_stats: {
        mean: Array([[0.059601, 0.059601, 0.059601, 0.059601, 0.059601]], dtype=float32),
    },
})
```

From this simplified example, you should be able to derive a full BatchNorm implementation, or any layer involving a state. To finish, let's add an optimizer to see how to play with both parameters updated by an optimizer and state variables.

This example isn't doing anything and is only for demonstration purposes.

```
from functools import partial

@partial(jax.jit, static_argnums=(0, 1))
def update_step(tx, apply_fn, x, opt_state, params, state):

  def loss(params):
    y, updated_state = apply_fn({'params': params, **state},
                                x, mutable=list(state.keys()))
    l = ((x - y) ** 2).sum()
    return l, updated_state

  (l, state), grads = jax.value_and_grad(loss, has_aux=True)(params)
  updates, opt_state = tx.update(grads, opt_state)
  params = optax.apply_updates(params, updates)
  return opt_state, params, state

x = jnp.ones((10,5))
variables = model.init(random.PRNGKey(0), x)
state, params = variables.pop('params')
del variables
tx = optax.sgd(learning_rate=0.02)
opt_state = tx.init(params)

for _ in range(3):
  opt_state, params, state = update_step(tx, model.apply, x, opt_state, params, state)
  print('Updated state: ', state)
```

```
Updated state:  FrozenDict({
    batch_stats: {
        mean: Array([[0.01, 0.01, 0.01, 0.01, 0.01]], dtype=float32),
    },
})
```

```
Updated state:  FrozenDict({
    batch_stats: {
        mean: Array([[0.0199, 0.0199, 0.0199, 0.0199, 0.0199]], dtype=float32),
    },
})
Updated state:  FrozenDict({
    batch_stats: {
        mean: Array([[0.029701, 0.029701, 0.029701, 0.029701, 0.029701]], dtype=float32),
    },
})
```

Note that the above function has a quite verbose signature and it would not actually work with `jax.jit()` because the function arguments are not "valid JAX types".

Flax provides a handy wrapper - `TrainState` - that simplifies the above code. Check out `flax.training.train_state.TrainState` to learn more.

Exporting to Tensorflow's SavedModel with jax2tf

JAX released an experimental converter called jax2tf, which allows converting trained Flax models into Tensorflow's SavedModel format (so it can be used for TF Hub, TF.lite, TF.js, or other downstream applications). The repository contains more documentation and has various examples for Flax.

Managing Parameters and State

We will show you how to…

- manage the variables from initialization to updates.
- split and re-assemble parameters and state.
- use `vmap` with batch-dependant state.

```
class BiasAdderWithRunningMean(nn.Module):
  momentum: float = 0.9

  @nn.compact
  def __call__(self, x):
    is_initialized = self.has_variable('batch_stats', 'mean')
    mean = self.variable('batch_stats', 'mean', jnp.zeros, x.shape[1:])
    bias = self.param('bias', lambda rng, shape: jnp.zeros(shape), x.shape[1:])
    if is_initialized:
      mean.value = (self.momentum * mean.value +
                    (1.0 - self.momentum) * jnp.mean(x, axis=0, keepdims=True))
    return mean.value + bias
```

This example model is a minimal example that contains both parameters (declared with `self.param`) and state variables (declared with `self.variable`).

The tricky part with initialization here is that we need to split the state variables and the parameters we're going to optimize for.

First we define `update_step` as follows (with a dummy loss that should be replaced with yours):

```
def update_step(apply_fn, x, opt_state, params, state):
  def loss(params):
    y, updated_state = apply_fn({'params': params, **state},
                                x, mutable=list(state.keys()))
    l = ((x - y) ** 2).sum() # Replace with your loss here.
    return l, updated_state

  (l, updated_state), grads = jax.value_and_grad(
      loss, has_aux=True)(params)
  updates, opt_state = tx.update(grads, opt_state)  # Defined below.
  params = optax.apply_updates(params, updates)
  return opt_state, params, updated_state
```

Then we can write the actual training code.

```
model = BiasAdderWithRunningMean()
variables = model.init(random.PRNGKey(0), dummy_input)
# Split state and params (which are updated by optimizer).
state, params = variables.pop('params')
del variables  # Delete variables to avoid wasting resources
tx = optax.sgd(learning_rate=0.02)
opt_state = tx.init(params)

for _ in range(num_epochs):
  opt_state, params, state = update_step(
      model.apply, dummy_input, opt_state, params, state)
```

`vmap` accross the batch dimension

When using `vmap` and managing state that depends on the batch dimension, for example when using `BatchNorm`, the setup above must be modified slightly. This is because any layer whose state depends on the batch dimension is not strictly vectorizable. In the case of `BatchNorm`, `lax.pmean()` must be used to average the statistics over the batch dimension so that the state is in sync for each item in the batch.

This requires two small changes. Firstly, we need to name the batch axis in our model definition. Here, this is done by specifying the `axis_name` argument of `BatchNorm`. In your own code this might require specifying the `axis_name` argument of `lax.pmean()` directly.

```
class MLP(nn.Module):
  hidden_size: int
  out_size: int

  @nn.compact
  def __call__(self, x, train=False):
    norm = partial(
        nn.BatchNorm,
        use_running_average=not train,
        momentum=0.9,
        epsilon=1e-5,
        axis_name="batch", # Name batch dim
    )
```

(continues on next page)

(continued from previous page)

```
    x = nn.Dense(self.hidden_size)(x)
    x = norm()(x)
    x = nn.relu(x)
    x = nn.Dense(self.hidden_size)(x)
    x = norm()(x)
    x = nn.relu(x)
    y = nn.Dense(self.out_size)(x)

    return y
```

Secondly, we need to specify the same name when calling `vmap` in our training code:

```
def update_step(apply_fn, x_batch, y_batch, opt_state, params, state):

  def batch_loss(params):
    def loss_fn(x, y):
      pred, updated_state = apply_fn(
        {'params': params, **state},
        x, mutable=list(state.keys())
      )
      return (pred - y) ** 2, updated_state

    loss, updated_state = jax.vmap(
      loss_fn, out_axes=(0, None),  # Do not vmap `updated_state`.
      axis_name='batch'  # Name batch dim
    )(x_batch, y_batch)  # vmap only `x`, `y`, but not `state`.
    return jnp.mean(loss), updated_state

  (loss, updated_state), grads = jax.value_and_grad(
    batch_loss, has_aux=True
  )(params)

  updates, opt_state = tx.update(grads, opt_state)  # Defined below.
  params = optax.apply_updates(params, updates)
  return opt_state, params, updated_state, loss
```

Note that we also need to specify that the model state does not have a batch dimension. Now we are able to train the model:

```
model = MLP(hidden_size=10, out_size=1)
variables = model.init(random.PRNGKey(0), dummy_input)
# Split state and params (which are updated by optimizer).
state, params = variables.pop('params')
del variables  # Delete variables to avoid wasting resources
tx = optax.sgd(learning_rate=0.02)
opt_state = tx.init(params)

for _ in range(num_epochs):
  opt_state, params, state, loss = update_step(
      model.apply, X, Y, opt_state, params, state)
```

`setup` vs `compact`

In Flax's module system (named Linen), submodules and variables (parameters or others) can be defined in two ways:

1. **Explicitly** (using `setup`):

   Assign submodules or variables to `self.<attr>` inside a *`setup`* method. Then use the submodules and variables assigned to `self.<attr>` in `setup` from any "forward pass" method defined on the class. This resembles how modules are defined in PyTorch.

2. **In-line** (using `nn.compact`):

   Write your network's logic directly within a single "forward pass" method annotated with *`nn.compact`*. This allows you to define your whole module in a single method, and "co-locate" submodules and variables next to where they are used.

Both of these approaches are perfectly valid, behave the same way, and interoperate with all of Flax.

Here is a short example of a module defined in both ways, with exactly the same functionality.

Using `setup`

```
class MLP(nn.Module):
  def setup(self):
    # Submodule names are derived by the attributes you assign to. In this
    # case, "dense1" and "dense2". This follows the logic in PyTorch.
    self.dense1 = nn.Dense(32)
    self.dense2 = nn.Dense(32)

  def __call__(self, x):
    x = self.dense1(x)
    x = nn.relu(x)
    x = self.dense2(x)
    return x
```

Using `nn.compact`

```
class MLP(nn.Module):

  @nn.compact
  def __call__(self, x):
    x = nn.Dense(32, name="dense1")(x)
    x = nn.relu(x)
    x = nn.Dense(32, name="dense2")(x)
    return x
```

So, how would you decide which style to use? It can be a matter of taste, but here are some pros and cons:

Reasons to prefer using `nn.compact`:

1. Allows defining submodules, parameters and other variables next to where they are used: less scrolling up/down to see how everything is defined.
2. Reduces code duplication when there are conditionals or for loops that conditionally define submodules, parameters or variables.
3. Code typically looks more like mathematical notation: `y = self.param('W', ...) @ x + self.param('b', ...)` looks similar to $y = Wx + b$)
4. If you are using shape inference, i.e. using parameters whose shape/value depend on shapes of the inputs (which are unknown at initialization), this is not possible using `setup`.

Reasons to prefer using `setup`:

1. Closer to the PyTorch convention, thus easier when porting models from PyTorch
2. Some people find it more natural to explicitly separate the definition of submodules and variables from where they are used
3. Allows defining more than one "forward pass" method (see *`MultipleMethodsCompactError`*)

Dealing with Flax Module arguments

Introduction

In Flax Linen we can define `Module` arguments either as dataclass attributes or as arguments to methods (usually `__call__`). Typically the distinction is clear:

- Completely fixed properties, such as the choice of kernel initializer or number of output features, are hyperparameters and should be defined as dataclass attributes. Typically two Module instances with different hyperparamaters cannot share in a meaningful way.
- Dynamic properties, such as input data and top-level "mode switches" like `train=True/False`, should be passed as arguments to `__call__` or another method.

Some cases are however less clear cut. Take for example the `Dropout` module. We have a number of clear hyperparameters:

1. The dropout rate
2. The axes for which a dropout mask is generated

And some clear call time arguments:

1. The input that should be masked using dropout
2. The (optional) rng used to sample the random mask

There is however one property that is ambiguous – the `deterministic` property in a Dropout module.

If `deterministic` is `True` no dropout mask is sampled. This is typically used during model evaluation. However, if we pass `eval=True` or `train=False` to a top-level Module. The `deterministic` argument needs to be applied everywhere and the boolean argument needs to be passed down to all the layers that might use `Dropout`. If instead `deterministic` is a dataclass attribute, we might do the following:

```
from functools import partial
from flax import linen as nn

class ResidualModel(nn.Module):
  drop_rate: float

  @nn.compact
  def __call__(self, x, *, train):
    dropout = partial(nn.Dropout, rate=self.drop_rate, deterministic=not train)
    for i in range(10):
      x += ResidualBlock(dropout=dropout, ...)(x)
```

It makes sense to pass `determinstic` to the constructor here because this way we can pass the dropout template to the sub-modules. Now the sub-module no longer needs to take care of train vs eval mode and can simply use the `dropout` argument. Note that because the dropout layer can only be constructed in the sub-module we can only partially apply `deterministic` to the constructor but not to `__call__`.

However, if `deterministic` is a dataclass attribute we run into trouble when using the setup pattern. We would **want** to write our module code like this:

```
class SomeModule(nn.Module):
  drop_rate: float

  def setup(self):
    self.dropout = nn.Dropout(rate=self.drop_rate)

  @nn.compact
  def __call__(self, x, *, train):
    # ...
    x = self.dropout(x, deterministic=not train)
    # ...
```

But, as defined above, `deterministic` would be an attribute, so this doesn't work. Here it makes sense to pass `deterministic` during `__call__` because it depends on the `train` argument.

Solution

We can support both use cases described before by allowing certain properties to be passed as dataclass attributes or as method argument (but not both!). This can be implemented as follows:

```
class MyDropout(nn.Module):
  drop_rate: float
  deterministic: Optional[bool] = None

  @nn.compact
  def __call__(self, x, deterministic=None):
    deterministic = nn.merge_param('deterministic', self.deterministic, deterministic)
    # ...
```

In this example `nn.merge_param` will ensure that either `self.deterministic` or `deterministic` is set but not both. An error is raised if both values are `None` or both values are not `None`. This avoids confusing behavior where 2 different parts of the code set the same parameter and one is overruled by the other. It also avoids a default value which would probably cause either the train step or eval step of a training procedure to be broken by default.

Functional Core

Functional core defines functions rather than classes. Therefore, there is no clear distinction between hyperparameters and call-time arguments. The only way to pre-determine the hyperparameters is by using `partial`. On the upside, there are no ambiguous cases where method arguments could also be attributes.

5.2.2 Data preprocesing

Processing the entire Dataset

For efficiency reasons, we form batches that contain multiple examples and process them in parallel. Especially when evaluating a model, it is important that we process all examples and **avoid losing the remainder** of examples that does not form a complete batch at the end.

The problem

When evaluating on a single device, one can either drop the last incomplete batch, or one can form a last batch with a shape different from the preceding batches. Doing the latter has the disadvantage that this will trigger a **recompilation** of the `eval_step()` because XLA is not shape polymorphic.

```
collections.Counter(
    tuple(batch['image'].shape)
    for batch in tfds.load('mnist', split='test').batch(per_device_batch_size)
)
# output:
# Counter({(272, 28, 28, 1): 1, (512, 28, 28, 1): 19})
```

The problem is accentuated when using multiple devices for data parallelism. If the batch size is not **divisible by the number devices**, then that last step must be executed on a single device (or a subset of devices). Usually one would drop the last batch, but this will lead to incorrect results.

```
sum(
    np.prod(batch['label'].shape)
    for batch in tfds.load('mnist', split='test')
        .batch(per_device_batch_size, drop_remainder=True)
        .batch(jax.local_device_count())
)
# output:
# 9728
```

Using multiple hosts further complicates the situation because JAX uses the SPMD paradigm and every host must execute the same program. We would usually form non-overlapping splits for different hosts with `tfds.split_for_jax_process()`, but this can lead to **different numbers for different hosts**, resulting in different JAX programs when all examples are to be processed.

```
process_count = 6
[
    len(tfds.load(dataset_name, split=tfds.split_for_jax_process(
        'test', process_index=process_index, process_count=process_count)))
```

(continues on next page)

(continued from previous page)

```
    for process_index in range(process_count)
]
# output:
# [1667, 1667, 1667, 1667, 1666, 1666]
```

The solution: padding

Even though it's possible to solve this problem by cleverly adjusting the number of batches executed by different devices on different hosts, such a solution quickly becomes complicated and makes the main eval loop hard to read with a lot of cumbersome logic.

The more straightforward solution to this problem is to use padding at the end of the dataset to make sure that the last batch has the same size as the preceding batches.

Manual implementation

The last batch is manually padded to contain the same number of examples as in the preceding batches. The predictions for the padded examples are discarded from the computation.

```
shard = lambda x: einops.rearrange(
    x, '(d b) ... -> d b ...', d=jax.local_device_count())
unshard = lambda x: einops.rearrange(x, 'd b ... -> (d b) ...')

correct = total = 0
for batch in ds.as_numpy_iterator():
  images = batch['image']
  n = len(images)
  padding = np.zeros([per_host_batch_size - n, *images.shape[1:]], images.dtype)
  padded_images = np.concatenate([images, padding])
  preds = unshard(get_preds(variables, shard(padded_images)))[:n]
  total += n
  correct += (batch['label'] == preds.argmax(axis=-1)).sum()
```

Using `pad_shard_unpad()`

The above pattern, namely the pad→shard→predict→unshard→unpad sequence, can be extracted into a utility wrapper `pad_shard_unpad()`, which greatly simplifies above evaluation loop.

```
correct = total = 0
for batch in ds.as_numpy_iterator():
  preds = flax.jax_utils.pad_shard_unpad(get_preds)(
      vs, batch['image'], min_device_batch=per_device_batch_size)
  total += len(batch['image'])
  correct += (batch['label'] == preds.argmax(axis=-1)).sum()
```

Computing metrics in `eval_step()`

Instead of returning the predictions and computing the metrics in the main evaluation loop, we would often want to make the metric computation part of the evaluation step, especially when using libraries like `clu.metrics`, or `clu.metrics`.

In that case we would want to pass the metrics as a `static_argnums` (i.e. do not shard/pad it), and treat the return value as `static_return` too (i.e. no un-sharding or un-padding):

```
def eval_step(metrics, variables, batch):
  print('retrigger compilation', {k: v.shape for k, v in batch.items()})
  preds = model.apply(variables, batch['image'])
  correct = (batch['mask'] & (batch['label'] == preds.argmax(axis=-1))).sum()
  total = batch['mask'].sum()
  return dict(
      correct=metrics['correct'] + jax.lax.psum(correct, axis_name='batch'),
      total=metrics['total'] + jax.lax.psum(total, axis_name='batch'),
  )

eval_step = jax.pmap(eval_step, axis_name='batch')
eval_step = flax.jax_utils.pad_shard_unpad(
    eval_step, static_argnums=(0, 1), static_return=True)
```

Adding "infinite padding"

The above solution works in most cases, but it has some limitations:

1. In the rare case where even splitting of the dataset on multiple hosts leads to a different number of batches. Imagine having a dataset of `n=4097` examples, and evaluating this on `h=8`, each having `d=8` local devices, and forming on-device batch sizes of `b=128`. With even dataset splitting, the first host would get `4096/8+1==513` examples, and all other hosts would get `4096/8==512` examples. Forming per-host batches of `d*b==512` this would lead to two batches on the first host, and a single batch on all other hosts, violating SPMD principles and hanging the multi-host setup in the last `psum()` directive (which would only be executed by the first host, but not the others).
2. When dropping examples dynamically by using `ds.filter()`.

In these more complicated cases we could add "infinite padding" to the dataset, on each of the hosts independently, and continuing processing examples until *all* hosts run out of unpadded examples.

```
correct = total = 0
for batch in ds.as_numpy_iterator():
  n = count_p(batch['mask'])[0].item()  # adds sync barrier
  if not n: break

  preds = get_preds(vs, batch['image']).argmax(axis=-1)
  total += n
  correct += count_correct_p(batch['label'], preds, batch['mask'])[0]
```

As for the other examples in this HOWTO, the complete executable code can be found in the Colab:

5.2.3 Training techniques

Batch normalization

In this guide, you will learn how to apply batch normalization using *flax.linen.BatchNorm*.

Batch normalization is a regularization technique used to speed up training and improve convergence. During training, it computes running averages over feature dimensions. This adds a new form of non-differentiable state that must be handled appropriately.

Throughout the guide, you will be able to compare code examples with and without Flax `BatchNorm`.

Defining the model with `BatchNorm`

In Flax, `BatchNorm` is a *flax.linen.Module* that exhibits different runtime behavior between training and inference. You explicitly specify it via the `use_running_average` argument, as demonstrated below.

A common pattern is to accept a `train` (`training`) argument in the parent Flax `Module`, and use it to define `BatchNorm`'s `use_running_average` argument.

Note: In other machine learning frameworks, like PyTorch or TensorFlow (Keras), this is specified via a mutable state or a call flag (for example, in torch.nn.Module.eval or `tf.keras.Model` by setting the training flag).

No BatchNorm

```
class MLP(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = nn.Dense(features=4)(x)

    x = nn.relu(x)
    x = nn.Dense(features=1)(x)
    return x
```

With BatchNorm

```
class MLP(nn.Module):
  @nn.compact
  def __call__(self, x, train: bool):
    x = nn.Dense(features=4)(x)
    x = nn.BatchNorm(use_running_average=not train)(x)
    x = nn.relu(x)
    x = nn.Dense(features=1)(x)
    return x
```

Once you create your model, initialize it by calling *flax.linen.init()* to get the `variables` structure. Here, the main difference between the code without `BatchNorm` and with `BatchNorm` is that the `train` argument must be provided.

The batch_stats collection

In addition to the params collection, BatchNorm also adds a batch_stats collection that contains the running average of the batch statistics.

Note: You can learn more in the flax.linen variables API documentation.

The batch_stats collection must be extracted from the variables for later use.

No BatchNorm

```
mlp = MLP()
x = jnp.ones((1, 3))
variables = mlp.init(jax.random.PRNGKey(0), x)
params = variables['params']


jax.tree_util.tree_map(jnp.shape, variables)
```

With BatchNorm

```
mlp = MLP()
x = jnp.ones((1, 3))
variables = mlp.init(jax.random.PRNGKey(0), x, train=False)
params = variables['params']
batch_stats = variables['batch_stats']

jax.tree_util.tree_map(jnp.shape, variables)
```

Flax BatchNorm adds a total of 4 variables: mean and var that live in the batch_stats collection, and scale and bias that live in the params collection.

No BatchNorm

```
FrozenDict({
  'params': {
    'Dense_0': {
        'bias': (4,),
        'kernel': (3, 4),
    },
    'Dense_1': {
        'bias': (1,),
        'kernel': (4, 1),
    },
  },
})
```

With BatchNorm

```
FrozenDict({
  'batch_stats': {
    'BatchNorm_0': {
        'mean': (4,),
        'var': (4,),
    },
  },
  'params': {
    'BatchNorm_0': {
        'bias': (4,),
        'scale': (4,),
    },
    'Dense_0': {
        'bias': (4,),
        'kernel': (3, 4),
    },
    'Dense_1': {
        'bias': (1,),
        'kernel': (4, 1),
    },
  },
})
```

Modifying `flax.linen.apply`

When using *flax.linen.apply* to run your model with the `train==True` argument (that is, you have `use_running_average==False` in the call to `BatchNorm`), you need to consider the following:

- `batch_stats` must be passed as an input variable.
- The `batch_stats` collection needs to be marked as mutable by setting `mutable=['batch_stats']`.
- The mutated variables are returned as a second output. The updated `batch_stats` must be extracted from here.

No BatchNorm

```
y = mlp.apply(
  {'params': params},
  x,
)
...
```

With BatchNorm

```
y, updates = mlp.apply(
  {'params': params, 'batch_stats': batch_stats},
  x,
  train=True, mutable=['batch_stats']
)
batch_stats = updates['batch_stats']
```

Training and evaluation

When integrating models that use `BatchNorm` into a training loop, the main challenge is handling the additional `batch_stats` state. To do this, you need to:

- Add a `batch_stats` field to a custom *flax.training.train_state.TrainState* class.
- Pass the `batch_stats` values to the `train_state.TrainState.create` method.

No BatchNorm

```
from flax.training import train_state


state = train_state.TrainState.create(
  apply_fn=mlp.apply,
  params=params,

  tx=optax.adam(1e-3),
)
```

With BatchNorm

```
from flax.training import train_state

class TrainState(train_state.TrainState):
  batch_stats: Any

state = TrainState.create(
  apply_fn=mlp.apply,
  params=params,
  batch_stats=batch_stats,
  tx=optax.adam(1e-3),
)
```

In addition, update your `train_step` function to reflect these changes:

- Pass all new parameters to `flax.linen.apply` (as previously discussed).
- The `updates` to the `batch_stats` must be propagated out of the `loss_fn`.
- The `batch_stats` from the `TrainState` must be updated.

No BatchNorm

```
@jax.jit
def train_step(state: TrainState, batch):
  """Train for a single step."""
  def loss_fn(params):
    logits = state.apply_fn(
      {'params': params},
      x=batch['image'])
    loss = optax.softmax_cross_entropy_with_integer_labels(
      logits=logits, labels=batch['label'])
    return loss, logits
  grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
  (loss, logits), grads = grad_fn(state.params)
  state = state.apply_gradients(grads=grads)

  metrics = {
    'loss': loss,
      'accuracy': jnp.mean(jnp.argmax(logits, -1) == batch['label']),
  }
  return state, metrics
```

With BatchNorm

```
@jax.jit
def train_step(state: TrainState, batch):
  """Train for a single step."""
  def loss_fn(params):
    logits, updates = state.apply_fn(
      {'params': params, 'batch_stats': state.batch_stats},
      x=batch['image'], train=True, mutable=['batch_stats'])
    loss = optax.softmax_cross_entropy_with_integer_labels(
      logits=logits, labels=batch['label'])
    return loss, (logits, updates)
  grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
  (loss, (logits, updates)), grads = grad_fn(state.params)
  state = state.apply_gradients(grads=grads)
  state = state.replace(batch_stats=updates['batch_stats'])
  metrics = {
    'loss': loss,
      'accuracy': jnp.mean(jnp.argmax(logits, -1) == batch['label']),
  }
  return state, metrics
```

The `eval_step` is much simpler. Because `batch_stats` is not mutable, no updates need to be propagated. Make sure you pass the `batch_stats` to `flax.linen.apply`, and the `train` argument is set to `False`:

No BatchNorm

```
@jax.jit
def eval_step(state: TrainState, batch):
  """Train for a single step."""
  logits = state.apply_fn(
    {'params': params},
    x=batch['image'])
  loss = optax.softmax_cross_entropy_with_integer_labels(
    logits=logits, labels=batch['label'])
  metrics = {
    'loss': loss,
      'accuracy': jnp.mean(jnp.argmax(logits, -1) == batch['label']),
  }
  return state, metrics
```

With BatchNorm

```
@jax.jit
def eval_step(state: TrainState, batch):
  """Train for a single step."""
  logits = state.apply_fn(
    {'params': params, 'batch_stats': state.batch_stats},
    x=batch['image'], train=False)
  loss = optax.softmax_cross_entropy_with_integer_labels(
    logits=logits, labels=batch['label'])
  metrics = {
    'loss': loss,
      'accuracy': jnp.mean(jnp.argmax(logits, -1) == batch['label']),
  }
  return state, metrics
```

Dropout

This guide provides an overview of how to apply dropout using *flax.linen.Dropout()*.

Dropout is a stochastic regularization technique that randomly removes hidden and visible units in a network.

Throughout the guide, you will be able to compare code examples with and without Flax `Dropout`.

Split the PRNG key

Since dropout is a random operation, it requires a pseudorandom number generator (PRNG) state. Flax uses JAX's (splittable) PRNG keys, which have a number of desirable properties for neutral networks. To learn more, refer to the Pseudorandom numbers in JAX tutorial.

Note: Recall that JAX has an explicit way of giving you PRNG keys: you can fork the main PRNG state (such as `key = jax.random.PRNGKey(seed=0)`) into multiple new PRNG keys with `key, subkey = jax.random.split(key)`. You can refresh your memory in JAX - The Sharp Bits Randomness and PRNG keys.

Begin by splitting the PRNG key using jax.random.split() into three keys, including one for Flax Linen `Dropout`.

No Dropout

```
root_key = jax.random.PRNGKey(seed=0)
main_key, params_key = jax.random.split(key=root_key)
```

With Dropout

```
root_key = jax.random.PRNGKey(seed=0)
main_key, params_key, dropout_key = jax.random.split(key=root_key, num=3)
```

Note: In Flax, you provide *PRNG streams* with *names*, so that you can use them later in your *flax.linen.Module()*. For example, you pass the stream `'params'` for initializing parameters, and `'dropout'` for applying *flax.linen.Dropout()*.

Define your model with `Dropout`

To create a model with dropout:

- Subclass *flax.linen.Module()*, and then use *flax.linen.Dropout()* to add a dropout layer. Recall that *flax.linen.Module()* is the base class for all neural network Modules, and all layers and models are subclassed from it.
- In *flax.linen.Dropout()*, the `deterministic` argument is required to be passed as a keyword argument, either:
  - When constructing the *flax.linen.Module()*; or
  - When calling *flax.linen.init()* or *flax.linen.apply()* on a constructed `Module`. (Refer to `flax.linen.module.merge_param()` for more details.)
- Because `deterministic` is a boolean:
  - If it's set to `False`, the inputs are masked (that is, set to zero) with a probability set by `rate`. And the remaining inputs are scaled by `1 / (1 - rate)`, which ensures that the means of the inputs are preserved.
  - If it's set to `True`, no mask is applied (the dropout is turned off), and the inputs are returned as-is.

A common pattern is to accept a `training` (or `train`) argument (a boolean) in the parent Flax `Module`, and use it to enable or disable dropout (as demonstrated in later sections of this guide). In other machine learning frameworks, like PyTorch or TensorFlow (Keras), this is specified via a mutable state or a call flag (for example, in torch.nn.Module.eval or `tf.keras.Model` by setting the training flag).

Note: Flax provides an implicit way of handling PRNG key streams via Flax *flax.linen.Module()*'s *flax.linen.Module.make_rng()* method. This allows you to split off a fresh PRNG key inside Flax Modules (or their sub-Modules) from the PRNG stream. The `make_rng` method guarantees to provide a unique key each time you call it. Internally, *flax.linen.Dropout()* makes use of *flax.linen.Module.make_rng()* to create a key for dropout. You can check out the source code. In short, *flax.linen.Module.make_rng() guarantees full reproducibility*.

No Dropout

```
class MyModel(nn.Module):
  num_neurons: int

  @nn.compact
  def __call__(self, x):
    x = nn.Dense(self.num_neurons)(x)

    return x
```

With Dropout

```
class MyModel(nn.Module):
  num_neurons: int

  @nn.compact
  def __call__(self, x, training: bool):
    x = nn.Dense(self.num_neurons)(x)
    # Set the dropout layer with a `rate` of 50%.
    # When the `deterministic` flag is `True`, dropout is turned off.
    x = nn.Dropout(rate=0.5, deterministic=not training)(x)
    return x
```

Initialize the model

After creating your model:

- Instantiate the model.
- Then, in the *flax.linen.init()* call, set `training=False`.
- Finally, extract the `params` from the variable dictionary.

Here, the main difference between the code without Flax `Dropout` and with `Dropout` is that the `training` (or `train`) argument must be provided if you need dropout enabled.

No Dropout

```
my_model = MyModel(num_neurons=3)
x = jnp.empty((3, 4, 4))

variables = my_model.init(params_key, x)
params = variables['params']
```

With Dropout

```
my_model = MyModel(num_neurons=3)
x = jnp.empty((3, 4, 4))
# Dropout is disabled with `training=False` (that is, `deterministic=True`).
variables = my_model.init(params_key, x, training=False)
params = variables['params']
```

Perform the forward pass during training

When using *flax.linen.apply()* to run your model:

- Pass `training=True` to *flax.linen.apply()*.
- Then, to draw PRNG keys during the forward pass (with dropout), provide a PRNG key to seed the `'dropout'` stream when you call *flax.linen.apply()*.

No Dropout

```
# No need to pass the `training` and `rngs` flags.
y = my_model.apply({'params': params}, x)
```

With Dropout

```
# Dropout is enabled with `training=True` (that is, `deterministic=False`).
y = my_model.apply({'params': params}, x, training=True, rngs={'dropout': dropout_key})
```

Here, the main difference between the code without Flax `Dropout` and with `Dropout` is that the `training` (or `train`) and `rngs` arguments must be provided if you need dropout enabled.

During evaluation, use the above code with no dropout enabled (this means you do not have to pass a RNG either).

`TrainState` and the training step

This section explains how to amend your code inside the training step function if you have dropout enabled.

Note: Recall that Flax has a common pattern where you create a dataclass that represents the whole training state, including parameters and the optimizer state. Then, you can pass a single parameter, `state: TrainState`, to the training step function. Refer to the *flax.training.train_state.TrainState()* API docs to learn more.

- First, add a `key` field to a custom *flax.training.train_state.TrainState()* class.
- Then, pass the `key` value—in this case, the `dropout_key`—to the `train_state.TrainState.create()` method.

No Dropout

```
from flax.training import train_state

state = train_state.TrainState.create(
  apply_fn=my_model.apply,
  params=params,

  tx=optax.adam(1e-3)
)
```

With Dropout

```
from flax.training import train_state

class TrainState(train_state.TrainState):
  key: jax.random.KeyArray

state = TrainState.create(
  apply_fn=my_model.apply,
  params=params,
  key=dropout_key,
  tx=optax.adam(1e-3)
)
```

- Next, in the Flax training step function, `train_step`, generate a new PRNG key from the `dropout_key` to apply dropout at each step. This can be done with one of the following:
  - jax.random.split(); or
  - jax.random.fold_in()

  Using `jax.random.fold_in()` is generally faster. When you use `jax.random.split()` you split off a PRNG key that can be reused afterwards. However, using `jax.random.fold_in()` makes sure to 1) fold in unique data; and 2) can result in longer sequences of PRNG streams.
- Finally, when performing the forward pass, pass the new PRNG key to `state.apply_fn()` as an extra parameter.

No Dropout

```
@jax.jit
def train_step(state: TrainState, batch):

  def loss_fn(params):
    logits = state.apply_fn(
      {'params': params},
      x=batch['image'],


      )
    loss = optax.softmax_cross_entropy_with_integer_labels(
```

(continues on next page)

(continued from previous page)

```
      logits=logits, labels=batch['label'])
    return loss, logits
  grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
  (loss, logits), grads = grad_fn(state.params)
  state = state.apply_gradients(grads=grads)
  return state
```

With Dropout

```
@jax.jit
def train_step(state: TrainState, batch, dropout_key):
  dropout_train_key = jax.random.fold_in(key=dropout_key, data=state.step)
  def loss_fn(params):
    logits = state.apply_fn(
      {'params': params},
      x=batch['image'],
      training=True,
      rngs={'dropout': dropout_train_key}
      )
    loss = optax.softmax_cross_entropy_with_integer_labels(
      logits=logits, labels=batch['label'])
    return loss, logits
  grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
  (loss, logits), grads = grad_fn(state.params)
  state = state.apply_gradients(grads=grads)
  return state
```

Flax examples with dropout

- A Transformer-based model trained on the WMT Machine Translation dataset. This example uses dropout and attention dropout.
- Applying word dropout to a batch of input IDs in a text classification context. This example uses a custom *flax.linen.Dropout()* layer.

More Flax examples that use Module `make_rng()`

- Defining a prediction token in a decoder of a sequence-to-sequence model.

Learning rate scheduling

The learning rate is considered one of the most important hyperparameters for training deep neural networks, but choosing it can be quite hard. Rather than simply using a fixed learning rate, it is common to use a learning rate scheduler. In this example, we will use the *cosine scheduler*. Before the cosine scheduler comes into play, we start with a so-called *warmup* period in which the learning rate increases linearly for `warmup_epochs` epochs. For more information about the cosine scheduler, check out the paper "SGDR: Stochastic Gradient Descent with Warm Restarts".

We will show you how to…

- define a learning rate schedule
- train a simple model using that schedule

```
def create_learning_rate_fn(config, base_learning_rate, steps_per_epoch):
  """Creates learning rate schedule."""
  warmup_fn = optax.linear_schedule(
      init_value=0., end_value=base_learning_rate,
      transition_steps=config.warmup_epochs * steps_per_epoch)
  cosine_epochs = max(config.num_epochs - config.warmup_epochs, 1)
  cosine_fn = optax.cosine_decay_schedule(
      init_value=base_learning_rate,
      decay_steps=cosine_epochs * steps_per_epoch)
  schedule_fn = optax.join_schedules(
      schedules=[warmup_fn, cosine_fn],
      boundaries=[config.warmup_epochs * steps_per_epoch])
  return schedule_fn
```

To use the schedule, we must create a learning rate function by passing the hyperparameters to the `create_learning_rate_fn` function and then pass the function to your `Optax` optimizer. For example using this schedule on MNIST would require changing the `train_step` function:

Default learning rate

```
@jax.jit
def train_step(state, batch):
  def loss_fn(params):
    logits = CNN().apply({'params': params}, batch['image'])
    one_hot = jax.nn.one_hot(batch['label'], 10)
    loss = jnp.mean(optax.softmax_cross_entropy(logits, one_hot))
    return loss, logits
  grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
  (_, logits), grads = grad_fn(state.params)
  new_state = state.apply_gradients(grads=grads)
  metrics = compute_metrics(logits, batch['label'])


  return new_state, metrics
```

Learning rate schedule

```
@functools.partial(jax.jit, static_argnums=2)
def train_step(state, batch, learning_rate_fn):
  def loss_fn(params):
    logits = CNN().apply({'params': params}, batch['image'])
    one_hot = jax.nn.one_hot(batch['label'], 10)
    loss = jnp.mean(optax.softmax_cross_entropy(logits, one_hot))
    return loss, logits
  grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
  (_, logits), grads = grad_fn(state.params)
  new_state = state.apply_gradients(grads=grads)
  metrics = compute_metrics(logits, batch['label'])
  lr = learning_rate_fn(state.step)
  metrics['learning_rate'] = lr
  return new_state, metrics
```

And the `train_epoch` function:

Default learning rate

```
def train_epoch(state, train_ds, batch_size, epoch, rng):
  """Trains for a single epoch."""
  train_ds_size = len(train_ds['image'])
  steps_per_epoch = train_ds_size // batch_size
  perms = jax.random.permutation(rng, len(train_ds['image']))
  perms = perms[:steps_per_epoch * batch_size]
  perms = perms.reshape((steps_per_epoch, batch_size))
  batch_metrics = []
  for perm in perms:
    batch = {k: v[perm, ...] for k, v in train_ds.items()}
    state, metrics = train_step(state, batch)
    batch_metrics.append(metrics)

  # compute mean of metrics across each batch in epoch.
  batch_metrics = jax.device_get(batch_metrics)
  epoch_metrics = {
      k: np.mean([metrics[k] for metrics in batch_metrics])
      for k in batch_metrics[0]}

  logging.info('train epoch: %d, loss: %.4f, accuracy: %.2f', epoch,
               epoch_metrics['loss'], epoch_metrics['accuracy'] * 100)

  return state, epoch_metrics
```

Learning rate schedule

```
def train_epoch(state, train_ds, batch_size, epoch, learning_rate_fn, rng):
  """Trains for a single epoch."""
  train_ds_size = len(train_ds['image'])
  steps_per_epoch = train_ds_size // batch_size
  perms = jax.random.permutation(rng, len(train_ds['image']))
  perms = perms[:steps_per_epoch * batch_size]
  perms = perms.reshape((steps_per_epoch, batch_size))
  batch_metrics = []
  for perm in perms:
    batch = {k: v[perm, ...] for k, v in train_ds.items()}
    state, metrics = train_step(state, batch, learning_rate_fn)
    batch_metrics.append(metrics)

  # compute mean of metrics across each batch in epoch.
  batch_metrics = jax.device_get(batch_metrics)
  epoch_metrics = {
      k: np.mean([metrics[k] for metrics in batch_metrics])
      for k in batch_metrics[0]}

  logging.info('train epoch: %d, loss: %.4f, accuracy: %.2f', epoch,
               epoch_metrics['loss'], epoch_metrics['accuracy'] * 100)

  return state, epoch_metrics
```

And the `create_train_state` function:

Default learning rate

```
def create_train_state(rng, config):
  """Creates initial `TrainState`."""
  cnn = CNN()
  params = cnn.init(rng, jnp.ones([1, 28, 28, 1]))['params']
  tx = optax.sgd(config.learning_rate, config.momentum)
  return train_state.TrainState.create(
      apply_fn=cnn.apply, params=params, tx=tx)
```

Learning rate schedule

```
def create_train_state(rng, config, learning_rate_fn):
  """Creates initial `TrainState`."""
  cnn = CNN()
  params = cnn.init(rng, jnp.ones([1, 28, 28, 1]))['params']
  tx = optax.sgd(learning_rate_fn, config.momentum)
  return train_state.TrainState.create(
      apply_fn=cnn.apply, params=params, tx=tx)
```

Transfer learning

This guide demonstrates various parts of the transfer learning workflow with Flax. Depending on the task, a pretrained model can be used just as a feature extractor or it can be fine-tuned as part of a larger model.

This guide demonstrates how to:

- Load a pretrained model from HuggingFace Transformers and extract a specific sub-module from that pretrained model.
- Create a classifier model.
- Transfer the pretrained parameters to the new model structure.
- Create an optimizer for training different parts of the model separately with Optax.
- Set up the model for training.

Depending on your task, some of the content in this guide may be suboptimal. For example, if you are only going to train a linear classifier on top of a pretrained model, it may be better to just extract the feature embeddings once, which can result in much faster training, and you can use specialized algorithms for linear regression or logistic classification. This guide shows how to do transfer learning with all the model parameters.

Setup

```
# Note that the Transformers library doesn't use the latest Flax version.
! pip install -q transformers[flax]
# Install/upgrade Flax and JAX. For JAX installation with GPU/TPU support,
# visit https://github.com/google/jax#installation.
! pip install -U -q flax jax jaxlib
```

Create a function for model loading

To load a pre-trained classifier, for convenience first create a function that returns a Flax `Module` and its pretrained variables.

In the code below, the `load_model` function uses HuggingFace's `FlaxCLIPVisionModel` model from the Transformers library and extracts a `FlaxCLIPModule` module.

```
%%capture
from IPython.display import clear_output
from transformers import FlaxCLIPModel

# Note: FlaxCLIPModel is not a Flax Module
def load_model():
  clip = FlaxCLIPModel.from_pretrained('openai/clip-vit-base-patch32')
  clear_output(wait=False) # Clear the loading messages
  module = clip.module # Extract the Flax Module
  variables = {'params': clip.params} # Extract the parameters
  return module, variables
```

Note that `FlaxCLIPVisionModel` itself is not a Flax `Module` which is why we need to do this extra step.

Extracting a submodule

Calling `load_model` from the snippet above returns the `FlaxCLIPModule`, which is composed of `text_model` and `vision_model` submodules.

An easy way to extract the `vision_model` sub-Module defined inside `.setup()` and its variables is to use `flax.linen.Module.bind` on the `clip` Module immediately followed by `flax.linen.Module.unbind` on the `vision_model` sub-Module.

```
import flax.linen as nn

clip, clip_variables = load_model()
vision_model, vision_model_vars = clip.bind(clip_variables).vision_model.unbind()
```

Creating a classifier

To create a classifier define a new Flax `Module` consisting of a `backbone` (the pretrained vision model) and a `head` (the classifier) submodules.

```
from typing import Callable
import jax.numpy as jnp
import jax

class Classifier(nn.Module):
  num_classes: int
  backbone: nn.Module


  @nn.compact
  def __call__(self, x):
    x = self.backbone(x).pooler_output
    x = nn.Dense(
      self.num_classes, name='head', kernel_init=nn.zeros)(x)
    return x
```

To construct a classifier `model`, the `vision_model` Module is passed as the `backbone` to `Classifier`. Then the model's `params` can be randomly initialized by passing fake data that is used to infer the parameter shapes.

```
num_classes = 3
model = Classifier(num_classes=num_classes, backbone=vision_model)

x = jnp.empty((1, 224, 224, 3))
variables = model.init(jax.random.PRNGKey(1), x)
params = variables['params']
```

Transfering the parameters

Since `params` are currently random, the pretrained parameters from `vision_model_vars` have to be transfered to the `params` structure at the appropriate location. This can be done by unfreezing `params`, updating the `backbone` parameters, and freezing the `params` again:

```
from flax.core.frozen_dict import freeze

params = params.unfreeze()
params['backbone'] = vision_model_vars['params']
params = freeze(params)
```

Note: if the model contains other variable collections such as `batch_stats`, these have to be transfered as well.

Optimization

If you need to to train different parts of the model separately, you have three options:

1. Use `stop_gradient`.
2. Filter the parameters for `jax.grad`.
3. Use multiple optimizers for different parameters.

For most situations we recommend using multiple optimizers via Optax's `multi_transform` as its both efficient and can be easily extended to implement many fine-tunning strategies.

optax.multi_transform

To use `optax.multi_transform` following must be defined:

1. The parameter partitions.
2. A mapping between partitions and their optimizer.
3. A pytree with the same shape as the parameters but its leaves containing the corresponding partition label.

To freeze layers with `optax.multi_transform` for the model above, the following setup can be used:

- Define the `trainable` and `frozen` parameter partitions.
- For the `trainable` parameters select the Adam (`optax.adam`) optimizer.
- For the `frozen` parameters select the `optax.set_to_zero` optimizer. This dummy optimizer zeros-out the gradients so no training is done.
- Map parameters to partitions using `flax.traverse_util.path_aware_map`, mark the leaves from the `backbone` as `frozen`, and the rest as `trainable`.

```
from flax import traverse_util
import optax

partition_optimizers = {'trainable': optax.adam(5e-3), 'frozen': optax.set_to_zero()}
param_partitions = freeze(traverse_util.path_aware_map(
  lambda path, v: 'frozen' if 'backbone' in path else 'trainable', params))
tx = optax.multi_transform(partition_optimizers, param_partitions)
```

(continues on next page)

(continued from previous page)

```
# visualize a subset of the param_partitions structure
flat = list(traverse_util.flatten_dict(param_partitions).items())
freeze(traverse_util.unflatten_dict(dict(flat[:2] + flat[-2:])))
```

```
FrozenDict({
    backbone: {
        embeddings: {
            class_embedding: 'frozen',
            patch_embedding: {
                kernel: 'frozen',
            },
        },
    },
    head: {
        bias: 'trainable',
        kernel: 'trainable',
    },
})
```

To implement differential learning rates, the `optax.set_to_zero` can be replaced with any other optimizer, different optimizers and partitioning schemes can be selected depending on the task. For more information on advanced optimizers, refer to Optax's Combining Optimizers documentation.

Creating the `TrainState`

Once the module, params, and optimizer are defined, the `TrainState` can be constructed as usual:

```
from flax.training.train_state import TrainState

state = TrainState.create(
  apply_fn=model.apply,
  params=params,
  tx=tx)
```

Since the optimizer takes care of the freezing or fine-tunning strategy, the `train_step` requires no additional changes, training can proceed normally.

Save and load checkpoints

This guide demonstrates how to save and load Flax checkpoints with Orbax.

Orbax provides a variety of features for saving and loading model data, which you will learn about in this doc:

- Support for various array types and storage formats
- Asynchronous saving to reduce training wait time
- Versioning and automatic bookkeeping of past checkpoints
- Flexible `transformations` to tweak and load old checkpoints
- `jax.sharding`-based API to save and load in multi-host scenarios

Ongoing migration to Orbax:

After July 30 2023, Flax's legacy `flax.training.checkpoints` API will be deprecated in favor of Orbax.

- **If you are a new Flax user**: Use the new `orbax.checkpoint` API, as demonstrated in this guide.
- **If you have legacy `flax.training.checkpoints` code in your project**: Consider the following options:
  - **Migrating your code to Orbax (Recommended)**: Migrate your API calls to `orbax.checkpoint` API by following this migration guide.
  - **Automatically use the Orbax backend**: Add `flax.config.update('flax_use_orbax_checkpointing', True)` to your project, which will let your `flax.training.checkpoints` calls automatically use the Orbax backend to save your checkpoints.
    - **Scheduled flip**: This will become the default mode after **May 2023** (tentative date).
    - Visit Orbax-as-backend troubleshooting section if you meet any issue in the automatic migration.

For backward-compatibility, this guide shows the Orbax-equivalent calls in the Flax legacy `flax.training.checkpoints` API.

If you need to learn more about `orbax.checkpoint`, refer to the Orbax docs.

Setup

Install/upgrade Flax and Orbax. For JAX installation with GPU/TPU support, visit this section on GitHub.

```
# replace with `pip install -U flax` after release 0.6.9.
! pip install -U -qq "git+https://github.com/google/flax.git@main#egg=flax"
```

Note: Before running `import jax`, create eight fake devices to mimic a multi-host environment in this notebook. Note that the order of imports is important here. The `os.environ["XLA_FLAGS"] = '--xla_force_host_platform_device_count=8'` command works only with the CPU backend, which means it won't work with GPU/TPU acceleration on if you're running this notebook in Google Colab. If you are already running the code on multiple devices (for example, in a 4x2 TPU environment), you can skip running the next cell.

```
import os
os.environ["XLA_FLAGS"] = '--xla_force_host_platform_device_count=8'
```

```
from typing import Optional, Any
import shutil

import numpy as np
import jax
from jax import random, numpy as jnp

import flax
from flax import linen as nn
from flax.training import checkpoints, train_state
from flax import struct, serialization
import orbax.checkpoint

import optax
```

```
2024-05-02 00:12:42.253844: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-
↪TRT Warning: Could not find TensorRT
```

```
WARNING:absl:GlobalAsyncCheckpointManager is not imported correctly. Checkpointing of␣
↪GlobalDeviceArrays will not be available.To use the feature, install tensorstore.
```

```
ckpt_dir = 'tmp'

if os.path.exists(ckpt_dir):
    shutil.rmtree(ckpt_dir)  # Remove any existing checkpoints from the last notebook␣
↪run.
```

Save checkpoints

In Orbax and Flax, you can save and load any given JAX pytree. This includes not only typical Python and NumPy containers, but also customized classes extended from `flax.struct.dataclass`. That means you can store almost any data generated — not only your model parameters, but any arrays/dictionaries, metadata/configs, and so on.

First, create a pytree with many data structures and containers, and play with it:

```
# A simple model with one linear layer.
key1, key2 = random.split(random.PRNGKey(0))
x1 = random.normal(key1, (5,))      # A simple JAX array.
model = nn.Dense(features=3)
variables = model.init(key2, x1)

# Flax's TrainState is a pytree dataclass and is supported in checkpointing.
# Define your class with `@flax.struct.dataclass` decorator to make it compatible.
tx = optax.sgd(learning_rate=0.001)      # An Optax SGD optimizer.
state = train_state.TrainState.create(
    apply_fn=model.apply,
    params=variables['params'],
    tx=tx)
# Perform a simple gradient update similar to the one during a normal training workflow.
state = state.apply_gradients(grads=jax.tree_map(jnp.ones_like, state.params))

# Some arbitrary nested pytree with a dictionary, a string, and a NumPy array.
config = {'dimensions': np.array([5, 3]), 'name': 'dense'}

# Bundle everything together.
ckpt = {'model': state, 'config': config, 'data': [x1]}
ckpt
```

```
WARNING:jax._src.xla_bridge:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_
↪LEVEL=0 and rerun for more info.)
```

```
{'model': TrainState(step=1, apply_fn=<bound method Module.apply of Dense(
     # attributes
     features = 3
     use_bias = True
     dtype = None
```

(continues on next page)

(continued from previous page)

```
    param_dtype = float32
    precision = None
    kernel_init = init
    bias_init = zeros
    dot_general = dot_general
)>, params=FrozenDict({
    bias: Array([-0.001, -0.001, -0.001], dtype=float32),
    kernel: Array([[ 0.26048955, -0.61399287, -0.23458514],
           [ 0.11050402, -0.8765793 ,  0.9800635 ],
           [ 0.36260957,  0.18276349, -0.6856061 ],
           [-0.8519373 , -0.6416717 , -0.4818122 ],
           [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32),
}), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7fba50382940>,␣
↪update=<function chain.<locals>.update_fn at 0x7fba5039ad30>), opt_state=(EmptyState(),
↪ EmptyState())),
 'config': {'dimensions': array([5, 3]), 'name': 'dense'},
 'data': [Array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],     ␣
↪dtype=float32)]}
```

With Orbax

Save the checkpoint with `orbax.checkpoint.PyTreeCheckpointer`, directly to the `tmp/orbax/single_save` directory.

Note: An optional `save_args` is provided. This is recommended for performance speedups, as it bundles smaller arrays in your pytree to a single large file instead of multiple smaller files.

```
from flax.training import orbax_utils

orbax_checkpointer = orbax.checkpoint.PyTreeCheckpointer()
save_args = orbax_utils.save_args_from_target(ckpt)
orbax_checkpointer.save('tmp/orbax/single_save', ckpt, save_args=save_args)
```

Next, to use versioning and automatic bookkeeping features, you need to wrap `orbax.checkpoint.CheckpointManager` over `orbax.checkpoint.PyTreeCheckpointer`.

In addition, provide `orbax.checkpoint.CheckpointManagerOptions` that customizes your needs, such as how often and on what criteria you prefer old checkpoints be deleted. See documentation for a full list of options offered.

`orbax.checkpoint.CheckpointManager` should be placed at the top-level outside your training steps to manage your saves.

```
options = orbax.checkpoint.CheckpointManagerOptions(max_to_keep=2, create=True)
checkpoint_manager = orbax.checkpoint.CheckpointManager(
    'tmp/orbax/managed', orbax_checkpointer, options)

# Inside a training loop
for step in range(5):
    # ... do your training
    checkpoint_manager.save(step, ckpt, save_kwargs={'save_args': save_args})

os.listdir('tmp/orbax/managed')  # Because max_to_keep=2, only step 3 and 4 are retained
```

```
['3', '4']
```

With the legacy API

And here's how to save with the legacy Flax checkpointing utilities (note that this provides less management features compared with `orbax.checkpoint.CheckpointManagerOptions`):

```
# Import Flax Checkpoints.
from flax.training import checkpoints

checkpoints.save_checkpoint(ckpt_dir='tmp/flax-checkpointing',
                            target=ckpt,
                            step=0,
                            overwrite=True,
                            keep=2)
```

```
'tmp/flax-checkpointing/checkpoint_0'
```

Restore checkpoints

With Orbax

In Orbax, call `.restore()` for either `orbax.checkpoint.PyTreeCheckpointer` or `orbax.checkpoint.CheckpointManager` to restore your checkpoint in the raw pytree format.

```
raw_restored = orbax_checkpointer.restore('tmp/orbax/single_save')
raw_restored
```

```
{'config': {'dimensions': array([5, 3]), 'name': 'dense'},
 'data': [array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
        dtype=float32)],
 'model': {'opt_state': [None, None],
  'params': {'bias': array([-0.001, -0.001, -0.001], dtype=float32),
   'kernel': array([[ 0.26048955, -0.61399287, -0.23458514],
          [ 0.11050402, -0.8765793 ,  0.9800635 ],
          [ 0.36260957,  0.18276349, -0.6856061 ],
          [-0.8519373 , -0.6416717 , -0.4818122 ],
          [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32)},
  'step': 1}}
```

Note that the `step` number is required for `CheckpointManger`. You can also use `.latest_step()` to find the latest step available.

```
step = checkpoint_manager.latest_step()  # step = 4
checkpoint_manager.restore(step)
```

```
{'config': {'dimensions': array([5, 3]), 'name': 'dense'},
 'data': [array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
        dtype=float32)],
```

(continues on next page)

(continued from previous page)

```
 'model': {'opt_state': [None, None],
  'params': {'bias': array([-0.001, -0.001, -0.001], dtype=float32),
   'kernel': array([[ 0.26048955, -0.61399287, -0.23458514],
          [ 0.11050402, -0.8765793 ,  0.9800635 ],
          [ 0.36260957,  0.18276349, -0.6856061 ],
          [-0.8519373 , -0.6416717 , -0.4818122 ],
          [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32)},
  'step': 1}}
```

With the legacy API

Note that with the migration to Orbax in progress, `flax.training.checkpointing.restore_checkpoint` can automatically identify whether a checkpoint is saved in the legacy Flax format or with an Orbax backend, and restore the pytree correctly. Therefore, adding `flax.config.update('flax_use_orbax_checkpointing', True)` won't hurt your ability to restore old checkpoints.

Here's how to restore checkpoints using the legacy API:

```
raw_restored = checkpoints.restore_checkpoint(ckpt_dir='tmp/flax-checkpointing',␣
↪target=None)
raw_restored
```

```
{'model': {'step': 1,
  'params': {'bias': array([-0.001, -0.001, -0.001], dtype=float32),
   'kernel': array([[ 0.26048955, -0.61399287, -0.23458514],
          [ 0.11050402, -0.8765793 ,  0.9800635 ],
          [ 0.36260957,  0.18276349, -0.6856061 ],
          [-0.8519373 , -0.6416717 , -0.4818122 ],
          [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32)},
  'opt_state': {'0': {}, '1': {}}},
 'config': {'dimensions': array([5, 3]), 'name': 'dense'},
 'data': {'0': array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
       dtype=float32)}}
```

Restore with custom dataclasses

With Orbax

- The pytrees restored in the previous examples are in the form of raw dictionaries. Original pytrees contain custom dataclasses like `TrainState` and `optax` states.
- This is because when restoring a pytree, the program does not yet know which structure it once belonged to.
- To resolve this, you should first provide an example pytree to let Orbax or Flax know exactly which structure to restore to.

This section demonstrates how to set up any custom Flax dataclass explicitly, and have the same structure as a saved checkpoint.

Note: Data that was a JAX NumPy array (`jnp.array`) format will be restored as a NumPy array (`numpy.array`). This would not affect your work because JAX will automatically convert NumPy arrays to JAX arrays once the computation starts.

```
empty_state = train_state.TrainState.create(
    apply_fn=model.apply,
    params=jax.tree_map(np.zeros_like, variables['params']),  # values of the tree leaf␣
↪doesn't matter
    tx=tx,
)
empty_config = {'dimensions': np.array([0, 0]), 'name': ''}
target = {'model': empty_state, 'config': empty_config, 'data': [jnp.zeros_like(x1)]}
state_restored = orbax_checkpointer.restore('tmp/orbax/single_save', item=target)
state_restored
```

```
{'config': {'dimensions': array([5, 3]), 'name': 'dense'},
 'data': [array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
       dtype=float32)],
 'model': TrainState(step=1, apply_fn=<bound method Module.apply of Dense(
    # attributes
    features = 3
    use_bias = True
    dtype = None
    param_dtype = float32
    precision = None
    kernel_init = init
    bias_init = zeros
    dot_general = dot_general
 )>, params=FrozenDict({
    bias: array([-0.001, -0.001, -0.001], dtype=float32),
    kernel: array([[ 0.26048955, -0.61399287, -0.23458514],
           [ 0.11050402, -0.8765793 ,  0.9800635 ],
           [ 0.36260957,  0.18276349, -0.6856061 ],
           [-0.8519373 , -0.6416717 , -0.4818122 ],
           [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32),
 }), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7fba50382940>,␣
↪update=<function chain.<locals>.update_fn at 0x7fba5039ad30>), opt_state=(EmptyState(),
↪ EmptyState()))}
```

With the legacy API

Alternatively, you can restore from Orbax `CheckpointManager` and from the legacy Flax code as follows:

```
checkpoint_manager.restore(4, items=target)
```

```
{'config': {'dimensions': array([5, 3]), 'name': 'dense'},
 'data': [array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
       dtype=float32)],
 'model': TrainState(step=1, apply_fn=<bound method Module.apply of Dense(
    # attributes
    features = 3
    use_bias = True
    dtype = None
    param_dtype = float32
    precision = None
```

(continues on next page)

(continued from previous page)

```
    kernel_init = init
    bias_init = zeros
    dot_general = dot_general
)>, params=FrozenDict({
    bias: array([-0.001, -0.001, -0.001], dtype=float32),
    kernel: array([[ 0.26048955, -0.61399287, -0.23458514],
           [ 0.11050402, -0.8765793 ,  0.9800635 ],
           [ 0.36260957,  0.18276349, -0.6856061 ],
           [-0.8519373 , -0.6416717 , -0.4818122 ],
           [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32),
}), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7fba50382940>,␣
↪update=<function chain.<locals>.update_fn at 0x7fba5039ad30>), opt_state=(EmptyState(),
↪ EmptyState()))}
```

```
checkpoints.restore_checkpoint(ckpt_dir='tmp/flax-checkpointing', target=target)
```

```
{'model': TrainState(step=1, apply_fn=<bound method Module.apply of Dense(
    # attributes
    features = 3
    use_bias = True
    dtype = None
    param_dtype = float32
    precision = None
    kernel_init = init
    bias_init = zeros
    dot_general = dot_general
)>, params=FrozenDict({
    bias: array([-0.001, -0.001, -0.001], dtype=float32),
    kernel: array([[ 0.26048955, -0.61399287, -0.23458514],
           [ 0.11050402, -0.8765793 ,  0.9800635 ],
           [ 0.36260957,  0.18276349, -0.6856061 ],
           [-0.8519373 , -0.6416717 , -0.4818122 ],
           [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32),
}), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7fba50382940>,␣
↪update=<function chain.<locals>.update_fn at 0x7fba5039ad30>), opt_state=(EmptyState(),
↪ EmptyState())),
 'config': {'dimensions': array([5, 3]), 'name': 'dense'},
 'data': [array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
        dtype=float32)]}
```

It's often recommended to refactor out the process of initializing a checkpoint's structure (for example, a `TrainState`), so that saving/loading is easier and less error-prone. This is because functions and complex objects like `apply_fn` and `tx` (optimizer) cannot be serialized into the checkpoint file and must be initialized by code.

Restore when checkpoint structures differ

During your development, your checkpoint structure will change when changing the model, adding/removing fields during tweaking, and so on.

This section explains how to load old data to your new code.

Below is a simple example — a `CustomTrainState` extended from `flax.training.train_state.TrainState` that contains an extra field called `batch_stats`. When working on a real-world model, you may need this when applying batch normalization.

Here, you store the new `CustomTrainState` as step 5, while step 4 contains the old/previous `TrainState`.

```
class CustomTrainState(train_state.TrainState):
    batch_stats: Any = None

custom_state = CustomTrainState.create(
    apply_fn=state.apply_fn,
    params=state.params,
    tx=state.tx,
    batch_stats=np.arange(10),
)

custom_ckpt = {'model': custom_state, 'config': config, 'data': [x1]}
# Use a custom state to read the old `TrainState` checkpoint.
custom_target = {'model': custom_state, 'config': None, 'data': [jnp.zeros_like(x1)]}

# Save it in Orbax.
custom_save_args = orbax_utils.save_args_from_target(custom_ckpt)
checkpoint_manager.save(5, custom_ckpt, save_kwargs={'save_args': custom_save_args})
```

```
True
```

It is recommended to keep your checkpoints up-to-date with your pytree dataclass definitions. However, you might be forced to restore the checkpoints with incompatible reference objects at runtime. When this happens, the checkpoint restoration will try to respect the structure of the reference when given.

Below are examples of a few common scenarios.

Scenario 1: When a reference object is partial

If your reference object is a subtree of your checkpoint, the restoration will ignore the additional field(s) and restore a checkpoint with the same structure as the reference.

Like in the example below, the `batch_stats` field in `CustomTrainState` was ignored, and the checkpoint was restored as a `TrainState`.

This can also be useful for reading only part of your checkpoint.

```
restored = checkpoint_manager.restore(5, items=target)
assert not hasattr(restored, 'batch_stats')
assert type(restored['model']) == train_state.TrainState
restored
```

```
{'config': {'dimensions': array([5, 3]), 'name': 'dense'},
 'data': [array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
       dtype=float32)],
 'model': TrainState(step=0, apply_fn=<bound method Module.apply of Dense(
    # attributes
    features = 3
    use_bias = True
    dtype = None
    param_dtype = float32
    precision = None
    kernel_init = init
    bias_init = zeros
    dot_general = dot_general
 )>, params=FrozenDict({
    bias: array([-0.001, -0.001, -0.001], dtype=float32),
    kernel: array([[ 0.26048955, -0.61399287, -0.23458514],
           [ 0.11050402, -0.8765793 ,  0.9800635 ],
           [ 0.36260957,  0.18276349, -0.6856061 ],
           [-0.8519373 , -0.6416717 , -0.4818122 ],
           [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32),
 }), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7fba50382940>,␣
↪update=<function chain.<locals>.update_fn at 0x7fba5039ad30>), opt_state=(EmptyState(),
↪ EmptyState()))}
```

Scenario 2: When a checkpoint is partial

On the other hand, if the reference object contains a value that is not available in the checkpoint, the checkpointing code will by default warn that some data is not compatible.

To bypass the error, you need to pass an Orbax `transform` that teaches Orbax how to conform this checkpoint into the structure of the `custom_target`.

In this case, pass a default `{}` that lets Orbax use values in the `custom_target` to fill in the blank. This allows you to restore an old checkpoint into a new data structure, the `CustomTrainState`.

```
try:
    checkpoint_manager.restore(4, items=custom_target)
except KeyError as e:
    print(f'KeyError when target state has an unmentioned field: {e}')
    print('')

# Step 4 is an original `TrainState`, without the `batch_stats`
restored = checkpoint_manager.restore(4, items=custom_target,
                                      restore_kwargs={'transforms': {}})
assert type(restored['model']) == CustomTrainState
np.testing.assert_equal(restored['model'].batch_stats,
                        custom_target['model'].batch_stats)
restored
```

```
KeyError when target state has an unmentioned field: 'batch_stats'
```

```
{'config': None,
 'data': [array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
       dtype=float32)],
 'model': CustomTrainState(step=1, apply_fn=<bound method Module.apply of Dense(
     # attributes
     features = 3
     use_bias = True
     dtype = None
     param_dtype = float32
     precision = None
     kernel_init = init
     bias_init = zeros
     dot_general = dot_general
 )>, params=FrozenDict({
     bias: array([-0.001, -0.001, -0.001], dtype=float32),
     kernel: array([[ 0.26048955, -0.61399287, -0.23458514],
            [ 0.11050402, -0.8765793 ,  0.9800635 ],
            [ 0.36260957,  0.18276349, -0.6856061 ],
            [-0.8519373 , -0.6416717 , -0.4818122 ],
            [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32),
 }), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7fba50382940>,␣
↪update=<function chain.<locals>.update_fn at 0x7fba5039ad30>), opt_state=(EmptyState(),
↪ EmptyState()), batch_stats=array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))}
```

With Orbax

If you have already saved your checkpoints with the Orbax backend, you can use `orbax_transforms` to access this `transforms` argument in the Flax API.

```
# Save in the "Flax-with-Orbax" backend.
flax.config.update('flax_use_orbax_checkpointing', True)
checkpoints.save_checkpoint(ckpt_dir='tmp/flax-checkpointing',
                            target=ckpt,
                            step=4,
                            overwrite=True,
                            keep=2)

checkpoints.restore_checkpoint('tmp/flax-checkpointing', target=custom_target, step=4,
                               orbax_transforms={})
```

```
{'model': CustomTrainState(step=1, apply_fn=<bound method Module.apply of Dense(
     # attributes
     features = 3
     use_bias = True
     dtype = None
     param_dtype = float32
     precision = None
     kernel_init = init
     bias_init = zeros
     dot_general = dot_general
 )>, params=FrozenDict({
```

(continues on next page)

(continued from previous page)

```
    bias: array([-0.001, -0.001, -0.001], dtype=float32),
    kernel: array([[ 0.26048955, -0.61399287, -0.23458514],
           [ 0.11050402, -0.8765793 ,  0.9800635 ],
           [ 0.36260957,  0.18276349, -0.6856061 ],
           [-0.8519373 , -0.6416717 , -0.4818122 ],
           [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32),
}), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7fba50382940>,␣
↪update=<function chain.<locals>.update_fn at 0x7fba5039ad30>), opt_state=(EmptyState(),
↪ EmptyState()), batch_stats=array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])),
'config': None,
'data': [array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
       dtype=float32)]}
```

With the legacy API

Using the legacy `flax.training.checkpoints` API, similar things are doable too, but they are not as flexible as the Orbax Transformations.

You need to restore the checkpoint to a raw dict with `target=None`, modify the structure accordingly, and then deserialize it back to the original target.

```
# Save using the legacy Flax `checkpoints` API.
flax.config.update('flax_use_orbax_checkpointing', False)
checkpoints.save_checkpoint(ckpt_dir='tmp/flax-checkpointing',
                            target=ckpt,
                            step=5,
                            overwrite=True,
                            keep=2)

# Pass no target to get a raw state dictionary first.
raw_state_dict = checkpoints.restore_checkpoint('tmp/flax-checkpointing', target=None,␣
↪step=5)
# Add/remove fields as needed.
raw_state_dict['model']['batch_stats'] = np.flip(np.arange(10))
# Restore the classes with correct target now
flax.serialization.from_state_dict(custom_target, raw_state_dict)
```

```
{'model': CustomTrainState(step=1, apply_fn=<bound method Module.apply of Dense(
    # attributes
    features = 3
    use_bias = True
    dtype = None
    param_dtype = float32
    precision = None
    kernel_init = init
    bias_init = zeros
    dot_general = dot_general
)>, params=FrozenDict({
    bias: array([-0.001, -0.001, -0.001], dtype=float32),
    kernel: array([[ 0.26048955, -0.61399287, -0.23458514],
           [ 0.11050402, -0.8765793 ,  0.9800635 ],
```

(continues on next page)

(continued from previous page)

```
           [ 0.36260957,  0.18276349, -0.6856061 ],
           [-0.8519373 , -0.6416717 , -0.4818122 ],
           [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32),
 }), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7fba50382940>,␣
↪update=<function chain.<locals>.update_fn at 0x7fba5039ad30>), opt_state=(EmptyState(),
↪ EmptyState()), batch_stats=array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])),
 'config': {'dimensions': array([5, 3]), 'name': 'dense'},
 'data': [array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
        dtype=float32)]}
```

Asynchronized checkpointing

Checkpointing is I/O heavy, and if you have a large amount of data to save, it may be worthwhile to put it into a background thread, while continuing with your training.

You can do this by creating an `orbax.checkpoint.AsyncCheckpointer` in place of the `orbax.checkpoint.PyTreeCheckpointer`.

Note: You should use the same `async_checkpointer` to handle all your async saves across your training steps, so that it can make sure that a previous async save is done before the next one begins. This enables bookkeeping, such as `keep` (the number of checkpoints) and `overwrite` to be consistent across steps.

Whenever you want to explicitly wait until an async save is done, you can call `async_checkpointer.wait_until_finished()`.

```
# `orbax.checkpoint.AsyncCheckpointer` needs some multi-process initialization, because␣
↪it was
# originally designed for multi-process large model checkpointing.
# For Python notebooks or other single-process settings, just set up with `num_
↪processes=1`.
# Refer to https://jax.readthedocs.io/en/latest/multi_process.html#initializing-the-
↪cluster
# for how to set it up in multi-process scenarios.
jax.distributed.initialize("localhost:8889", num_processes=1, process_id=0)

async_checkpointer = orbax.checkpoint.AsyncCheckpointer(
    orbax.checkpoint.PyTreeCheckpointHandler(), timeout_secs=50)

# Save your job:
async_checkpointer.save('tmp/orbax/single_save_async', ckpt, save_args=save_args)
# ... Continue with your work...

# ... Until a time when you want to wait until the save completes:
async_checkpointer.wait_until_finished()  # Blocks until the checkpoint saving is␣
↪completed.
async_checkpointer.restore('tmp/orbax/single_save_async', item=target)
```

```
{'config': {'dimensions': array([5, 3]), 'name': 'dense'},
 'data': [array([0.59902626, 0.2172144 , 2.4202902 , 0.03266738, 1.2164948 ],
        dtype=float32)],
 'model': TrainState(step=1, apply_fn=<bound method Module.apply of Dense(
     # attributes
```

(continues on next page)

(continued from previous page)

```
    features = 3
    use_bias = True
    dtype = None
    param_dtype = float32
    precision = None
    kernel_init = init
    bias_init = zeros
    dot_general = dot_general
)>, params=FrozenDict({
    bias: array([-0.001, -0.001, -0.001], dtype=float32),
    kernel: array([[ 0.26048955, -0.61399287, -0.23458514],
           [ 0.11050402, -0.8765793 ,  0.9800635 ],
           [ 0.36260957,  0.18276349, -0.6856061 ],
           [-0.8519373 , -0.6416717 , -0.4818122 ],
           [-0.6886102 , -0.33987316, -0.05898903]], dtype=float32),
}), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7fba50382940>,␣
↪update=<function chain.<locals>.update_fn at 0x7fba5039ad30>), opt_state=(EmptyState(),
↪ EmptyState()))}
```

If you are using Orbax `CheckpointManager`, just pass in the async_checkpointer when initializing it. Then, in practice, call `async_checkpoint_manager.wait_until_finished()` instead.

```
async_checkpoint_manager = orbax.checkpoint.CheckpointManager(
    'tmp/orbax/managed_async', async_checkpointer, options)
async_checkpoint_manager.wait_until_finished()
```

Multi-host/multi-process checkpointing

JAX provides a few ways to scale up your code on multiple hosts at the same time. This usually happens when the number of devices (CPU/GPU/TPU) is so large that different devices are managed by different hosts (CPU). To get started on JAX in multi-process settings, check out Using JAX in multi-host and multi-process environments and the distributed array guide.

In the Single Program Multi Data (SPMD) paradigm with JAX `jit`, a large multi-process array can have its data sharded across different devices. (Note that JAX `pjit` and `jit` have been merged into a single unified interface. To learn about compiling and executing JAX functions in multi-host or multi-core environments, refer to this guide and the jax.Array migration guide.) When a multi-process array is serialized, each host dumps its data shards to a single shared storage, such as a Google Cloud bucket.

Orbax supports saving and loading pytrees with multi-process arrays in the same fashion as single-process pytrees. However, it's recommended to use the asynchronized `orbax.AsyncCheckpointer` to save large multi-process arrays on another thread, so that you can perform computation alongside the saves. With pure Orbax, saving checkpoints in a multi-process context uses the same API as in a single-process context.

```
from jax.sharding import PartitionSpec, NamedSharding

# Create an array sharded across multiple devices.
mesh_shape = (4, 2)
devices = np.asarray(jax.devices()).reshape(*mesh_shape)
mesh = jax.sharding.Mesh(devices, ('x', 'y'))

mp_array = jax.device_put(np.arange(8 * 2).reshape(8, 2),
```

(continues on next page)

(continued from previous page)

```
                            NamedSharding(mesh, PartitionSpec('x', 'y')))

# Make it a pytree.
mp_ckpt = {'model': mp_array}
```

```
async_checkpoint_manager.save(0, mp_ckpt)
async_checkpoint_manager.wait_until_finished()
```

When restoring a checkpoint with multi-process arrays, you need to specify what `sharding` each array should be restored back to. Otherwise, they will be restored as large `np.array`s on process 0, costing time and memory.

(In this notebook, since we are on single-process, it will be restored as `np.array` even if we provide shardings.)

With Orbax

Orbax allows you to specify this by passing a pytree of `sharding`s in `restore_args`. If you already have a reference pytree that has all the arrays with the right sharding, you can use `orbax_utils.restore_args_from_target` to transform it into the `restore_args` that Orbax needs.

```
# The reference doesn't need to be as large as your checkpoint!
# Just make sure it has the `.sharding` you want.
mp_smaller = jax.device_put(np.arange(8).reshape(4, 2),
                            NamedSharding(mesh, PartitionSpec('x', 'y')))
ref_ckpt = {'model': mp_smaller}

restore_args = orbax_utils.restore_args_from_target(ref_ckpt)
async_checkpoint_manager.restore(
    0, items=ref_ckpt, restore_kwargs={'restore_args': restore_args})
```

```
{'model': array([[ 0,  1],
        [ 2,  3],
        [ 4,  5],
        [ 6,  7],
        [ 8,  9],
        [10, 11],
        [12, 13],
        [14, 15]], dtype=int32)}
```

With the legacy Flax: use `save_checkpoint_multiprocess`

In legacy Flax, to save multi-process arrays, use `flax.training.checkpoints.save_checkpoint_multiprocess()` in place of `save_checkpoint()` and with the same arguments.

If your checkpoint is too large, you can specify `timeout_secs` in the manager and give it more time to finish writing.

```
async_checkpointer = orbax.checkpoint.AsyncCheckpointer(orbax.checkpoint.
↪PyTreeCheckpointHandler(), timeout_secs=50)
checkpoints.save_checkpoint_multiprocess(ckpt_dir,
                                         mp_ckpt,
                                         step=3,
```

(continues on next page)

(continued from previous page)

```
                                          overwrite=True,
                                          keep=4,
                                          orbax_checkpointer=async_checkpointer)
```

```
'tmp/checkpoint_3'
```

```
mp_restored = checkpoints.restore_checkpoint(ckpt_dir,
                                             target=ref_ckpt,
                                             step=3,
                                             orbax_checkpointer=async_checkpointer)
mp_restored
```

```
{'model': array([[ 0,  1],
        [ 2,  3],
        [ 4,  5],
        [ 6,  7],
        [ 8,  9],
        [10, 11],
        [12, 13],
        [14, 15]], dtype=int32)}
```

Orbax-as-backend troubleshooting

As an intermediate stage of the migration (to Orbax from the legacy Flax `checkpoints` API), `flax.training.checkpoints` APIs will start to use Orbax as their backend when saving checkpoints starting from May 15, 2023.

Checkpoints saved with the Orbax backend can be readable by either `flax.training.checkpoints.restore_checkpoint` or `orbax.checkpoint.PyTreeCheckpointer`.

Code-wise, this is equivalent to setting the config flag `flax.config.flax_use_orbax_checkpointing` default to `True`. You can overwrite this value in your project with `flax.config.update('flax_use_orbax_checkpointing', <BoolValue>)` at any time.

In general, this automatic migration will not affect most users. However, you may encounter issues if your API usage follows some specific pattern. Check out the sections below for troubleshooting.

If your devices hang when writing checkpoints

If you are running in a multi-host environment (usually anything larger than 8 TPU devices) and your devices hang when writing checkpoints, check if your code is in the following pattern (that is, the `save_checkpoint` only ran on host `0`):

```
if jax.process_index() == 0:
  flax.training.checkpoints.save_checkpoint(...)
```

Unfortunately this is a legacy pattern that will be deprecated and won't be supported, because in a multi-process environment, the checkpointing code should coordinate among hosts instead of being triggered only on the host `0`. Replacing the code above with the following should resolve the hang issue:

```
flax.training.checkpoints.save_checkpoint_multiprocess(...)
```

If you don't save pytrees

Orbax uses `orbax.checkpoint.PyTreeCheckpointHandler` to save checkpoints, which means they only save pytrees.

If you want to save singular arrays or numbers, you have two options:

1. Use `orbax.ArrayCheckpointHandler` to save them following this migration section.
2. Wrap it inside a pytree and save as usual.

5.2.4 Parallel training

Ensembling on multiple devices

We show how to train an ensemble of CNNs on the MNIST dataset, where the size of the ensemble is equal to the number of available devices. In short, this change be described as:

- make a number of functions parallel using `jax.pmap()`,
- split the random seed to obtain different parameter initialization,
- replicate the inputs and unreplicate the outputs where necessary,
- average probabilities across devices to compute the predictions.

In this HOWTO we omit some of the code such as imports, the CNN module, and metrics computation, but they can be found in the MNIST example.

Parallel functions

We start by creating a parallel version of `create_train_state()`, which retrieves the initial parameters of the models. We do this using `jax.pmap()`. The effect of "pmapping" a function is that it will compile the function with XLA (similar to `jax.jit()`), but execute it in parallel on XLA devices (e.g., GPUs/TPUs).

Single-model

```
def create_train_state(rng, learning_rate, momentum):
  cnn = CNN()
  params = cnn.init(rng, jnp.ones([1, 28, 28, 1]))['params']
  tx = optax.sgd(learning_rate, momentum)
  return train_state.TrainState.create(
      apply_fn=cnn.apply, params=params, tx=tx)
```

Ensemble

```
@functools.partial(jax.pmap, static_broadcasted_argnums=(1, 2))
def create_train_state(rng, learning_rate, momentum):
  cnn = CNN()
  params = cnn.init(rng, jnp.ones([1, 28, 28, 1]))['params']
  tx = optax.sgd(learning_rate, momentum)
  return train_state.TrainState.create(
      apply_fn=cnn.apply, params=params, tx=tx)
```

Note that for the single-model code above, we use `jax.jit()` to lazily initialize the model (see Module.init's documentation for more details). For the ensembling case, `jax.pmap()` will map over the first axis of the provided argument `rng` by default, so we should make sure that we provide a different value for each device when we call this function later on.

Note also how we specify that `learning_rate` and `momentum` are static arguments, which means the concrete values of these arguments will be used, rather than abstract shapes. This is necessary because the provided arguments will be scalar values. For more details see JIT mechanics: tracing and static variables.

Next we simply do the same for the functions `apply_model()` and `update_model()`. To compute the predictions from the ensemble, we take the average of the individual probabilities. We use `jax.lax.pmean()` to compute the average *across devices*. This also requires us to specify the `axis_name` to both `jax.pmap()` and `jax.lax.pmean()`.

Single-model

```
@jax.jit
def apply_model(state, images, labels):
  def loss_fn(params):
    logits = CNN().apply({'params': params}, images)
    one_hot = jax.nn.one_hot(labels, 10)
    loss = optax.softmax_cross_entropy(logits=logits, labels=one_hot).mean()
    return loss, logits

  grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
  (loss, logits), grads = grad_fn(state.params)

  accuracy = jnp.mean(jnp.argmax(logits, -1) == labels)
  return grads, loss, accuracy

@jax.jit
def update_model(state, grads):
  return state.apply_gradients(grads=grads)
```

Ensemble

```
@functools.partial(jax.pmap, axis_name='ensemble')
def apply_model(state, images, labels):
  def loss_fn(params):
    logits = CNN().apply({'params': params}, images)
    one_hot = jax.nn.one_hot(labels, 10)
    loss = optax.softmax_cross_entropy(logits=logits, labels=one_hot).mean()
    return loss, logits

  grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
  (loss, logits), grads = grad_fn(state.params)
  probs = jax.lax.pmean(jax.nn.softmax(logits), axis_name='ensemble')
  accuracy = jnp.mean(jnp.argmax(probs, -1) == labels)
  return grads, loss, accuracy

@jax.pmap
def update_model(state, grads):
  return state.apply_gradients(grads=grads)
```

Training the Ensemble

Next we transform the `train_epoch()` function. When calling the pmapped functions from above, we mainly need to take care of duplicating the arguments for all devices where necessary, and de-duplicating the return values.

Single-model

```
def train_epoch(state, train_ds, batch_size, rng):
  train_ds_size = len(train_ds['image'])
  steps_per_epoch = train_ds_size // batch_size

  perms = jax.random.permutation(rng, len(train_ds['image']))
  perms = perms[:steps_per_epoch * batch_size]
  perms = perms.reshape((steps_per_epoch, batch_size))

  epoch_loss = []
  epoch_accuracy = []

  for perm in perms:
    batch_images = train_ds['image'][perm, ...]
    batch_labels = train_ds['label'][perm, ...]
    grads, loss, accuracy = apply_model(state, batch_images, batch_labels)
    state = update_model(state, grads)
    epoch_loss.append(loss)
    epoch_accuracy.append(accuracy)
  train_loss = np.mean(epoch_loss)
  train_accuracy = np.mean(epoch_accuracy)
  return state, train_loss, train_accuracy
```

Ensemble

```
def train_epoch(state, train_ds, batch_size, rng):
  train_ds_size = len(train_ds['image'])
  steps_per_epoch = train_ds_size // batch_size

  perms = jax.random.permutation(rng, len(train_ds['image']))
  perms = perms[:steps_per_epoch * batch_size]
  perms = perms.reshape((steps_per_epoch, batch_size))

  epoch_loss = []
  epoch_accuracy = []

  for perm in perms:
    batch_images = jax_utils.replicate(train_ds['image'][perm, ...])
    batch_labels = jax_utils.replicate(train_ds['label'][perm, ...])
    grads, loss, accuracy = apply_model(state, batch_images, batch_labels)
    state = update_model(state, grads)
    epoch_loss.append(jax_utils.unreplicate(loss))
    epoch_accuracy.append(jax_utils.unreplicate(accuracy))
  train_loss = np.mean(epoch_loss)
  train_accuracy = np.mean(epoch_accuracy)
  return state, train_loss, train_accuracy
```

As can be seen, we do not have to make any changes to the logic around the `state`. This is because, as we will see below in our training code, the train state is replicated already, so when we pass it to `train_step()`, things will just work fine since `train_step()` is pmapped. However, the train dataset is not yet replicated, so we do that here. Since replicating the entire train dataset is too memory intensive we do it at the batch level.

We can now rewrite the actual training logic. This consists of two simple changes: making sure the RNGs are replicated when we pass them to `create_train_state()`, and replicating the test dataset, which is much smaller than the train dataset so we can do this for the entire dataset directly.

Single-model

```
train_ds, test_ds = get_datasets()

rng = jax.random.PRNGKey(0)

rng, init_rng = jax.random.split(rng)
state = create_train_state(init_rng, learning_rate, momentum)


for epoch in range(1, num_epochs + 1):
  rng, input_rng = jax.random.split(rng)
  state, train_loss, train_accuracy = train_epoch(
      state, train_ds, batch_size, input_rng)

  _, test_loss, test_accuracy = apply_model(
      state, test_ds['image'], test_ds['label'])
```

(continues on next page)

(continued from previous page)

```
  logging.info(
      'epoch:% 3d, train_loss: %.4f, train_accuracy: %.2f, '
      'test_loss: %.4f, test_accuracy: %.2f'
      % (epoch, train_loss, train_accuracy * 100, test_loss,
         test_accuracy * 100))
```

Ensemble

```
train_ds, test_ds = get_datasets()
test_ds = jax_utils.replicate(test_ds)
rng = jax.random.PRNGKey(0)

rng, init_rng = jax.random.split(rng)
state = create_train_state(jax.random.split(init_rng, jax.device_count()),
                           learning_rate, momentum)

for epoch in range(1, num_epochs + 1):
  rng, input_rng = jax.random.split(rng)
  state, train_loss, train_accuracy = train_epoch(
      state, train_ds, batch_size, input_rng)

  _, test_loss, test_accuracy = jax_utils.unreplicate(
      apply_model(state, test_ds['image'], test_ds['label']))

  logging.info(
      'epoch:% 3d, train_loss: %.4f, train_accuracy: %.2f, '
      'test_loss: %.4f, test_accuracy: %.2f'
      % (epoch, train_loss, train_accuracy * 100, test_loss,
         test_accuracy * 100))
```

Scale up Flax Modules on multiple devices with `pjit`

This guide shows how to scale up Flax Modules on multiple devices and hosts using JAX's `pjit` and `flax.linen.spmd`.

Flax and `pjit`

`jax.experimental.pjit` provides a way to automatically compile and scale up JAX computations. `pjit` has the following benefits:

- `pjit` has the similar interface of `jax.jit` and works as a decorator on a function that needs to be compiled.
- When using `pjit`, you can write code as if it runs on a single device, and `pjit` will automatically compile and run it on multiple devices using the Single Program Multi Data (SPMD) paradigm.
- With `pjit` you can state how the input and output of your code is partitioned across devices, and the compiler will figure out how to: 1) partition everything inside; and 2) compile inter-device communications.

To learn more, refer to JAX-101 pjit tutorial and JAX in multi-process environments.

Flax provides several functionalities that can help you use `pjit` on Flax Modules, including:

1. An interface to specify partitions of your data when defining `flax.linen.Module`.
2. Utility functions to generate the partition information that `pjit` requires to run.
3. An interface to customize your axis names called "logical axis annotations" to decouple both your Module code and partition plan to experiment with different partition layouts more easily.

Setup

Install Flax from HEAD:

```
# Once Flax v0.6.4 is released, use `pip3 install flax`.
! pip3 install -qq "git+https://github.com/google/flax.git@main#egg=flax"
```

Imports

Import some necessary dependencies.

Note: This guide uses the `--xla_force_host_platform_device_count=8` flag to emulate multiple devices in a CPU environment in a Google Colab/Jupyter Notebook. Check out the JAX-101 pjit tutorial to learn more about emulating a multi-device TPU environment (in which case you should ignore running `os.environ`).

```
import os
os.environ["XLA_FLAGS"] = '--xla_force_host_platform_device_count=8'
```

```
import functools
import numpy as np
import jax

from jax import lax, random, numpy as jnp

import flax
from flax import struct, traverse_util, linen as nn
from flax.linen import spmd # Flax Linen SPMD.
from flax.core import freeze, unfreeze
from flax.training import train_state, checkpoints

import optax # Optax for common losses and optimizers.
```

```
2024-05-02 00:11:31.353559: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-
↪TRT Warning: Could not find TensorRT
```

```
WARNING:absl:GlobalAsyncCheckpointManager is not imported correctly. Checkpointing of␣
↪GlobalDeviceArrays will not be available.To use the feature, install tensorstore.
```

Next, import all the `pjit`-related libraries.

> **Note:** `jax.experimental.pjit` is still in the experimental package of JAX, so there may be changes in the API in future.

1. Start a 2x4 device mesh (8 devices)—this is the same as the layout of TPU v3-8.
2. Annotate each axis with a name. A typical way to annotate axis names is `('data', 'model')`, where:

- `'data'`: the mesh dimension used for data-parallel sharding of the batch dimension of inputs and activations.

- 'model': the mesh dimension used for sharding parameters of the model across devices.

```
from jax.experimental.pjit import pjit, with_sharding_constraint
from jax.sharding import Mesh, PartitionSpec
from jax.experimental import mesh_utils

# Start a device mesh.
device_mesh = mesh_utils.create_device_mesh((2, 4))
print(device_mesh)
# Annotate each axis with a name.
mesh = Mesh(devices=device_mesh, axis_names=('data', 'model'))
mesh
```

```
WARNING:jax._src.xla_bridge:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_
↪LEVEL=0 and rerun for more info.)
```

```
[[CpuDevice(id=0) CpuDevice(id=1) CpuDevice(id=2) CpuDevice(id=3)]
 [CpuDevice(id=4) CpuDevice(id=5) CpuDevice(id=6) CpuDevice(id=7)]]
```

```
Mesh(device_ids=array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), axis_names=('data', 'model'))
```

Define a layer

Before defining a model, create an example layer called `DotReluDot` (by subclassing `flax.linen.Module`), which creates two parameters `W1` and `W2` for dot product multiplication, and uses the `jax.nn.relu` (ReLU) activation function in-between.

To use this layer in `pjit` efficiently, apply the following APIs to annotate the parameters and intermediate variables correctly:

1. Use `flax.linen.with_partitioning` to decorate the initializer function when creating parameters `W1` and `W2`.
2. Apply `pjit.with_sharding_constraint` to annotate intermediate variables like `y` and `z` to force a particular sharding pattern under `pjit` when the ideal constraint is known.

- This step is optional, but can sometimes help auto-SPMD to partition efficiently. In the example below, the call is not required, because `pjit` will figure out the same sharding layout for `y` and `z` regardless.

```
class DotReluDot(nn.Module):
  depth: int
  @nn.compact
  def __call__(self, x):
    W1 = self.param(
        'W1',
        nn.with_partitioning(nn.initializers.xavier_normal(), (None, 'model')),
        (x.shape[-1], self.depth))

    y = jax.nn.relu(jnp.dot(x, W1))
    # Force a local sharding annotation.
    y = with_sharding_constraint(y, PartitionSpec('data', 'model'))
```

(continues on next page)

(continued from previous page)

```
    W2 = self.param(
        'W2',
        nn.with_partitioning(nn.initializers.xavier_normal(), ('model', None)),
        (self.depth, x.shape[-1]))

    z = jnp.dot(y, W2)
    # Force a local sharding annotation.
    z = with_sharding_constraint(z, PartitionSpec('data', None))

    # Return a tuple to conform with the API `flax.linen.scan` as shown in the cell below.
    return z, None
```

Note that device axis names like `'data'`, `'model'` or `None` are passed into both `flax.linen.with_partitioning` and `pjit_with_sharding_constraint` API calls. This refers to how each dimension of this data should be sharded — either across one of the device mesh dimensions, or not sharded at all.

For example:

- When you define `W1` with shape `(x.shape[-1], self.depth)` and annotate as `(None, 'model')`:
  - The first dimension (of length `x.shape[-1]`) will be replicated across all devices.
  - The second dimension (of length `self.depth`) will be sharded over the `'model'` axis of the device mesh. This means `W1` will be sharded 4-way on devices `(0, 4)`, `(1, 5)`, `(2, 6)` and `(3, 7)`, on this dimension.
- When you annotate the output `z` as `('data', None)`:
  - The first dimension — the batch dimension — will be sharded over the `'data'` axis. This means half of the batch will be processed on devices `0-3` (first four devices), and another half on devices `4-7` (the remaining four devices).
  - The second dimension — the data depth dimension — will be replicated across all devices.

Define a model with `flax.linen.scan` lifted transformation

This guide uses `flax.linen.scan` to demonstrate how Flax lifted transforms, such as `scan`, can work together with JAX `pjit`.

Having created `DotReluDot`, define the `MLP` model (by subclassing `flax.linen.Module`) as multiple layers of `DotReluDot`.

To replicate identical layers, you can either use `flax.linen.scan`, or a for-loop:

- `flax.linen.scan` can offer faster compilation times.
- The for-loop can be faster on runtime.

The code below shows how to apply both methods.

Note: `flax.linen.scan` has another dimension for the parameters (the dimension over which `scan` is applied). You need to use the `metadata_params` argument to annotate the partition of this dimension. Since the parameters inside your `DotReluDot` (a sub-`Module`) are already sharded along the `model` axis, you don't need to partition multiple layers across the `model` dimension here, and therefore you should denote it as `None`.

```
class MLP(nn.Module):
  num_layers: int
  depth: int
  use_scan: bool
```

(continues on next page)

(continued from previous page)

```
  @nn.compact
  def __call__(self, x):
    if self.use_scan:
      x, _ = nn.scan(DotReluDot, length=self.num_layers,
                     variable_axes={"params": 0},
                     split_rngs={"params": True},
                     metadata_params={nn.PARTITION_NAME: None}
                     )(self.depth)(x)
    else:
      for i in range(self.num_layers):
        x, _ = DotReluDot(self.depth)(x)
    return x
```

Specify sharding (includes initialization and `TrainState` creation)

Next, generate the `jax.sharding.PartitionSpec` that `pjit` should receive as annotations of *input* and *output* data. `PartitionSpec` is a tuple of 2 axes (in a 2x4 mesh). To learn more, refer to JAX-101: Introduction to `pjit`.

Specify the input

For data parallelism, you can shard the batched *input* `x` across the `data` axis by denoting the batch axis as `data`:

```
x_spec = PartitionSpec('data', None)  # dimensions: (batch, length)
x_spec
```

```
PartitionSpec('data', None)
```

Generate a `PartitionSpec` for the output

Next, generate a `PartitionSpec` for the *output*, you need to use some actual output as a reference.

1. Instantiate a model.
2. Evaluate `model.init` abstractly using `jax.eval_shape`.
3. Use `flax.linen.get_partition_spec` to automatically generate the `PartitionSpec`.

The code below shows how to get the output spec if you use `flax.training.train_state` to carry out your initialization and training steps, in which case your `pjit`ted function will output a `TrainState`.

(In a simpler case, people might choose the variable dict as in `variables = model.init(k, x)` as their `pjit`ted function's output. That works too.)

```
# MLP hyperparameters.
BATCH, LAYERS, DEPTH, USE_SCAN = 8, 4, 1024, True
# Create fake inputs.
x = jnp.ones((BATCH, DEPTH))
# Initialize a PRNG key.
k = random.PRNGKey(0)
```

(continues on next page)

(continued from previous page)

```
# Create an Optax optimizer.
optimizer = optax.adam(learning_rate=0.001)
# Instantiate the model.
model = MLP(LAYERS, DEPTH, USE_SCAN)

# A functional way of model initialization.
def init_fn(k, x, model, optimizer):
  variables = model.init(k, x) # Initialize the model.
  state = train_state.TrainState.create( # Create a `TrainState`.
    apply_fn=model.apply,
    params=variables['params'],
    tx=optimizer)
  return state

with mesh:
  # Create an abstract closure to wrap the function before feeding it in
  # because `jax.eval_shape` only takes pytrees as arguments`.
  abstract_variables = jax.eval_shape(
      functools.partial(init_fn, model=model, optimizer=optimizer), k, x)
# This `state_spec` has the same pytree structure as the output
# of the `init_fn`.
state_spec = nn.get_partition_spec(abstract_variables)
state_spec
```

```
TrainState(step=PartitionSpec(), apply_fn=<bound method Module.apply of MLP(
    # attributes
    num_layers = 4
    depth = 1024
    use_scan = True
)>, params=FrozenDict({
    ScanDotReluDot_0: {
        W1: PartitionSpec(None, None, 'model'),
        W2: PartitionSpec(None, 'model', None),
    },
}), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7f83a5303430>,␣
↪update=<function chain.<locals>.update_fn at 0x7f83a53034c0>), opt_
↪state=(ScaleByAdamState(count=PartitionSpec(), mu=FrozenDict({
    ScanDotReluDot_0: {
        W1: PartitionSpec(None, None, 'model'),
        W2: PartitionSpec(None, 'model', None),
    },
}), nu=FrozenDict({
    ScanDotReluDot_0: {
        W1: PartitionSpec(None, None, 'model'),
        W2: PartitionSpec(None, 'model', None),
    },
})), EmptyState()))
```

Apply `pjit` to compile the code

Now you can apply JAX `pjit` to your `init_fn` in a similar fashion as `jax.jit` but with two extra arguments: `in_axis_resources` and `out_axis_resources`.

You need to add a `with mesh:` context when running a `pjit`ted function, so that it can refer to `mesh` (an instance of `jax.sharding.Mesh`) to allocate data on devices correctly.

```
pjit_init_fn = pjit(init_fn,
                    static_argnums=(2, 3),
                    in_axis_resources=(PartitionSpec(None), x_spec),  # PRNG key and x
                    out_axis_resources=state_spec
                    )
with mesh:
  initialized_state = pjit_init_fn(k, x, model, optimizer)
jax.tree_map(jnp.shape, initialized_state)
```

```
TrainState(step=(), apply_fn=<bound method Module.apply of MLP(
    # attributes
    num_layers = 4
    depth = 1024
    use_scan = True
)>, params=FrozenDict({
    ScanDotReluDot_0: {
        W1: Partitioned(value=(4, 1024, 1024), names=(None, None, 'model'), mesh=None),
        W2: Partitioned(value=(4, 1024, 1024), names=(None, 'model', None), mesh=None),
    },
}), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7f83a5303430>,␣
↪update=<function chain.<locals>.update_fn at 0x7f83a53034c0>), opt_
↪state=(ScaleByAdamState(count=(), mu=FrozenDict({
    ScanDotReluDot_0: {
        W1: Partitioned(value=(4, 1024, 1024), names=(None, None, 'model'), mesh=None),
        W2: Partitioned(value=(4, 1024, 1024), names=(None, 'model', None), mesh=None),
    },
}), nu=FrozenDict({
    ScanDotReluDot_0: {
        W1: Partitioned(value=(4, 1024, 1024), names=(None, None, 'model'), mesh=None),
        W2: Partitioned(value=(4, 1024, 1024), names=(None, 'model', None), mesh=None),
    },
})), EmptyState()))
```

Inspect the Module output

Note that in the output of `initialized_state`, the `params` `W1` and `W2` are of type `flax.linen.Partitioned`. This is a wrapper around the actual `jax.Array` that allows Flax to record metadata associated with it. You can access the raw `jax.Array` by adding `.value` or running `.unbox()`.

You can also check the underlying `jax.sharding` of the JAX array, which gives a hint on the way it is partitioned.

```
print(type(initialized_state.params['ScanDotReluDot_0']['W1']))
print(type(initialized_state.params['ScanDotReluDot_0']['W1'].value))
print(initialized_state.params['ScanDotReluDot_0']['W1'].value.shape)
```

```
<class 'flax.core.meta.Partitioned'>
<class 'jaxlib.xla_extension.ArrayImpl'>
(4, 1024, 1024)
```

```
print(initialized_state.params['ScanDotReluDot_0']['W1'].value.sharding)
```

```
NamedSharding(mesh={'data': 2, 'model': 4}, spec=PartitionSpec(None, None, 'model'))
```

You can use `jax.tree_map` to perform mass computation on a dict of boxed params, in the same way as on a dict of JAX arrays.

```
diff = jax.tree_map(
    lambda a, b: a - b,
    initialized_state.params['ScanDotReluDot_0'], initialized_state.params[
↪'ScanDotReluDot_0'])
print(jax.tree_map(jnp.shape, diff))
diff_array = diff['W1'].unbox()
print(type(diff_array))
print(diff_array.shape)
```

```
FrozenDict({
    W1: Partitioned(value=(4, 1024, 1024), names=(None, None, 'model'), mesh=None),
    W2: Partitioned(value=(4, 1024, 1024), names=(None, 'model', None), mesh=None),
})
<class 'jaxlib.xla_extension.ArrayImpl'>
(4, 1024, 1024)
```

Apply `pjit` to the train step and inference

Now, you create a `pjit`ted training step:

```
def train_step(state, x):
  # A fake loss function.
  def loss_unrolled(params):
    y = model.apply({'params': params}, x)
    return y.sum()
  grad_fn = jax.grad(loss_unrolled)
  grads = grad_fn(state.params)
  state = state.apply_gradients(grads=grads)
  return state

pjit_step_fn = pjit(train_step,
                    in_axis_resources=(state_spec, x_spec),  # input annotations
                    out_axis_resources=state_spec,           # output annotations
                    )
with mesh:
  new_state = pjit_step_fn(initialized_state, x)
```

Apply `pjit` to inference. Note that, similar to `jax.jit`, you can use a decorator like `@functools.partial(pjit, ...)` to directly compile your function.

```
@functools.partial(pjit, in_axis_resources=(state_spec, x_spec), out_axis_resources=x_
↪spec)
def pjit_apply_fn(state, x):
  return state.apply_fn({'params': state.params}, x)

with mesh:
  y = pjit_apply_fn(new_state, x)
print(type(y))
print(y.dtype)
print(y.shape)
```

```
<class 'jaxlib.xla_extension.ArrayImpl'>
float32
(8, 1024)
```

Profiling

If you are running on a TPU pod or a pod slice, you can use a custom `block_all` utility function, as defined below, to measure the performance:

```
%%timeit

def block_all(xs):
  jax.tree_map(lambda x: x.block_until_ready(), xs)
  return xs

with mesh:
  new_state = block_all(pjit_step_fn(initialized_state, x))
```

```
519 ms ± 5.34 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Logical axis annotation

JAX auto SPMD encourages users to explore different sharding layouts to find the optimal one. To this end, in Flax you actually can annotate the dimensions of any data with more descriptive axis names (not just device mesh axis names like `'data'` and `'model'`).

The `LogicalDotReluDot` and `LogicalMLP` Module definition below are similar to the Modules you created earlier, except for the following:

1. All axes are annotated with more concrete, meaningful names, such as `'embed'`, `'hidden'`, `'batch'` and `'layer'`. These names are referred to as *logical axis names* in Flax. They make the dimensional changes inside model definitions more readable.
2. `flax.linen.spmd.with_logical_partitioning` replaces `flax.linen.with_partitioning`; and `flax.linen.spmd.with_logical_constraint` replaces `pjit.with_sharding_constraint`, to recognize the logical axis names.

```
class LogicalDotReluDot(nn.Module):
  depth: int
  @nn.compact
```

(continues on next page)

(continued from previous page)

```
  def __call__(self, x):
    W1 = self.param(
        'W1',
        spmd.with_logical_partitioning(nn.initializers.xavier_normal(), ('embed', 'hidden
↪')),
        (x.shape[-1], self.depth))

    y = jax.nn.relu(jnp.dot(x, W1))
    # Force a local sharding annotation.
    y = spmd.with_logical_constraint(y, ('batch', 'hidden'))

    W2 = self.param(
        'W2',
        spmd.with_logical_partitioning(nn.initializers.xavier_normal(), ('hidden', 'embed
↪')),
        (self.depth, x.shape[-1]))

    z = jnp.dot(y, W2)
    # Force a local sharding annotation.
    z = spmd.with_logical_constraint(z, ('batch', 'embed'))
    return z, None

class LogicalMLP(nn.Module):
  num_layers: int
  depth: int
  use_scan: bool
  @nn.compact
  def __call__(self, x):
    if self.use_scan:
      x, _ = nn.scan(LogicalDotReluDot, length=self.num_layers,
                     variable_axes={"params": 0},
                     split_rngs={"params": True},
                     metadata_params={nn.PARTITION_NAME: 'layer'}
                     )(self.depth)(x)
    else:
      for i in range(self.num_layers):
        x, _ = DotReluDot(self.depth)(x)
    return x
```

The `LogicalMLP` model definition generates a set of `PartitionSpec` with logical axis names.

Repeat the steps from earlier: instantiate a model, evaluate the `init_fn` abstractly, and use `flax.linen.get_partition_spec` to automatically generate the `PartitionSpec`:

```
logical_model = LogicalMLP(LAYERS, DEPTH, USE_SCAN)
logical_abstract_variables = jax.eval_shape(
    functools.partial(init_fn, model=logical_model, optimizer=optimizer), k, x)
logical_output_spec = nn.get_partition_spec(logical_abstract_variables)
logical_output_spec
```

```
TrainState(step=PartitionSpec(), apply_fn=<bound method Module.apply of LogicalMLP(
    # attributes
```

(continues on next page)

(continued from previous page)

```
    num_layers = 4
    depth = 1024
    use_scan = True
)>, params=FrozenDict({
    ScanLogicalDotReluDot_0: {
        W1: PartitionSpec('layer', 'embed', 'hidden'),
        W2: PartitionSpec('layer', 'hidden', 'embed'),
    },
}), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7f83a5303430>,␣
↪update=<function chain.<locals>.update_fn at 0x7f83a53034c0>), opt_
↪state=(ScaleByAdamState(count=PartitionSpec(), mu=FrozenDict({
    ScanLogicalDotReluDot_0: {
        W1: PartitionSpec('layer', 'embed', 'hidden'),
        W2: PartitionSpec('layer', 'hidden', 'embed'),
    },
}), nu=FrozenDict({
    ScanLogicalDotReluDot_0: {
        W1: PartitionSpec('layer', 'embed', 'hidden'),
        W2: PartitionSpec('layer', 'hidden', 'embed'),
    },
})), EmptyState()))
```

To allow the device mesh to take your model correctly, you need to decide which of these logical axis names are mapped to the device axis `'data'` or `'model'`. This rule is a list of (`logical_axis_name`, `device_axis_name`) tuples, and `jax.linen.spmd.logical_to_mesh` will convert them to the spec that `pjit` accepts.

This allows you to change the rules and try out new partition layouts without modifying the model definition.

```
# Unspecified rule means unsharded by default, so no need to specify `('embed', None)` and␣
↪`('layer', None)`.
rules = (('batch', 'data'),
         ('hidden', 'model'))

logical_state_spec = spmd.logical_to_mesh(logical_output_spec, rules)
logical_state_spec
```

```
TrainState(step=PartitionSpec(), apply_fn=<bound method Module.apply of LogicalMLP(
    # attributes
    num_layers = 4
    depth = 1024
    use_scan = True
)>, params=FrozenDict({
    ScanLogicalDotReluDot_0: {
        W1: PartitionSpec(None, None, 'model'),
        W2: PartitionSpec(None, 'model', None),
    },
}), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7f83a5303430>,␣
↪update=<function chain.<locals>.update_fn at 0x7f83a53034c0>), opt_
↪state=(ScaleByAdamState(count=PartitionSpec(), mu=FrozenDict({
    ScanLogicalDotReluDot_0: {
        W1: PartitionSpec(None, None, 'model'),
        W2: PartitionSpec(None, 'model', None),
```

(continues on next page)

(continued from previous page)

```
    },
}), nu=FrozenDict({
    ScanLogicalDotReluDot_0: {
        W1: PartitionSpec(None, None, 'model'),
        W2: PartitionSpec(None, 'model', None),
    },
})), EmptyState()))
```

You can verify that the `logical_state_spec` here has the same content as `state_spec` in the previous ("non-logical") example. This will be the `out_axis_resources` you specify when creating `pjit`ted functions.

```
state_spec.params['ScanDotReluDot_0'] == logical_state_spec.params[
↪'ScanLogicalDotReluDot_0']
```

```
True
```

```
logical_pjit_init_fn = pjit(init_fn,
                            static_argnums=(2, 3),
                            in_axis_resources=(PartitionSpec(None), x_spec),  # RNG key␣
↪and x
                            out_axis_resources=logical_state_spec
                            )
with mesh:
  logical_initialized_state = logical_pjit_init_fn(k, x, logical_model, optimizer)
jax.tree_map(jnp.shape, logical_initialized_state)
```

```
TrainState(step=(), apply_fn=<bound method Module.apply of LogicalMLP(
    # attributes
    num_layers = 4
    depth = 1024
    use_scan = True
)>, params=FrozenDict({
    ScanLogicalDotReluDot_0: {
        W1: LogicallyPartitioned(value=(4, 1024, 1024), names=('layer', 'embed', 'hidden
↪'), mesh=None, rules=None),
        W2: LogicallyPartitioned(value=(4, 1024, 1024), names=('layer', 'hidden', 'embed
↪'), mesh=None, rules=None),
    },
}), tx=GradientTransformation(init=<function chain.<locals>.init_fn at 0x7f83a5303430>,␣
↪update=<function chain.<locals>.update_fn at 0x7f83a53034c0>), opt_
↪state=(ScaleByAdamState(count=(), mu=FrozenDict({
    ScanLogicalDotReluDot_0: {
        W1: LogicallyPartitioned(value=(4, 1024, 1024), names=('layer', 'embed', 'hidden
↪'), mesh=None, rules=None),
        W2: LogicallyPartitioned(value=(4, 1024, 1024), names=('layer', 'hidden', 'embed
↪'), mesh=None, rules=None),
    },
}), nu=FrozenDict({
    ScanLogicalDotReluDot_0: {
        W1: LogicallyPartitioned(value=(4, 1024, 1024), names=('layer', 'embed', 'hidden
↪'), mesh=None, rules=None),
```

(continues on next page)

(continued from previous page)

```
        W2: LogicallyPartitioned(value=(4, 1024, 1024), names=('layer', 'hidden', 'embed
↪'), mesh=None, rules=None),
    },
})), EmptyState()))
```

When to use device axis / logical axis

Choosing when to use a device or logical axis depends on how much you want to control the partitioning of your model.

If you want a very simple model, or you are very confident of your way of partitioning, defining it with **device mesh axis** can potentially save you a few extra lines of code of converting the logical naming back to the device naming.

On the other hand, the **logical naming** helpers are useful for exploring different sharding layouts. Use this if you want to experiment around and find the most optimal partition layout for your model.

In really advanced use cases, you may have more complicated sharding patterns that require annotating *activation* dimension names differently from *parameter* dimension names. When people wish to have more fine-grained control on manual mesh assignments, directly using **device axis names** could be more helpful.

Save the data

You can use `flax.training.checkpoints` to save the cross-device array, as shown in the Save and load checkpoints guide - Multi-host/multi-process checkpointing. This is especially required if you are running on a multi-host environment (for example, a TPU pod).

Keep in mind that to restore the arrays to the desired partition, you need to provide a sample `target` pytree that has the same structure and has the desired `PartitionSpec` in place for each JAX array. The `PartitionSpec` you use to restore the array doesn't necessarily need to be the same as the ones you used to store the array.

5.2.5 Model inspection

Model surgery

Usually, Flax modules and optimizers track and update the params for you. But there may be some time when you want to do some model surgery and tweak the param tensors yourself. This guide shows you how to do the trick.

Setup

```
!pip install --upgrade -q pip jax jaxlib flax
```

```
import functools

import jax
import jax.numpy as jnp
from flax import traverse_util
from flax import linen as nn
from flax.core import freeze
import jax
import optax
```

Surgery with Flax Modules

Let's create a small convolutional neural network model for our demo.

As usual, you can run `CNN.init(...)['params']` to get the `params` to pass and modify it in every step of your training.

```
class CNN(nn.Module):
    @nn.compact
    def __call__(self, x):
      x = nn.Conv(features=32, kernel_size=(3, 3))(x)
      x = nn.relu(x)
      x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
      x = nn.Conv(features=64, kernel_size=(3, 3))(x)
      x = nn.relu(x)
      x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
      x = x.reshape((x.shape[0], -1))
      x = nn.Dense(features=256)(x)
      x = nn.relu(x)
      x = nn.Dense(features=10)(x)
      x = nn.log_softmax(x)
      return x

def get_initial_params(key):
    init_shape = jnp.ones((1, 28, 28, 1), jnp.float32)
    initial_params = CNN().init(key, init_shape)['params']
    return initial_params

key = jax.random.PRNGKey(0)
params = get_initial_params(key)

jax.tree_util.tree_map(jnp.shape, params)
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more␣
↪info.)
```

```
FrozenDict({
    Conv_0: {
        bias: (32,),
        kernel: (3, 3, 1, 32),
    },
    Conv_1: {
        bias: (64,),
        kernel: (3, 3, 32, 64),
    },
    Dense_0: {
        bias: (256,),
        kernel: (3136, 256),
    },
    Dense_1: {
        bias: (10,),
        kernel: (256, 10),
    },
```

(continues on next page)

(continued from previous page)

```
})
```

Note that what returned as `params` is a `FrozenDict`, which contains a few JAX arrays as kernel and bias.

A `FrozenDict` is nothing more than a read-only dict, and Flax made it read-only because of the functional nature of JAX: JAX arrays are immutable, and the new `params` need to replace the old `params`. Making the dict read-only ensures that no in-place mutation of the dict can happen accidentally during the training and updating.

One way to actually modify the params outside of a Flax module is to explicitly flatten it and creates a mutable dict. Note that you can use a separator `sep` to join all nested keys. If no `sep` is given, the key will be a tuple of all nested keys.

```
# Get a flattened key-value list.
flat_params = traverse_util.flatten_dict(params, sep='/')

jax.tree_util.tree_map(jnp.shape, flat_params)
```

```
{'Conv_0/bias': (32,),
 'Conv_0/kernel': (3, 3, 1, 32),
 'Conv_1/bias': (64,),
 'Conv_1/kernel': (3, 3, 32, 64),
 'Dense_0/bias': (256,),
 'Dense_0/kernel': (3136, 256),
 'Dense_1/bias': (10,),
 'Dense_1/kernel': (256, 10)}
```

Now you can do whatever you want with the params. When you are done, unflatten it back and use it in future training.

```
# Somehow modify a layer
dense_kernel = flat_params['Dense_1/kernel']
flat_params['Dense_1/kernel'] = dense_kernel / jnp.linalg.norm(dense_kernel)

# Unflatten.
unflat_params = traverse_util.unflatten_dict(flat_params, sep='/')
# Refreeze.
unflat_params = freeze(unflat_params)
jax.tree_util.tree_map(jnp.shape, unflat_params)
```

```
FrozenDict({
    Conv_0: {
        bias: (32,),
        kernel: (3, 3, 1, 32),
    },
    Conv_1: {
        bias: (64,),
        kernel: (3, 3, 32, 64),
    },
    Dense_0: {
        bias: (256,),
        kernel: (3136, 256),
    },
    Dense_1: {
        bias: (10,),
```

(continues on next page)

(continued from previous page)

```
        kernel: (256, 10),
    },
})
```

Surgery with Optimizers

When using `Optax` as an optimizer, the `opt_state` is actually a nested tuple of the states of individual gradient transformations that compose the optimizer. These states contain pytrees that mirror the parameter tree, and can be modified the same way: flattening, modifying, unflattening, and then recreating a new optimizer state that mirrors the original state.

```
tx = optax.adam(1.0)
opt_state = tx.init(params)

# The optimizer state is a tuple of gradient transformation states.
jax.tree_util.tree_map(jnp.shape, opt_state)
```

```
(ScaleByAdamState(count=(), mu=FrozenDict({
    Conv_0: {
        bias: (32,),
        kernel: (3, 3, 1, 32),
    },
    Conv_1: {
        bias: (64,),
        kernel: (3, 3, 32, 64),
    },
    Dense_0: {
        bias: (256,),
        kernel: (3136, 256),
    },
    Dense_1: {
        bias: (10,),
        kernel: (256, 10),
    },
}), nu=FrozenDict({
    Conv_0: {
        bias: (32,),
        kernel: (3, 3, 1, 32),
    },
    Conv_1: {
        bias: (64,),
        kernel: (3, 3, 32, 64),
    },
    Dense_0: {
        bias: (256,),
        kernel: (3136, 256),
    },
    Dense_1: {
        bias: (10,),
        kernel: (256, 10),
```

(continues on next page)

(continued from previous page)

```
    },
 })),
 EmptyState())
```

The pytrees inside the optimizer state follow the same structure as the parameters and can be flattened / modified exactly the same way.

```
flat_mu = traverse_util.flatten_dict(opt_state[0].mu, sep='/')
flat_nu = traverse_util.flatten_dict(opt_state[0].nu, sep='/')

jax.tree_util.tree_map(jnp.shape, flat_mu)
```

```
{'Conv_0/bias': (32,),
 'Conv_0/kernel': (3, 3, 1, 32),
 'Conv_1/bias': (64,),
 'Conv_1/kernel': (3, 3, 32, 64),
 'Dense_0/bias': (256,),
 'Dense_0/kernel': (3136, 256),
 'Dense_1/bias': (10,),
 'Dense_1/kernel': (256, 10)}
```

After modification, re-create optimizer state. Use this for future training.

```
opt_state = (
    opt_state[0]._replace(
        mu=traverse_util.unflatten_dict(flat_mu, sep='/'),
        nu=traverse_util.unflatten_dict(flat_nu, sep='/'),
    ),
) + opt_state[1:]
jax.tree_util.tree_map(jnp.shape, opt_state)
```

```
(ScaleByAdamState(count=(), mu={'Conv_0': {'bias': (32,), 'kernel': (3, 3, 1, 32)},
↪'Conv_1': {'bias': (64,), 'kernel': (3, 3, 32, 64)}, 'Dense_0': {'bias': (256,),
↪'kernel': (3136, 256)}, 'Dense_1': {'bias': (10,), 'kernel': (256, 10)}}, nu={'Conv_0
↪': {'bias': (32,), 'kernel': (3, 3, 1, 32)}, 'Conv_1': {'bias': (64,), 'kernel': (3, 3,
↪ 32, 64)}, 'Dense_0': {'bias': (256,), 'kernel': (3136, 256)}, 'Dense_1': {'bias': (10,
↪), 'kernel': (256, 10)}}),
 EmptyState())
```

Extracting intermediate values

This guide will show you how to extract intermediate values from a module. Let's start with this simple CNN that uses `nn.compact`.

```
from flax import linen as nn
import jax
import jax.numpy as jnp
from typing import Sequence

class CNN(nn.Module):
  @nn.compact
```

(continues on next page)

(continued from previous page)

```
  def __call__(self, x):
    x = nn.Conv(features=32, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = nn.Conv(features=64, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = x.reshape((x.shape[0], -1))  # flatten
    x = nn.Dense(features=256)(x)
    x = nn.relu(x)
    x = nn.Dense(features=10)(x)
    x = nn.log_softmax(x)
    return x
```

Because this module uses `nn.compact`, we don't have direct access to intermediate values. There are a few ways to expose them:

Store intermediate values in a new variable collection

The CNN can be augmented with calls to `sow` to store intermediates as following:

Default CNN

```
class CNN(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = nn.Conv(features=32, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = nn.Conv(features=64, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = x.reshape((x.shape[0], -1))  # flatten

    x = nn.Dense(features=256)(x)
    x = nn.relu(x)
    x = nn.Dense(features=10)(x)
    x = nn.log_softmax(x)
    return x
```

CNN using sow API

```
class SowCNN(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = nn.Conv(features=32, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = nn.Conv(features=64, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = x.reshape((x.shape[0], -1))  # flatten
    self.sow('intermediates', 'features', x)
    x = nn.Dense(features=256)(x)
    x = nn.relu(x)
    x = nn.Dense(features=10)(x)
    x = nn.log_softmax(x)
    return x
```

`sow` acts as a no-op when the variable collection is not mutable. Therefore, it works perfectly for debugging and optional tracking of intermediates. The 'intermediates' collection is also used by the `capture_intermediates` API (see the *Use capture_intermediates* section).

Note that, by default `sow` appends values every time it is called:

- This is necessary because once instantiated, a module could be called multiple times in its parent module, and we want to catch all the sowed values.
- Therefore you want to make sure that you **do not** feed intermediate values back into `variables`. Otherwise every call will increase the length of that tuple and trigger a recompile.
- To override the default append behavior, specify `init_fn` and `reduce_fn` - see *Module.sow()*.

```
class SowCNN2(nn.Module):
  @nn.compact
  def __call__(self, x):
    mod = SowCNN(name='SowCNN')
    return mod(x) + mod(x)  # Calling same module instance twice.

@jax.jit
def init(key, x):
  variables = SowCNN2().init(key, x)
  # By default the 'intermediates' collection is not mutable during init.
  # So variables will only contain 'params' here.
  return variables

@jax.jit
def predict(variables, x):
  # If mutable='intermediates' is not specified, then .sow() acts as a noop.
  output, mod_vars = SowCNN2().apply(variables, x, mutable='intermediates')
  features = mod_vars['intermediates']['SowCNN']['features']
  return output, features

batch = jnp.ones((1,28,28,1))
variables = init(jax.random.PRNGKey(0), batch)
```

(continues on next page)

(continued from previous page)

```
preds, feats = predict(variables, batch)

assert len(feats) == 2  # Tuple with two values since module was called twice.
```

Refactor module into submodules

This is a useful pattern for cases where it's clear in what particular way you want to split your submodules. Any submodule you expose in `setup` can be used directly. In the limit, you can define all submodules in `setup` and avoid using `nn.compact` altogether.

```
class RefactoredCNN(nn.Module):
  def setup(self):
    self.features = Features()
    self.classifier = Classifier()

  def __call__(self, x):
    x = self.features(x)
    x = self.classifier(x)
    return x

class Features(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = nn.Conv(features=32, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = nn.Conv(features=64, kernel_size=(3, 3))(x)
    x = nn.relu(x)
    x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
    x = x.reshape((x.shape[0], -1))  # flatten
    return x

class Classifier(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = nn.Dense(features=256)(x)
    x = nn.relu(x)
    x = nn.Dense(features=10)(x)
    x = nn.log_softmax(x)
    return x

@jax.jit
def init(key, x):
  variables = RefactoredCNN().init(key, x)
  return variables['params']

@jax.jit
def features(params, x):
  return RefactoredCNN().apply({"params": params}, x,
    method=lambda module, x: module.features(x))
```

(continues on next page)

(continued from previous page)

```
params = init(jax.random.PRNGKey(0), batch)

features(params, batch)
```

Use `capture_intermediates`

Linen supports the capture of intermediate return values from submodules automatically without any code changes. This pattern should be considered the "sledge hammer" approach to capturing intermediates. As a debugging and inspection tool it is very useful, but using the other patterns described in this guide will give you more fine-grained control over what intermediates you want to extract.

In the following code example we check if any intermediate activations are non-finite (NaN or infinite):

```
@jax.jit
def init(key, x):
  variables = CNN().init(key, x)
  return variables

@jax.jit
def predict(variables, x):
  y, state = CNN().apply(variables, x, capture_intermediates=True, mutable=[
↪"intermediates"])
  intermediates = state['intermediates']
  fin = jax.tree_util.tree_map(lambda xs: jnp.all(jnp.isfinite(xs)), intermediates)
  return y, fin

variables = init(jax.random.PRNGKey(0), batch)
y, is_finite = predict(variables, batch)
all_finite = all(jax.tree_util.tree_leaves(is_finite))
assert all_finite, "non-finite intermediate detected!"
```

By default only the intermediates of `__call__` methods are collected. Alternatively, you can pass a custom filter function based on the `Module` instance and the method name.

```
filter_Dense = lambda mdl, method_name: isinstance(mdl, nn.Dense)
filter_encodings = lambda mdl, method_name: method_name == "encode"

y, state = CNN().apply(variables, batch, capture_intermediates=filter_Dense, mutable=[
↪"intermediates"])
dense_intermediates = state['intermediates']
```

Note that `capture_intermediates` will only apply to layers. You can use `self.sow` to manually store non-layer intermediates, but the filter function won't be applied to it.

Capturing all layer intermediates

```
class Model(nn.Module):
  @nn.compact
  def __call__(self, x):
    a = nn.Dense(4)(x) # Dense_0
    b = nn.Dense(4)(x) # Dense_1
    c = a + b # not a Flax layer, so won't be stored as an intermediate
    d = nn.Dense(4)(c) # Dense_2
    return d

@jax.jit
def init(key, x):
  variables = Model().init(key, x)
  return variables['params']

@jax.jit
def predict(params, x):
  return Model().apply({"params": params}, x, capture_intermediates=True)

batch = jax.random.uniform(jax.random.PRNGKey(1), (1,3))
params = init(jax.random.PRNGKey(0), batch)
preds, feats = predict(params, batch)
feats # intermediate c in Model was not stored because it's not a Flax layer
```

Using filter function and `self.sow()`

```
class Model(nn.Module):
  @nn.compact
  def __call__(self, x):
    a = nn.Dense(4)(x) # Dense_0
    b = nn.Dense(4)(x) # Dense_1
    c = a + b
    self.sow('intermediates', 'c', c) # store intermediate c
    d = nn.Dense(4)(c) # Dense_2
    return d

@jax.jit
def init(key, x):
  variables = Model().init(key, x)
  return variables['params']

@jax.jit
def predict(params, x):
  # filter specifically for only the Dense_0 and Dense_2 layer
  filter_fn = lambda mdl, method_name: isinstance(mdl.name, str) and (mdl.name in {
↪'Dense_0', 'Dense_2'})
  return Model().apply({"params": params}, x, capture_intermediates=filter_fn)

batch = jax.random.uniform(jax.random.PRNGKey(1), (1,3))
params = init(jax.random.PRNGKey(0), batch)
```

(continues on next page)

(continued from previous page)

```
preds, feats = predict(params, batch)
feats # intermediate c in Model is stored and isn't filtered out by the filter function
```

To separate the intermediates extracted from `self.sow` from the intermediates extracted from `capture_intermediates`, we can either define a separate collection like `self.sow('sow_intermediates', 'c', c)`, or manually filter out the intermediates after calling `.apply()`. For example:

```
flattened_dict = flax.traverse_util.flatten_dict(feats['intermediates'], sep='/')
flattened_dict['c']
```

In terms of efficiency, as long as everything is jitted, then any intermediates you don't end up using should be optimized away by XLA.

Use `Sequential`

You could also define `CNN` using a simple implementation of a `Sequential` combinator (this is quite common in more stateful approaches). This may be useful for very simple models and gives you arbitrary model surgery. But it can be very limiting – if you even want to add one conditional, you are forced to refactor away from `Sequential` and structure your model more explicitly.

```
class Sequential(nn.Module):
  layers: Sequence[nn.Module]

  def __call__(self, x):
    for layer in self.layers:
      x = layer(x)
    return x

def SeqCNN():
  return Sequential([
    nn.Conv(features=32, kernel_size=(3, 3)),
    nn.relu,
    lambda x: nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2)),
    nn.Conv(features=64, kernel_size=(3, 3)),
    nn.relu,
    lambda x: nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2)),
    lambda x: x.reshape((x.shape[0], -1)),  # flatten
    nn.Dense(features=256),
    nn.relu,
    nn.Dense(features=10),
    nn.log_softmax,
  ])

@jax.jit
def init(key, x):
  variables = SeqCNN().init(key, x)
  return variables['params']

@jax.jit
def features(params, x):
  return Sequential(SeqCNN().layers[0:7]).apply({"params": params}, x)
```

(continues on next page)

(continued from previous page)

```
batch = jnp.ones((1,28,28,1))
params = init(jax.random.PRNGKey(0), batch)
features(params, batch)
```

Extracting gradients of intermediate values

For debugging purposes, it can be useful to extract the gradients of intermediate values. This can be done by using the *Module.perturb()* method over the desired values.

```
class Model(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = nn.relu(nn.Dense(8)(x))
    x = self.perturb('hidden', x)
    x = nn.Dense(2)(x)
    x = self.perturb('logits', x)
    return x
```

`perturb` adds a variable to a `perturbations` collection by default, it behaves like an identity function and the gradient of the perturbation matches the gradient of the input. To get the perturbations just initialize the model:

```
x = jnp.empty((1, 4)) # random data
y = jnp.empty((1, 2)) # random data

model = Model()
variables = model.init(jax.random.PRNGKey(1), x)
params, perturbations = variables['params'], variables['perturbations']
```

Finally compute the gradients of the loss with respect to the perturbations, these will match the gradients of the intermediates:

```
def loss_fn(params, perturbations, x, y):
  y_pred = model.apply({'params': params, 'perturbations': perturbations}, x)
  return jnp.mean((y_pred - y) ** 2)

intermediate_grads = jax.grad(loss_fn, argnums=1)(params, perturbations, x, y)
```

5.2.6 Converting and upgrading

Convert PyTorch models to Flax

We will show how to convert PyTorch models to Flax. We will cover convolutions, fc layers, batch norm, and average pooling.

FC Layers

Let's start with fc layers. The only thing to be aware of here is that the PyTorch kernel has shape [outC, inC] and the Flax kernel has shape [inC, outC]. Transposing the kernel will do the trick.

```
t_fc = torch.nn.Linear(in_features=3, out_features=4)

kernel = t_fc.weight.detach().cpu().numpy()
bias = t_fc.bias.detach().cpu().numpy()

# [outC, inC] -> [inC, outC]
kernel = jnp.transpose(kernel, (1, 0))

key = random.PRNGKey(0)
x = random.normal(key, (1, 3))

variables = {'params': {'kernel': kernel, 'bias': bias}}
j_fc = nn.Dense(features=4)
j_out = j_fc.apply(variables, x)

t_x = torch.from_numpy(np.array(x))
t_out = t_fc(t_x)
t_out = t_out.detach().cpu().numpy()

np.testing.assert_almost_equal(j_out, t_out)
```

Convolutions

Let's now look at 2D convolutions. PyTorch uses the NCHW format and Flax uses NHWC. Consequently, the kernels will have different shapes. The kernel in PyTorch has shape [outC, inC, kH, kW] and the Flax kernel has shape [kH, kW, inC, outC]. Transposing the kernel will do the trick.

```
t_conv = torch.nn.Conv2d(in_channels=3, out_channels=4, kernel_size=2, padding='valid')

kernel = t_conv.weight.detach().cpu().numpy()
bias = t_conv.bias.detach().cpu().numpy()

# [outC, inC, kH, kW] -> [kH, kW, inC, outC]
kernel = jnp.transpose(kernel, (2, 3, 1, 0))

key = random.PRNGKey(0)
x = random.normal(key, (1, 6, 6, 3))

variables = {'params': {'kernel': kernel, 'bias': bias}}
j_conv = nn.Conv(features=4, kernel_size=(2, 2), padding='valid')
j_out = j_conv.apply(variables, x)

# [N, H, W, C] -> [N, C, H, W]
t_x = torch.from_numpy(np.transpose(np.array(x), (0, 3, 1, 2)))
t_out = t_conv(t_x)
# [N, C, H, W] -> [N, H, W, C]
t_out = np.transpose(t_out.detach().cpu().numpy(), (0, 2, 3, 1))
```

(continues on next page)

(continued from previous page)

```
np.testing.assert_almost_equal(j_out, t_out, decimal=6)
```

Convolutions and FC Layers

We have to be careful, when we have a model that uses convolutions followed by fc layers (ResNet, VGG, etc). In PyTorch, the activations will have shape [N, C, H, W] after the convolutions and are then reshaped to [N, C * H * W] before being fed to the fc layers. When we port our weights from PyToch to Flax, the activations after the convolutions will be of shape [N, H, W, C] in Flax. Before we reshape the activations for the fc layers, we have to transpose them to [N, C, H, W].

Consider this PyTorch model:

```
class TModel(torch.nn.Module):

  def __init__(self):
    super(TModel, self).__init__()
    self.conv = torch.nn.Conv2d(in_channels=3, out_channels=4, kernel_size=2, padding=
↪'valid')
    self.fc = torch.nn.Linear(in_features=100, out_features=2)

  def forward(self, x):
    x = self.conv(x)
    x = x.reshape(x.shape[0], -1)
    x = self.fc(x)
    return x


t_model = TModel()
```

Now, if you want to use the weights from this model in Flax, the corresponding Flax model has to look like this:

```
class JModel(nn.Module):

  @nn.compact
  def __call__(self, x):
    x = nn.Conv(features=4, kernel_size=(2, 2), padding='valid', name='conv')(x)
    # [N, H, W, C] -> [N, C, H, W]
    x = jnp.transpose(x, (0, 3, 1, 2))
    x = jnp.reshape(x, (x.shape[0], -1))
    x = nn.Dense(features=2, name='fc')(x)
    return x


j_model = JModel()
```

The model looks very similar to the PyTorch model, except that we included a transpose operation before reshaping our activations for the fc layer. We can omit the transpose operation if we apply pooling before reshaping such that the spatial dimensions are 1x1.

Other than the transpose operation before reshaping, we can convert the weights the same way as we did before:

```
conv_kernel = t_model.state_dict()['conv.weight'].detach().cpu().numpy()
conv_bias = t_model.state_dict()['conv.bias'].detach().cpu().numpy()
fc_kernel = t_model.state_dict()['fc.weight'].detach().cpu().numpy()
fc_bias = t_model.state_dict()['fc.bias'].detach().cpu().numpy()

# [outC, inC, kH, kW] -> [kH, kW, inC, outC]
conv_kernel = jnp.transpose(conv_kernel, (2, 3, 1, 0))

# [outC, inC] -> [inC, outC]
fc_kernel = jnp.transpose(fc_kernel, (1, 0))

variables = {'params': {'conv': {'kernel': conv_kernel, 'bias': conv_bias},
                        'fc': {'kernel': fc_kernel, 'bias': fc_bias}}}

key = random.PRNGKey(0)
x = random.normal(key, (1, 6, 6, 3))

j_out = j_model.apply(variables, x)

# [N, H, W, C] -> [N, C, H, W]
t_x = torch.from_numpy(np.transpose(np.array(x), (0, 3, 1, 2)))
t_out = t_model(t_x)
t_out = t_out.detach().cpu().numpy()

np.testing.assert_almost_equal(j_out, t_out, decimal=6)
```

Batch Norm

`torch.nn.BatchNorm2d` uses `0.1` as the default value for the `momentum` parameter while `nn.BatchNorm` uses `0.9`. However, this corresponds to the same computation, because PyTorch multiplies the estimated statistic with `(1 momentum)` and the new observed value with `momentum`, while Flax multiplies the estimated statistic with `momentum` and the new observed value with `(1  momentum)`.

```
t_bn = torch.nn.BatchNorm2d(num_features=3, momentum=0.1)
t_bn.eval()

scale = t_bn.weight.detach().cpu().numpy()
bias = t_bn.bias.detach().cpu().numpy()
mean = t_bn.running_mean.detach().cpu().numpy()
var = t_bn.running_var.detach().cpu().numpy()

variables = {'params': {'scale': scale, 'bias': bias},
             'batch_stats': {'mean': mean, 'var': var}}

key = random.PRNGKey(0)
x = random.normal(key, (1, 6, 6, 3))

j_bn = nn.BatchNorm(momentum=0.9, use_running_average=True)

j_out = j_bn.apply(variables, x)
```

(continues on next page)

(continued from previous page)

```
# [N, H, W, C] -> [N, C, H, W]
t_x = torch.from_numpy(np.transpose(np.array(x), (0, 3, 1, 2)))
t_out = t_bn(t_x)
# [N, C, H, W] -> [N, H, W, C]
t_out = np.transpose(t_out.detach().cpu().numpy(), (0, 2, 3, 1))

np.testing.assert_almost_equal(j_out, t_out)
```

Average Pooling

`torch.nn.AvgPool2d` and `nn.avg_pool()` are compatible when using default parameters. However, `torch.nn.AvgPool2d` has a parameter `count_include_pad`. When `count_include_pad=False`, the zero-padding will not be considered for the average calculation. There does not exist a similar parameter for `nn.avg_pool()`. However, we can easily implement a wrapper around the pooling operation. `nn.pool()` is the core function behind `nn.avg_pool()` and `nn.max_pool()`.

```
def avg_pool(inputs, window_shape, strides=None, padding='VALID'):
  """
  Pools the input by taking the average over a window.
  In comparison to nn.avg_pool(), this pooling operation does not
  consider the padded zero's for the average computation.
  """
  assert len(window_shape) == 2

  y = nn.pool(inputs, 0., jax.lax.add, window_shape, strides, padding)
  counts = nn.pool(jnp.ones_like(inputs), 0., jax.lax.add, window_shape, strides,␣
↪padding)
  y = y / counts
  return y


key = random.PRNGKey(0)
x = random.normal(key, (1, 6, 6, 3))

j_out = avg_pool(x, window_shape=(2, 2), strides=(1, 1), padding=((1, 1), (1, 1)))
t_pool = torch.nn.AvgPool2d(kernel_size=2, stride=1, padding=1, count_include_pad=False)

# [N, H, W, C] -> [N, C, H, W]
t_x = torch.from_numpy(np.transpose(np.array(x), (0, 3, 1, 2)))
t_out = t_pool(t_x)
# [N, C, H, W] -> [N, H, W, C]
t_out = np.transpose(t_out.detach().cpu().numpy(), (0, 2, 3, 1))

np.testing.assert_almost_equal(j_out, t_out)
```

Transposed Convolutions

`torch.nn.ConvTranspose2d` and `nn.ConvTranspose` are not compatible. `nn.ConvTranspose` is a wrapper around `jax.lax.conv_transpose` which computes a fractionally strided convolution, while `torch.nn.ConvTranspose2d` computes a gradient based transposed convolution. Currently, there is no implementation of a gradient based transposed convolution is `Jax`. However, there is a pending pull request that contains an implementation.

Migrate checkpointing to Orbax

This guide shows how to convert Flax's checkpoint saving and restoring calls — flax.training.checkpoints.save_checkpoint and restore_checkpoint — to the equivalent Orbax methods. Orbax provides a flexible and customizable API for managing checkpoints for various objects. Note that as Flax's checkpointing is being migrated to Orbax from `flax.training.checkpoints`, all existing features in the Flax API will continue to be supported, but the API will change.

You will learn how to migrate to Orbax through the following scenarios:

- The most common use case: Saving/loading and managing checkpoints
- A "lightweight" use case: "Pure" saving/loading without the top-level checkpoint manager
- Restoring checkpoints without a target pytree
- Async checkpointing
- Saving/loading a single JAX or NumPy Array

To learn more about Orbax, check out the quick start introductory Colab notebook and the official Orbax documentation.

You can click on "Open in Colab" above to run the code from this guide.

Throughout the guide, you will be able to compare code examples with and without the Orbax code.

Setup

```
# Create some dummy variables for this example.
MAX_STEPS = 5
CKPT_PYTREE = [12, {'foo': 'str', 'bar': np.array((2, 3))}, [1, 4, 10]]
TARGET_PYTREE = [0, {'foo': '', 'bar': np.array((0))}, [0, 0, 0]]
```

Most common use case: Saving/loading and managing checkpoints

This section covers the following scenario:

- Your original Flax `save_checkpoint()` or `save_checkpoint_multiprocess()` call contains the following arguments: `prefix`, `keep`, `keep_every_n_steps`; or
- You want to use some automatic management logic for your checkpoints (for example, for deleting old data, deleting data based on metrics/loss, and so on).

In this case, you need to use `orbax.CheckpointManager`. This allows you to not only save and load your model, but also manage your checkpoints and delete outdated checkpoints *automatically*.

To upgrade your code:

1. Create and keep an `orbax.CheckpointManager` instance at the top level, customized with `orbax.CheckpointManagerOptions`.
2. At runtime, call `orbax.CheckpointManager.save()` to save your data.
3. Then, call `orbax.CheckpointManager.restore()` to restore your data.
4. And, if your checkpoint includes some multi-host/multi-process array, pass the correct `mesh` into `flax.training.orbax_utils.restore_args_from_target()` to generate the correct `restore_args` before restoring.

For example:

flax.checkpoints

```
CKPT_DIR = './tmp/'
flax.config.update('flax_use_orbax_checkpointing', False)

# Inside your training loop
for step in range(MAX_STEPS):
  # do training
  checkpoints.save_checkpoint(CKPT_DIR, CKPT_PYTREE, step=step,
                              prefix='test_', keep=3, keep_every_n_steps=2)


checkpoints.restore_checkpoint(CKPT_DIR, target=TARGET_PYTREE, step=4, prefix='test_')
```

orbax.checkpoint

```
CKPT_DIR = './tmp/orbax'

# At the top level
mgr_options = orbax.checkpoint.CheckpointManagerOptions(
  create=True, max_to_keep=3, keep_period=2, step_prefix='test_')
ckpt_mgr = orbax.checkpoint.CheckpointManager(
  CKPT_DIR,
  orbax.checkpoint.Checkpointer(orbax.checkpoint.PyTreeCheckpointHandler()), mgr_options)

# Inside your training loop
for step in range(MAX_STEPS):
  # do training
  save_args = flax.training.orbax_utils.save_args_from_target(CKPT_PYTREE)
  ckpt_mgr.save(step, CKPT_PYTREE, save_kwargs={'save_args': save_args})


restore_args = flax.training.orbax_utils.restore_args_from_target(TARGET_PYTREE,␣
↪mesh=None)
ckpt_mgr.restore(4, items=TARGET_PYTREE, restore_kwargs={'restore_args': restore_args})
```

A "lightweight" use case: "Pure" saving/loading without the top-level checkpoint manager

If you prefer to not maintain a top-level checkpoint manager, you can still save and restore any individual checkpoint with an `orbax.checkpoint.Checkpointer`. Note that this means you cannot use all the Orbax management features.

To migrate to Orbax code, instead of using the `overwrite` argument in `flax.save_checkpoint()` use the `force` argument in `orbax.checkpoint.Checkpointer.save()`.

For example:

flax.checkpoints

```
PURE_CKPT_DIR = './tmp/pure'
flax.config.update('flax_use_orbax_checkpointing', False)

checkpoints.save_checkpoint(PURE_CKPT_DIR, CKPT_PYTREE, step=0, overwrite=True)
checkpoints.restore_checkpoint(PURE_CKPT_DIR, target=TARGET_PYTREE)
```

orbax.checkpoint

```
PURE_CKPT_DIR = './tmp/pure'

ckptr = orbax.checkpoint.Checkpointer(orbax.checkpoint.PyTreeCheckpointHandler())  # A␣
↪stateless object, can be created on the fly.
ckptr.save(PURE_CKPT_DIR, CKPT_PYTREE,
           save_args=flax.training.orbax_utils.save_args_from_target(CKPT_PYTREE),␣
↪force=True)
ckptr.restore(PURE_CKPT_DIR, item=TARGET_PYTREE,
              restore_args=flax.training.orbax_utils.restore_args_from_target(TARGET_
↪PYTREE, mesh=None))
```

Restoring checkpoints without a target pytree

If you need to restore your checkpoints without a target pytree, pass `item=None` to `orbax.checkpoint.Checkpointer` or `items=None` to `orbax.CheckpointManager`'s `.restore()` method, which should trigger the restoration.

For example:

flax.checkpoints

```
NOTARGET_CKPT_DIR = './tmp/no_target'
flax.config.update('flax_use_orbax_checkpointing', False)

checkpoints.save_checkpoint(NOTARGET_CKPT_DIR, CKPT_PYTREE, step=0)
checkpoints.restore_checkpoint(NOTARGET_CKPT_DIR, target=None)
```

orbax.checkpoint

```
NOTARGET_CKPT_DIR = './tmp/no_target'

# A stateless object, can be created on the fly.
ckptr = orbax.checkpoint.Checkpointer(orbax.checkpoint.PyTreeCheckpointHandler())
ckptr.save(NOTARGET_CKPT_DIR, CKPT_PYTREE,
           save_args=flax.training.orbax_utils.save_args_from_target(CKPT_PYTREE))
ckptr.restore(NOTARGET_CKPT_DIR, item=None)
```

Async checkpointing

To make your checkpoint-saving asynchronous, substitute `orbax.checkpoint.Checkpointer` with `orbax.checkpoint.AsyncCheckpointer`.

Then, you can call `orbax.checkpoint.AsyncCheckpointer.wait_until_finished()` or Orbax's `CheckpointerManager.wait_until_finished()` to wait for the save the complete.

For more details, read the checkpoint guide.

Saving/loading a single JAX or NumPy Array

The `orbax.checkpoint.PyTreeCheckpointHandler` class, as the name suggests, can only be used for pytrees. Therefore, if you need to save/restore a single pytree leaf (for example, an array), use `orbax.checkpoint.ArrayCheckpointHandler` instead.

For example:

flax.checkpoints

```
ARR_CKPT_DIR = './tmp/singleton'
flax.config.update('flax_use_orbax_checkpointing', False)

checkpoints.save_checkpoint(ARR_CKPT_DIR, jnp.arange(10), step=0)
checkpoints.restore_checkpoint(ARR_CKPT_DIR, target=None)
```

orbax.checkpoint

```
ARR_CKPT_DIR = './tmp/singleton'

ckptr = orbax.checkpoint.Checkpointer(orbax.checkpoint.ArrayCheckpointHandler())
ckptr.save(ARR_CKPT_DIR, jnp.arange(10))
ckptr.restore(ARR_CKPT_DIR, item=None)
```

Final words

This guide provides an overview of how to migrate from the "legacy" Flax checkpointing API to the Orbax API. Orbax provides more functionalities and the Orbax team is actively developing new features. Stay tuned and follow the official Orbax GitHub repository for more!

Upgrading my codebase to Optax

We have proposed to replace `flax.optim` with Optax in 2021 with FLIP #1009 and the Flax optimizers have been removed in v0.6.0 - this guide is targeted towards `flax.optim` users to help them update their code to Optax.

See also Optax's quick start documentation: https://optax.readthedocs.io/en/latest/optax-101.html

Replacing `flax.optim` with `optax`

Optax has drop-in replacements for all of Flax's optimizers. Refer to Optax's documentation Common Optimizers for API details.

The usage is very similar, with the difference that `optax` does not keep a copy of the `params`, so they need to be passed around separately. Flax provides the utility *`TrainState`* to store optimizer state, parameters, and other associated data in a single dataclass (not used in code below).

flax.optim

```
@jax.jit
def train_step(optimizer, batch):
  grads = jax.grad(loss)(optimizer.target, batch)


  return optimizer.apply_gradient(grads)

optimizer_def = flax.optim.Momentum(
    learning_rate, momentum)
optimizer = optimizer_def.create(variables['params'])

for batch in get_ds_train():
  optimizer = train_step(optimizer, batch)
```

optax

```
@jax.jit
def train_step(params, opt_state, batch):
  grads = jax.grad(loss)(params, batch)
  updates, opt_state = tx.update(grads, opt_state)
  params = optax.apply_updates(params, updates)
  return params, opt_state

tx = optax.sgd(learning_rate, momentum)
params = variables['params']
opt_state = tx.init(params)

for batch in ds_train:
  params, opt_state = train_step(params, opt_state, batch)
```

Composable Gradient Transformations

The function `optax.sgd()` used in the code snippet above is simply a wrapper for the sequential application of two gradient transformations. Instead of using this alias, it is common to use `optax.chain()` to combine multiple of these generic building blocks.

Pre-defined alias

```
# Note that the aliases follow the convention to use positive
# values for the learning rate by default.
tx = optax.sgd(learning_rate, momentum)
```

Combining transformations

```
#

tx = optax.chain(
    # 1. Step: keep a trace of past updates and add to gradients.
    optax.trace(decay=momentum),
    # 2. Step: multiply result from step 1 with negative learning rate.
    # Note that `optax.apply_updates()` simply adds the final updates to the
    # parameters, so we must make sure to flip the sign here for gradient
    # descent.
    optax.scale(-learning_rate),
)
```

Weight Decay

Some of Flax's optimizers also include a weight decay. In Optax, some optimizers also have a weight decay parameter (such as `optax.adamw()`), and to others the weight decay can be added as another "gradient transformation" `optax.add_decayed_weights()` that adds an update derived from the parameters.

flax.optim

```
optimizer_def = flax.optim.Adam(
    learning_rate, weight_decay=weight_decay)
optimizer = optimizer_def.create(variables['params'])
```

optax

```
# (Note that you could also use `optax.adamw()` in this case)
tx = optax.chain(
    optax.scale_by_adam(),
    optax.add_decayed_weights(weight_decay),
    # params -= learning_rate * (adam(grads) + params * weight_decay)
    optax.scale(-learning_rate),
)
# Note that you'll need to specify `params` when computing the udpates:
# tx.update(grads, opt_state, params)
```

Gradient Clipping

Training can be stabilized by clipping gradients to a global norm (Pascanu et al, 2012). In Flax this is often done by processing the gradients before passing them to the optimizer. With Optax this becomes just another gradient transformation `optax.clip_by_global_norm()`.

flax.optim

```
def train_step(optimizer, batch):
  grads = jax.grad(loss)(optimizer.target, batch)
  grads_flat, _ = jax.tree_util.tree_flatten(grads)
  global_l2 = jnp.sqrt(sum([jnp.vdot(p, p) for p in grads_flat]))
  g_factor = jnp.minimum(1.0, grad_clip_norm / global_l2)
  grads = jax.tree_util.tree_map(lambda g: g * g_factor, grads)
  return optimizer.apply_gradient(grads)
```

optax

```
tx = optax.chain(
    optax.clip_by_global_norm(grad_clip_norm),
    optax.trace(decay=momentum),
    optax.scale(-learning_rate),
)
```

Learning Rate Schedules

For learning rate schedules, Flax allows overwriting hyper parameters when applying the gradients. Optax maintains a step counter and provides this as an argument to a function for scaling the updates added with `optax.scale_by_schedule()`. Optax also allows specifying a functions to inject arbitrary scalar values for other gradient updates via `optax.inject_hyperparams()`.

Read more about learning rate schedules in the *Learning rate scheduling* guide.

Read more about schedules defined in Optax under Optimizer Schedules. the standard optimizers (like `optax.adam()`, `optax.sgd()` etc.) also accept a learning rate schedule as a parameter for `learning_rate`.

flax.optim

```
def train_step(step, optimizer, batch):
  grads = jax.grad(loss)(optimizer.target, batch)
  return step + 1, optimizer.apply_gradient(grads, learning_rate=schedule(step))
```

optax

```
tx = optax.chain(
    optax.trace(decay=momentum),
    # Note that we still want a negative value for scaling the updates!
    optax.scale_by_schedule(lambda step: -schedule(step)),
)
```

Multiple Optimizers / Updating a Subset of Parameters

In Flax, traversals are used to specify which parameters should be updated by an optimizer. And you can combine traversals using `flax.optim.MultiOptimizer` to apply different optimizers on different parameters. The equivalent in Optax is `optax.masked()` and `optax.chain()`.

Note that the example below is using *`flax.traverse_util`* to create the boolean masks required by `optax.masked()` - alternatively you could also create them manually, or use `optax.multi_transform()` that takes a multivalent pytree to specify gradient transformations.

Beware that `optax.masked()` flattens the pytree internally and the inner gradient transformations will only be called with that partial flattened view of the params/gradients. This is not a problem usually, but it makes it hard to nest multiple levels of masked gradient transformations (because the inner masks will expect the mask to be defined in terms of the partial flattened view that is not readily available outside the outer mask).

flax.optim

```
kernels = flax.traverse_util.ModelParamTraversal(lambda p, _: 'kernel' in p)
biases = flax.traverse_util.ModelParamTraversal(lambda p, _: 'bias' in p)

kernel_opt = flax.optim.Momentum(learning_rate, momentum)
bias_opt = flax.optim.Momentum(learning_rate * 0.1, momentum)


optimizer = flax.optim.MultiOptimizer(
    (kernels, kernel_opt),
    (biases, bias_opt)
).create(variables['params'])
```

optax

```
kernels = flax.traverse_util.ModelParamTraversal(lambda p, _: 'kernel' in p)
biases = flax.traverse_util.ModelParamTraversal(lambda p, _: 'bias' in p)

all_false = jax.tree_util.tree_map(lambda _: False, params)
kernels_mask = kernels.update(lambda _: True, all_false)
biases_mask = biases.update(lambda _: True, all_false)

tx = optax.chain(
    optax.trace(decay=momentum),
    optax.masked(optax.scale(-learning_rate), kernels_mask),
    optax.masked(optax.scale(-learning_rate * 0.1), biases_mask),
)
```

Final Words

All above patterns can of course also be mixed and Optax makes it possible to encapsulate all these transformations into a single place outside the main training loop, which makes testing much easier.

Upgrading my codebase to Linen

As of Flax v0.4.0, `flax.nn` no longer exists, and is replaced with the new Linen API at `flax.linen`. If your codebase is still using the old API, you can use this upgrade guide to upgrade it to Linen.

Defining simple Flax Modules

Old Flax

```
from flax import nn

class Dense(base.Module):
  def apply(self,
            inputs,
            features,
            use_bias=True,
            kernel_init=default_kernel_init,
            bias_init=initializers.zeros_init()):

    kernel = self.param('kernel',
      (inputs.shape[-1], features), kernel_init)
    y = jnp.dot(inputs, kernel)
    if use_bias:
      bias = self.param(
        'bias', (features,), bias_init)
      y = y + bias
    return y

  return new_state, metrics
```

Linen

```
from flax import linen as nn  # [1]

class Dense(nn.Module):
  features: int  # [2]
  use_bias: bool = True
  kernel_init: Callable[[PRNGKey, Shape, Dtype], Array] = default_kernel_init
  bias_init: Callable[[PRNGKey, Shape, Dtype], Array] = initializers.zeros_init()

  @nn.compact
  def __call__(self, inputs):  # [3]
    kernel = self.param('kernel',
      self.kernel_init, (inputs.shape[-1], self.features))  # [4]
    y = jnp.dot(inputs, kernel)
    if self.use_bias:
      bias = self.param(
        'bias', self.bias_init, (self.features,))  # [5]
      y = y + bias
    return y
```

1. Replace `from flax import nn` with `from flax import linen as nn`.
2. Move arguments to `apply` into dataclass attributes. Add type annotations (or use type `Any` to bypass).
3. Rename method `apply` to `__call__` and (optionally) wrap with `@compact`. Methods wrapped in `@compact` can define submodules directly within the method (like in old Flax). You can only wrap a single method with `@compact`. Alternatively, you can define a `setup` method. For more details, please see our other HOWTO Should I use setup or nn.compact?.
4. Access dataclass attributes values by `self.<attr>` inside methods, e.g. `self.features`.

5. Move shape to the end of the arguments to `self.param` (initializer functions can take arbitrary argument lists).

Using Flax Modules inside other Modules

Old Flax

```
class Encoder(nn.Module):

  def apply(self, x):
    x = nn.Dense(x, 500)
    x = nn.relu(x)
    z = nn.Dense(x, 500, name="latents")
    return z
```

Linen

```
class Encoder(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = nn.Dense(500)(x)  # [1]
    x = nn.relu(x)
    z = nn.Dense(500, name='latents')(x)  # [2]
    return z
```

1. Module constructors no longer return the outputs. Instead, they work like normal constructors and return module instances. These instances can be shared like in normal Python (instead of using `.shared()` in old Flax). Since most modules implement `__call__`, you can retain the conciseness of old Flax.
2. Names can be optionally passed to all module constructors.

Sharing submodules and defining multiple methods

Old Flax

```
class AutoEncoder(nn.Module):
  def _create_submodules(self):
    return Decoder.shared(name="encoder")

  def apply(self, x, z_rng, latents=20):
    decoder = self._create_decoder()
    z = Encoder(x, latents, name="encoder")
    return decoder(z)

  @nn.module_method
```

(continues on next page)

(continued from previous page)

```
  def generate(self, z, **unused_kwargs):
    decoder = self._create_decoder()
    return nn.sigmoid(decoder(z))
```

Linen

```
class AutoEncoder(nn.Module):
  latents: int = 20

  def setup(self):  # [1]
    self.encoder = Encoder(self.latents)  # [2]
    self.decoder = Decoder()

  def __call__(self, x):  # [3]
    z = self.encoder(x)
    return self.decoder(z)

  def generate(self, z):  # [4]
    return nn.sigmoid(self.decoder(z))
```

1. Use `setup` instead of `__init__`, which is already defined in the dataclasses library. Flax calls setup right after modules are ready to be used. (You can do this for all modules if you like instead of using `@compact`, but we like how `@compact` co-locates where modules are defined and used, especially if you have loops or conditionals).
2. Like regular Python, share submodules by assigning to self during initialization. Similar to PyTorch, `self.encoder` automatically has the name `"encoder"`.
3. We don't use `@compact` here because we're not defining any inline submodules (all submodules are defined in setup).
4. Define additional methods just like in regular Python.

`Module.partial` inside other modules

Old Flax

```
# no import

class ResNet(nn.Module):
  """ResNetV1."""


  def apply(self, x,
            stage_sizes,
            num_filters=64,
            train=True):
    conv = nn.Conv.partial(bias=False)
    norm = nn.BatchNorm.partial(
```

(continues on next page)

(continued from previous page)

```
        use_running_average=not train,
        momentum=0.9, epsilon=1e-5)

    x = conv(x, num_filters, (7, 7), (2, 2),
             padding=[(3, 3), (3, 3)],
             name='conv_init')
    x = norm(x, name='bn_init')

    # [...]
    return x
```

Linen

```
from functools import partial

class ResNet(nn.Module):
  """ResNetV1."""
  stage_sizes: Sequence[int]
  num_filters: int = 64
  train: bool = True

  @nn.compact
  def __call__(self, x):
    conv = partial(nn.Conv, use_bias=False)
    norm = partial(nn.BatchNorm,
                   use_running_average=not self.train,
                   momentum=0.9, epsilon=1e-5)

    x = conv(self.num_filters, (7, 7), (2, 2),
             padding=[(3, 3), (3, 3)],
             name='conv_init')(x)
    x = norm(name='bn_init')(x)

    # [...]
    return x
```

Use normal `functools.partial` instead of `Module.partial`. The rest stays the same.

Top-level training code patterns

Old Flax

```
def create_model(key):
  _, initial_params = CNN.init_by_shape(
    key, [((1, 28, 28, 1), jnp.float32)])
  model = nn.Model(CNN, initial_params)
  return model

def create_optimizer(model, learning_rate):
  optimizer_def = optim.Momentum(learning_rate=learning_rate)
  optimizer = optimizer_def.create(model)
  return optimizer

def cross_entropy_loss(*, logits, labels):
  one_hot_labels = jax.nn.one_hot(labels, num_classes=10)
  return -jnp.mean(jnp.sum(one_hot_labels * logits, axis=-1))

def loss_fn(model):
  logits = model(batch['image'])
  one_hot = jax.nn.one_hot(batch['label'], num_classes=10)
  loss = -jnp.mean(jnp.sum(one_hot_labels * batch['label'],
                           axis=-1))
  return loss, logits
```

Linen

```
def create_train_state(rng, config):  # [1]
  variables = CNN().init(rng, jnp.ones([1, 28, 28, 1]))  # [2]
  params = variables['params']  # [3]
  tx = optax.sgd(config.learning_rate, config.momentum)  # [4]
  return train_state.TrainState.create(
      apply_fn=CNN.apply, params=params, tx=tx)


def loss_fn(params):
  logits = CNN().apply({'params': params}, batch['image'])  # [5]
  one_hot = jax.nn.one_hot(batch['label'], 10)
  loss = jnp.mean(optax.softmax_cross_entropy(logits=logits,
                                              labels=one_hot))
  return loss, logits
```

1. We no longer use the `Model` abstraction – instead we pass parameters around directly, usually encapsulated in a TrainState object, which can directly be passed to JAX transformations.
2. To compute initial parameters, construct a module instance and call `init` or `init_with_output`. We haven't ported over `init_by_shape` because this function did some magic we did not like (it evaluated the function by shape. but returned real values anyway). Therefore, you should now pass concrete values to the initializer

functions, and you can optimize the initialization by wrapping it with `jax.jit`, which is highly recommended to avoid running a full forward pass.

3. Linen generalizes parameters into variables. Parameters are one "collection" of variables. Variables are nested dicts, where the top-level keys reflect the different variable collections, of which "param" is one of. See the Variables documentation for more details.
4. We recommend using Optax optimizers. See our separate HOWTO called Upgrading my codebase to Optax for more details.
5. To make predictions with your model, make an instance at the top level (this is free – just a wrapper around constructor attributes) and call the `apply` method (which will call `__call__` internally).

Non-trainable variables ("state"): Use within Modules

Old Flax

```
class BatchNorm(nn.Module):
  def apply(self, x, ...):
    # [...]
    ra_mean = self.state(
      'mean', (x.shape[-1], ), initializers.zeros_init())
    ra_var = self.state(
      'var', (x.shape[-1], ), initializers.ones_init())
    # [...]
```

Linen

```
class BatchNorm(nn.Module):
  def __call__(self, x):
    # [...]
    ra_mean = self.variable(
      'batch_stats', 'mean', initializers.zeros_init(), (x.shape[-1], ))
    ra_var = self.variable(
      'batch_stats', 'var', initializers.ones_init(), (x.shape[-1], ))
    # [...]
```

The first argument is the name of the variable collection ("param" is the only variable collection that's always available). Some colllections may be treated as mutable, and others as immutable at top-level training code (see next section for details). Flax also lets you treat each variable collection differently when using JAX transformations inside modules.

Non-trainable variables ("state"): Top-level training code patterns

Old Flax

```
# initial params and state
def initial_model(key, init_batch):
  with nn.stateful() as initial_state:
    _, initial_params = ResNet.init(key, init_batch)
  model = nn.Model(ResNet, initial_params)
  return model, init_state


# updates batch statistics during training
def loss_fn(model, model_state):
  with nn.stateful(model_state) as new_model_state:
    logits = model(batch['image'])
  # [...]



# reads immutable batch statistics during evaluation
def eval_step(model, model_state, batch):
with nn.stateful(model_state, mutable=False):
    logits = model(batch['image'], train=False)
  return compute_metrics(logits, batch['label'])
```

Linen

```
# initial variables ({"param": ..., "batch_stats": ...})
def initial_variables(key, init_batch):
  return ResNet().init(key, init_batch)  # [1]



# updates batch statistics during training
def loss_fn(params, batch_stats):
  variables = {'params': params, 'batch_stats': batch_stats}  # [2]
  logits, new_variables = ResNet(train=true).apply(
    variables, batch['image'], mutable=['batch_stats'])  # [3]
  new_batch_stats = new_variables['batch_stats']
  # [...]


# reads immutable batch statistics during evaluation
def eval_step(params, batch_stats, batch):
  variables = {'params': params, 'batch_stats': batch_stats}
  logits = ResNet(train=False).apply(
    variables, batch['image'], mutable=False)  # [4]
  return compute_metrics(logits, batch['label'])
```

1. `init` returns a variable dict, e.g. `{"param": ..., "batch_stats": ...}` (see Variables documentation).
2. Combine the different variable collections into a variable dict.
3. During training, the `batch_stats` variable collection changes. Since we specify that in the mutable argument, the return value from `module.apply` becomes an ordered pair of `output, new_variables`.
4. During evaluation, we want to raise an error if we're accidentally applying Batch Norm in training mode. By passing `mutable=False` into `module.apply` we enforce that. Since no variables are mutated, the return value is once again just the output.

Loading pre-Linen checkpoints

While most Linen modules should be able to use pre-Linen weights without any modification, there is one catch: In pre-Linen API submodules were numbered incrementally, independent of the submodule class. With Linen this behavior has changed to keep separate submodule counts per module class.

In pre-Linen, params have the following structure:

`{'Conv_0': { ... }, 'Dense_1': { ... } }`

In Linen this is instead:

`{'Conv_0': { ... }, 'Dense_0': { ... } }`

TODO: Add an example here how to load a new `TrainState` object.

Randomness

Old Flax

```
def dropout(inputs, rate, deterministic=False):
  keep_prob = 1. - rate
  if deterministic:
    return inputs
  else:
    mask = random.bernoulli(
    make_rng(), p=keep_prob, shape=inputs.shape)
    return lax.select(
      mask, inputs / keep_prob, jnp.zeros_like(inputs))


def loss_fn(model, dropout_rng):
  with nn.stochastic(dropout_rng):
    logits = model(inputs)
```

Linen

```
class Dropout(nn.Module):
  rate: float

  @nn.compact
  def __call__(self, inputs, deterministic=False):
    keep_prob = 1. - self.rate
    if deterministic:
      return inputs
    else:
      mask = random.bernoulli(
        self.make_rng('dropout'), p=keep_prob, shape=inputs.shape)  # [1]
      return lax.select(
        mask, inputs / keep_prob, jnp.zeros_like(inputs))


def loss_fn(params, dropout_rng):
  logits = Transformer().apply(
    {'params': params}, inputs, rngs={'dropout': dropout_rng})  # [2]
```

1. RNGs in Linen have "kinds" – in this case `'dropout'`. Different kinds can be treated different in JAX transformations (for example, do you want the same dropout mask for each timestep in a sequence model or a different one?)
2. Instead of using the `nn.stochastic` context manager, you pass in RNGs explicitly to `module.apply`. During evaluation you wouldn't pass any RNGs – then if you accidentally use dropout in non-deterministic mode, `self.make_rng('dropout')` would raise an error.

Lifted transformations

In Linen, rather than using JAX transformation directly, we are using "lifted transforms", which are JAX transformations applied to Flax Modules.

For more information, please see the design note on Lifted transformations.

TODO: Given an example of `jax.scan_in_dim` (pre-Linen) vs. `nn.scan` (Linen).

5.2.7 Flax - The Sharp Bits

Flax exposes the full power of JAX. And just like when using JAX, there are certain *"sharp bits"* you may experience when working with Flax. This evolving document is designed to assist you with them.

First, install and/or update Flax:

```
! pip install -qq flax
```

`flax.linen.Dropout` layer and randomness

TL;DR

When working on a model with dropout (subclassed from Flax `Module`), add the `'dropout'` PRNGkey only during the forward pass.

1. Start with `jax.random.split()` to explicitly create PRNG keys for `'params'` and `'dropout'`.
2. Add the `flax.linen.Dropout` layer(s) to your model (subclassed from Flax `Module`).
3. When initializing the model (`flax.linen.init()`), there's no need to pass in an extra `'dropout'` PRNG key—just the `'params'` key like in a "simpler" model.
4. During the forward pass with `flax.linen.apply()`, pass in `rngs={'dropout': dropout_key}`.

Check out a full example below.

Why this works

- Internally, `flax.linen.Dropout` makes use of `flax.linen.Module.make_rng` to create a key for dropout (check out the source code).
- Every time `make_rng` is called (in this case, it's done implicitly in `Dropout`), you get a new PRNG key split from the main/root PRNG key.
- `make_rng` still *guarantees full reproducibility*.

Background

The dropout stochastic regularization technique randomly removes hidden and visible units in a network. Dropout is a random operation, requiring a PRNG state, and Flax (like JAX) uses Threefry PRNG that is splittable.

> Note: Recall that JAX has an explicit way of giving you PRNG keys: you can fork the main PRNG state (such as `key = jax.random.PRNGKey(seed=0)`) into multiple new PRNG keys with `key, subkey = jax.random.split(key)`. Refresh your memory in JAX - The Sharp Bits Randomness and PRNG keys.

Flax provides an *implicit* way of handling PRNG key streams via Flax `Module`'s `flax.linen.Module.make_rng` helper function. It allows the code in Flax `Modules` (or its sub-`Modules`) to "pull PRNG keys". `make_rng` guarantees to provide a unique key each time you call it.

> Note: Recall that `flax.linen.Module` is the base class for all neural network modules. All layers and models are subclassed from it.

Example

Remember that each of the Flax PRNG streams has a name. The example below uses the `'params'` stream for initializing parameters, as well as the `'dropout'` stream. The PRNG key provided to `flax.linen.init()` is the one that seeds the `'params'` PRNG key stream. To draw PRNG keys during the forward pass (with dropout), provide a PRNG key to seed that stream (`'dropout'`) when you call `Module.apply()`.

```
# Setup.
import jax
```

(continues on next page)

(continued from previous page)

```
import jax.numpy as jnp
import flax.linen as nn
```

```
# Randomness.
seed = 0
root_key = jax.random.PRNGKey(seed=seed)
main_key, params_key, dropout_key = jax.random.split(key=root_key, num=3)

# A simple network.
class MyModel(nn.Module):
  num_neurons: int
  training: bool
  @nn.compact
  def __call__(self, x):
    x = nn.Dense(self.num_neurons)(x)
    # Set the dropout layer with a rate of 50% .
    # When the `deterministic` flag is `True`, dropout is turned off.
    x = nn.Dropout(rate=0.5, deterministic=not self.training)(x)
    return x

# Instantiate `MyModel` (you don't need to set `training=True` to
# avoid performing the forward pass computation).
my_model = MyModel(num_neurons=3, training=False)

x = jax.random.uniform(key=main_key, shape=(3, 4, 4))

# Initialize with `flax.linen.init()`.
# The `params_key` is equivalent to a dictionary of PRNGs.
# (Here, you are providing only one PRNG key.)
variables = my_model.init(params_key, x)

# Perform the forward pass with `flax.linen.apply()`.
y = my_model.apply(variables, x, rngs={'dropout': dropout_key})
```

```
No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and rerun for more␣
↪info.)
```

Real-life examples:

- Applying word dropout to a batch of input IDs (in a text classification context).
- Defining a prediction token in a decoder of a sequence-to-sequence model.

5.3 Examples

5.3.1 Core examples

Core examples are hosted on the GitHub Flax repository in the examples directory.

Each example is designed to be **self-contained and easily forkable**, while reproducing relevant results in different areas of machine learning.

As discussed in #231, we decided to go for a standard pattern for all examples including the simplest ones (like MNIST). This makes every example a bit more verbose, but once you know one example, you know the structure of all of them. Having unit tests and integration tests is also very useful when you fork these examples.

Some of the examples below have a link "Interactive" that lets you run them directly in Colab.

Image classification

- MNIST - Interactive: Convolutional neural network for MNIST classification (featuring simple code).
- ImageNet - Interactive: Resnet-50 on ImageNet with weight decay (featuring multi-host SPMD, custom preprocessing, checkpointing, dynamic scaling, mixed precision).

Reinforcement learning

- Proximal Policy Optimization: Learning to play Atari games (featuring single host SPMD, RL setup).

Natural language processing

- Sequence to sequence for number addition: (featuring simple code, LSTM state handling, on the fly data generation).
- Parts-of-speech tagging: Simple transformer encoder model using the universal dependency dataset.
- Sentiment classification: with a LSTM model.
- Transformer encoder/decoder model trained on WMT: Translating English/German (featuring multihost SPMD, dynamic bucketing, attention cache, packed sequences, recipe for TPU training on GCP).
- Transformer encoder trained on one billion word benchmark: for autoregressive language modeling, based on the WMT example above.

Generative models

- Variational auto-encoder: Trained on binarized MNIST (featuring simple code, vmap).

Graph modeling

- Graph Neural Networks: Molecular predictions on ogbg-molpcba from the Open Graph Benchmark.

Contributing to core Flax examples

Most of the core Flax examples on GitHub follow a structure that the Flax dev team found works well with Flax projects. The team strives to make these examples easy to explore and fork. In particular (as per GitHub Issue #231):

- README: contains links to paper, command line, TensorBoard metrics.
- Focus: an example is about a single model/dataset.
- Configs: we use `ml_collections.ConfigDict` stored under `configs/`.
- Tests: executable `main.py` loads `train.py` which has `train_test.py`.
- Data: is read from TensorFlow Datasets.
- Standalone: every directory is self-contained.
- Requirements: versions are pinned in `requirements.txt`.
- Boilerplate: is reduced by using clu.
- Interactive: the example can be explored with a Colab.

5.3.2 Google Research examples

A collection of research by Google Research made with Flax.

Attention

Fast Attention (FAVOR+) and Rethinking Attention with Performers

- Code on GitHub:
  - Performer's Fast Attention (FAVOR+) module
- Research paper:
  - Rethinking Attention with Performers (Choromanski et al., 2020)
    - Introduces *"Performers, Transformer architectures which can estimate regular (softmax) full-rank-attention Transformers with provable accuracy, but using only linear (as opposed to quadratic) space and time complexity, without relying on any priors such as sparsity or low-rankness. To approximate softmax attention-kernels, Performers use a novel Fast Attention Via positive Orthogonal Random features approach (FAVOR+), which may be of independent interest for scalable kernel methods. FAVOR+ can be also used to efficiently model kernelizable attention mechanisms beyond softmax."*

Self-attention Does Not Need O(n^2) Memory

- Code on GitHub
- Colab notebook
- Research paper:
  - Self-attention Does Not Need O(n^2) Memory (Rabe and Staats, 2021)
    - *"We present a very simple algorithm for attention that requires O(1) memory with respect to sequence length and an extension to self-attention that requires O(log n) memory. This is in contrast with the frequently stated belief that self-attention requires O(n^2) memory. While the time complexity is still O(n^2), device memory rather than compute capability is often the limiting factor on modern accelerators. Thus, reducing the memory requirements of attention allows processing of longer sequences than might otherwise be feasible…"*

Computer vision

Colorization Transformer (ColTran)

- Code on GitHub
- Research paper:
  - Colorization Transformer (Kumar et al., 2020)
    - *"We presented the Colorization Transformer (ColTran), an architecture that entirely relies on self-attention for image colorization. We introduce conditional transformer layers, a novel building block for conditional, generative models based on self-attention. Our ablations show the superiority of employing this mechanism over a number of different baselines. Finally, we demonstrate that ColTran can generate diverse, high-fidelity colorizations on ImageNet, which are largely indistinguishable from the ground-truth even for human raters."*

Vision Transformer (ViT), MLP-Mixer Architectures *and* Big Vision

- Code on GitHub:
  - Vision Transformer and MLP-Mixer Architectures
  - Big Vision
    - *"This codebase is designed for training large-scale vision models using Cloud TPU VMs or GPU machines. It is based on Jax/Flax libraries, and uses tf.data and TensorFlow Datasets for scalable and reproducible input pipelines."*
- Colab notebooks:
  - The JAX code of Vision Transformers and MLP Mixers
  - More than 50k Vision Transformer and hybrid checkpoints that were used to generate the data of "How to train your ViT?"
- Research papers:
  - An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (Dosovitskiy et al., 2020)
    - *"In vision, attention is either applied in conjunction with convolutional networks, or used to replace certain components of convolutional networks while keeping their overall structure in place. We show*

that this reliance on CNNs is not necessary and a pure transformer applied directly to sequences of image patches can perform very well on image classification tasks. When pre-trained on large amounts of data and transferred to multiple mid-sized or small image recognition benchmarks (ImageNet, CIFAR-100, VTAB, etc.), Vision Transformer (ViT) attains excellent results compared to state-of-the-art convolutional networks while requiring substantially fewer computational resources to train."

- MLP-Mixer: An All-MLP Architecture for Vision (Tolstikhin et al., 2021)
  - *"In this paper we show that while convolutions and attention are both sufficient for good performance, neither of them are necessary. We present MLP-Mixer, an architecture based exclusively on multi-layer perceptrons (MLPs). MLP-Mixer contains two types of layers: one with MLPs applied independently to image patches (i.e. "mixing" the per-location features), and one with MLPs applied across patches (i.e. "mixing" spatial information). When trained on large datasets, or with modern regularization schemes, MLP-Mixer attains competitive scores on image classification benchmarks, with pre-training and inference cost comparable to state-of-the-art models."*
- How to Train Your ViT? Data, Augmentation, and Regularization in Vision Transformers (Steiner et al., 2021)
  - *"Vision Transformers (ViT) have been shown to attain highly competitive performance for a wide range of vision applications, such as image classification, object detection and semantic image segmentation. In comparison to convolutional neural networks, the Vision Transformer's weaker inductive bias is generally found to cause an increased reliance on model regularization or data augmentation ("AugReg" for short) when training on smaller training datasets. We conduct a systematic empirical study in order to better understand the interplay between the amount of training data, AugReg, model size and compute budget."*
- When Vision Transformers Outperform ResNets without Pretraining or Strong Data Augmentations (X. Chen et al., 2021)
  - *"Vision Transformers (ViTs) and MLPs signal further efforts on replacing hand-wired features or inductive biases with general-purpose neural architectures. Existing works empower the models by massive data, such as large-scale pre-training and/or repeated strong data augmentations, and still report optimization-related problems (e.g., sensitivity to initialization and learning rates). Hence, this paper investigates ViTs and MLP-Mixers from the lens of loss geometry, intending to improve the models' data efficiency at training and generalization at inference."*
- LiT: Zero-Shot Transfer with Locked-image Text Tuning (X. Zhai et al., 2021)
  - *"This paper presents contrastive-tuning, a simple method employing contrastive training to align image and text models while still taking advantage of their pre-training. In our empirical study we find that locked pre-trained image models with unlocked text models work best. We call this instance of contrastive-tuning "Locked-image Tuning" (LiT), which just teaches a text model to read out good representations from a pre-trained image model for new tasks. A LiT model gains the capability of zero-shot transfer to new vision tasks, such as image classification or retrieval. The proposed LiT is widely applicable; it works reliably with multiple pre-training methods (supervised and unsupervised) and across diverse architectures (ResNet, Vision Transformers and MLP-Mixer) using three different image-text datasets."*

Scaling Vision with Sparse Mixture of Experts (MoE)

- Code on GitHub
- Research paper:
  - Scaling Vision with Sparse Mixture of Experts (Riquelme et al., 2021)
    - *"Sparsely-gated Mixture of Experts networks (MoEs) have demonstrated excellent scalability in Natural Language Processing. In Computer Vision, however, almost all performant networks are "dense", that is, every input is processed by every parameter. We present a Vision MoE (V-MoE), a sparse version of the Vision Transformer, that is scalable and competitive with the largest dense networks… we demonstrate the potential of V-MoE to scale vision models, and train a 15B parameter model that attains 90.35% on ImageNet…"*

Diffusion

Variational Diffusion Models

- Code on GitHub
- Colab notebooks
- Research paper:
  - Variational Diffusion Models (Kingma et al., 2021)
    - *"Diffusion-based generative models have demonstrated a capacity for perceptually impressive synthesis, but can they also be great likelihood-based models? We answer this in the affirmative, and introduce a family of diffusion-based generative models that obtain state-of-the-art likelihoods on standard image density estimation benchmarks. Unlike other diffusion-based models, our method allows for efficient optimization of the noise schedule jointly with the rest of the model. We show that the variational lower bound (VLB) simplifies to a remarkably short expression in terms of the signal-to-noise ratio of the diffused data, thereby improving our theoretical understanding of this model class. Using this insight, we prove an equivalence between several models proposed in the literature. In addition, we show that the continuous-time VLB is invariant to the noise schedule, except for the signal-to-noise ratio at its endpoints. This enables us to learn a noise schedule that minimizes the variance of the resulting VLB estimator, leading to faster optimization…"*

Domain adaptation

GIFT (Gradual Interpolation of Features toward Target)

- Code on GitHub
- Research paper:
  - Gradual Domain Adaptation in the Wild: When Intermediate Distributions are Absent (Abnar et al., 2021)
    - *"We focus on the problem of domain adaptation when the goal is shifting the model towards the target distribution, rather than learning domain invariant representations. It has been shown that under the following two assumptions: (a) access to samples from intermediate distributions, and (b) samples being annotated with the amount of change from the source distribution, self-training can be successfully applied on gradually shifted samples to adapt the model toward the target distribution. We hypothesize having (a) is enough to enable iterative self-training to slowly adapt the model to the target distribution, by making use of an implicit curriculum. In the case where (a) does not hold, we observe that iterative*

self-training falls short. We propose GIFT, a method that creates virtual samples from intermediate distributions by interpolating representations of examples from source and target domains…"

Generalization

Surrogate Gap Minimization Improves Sharpness-Aware Training

- Code on GitHub
- Research paper:
  - Surrogate Gap Minimization Improves Sharpness-Aware Training (J. Zhuang et al., 2022)
    - *"The recently proposed Sharpness-Aware Minimization (SAM) improves generalization by minimizing a perturbed loss defined as the maximum loss within a neighborhood in the parameter space. However, we show that both sharp and flat minima can have a low perturbed loss, implying that SAM does not always prefer flat minima. Instead, we define a surrogate gap, a measure equivalent to the dominant eigenvalue of Hessian at a local minimum when the radius of neighborhood (to derive the perturbed loss) is small. The surrogate gap is easy to compute and feasible for direct minimization during training. Based on the above observations, we propose Surrogate Gap Guided Sharpness-Aware Minimization (GSAM), a novel improvement over SAM with negligible computation overhead…"*

Meta learning

`learned_optimization`

- Code on GitHub: learned_optimization
- Colab notebooks
- Research papers:
  - Unbiased Gradient Estimation in Unrolled Computation Graphs with Persistent Evolution Strategies (Vicol et al., 2021)
    - *"We introduce a method called Persistent Evolution Strategies (PES), which divides the computation graph into a series of truncated unrolls, and performs an evolution strategies-based update step after each unroll. PES eliminates bias from these truncations by accumulating correction terms over the entire sequence of unrolls. PES allows for rapid parameter updates, has low memory usage, is unbiased, and has reasonable variance characteristics."*
  - Gradients Are Not All You Need (Metz et al., 2021)
    - *"…In this short report, we discuss a common chaos based failure mode which appears in a variety of differentiable circumstances, ranging from recurrent neural networks and numerical physics simulation to training learned optimizers. We trace this failure to the spectrum of the Jacobian of the system under study, and provide criteria for when a practitioner might expect this failure to spoil their differentiation based optimization algorithms."*

Model efficiency

Efficiently Scaling Transformer Inference

- Code on GitHub:
  - T5X
  - AQT: Accurate Quantized Training
- Research paper:
  - Efficiently Scaling Transformer Inference (Pope et al., 2022)
    - *"We develop a simple analytical model for inference efficiency to select the best multi-dimensional partitioning techniques optimized for TPU v4 slices based on the application requirements. We combine these with a suite of low-level optimizations to achieve a new Pareto frontier on the latency and model FLOPS utilization (MFU) tradeoffs on 500B+ parameter models that outperforms the FasterTransformer suite of benchmarks. We further show that with appropriate partitioning, the lower memory requirements of multiquery attention (i.e. multiple query heads share single key/value head) enables scaling up to 32× larger context lengths."*

Neural rendering / NeRF

Generalizable Patch-Based Neural Rendering

- Code on GitHub
- Research paper:
  - Generalizable Patch-Based Neural Rendering (Suhail et al., 2022)
    - *"…We propose a different paradigm, where no deep features and no NeRF-like volume rendering are needed. Our method is capable of predicting the color of a target ray in a novel scene directly, just from a collection of patches sampled from the scene."*

Voxel-based Radiance Fields in JAX and Flax

- Colab notebook (Velez and Dellaert, 2022)
  - *"In this notebook we show how with JAX/Flax, it is relatively easy to quickly get a voxel-based NeRF variant up and running. Specifically, we will develop a simplified version of DVGO that directly regresses color instead of having a small MLP. It works remarkably well."*

Optimization

Amos Optimizer *and* JEstimator

- Code on GitHub:
  - Amos and JEstimator
    - *"… implements Amos, an optimizer compatible with the optax library, and JEstimator, a light-weight library with a tf.Estimator-like interface to manage T5X-compatible checkpoints for machine learning programs in JAX, which we use to run experiments in the paper."*

- Research paper:
  - Amos: An Adam-style Optimizer with Adaptive Weight Decay towards Model-Oriented Scale (Tian and Parikh, 2022)
    * Presents *"Amos, an optimizer compatible with the optax library, and JEstimator, a light-weight library with a tf.Estimator-like interface to manage T5X-compatible checkpoints for machine learning programs in JAX." "When used for pre-training BERT variants and T5, Amos consistently converges faster than the state-of-the-art settings of AdamW, achieving better validation loss within <=70% training steps and time, while requiring <=51% memory for slot variables."*

Quantization

Pareto-Optimal Quantized ResNet Is Mostly 4-bit *and* AQT: Accurate Quantized Training

- Code on GitHub:
  - AQT: Accurate Quantized Training
- Research paper:
  - Pareto-Optimal Quantized ResNet Is Mostly 4-bit (Abdolrashidi et al., 2021)
    * *"In this work, we use ResNet as a case study to systematically investigate the effects of quantization on inference compute cost-quality tradeoff curves. Our results suggest that for each bfloat16 ResNet model, there are quantized models with lower cost and higher accuracy; in other words, the bfloat16 compute cost-quality tradeoff curve is Pareto-dominated by the 4-bit and 8-bit curves, with models primarily quantized to 4-bit yielding the best Pareto curve… The quantization method we used is optimized for practicality: It requires little tuning and is designed with hardware capabilities in mind… As part of this work, we contribute a quantization library written in JAX…"*

Reinforcement learning

Continuous Control with Action Quantization from Demonstrations (AQuaDem)

- Code on GitHub
- Research paper:
  - Continuous Control with Action Quantization from Demonstrations (Dadashi et al., 2021)
    * Proposes *"a novel Reinforcement Learning (RL) framework for problems with continuous action spaces: Action Quantization from Demonstrations (AQuaDem). The proposed approach consists in learning a discretization of continuous action spaces from human demonstrations. This discretization returns a set of plausible actions (in light of the demonstrations) for each input state, thus capturing the priors of the demonstrator and their multimodal behavior. By discretizing the action space, any discrete action deep RL technique can be readily applied to the continuous control problem. Experiments show that the proposed approach outperforms state-of-the-art methods such as SAC in the RL setup, and GAIL in the Imitation Learning setup."*

Sequence models / Model parallelism

T5X: Scaling Up Models and Data with `t5x` and `seqio`

- Code on GitHub
  - *"T5X is a modular, composable, research-friendly framework for high-performance, configurable, self-service training, evaluation, and inference of sequence models (starting with language) at many scales."*
- Research paper:
  - T5X: Scaling Up Models and Data with t5x and seqio (Roberts et al., 2022)
    - *"Recent neural network-based language models have benefited greatly from scaling up the size of training datasets and the number of parameters in the models themselves. Scaling can be complicated due to various factors including the need to distribute computation on supercomputer clusters (e.g., TPUs), prevent bottlenecks when infeeding data, and ensure reproducible results. In this work, we present two software libraries that ease these issues: t5x simplifies the process of building and training large language models at scale while maintaining ease of use, and seqio provides a task-based API for simple creation of fast and reproducible training data and evaluation pipelines. These open-source libraries have been used to train models with hundreds of billions of parameters on datasets with multiple terabytes of training data. Along with the libraries, we release configurations and instructions for T5-like encoder-decoder models as well as GPT-like decoder-only architectures."*

Simulation

Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation

- Code on GitHub
- Colab notebooks
- Research paper:
  - Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation (Freeman et al., 2021)
    - *"We present Brax, an open source library for rigid body simulation with a focus on performance and parallelism on accelerators, written in JAX. We present results on a suite of tasks inspired by the existing reinforcement learning literature, but remade in our engine. Additionally, we provide reimplementations of PPO, SAC, ES, and direct policy optimization in JAX that compile alongside our environments, allowing the learning algorithm and the environment processing to occur on the same device, and to scale seamlessly on accelerators."*

5.3.3 Repositories that use Flax

The following code bases use Flax and provide training frameworks and a wealth of examples. In many cases, you can also find pre-trained weights:

Hugging Face

Hugging Face is a very popular library for building, training, and deploying state of the art machine learning models. These models can be applied on text, images, and audio. After organizing the JAX/Flax community week, they have now over 5,000 Flax/JAX models in their repository.

DALLE Mini

DALLE Mini is a Transformer-based text-to-image model implemented in JAX/Flax that follows the ideas from the original DALLE paper by OpenAI.

Scenic

Scenic is a codebase/library for computer vision research and beyond. Scenic's main focus is around attention-based models. Scenic has been successfully used to develop classification, segmentation, and detection models for multiple modalities including images, video, audio, and multimodal combinations of them.

Big Vision

Big Vision is a codebase designed for training large-scale vision models using Cloud TPU VMs or GPU machines. It is based on Jax/Flax libraries, and uses tf.data and TensorFlow Datasets for scalable and reproducible input pipelines. This is the original codebase of ViT, MLP-Mixer, LiT, UViM, and many more models.

T5X

T5X is a modular, composable, research-friendly framework for high-performance, configurable, self-service training, evaluation, and inference of sequence models (starting with language) at many scales.

5.3.4 Community examples

In addition to the curated list of official Flax examples on GitHub, there is a growing community of people using Flax to build new types of machine learning models. We are happy to showcase any example built by the community here!

If you want to submit your own Flax example, you can start by forking one of the official Flax examples on GitHub.

Models

Link	Author	Task type	Reference
matthias-wright/flaxmodels	@matthias-wright	Various	GPT-2, ResNet, StyleGAN-2, VGG, …
DarshanDeshpande/jax-models	@Darshan-Deshpande	Various	Segformer, Swin Transformer, … also some stand-alone layers
google/vision_transformer	@andsteing	Image classification, image/text	https://arxiv.org/abs/2010.11929, https://arxiv.org/abs/2105.01601, https://arxiv.org/abs/2111.07991, …
jax-resnet	@n2cholas	Various resnet implementations	torch.hub
Wav2Vec2 fine-tuning	@vasude-vgupta7	Automatic Speech Recognition	https://arxiv.org/abs/2006.11477

Examples

Link	Author	Task type	Reference
JAX-RL	@henry-prior	Reinforcement learning	N/A
BigBird Fine-tuning	@vasude-vgupta7	Question-Answering	https://arxiv.org/abs/2007.14062
Bayesian Networks with BlackJAX	@rlouf	Bayesian Inference, SGMCMC	https://arxiv.org/abs/1402.4102
DCGAN	@bkkaggle	Image Synthesis	https://arxiv.org/abs/1511.06434
denoising-diffusion-flax	@yiyixuxu	Image generation	https://arxiv.org/abs/2006.11239

Tutorials

Link	Author	Task type	Reference

Contributing policy

If you are interested in adding a project to the Community Examples section, take the following into consideration:

- **Code examples**: Examples must contain a README that is helpful, clear, and explains how to run the code. The code itself should be easy to follow.
- **Tutorials**: These docs should preferrably be a Jupyter Notebook format (refer to Contributing to learn how to convert a Jupyter Notebook into a Markdown file with *jupytext*). Your tutorial should be well-written, and discuss/describe an interesting topic/task. To avoid duplication, the content of these docs must be different from existing docs on the Flax documentation site or other community examples mentioned in this document.
- **Models**: repositories with models ported to Flax must provide at least one of the following:
  - Metrics that are comparable to the original work when the model is trained to completion. Having available plots of the metric's history during training is highly encouraged.
  - Tests to verify numerical equivalence against a well known implementation (same inputs + weights = same outputs) preferably using pretrained weights.

In all cases mentioned above, the code must work with the latest stable versions of the following packages: `jax`, `flax`, and `optax`, and make substantial use of Flax. Note that both `jax` and `optax` are required packages of `flax` (refer to the installation instructions for more details).

5.4 Glossary

For additional terms, refer to the Jax glossary.

Bound Module
: When a *`Module`* is created through regular Python object construction (e.g. *module = SomeModule(args...)*, it is in an *unbound* state. This means that only dataclass attributes are set, and no variables are bound to the module. When the pure functions *`Module.init()`* or *`Module.apply()`* are called, Flax clones the Module and binds the variables to it, and the module's method code is executed in a locally bound state, allowing things like calling submodules directly without providing variables. For more details, refer to the module lifecycle.

Compact / Non-compact Module
Modules with a single method are able to declare submodules and variables inline by using the *@nn.compact* decorator. These are referred to as "compact-style modules", whereas modules defining a *setup()* method (usually but not always with multiple callable methods) are referred to as "setup-style modules". To learn more, refer to the setup vs compact guide.

Folding in
Generating a new PRNG key given an input PRNG key and integer. Typically used when you want to generate a new key but still be able to use the original rng key afterwards. You can also do this with jax.random.split but this will effectively create two RNG keys, which is slower.

FrozenDict
An immutable dictionary which can be "unfrozen" to a regular, mutable dictionary. Internally, Flax uses FrozenDicts to ensure variable dicts aren't accidentally mutated. Note: We are considering returning to regular dicts from our APIs, and only using FrozenDicts internally. (see #1223).

Functional core
The flax core library implements the simple container Scope API for threading variables and PRNGs through a model, as well as the lifting machinery needed to transform functions passing Scope objects. The python class-based module API is built on top of this core library.

Lazy initialization
Variables in Flax are initialized late, only when needed. That is, during normal execution of a module, if a requested variable name isn't found in the provided variable collection data, we call the initializer function to create it. This allows us to treat initialization and application under the same code-paths, simplifying the use of JAX transforms with layers.

Lifted transformation
Refer to the Flax docs.

Module
A dataclass allowing the definition and initialization of parameters in a referentially-transparent form. This is responsible for storing and updating variables and parameters within itself. Modules can be readily transformed into functions, allowing them to be trivially used with JAX transformations like *vmap* and *scan*.

Params / parameters
"params" is the canonical variable collection in the variable dictionary (dict). The "params" collection generally contains the trainable weights.

RNG sequences
Inside Flax *Modules*, you can obtain a new PRNG key through *Module.make_rng()*. These keys can be used to generate random numbers through JAX's functional random number generators. Having different RNG sequences (e.g. for "params" and "dropout") allows fine-grained control in a multi-host setup (e.g. initializing parameters identically on different hosts, but have different dropout masks) and treating these sequences differently when lifting transformations.

Scope
A container class for holding the variables and PRNG keys for each layer.

Shape inference
Modules do not need to specify the shape of the input array in their definitions. Flax upon initialization inspects the input array, and infers the correct shapes for parameters in the model.

TrainState
Refer to *flax.training.train_state.TrainState*.

Variable
The weights / parameters / data / arrays residing in the leaves of *variable collections*. Variables are defined inside modules using *Module.variable()*. A variable of collection "params" is simply called a param and can be set using *Module.param()*.

Variable collections
: Entries in the variable dict, containing weights / parameters / data / arrays that are used by the model. "params" is the canonical collection in the variable dict. They are typically differentiable, updated by an outer SGD-like loop / optimizer, rather than modified directly by forward-pass code.

Variable dictionary
: A dictionary containing *variable collections*. Each variable collection is a mapping from a string name (e.g., "*params*" or "batch_stats") to a (possibly nested) dictionary with *Variables* as leaves, matching the submodule tree structure. Read more about pytrees and leaves in the Jax docs.

5.5 Developer notes

5.5.1 The Flax Module lifecycle

This design note is intended for users who are already familiar with Flax Linen Modules but want to understand more about the design principles behind the abstraction. This note should give you a good understanding of the assumptions and guarantees the Module API is built upon. If you have no practical experience with Modules yet, check out the Getting started notebook.

Flax Linen Modules offer a Pythonic abstraction on top of Flax core. The Module abstraction allows you to create classes that have state, parameters and randomness on top of JAX. This is a practical guide to the design and behavior of the `Module` class. By the end, you should feel comfortable to go off the beaten track and use Modules in new ways.

Overview

Definition

Let's start with a high-level overview of the Module lifecycle. First, define a simple Module:

```
class MLP(nn.Module):
  # 1. Attribute annotations
  hidden_size: int
  out_size: int

  # 2. The ``setup`` method
  def setup(self):
    self.hidden = nn.Dense(self.hidden_size)
    self.out = nn.Dense(self.out_size)

  # 3. User methods
  def __call__(self, x):
    a = self.hidden(x)
    h = nn.relu(a)
    return self.out(h)
```

This Module consists of:

1. **Attribute annotations**, defined as dataclass fields. These annotations automatically define a constructor.
2. **The ``setup`` method**, which creates submodules and assigns them to attributes.
3. **User methods**. By convention, most Modules have just one `__call__` method, but you can define multiple methods or use different method names.

Construction/initialization

Now we want to construct and use the `MLP` Module:

```
mlp = MLP(hidden_size=5, out_size=3)
x = jax.numpy.ones((1, 2))
variables = mlp.init(random.PRNGKey(0), x)
y = mlp.apply(variables, x)
```

First, we construct an instance of `MLP` and pass the construction attributes. Note that construction here is different from what you might expect if you are not used to Functional Programming patterns. The `MLP` constructor does not actually create variables or any internal state whatsoever. It's best to think of it as a specification or template of the Module that contains functionality but no data.

Let's take a closer look at initialization. Surprisingly, there actually is no separate initialization path in Flax. Calling `init` is just a special case of `apply`, which you can also write as:

```
# equivalent to: variables = mlp.init(random.PRNGKey(0), x)
_, variables = mlp.apply({}, x, rngs={"params": random.PRNGKey(0)}, mutable=True)
```

Thus, `init` is nothing more than a wrapper around `apply` where:

1. We call a Module without any initial variables (an empty dict).
2. A PRNG generator named `"params"` is always passed for randomly initializing parameters (using the parameter initialization function).
3. All variable collections are set to mutable (`mutable=True`). When a collection is mutable, existing variables can be updated and new variables can be created. Thus, inside `init` variables can be initialized in any variable collection and they are all added to the returned variable dictionary.

Lifecycle

Now that you have learned about `init` being a special case of `apply`, let's look at `.apply(...)` in more detail. In fact, most of the complexity of Modules resides in the `apply` method. The "Module lifecycle" consists of constructing and `apply`-ing a Module. We can summarize the Module lifecycle as follows:

1. We construct `mlp = MLP(hidden_size=5, out_size=3)`, such that `mlp.hidden_size=5` and `mlp.out_size=3`.
2. Then, call `mlp.apply`, which:
    1. Makes a clone of `mlp`, let's call it `mlp_copy`.
    2. Calls `mlp_copy.setup()`.
    3. Returns the output of `mlp_copy.__call__()` and optionally the variable collections that were specified as mutable using the keyword argument `mutable=`.

Notice that the lifecycle includes cloning the Module instance. This is done to ensure that `apply` can be treated as a pure function (i.e., if you pass the same arguments in, it will return the same outputs). You will learn about this in more detail later in the *Top-level Modules* section.

Variables

The word "variable" is ubiquitous in programming and math. However, it's important to have a good understanding of what variables are in the context of JAX and Flax. Inside Flax Modules, variables act like you expect from Python. They are initialized once, read, and perhaps even updated every so often. However, JAX has no concept of variables. Instead, values are stored in arrays similar to NumPy arrays - with one important difference: they are immutable.

The `init` and `apply` methods return the variables as a nested dictionary with string keys and JAX arrays at the leaves. At the top level each key corresponds to a variable collection. Inside each collection the nested dict structure corresponds with the `Module` hierarchy. The variable dict is immutable and therefore really just a snapshot of state the variables are in. When `apply` is called again, the variable dict is passed as an argument. Such that the variables are in the same state as when the previous `init` / `apply` call finished.

Note: Module fields are declared using the *field_name: TypeHint* syntax (same as dataclasses). Without a type hint, an attribute is considered a static property of the class. In case you cannot specify the type you can use `typing.Any` as a wildcard type.

Compact Modules

Linen provides an alternative API for defining modules more compactly. This is especially useful for the common case where the Module consists of only one method that uses parameters and/or sub-modules. Using the compact API the MLP can be rewritten as follows:

```
class CompactMLP(nn.Module):
  hidden_size: int
  out_size: int

  @nn.compact
  def __call__(self, x):
    a = nn.Dense(self.hidden_size)(x)
    h = nn.relu(a)
    return nn.Dense(self.out_size)(h)
```

A compact `Module` is similar in spirit to a function. It offers a concise notation and restricts external interaction to the inputs and return values of the function. In this case the concise notation might make it easier for others to understand what the Module does. There is no need to jump back and forth between the `setup` and `__call__` method to understand what the submodules are doing. Instead, simply reading the `__call__` method from top to bottom once should provide a concise overview. This can make a significant difference if you are implementing complex Modules with many hyperparameters. See setup or compact for a practical guide on deciding between setup and compact.

Another benefit of defining submodules and/or variables inline is that you can add arguments to your method when constructing variables. The most common example of this is using shape information to determine the shape of a parameter like this:

```
class CompactScaledMLP(nn.Module):
  hidden_size: int
  out_size: int

  @nn.compact
  def __call__(self, x):
    scale = self.param("scale", nn.initializers.ones_init(), x.shape[-1:])
    x *= scale[None]
```

(continues on next page)

(continued from previous page)

```
    a = nn.Dense(self.hidden_size)(x)
    h = nn.relu(a)
    return nn.Dense(self.out_size)(h)
```

Many of the standard Linen Modules like `nn.Dense` use shape inference already to avoid the need to specify input shapes (like the number of input features to a Dense layer).

Compact control flow

The order in which you define submodules determines the name of a submodule if none is provided explicitly (using the `name=` keyword argument passed to the Module's constructor). Because the `name` determines how parameters are mapped to submodules, you must be careful about mixing control flow with auto-generated names. Using control flow can change the order or remove certain submodules altogether. This is useful in case a submodule should only exist depending on some construction argument. However, when control flow depends on the input arguments to the Module, you should be careful. For example, the following Module will break:

```
class WrongModule(nn.Module):
  @nn.compact
  def __call__(self, x, mode):
    if mode == "encode":
      return nn.Dense(features=8)(x)
    elif mode == "decode":
      return nn.Dense(features=4)(x)
```

The above Module will break because either the encoder or decoder path will construct a Module named "Dense_0". This means the two Modules will share parameters which is not intended here. Actually, the two Modules cannot share parameters because they each have a different number of features.

This problem can be solved in various ways:

- Provide explicit names
- create the modules in `setup`
- or move the constructor out of the control flow.

The latter is done as follows:

```
class CorrectModule(nn.Module):
  @nn.compact
  def __call__(self, x, mode):
    encoder = nn.Dense(8)
    decoder = nn.Dense(4)
    if mode == "encode":
      return encoder(x)
    elif mode == "decode":
      return decoder(x)
```

In the above example the construction order is fixed. After construction the submodules can be used in an arbitrary order.

Note: compact modules show a strong resemblance to React hooks.

Top-level Modules

When a Module instance is created at the "top-level", it will be in an "unbound" state - that is, it has no variables attached. "Top-level" means it is not constructed as a sub-Module inside another Module class. Apart from calling `init` and `apply`, there is not much you can do with an unbound Module. Note also that `setup` is not called on unbound Modules, so you can only access the construction arguments. Refer to the *Future work* section to learn how this might change in the future.

Why are top-level Modules always unbound?

When we call `apply`, a copy of the top-level Module is created which will actually hold the variables and PRNG sequences. This stateful, "bound", clone only exists while we are executing the apply method. The reason for this is that if you create a stateful object and destroy it before the apply function returns, the `apply` function itself behaves like a pure function. A pure function has two constraints:

1. If you put the same arguments in, it will return the same outputs
2. It does not change anything outside the function. This means you cannot manipulate stateful objects that are accessible outside the pure function.

Pure functions have many advantages but when using JAX they are often essential. For example, most code requires compilation using `jax.jit` to be fast and once you created a Module you probably want to optimize its parameters using `jax.grad`. However, these APIs expect a pure function and don't work on stateful bound `Module` instances directly. Moreover, pure functions allow for flexible interoperability with other libraries. For example, We recommend Optax for optimizing parameters. The optimizers in Optax expect and return a PyTree of JAX arrays to optimize, just like the `apply` function of a Linen Module.

Cloning

To make this approach work reliably we need well-defined cloning behavior. Rather than relying on a complex nested cloning procedure like Python's `deepcopy`, Flax enforces that a `Module` is exactly defined by its construction arguments. Therefore cloning a Module reduces to calling the constructor with its original construction arguments. Because `Module` acts as an immutable dataclass, the construction arguments are mapped directly to instance attributes. Non-construction attributes that are computed in `setup` or `__post_init__` should also depend only on the construciton arguments to ensure a well-defined clone.

Bind

Sometimes it's useful to have a bound, top-level Module without having to wrap the code in a function. For example: to interact with a Module inside a Jupyter notebook. The bind method returns a bound clone with an unlimited lifetime. The downside of this is that you cannot combine it with JAX transformations or integrate it into a vanilla JAX codebase that expects stateless code. For example, Optax can optimze a Pytree of parameters but it cannot directly optimize a bound `Module` instance created with `.bind` (because that's not a Pytree). Thus, you cannot combine the `bind` API with a functional optimizer API like Optax.

Setup

The `setup` method is often used like the constructor hook (`__init__`) in normal Python classes. However, for more advanced use cases it's good to realize that it is not quite the same as a constructor.

`setup` is only called after a Module becomes bound. Normally, this is not an issue because most Modules are bound (almost) immediately (as part of `init` and `apply`). Inside `setup`, sub-modules become bound when they are assigned to an attribute. Inside an `nn.compact` decorated method, sub-modules are bound immediately when constructed. As explained in the previous section, top-level Modules are never bound and thus setup is not called when they are constructed. This means you cannot access attributes assigned in setup from an unbound, top-level module.

```
class TopLevelAccess(nn.Module):

  def setup(self):
    self.foo = nn.Dense(2)

mdl = TopLevelAccess()
assert not hasattr(mdl, "foo")  # foo is not defined because setup is not called
```

The `setup` method is not called immediately after the `Module` becomes bound but only when you interact with the `Module` instance (e.g.: call a method or access an attribute). This should not impact the behavior of a `Module` but the lazy execution does sometimes affect log statements and stack traces during debugging. The section on functionalization will explain why we need `setup` to be lazy in the first place.

Functionalization

So far we had a pure `apply` function that is typically transformed with some JAX transformations and inside `apply` we have a stateful Module instance to work with. In other words: Outside of a Module we are in a functional world where we have the power of JAX's functional transformations and inside the Module we get the power of Flax's stateful variables and PRNG sequence, and the `apply` method is our bridge between these two worlds.

But what if we want to use JAX transformations **inside** Modules? The answer to this is functionalization.

This procedure itself is tedious and error-prone but handled internally by Flax. At a high-level we can summarize it as follows. For a method `fn` defined within a Module:

1. Collect the state (variables & PRNG sequences) of the Module(s) that should be available inside the JAX transformation and take a snapshot of it.
2. Call the JAX transformation with the original arguments and the collected state. Then inside the transformation:
    1. Unpack the state and recreate the Modules
    2. Call the user code `fn`
    3. Collect the updated variables and rng and return it together with the original return values from `fn`
3. Update the original state with the updated state returned from the transformation.

A more in depth explanation of functionalization and lifting can be found in the Lifted Transformation design note.

Practical consequences

For the most part functionalization is something that is handled automatically for you. Still there are some constraints that you must take into account. Most importantly, Flax only handles the stateful primitives (Linen variables and RNGs) and not arbitrary stateful Python code. Most importantly: You cannot close over stateful objects and `Module` objects because they are invisible to Flax's internals (and to JAX in general).

```
class Foo(nn.Module):
  @nn.compact
  def __call__(self, x):
    dense = nn.Dense(x.shape[-1])
    fn = lambda x: dense(x) + 1
    # simply calling inner works fine
    # return self.inner(x, fn)
    # but applying a transformation doesn't:
    vmap_inner = nn.vmap(Foo.inner, in_axes=0, variable_axes={"params": 0}, split_rngs={
↪"params": True})
    return vmap_inner(self, x, fn)

  def inner(self, x, fn):
    for i in range(3):
      x = fn(x)
    return x
```

Here `inner` takes a function that closes over a Module instance. In this example, that works fine because we are not transforming the inner method with a lifted transformation. Most methods are not transformed but it is good to know how to make Module methods transformable.

The main obstacle for transformability are types that JAX does not recognize. JAX only understands Pytree arguments. That's arbitrarily nested Python containers (dict, list, tuple) of (Jax) numpy ndarrays and Python numbers/bools. Flax allows to define dataclasses which are Pytree compatible using the flax.struct API.

Function closure is the most common way to accidentally hide a JAX array or Linen Module from a transformation. There is however an easy workaround if you want to pass closures that are also compatible with JAX and Linen transformations:

```
class Partial(flax.struct.PyTreeNode):
  fn: Callable = flax.struct.field(pytree_node=False)
  args: Iterable[Any]

  def __call__(self, *args, **kwargs):
    return self.fn(*(tuple(self.args) + args), **kwargs)

class Foo(nn.Module):

  @nn.compact
  def __call__(self, x):
    dense = nn.Dense(x.shape[-1])
    fn = lambda mdl, x: mdl(x) + 1
    vmap_inner = nn.vmap(Foo.inner, in_axes=0, variable_axes={"params": 0}, split_rngs={
↪"params": True})
    return vmap_inner(self, x, Partial(fn, [dense]))

  def inner(self, x, fn):
```

(continues on next page)

(continued from previous page)

```
    for i in range(3):
      x = fn(x)
    return x
```

Here the closure is implemented using a Flax dataclass. The function itself is annotated with `flax.struct.field(pytree_node=False)` to indicate that it does not contain JAX Arrays or Linen Modules. The partially applied `args` on the other hand is treated as a pytree container. We rewrite the closure to use Partial. Now the inner method can be transformed using lifted transformations.

Future work

Setup for unbound Modules

The current Module abstraction is particularly restrictive when it comes to initializing fields after construction. In the current Module API, the `setup` method is the place to initialize the fields of the Module instance. Because `setup` is only called on a bound Module, the full Module API is available inside `setup`, including variable declaration. However, oftentimes we don't actually require any stateful API's to initialize a field. In fact, most commonly we simply want to declare a submodule. More importantly, it's often useful to inspect submodules for debugging or to partially run the model. Consider for example:

```
class AutoEncoder(nn.Module):
  def setup(self):
    self.encoder = Encoder(...)
    self.decoder = Decoder(...)
```

Imagine we want to call just the decoder using *auto_encoder.decoder.apply(decoder_variables, x)*. With the current setup API this does not work because we must first bind the variables before setup is called and the decoder attribute is defined. Of course we can manually construct the Decoder Module with the same attributes as in setup but this is not ideal in many cases.

There are two possible solutions to make this use case more ergonomic. First, setup could be made to run immediately after construction before it becomes bound. This means you can still create sub modules but you can no longer define or manipulate variables. Therefore, this would be a breaking change and it would require a new API for defining variables lazily

Alternatively, an additional special method could be introduced that runs right away after Module construction and before it becomes bound. In this case, the `setup` method would preserve its original semantics.

5.5.2 Lifted transformations

Advanced topic

This design note explains the underlying implementation of `flax.linen.transform`, which enables JAX transformations inside Flax `Modules`.

Introduction

JAX uses a functional API meaning that it only guarantees correct behavior when using functions without side effects (JAX docs). Typically, these side effects are the result of mutating an object that lives outside the function.

The functional paradigm has some advantages like the ability to explicitly reason about state and stochasticity. The function output only changes when an input argument changes. Therefore, a function is guaranteed to behave deterministically.

But pure functions offer another big advantage to JAX: specifically, they enable functional transformations. For example `jax.vmap(f)` will vectorize a function `f`. Because `f` cannot have side effects the vectorized/parallel version of `f` is well-defined. To see why we need this restriction, consider what happens if `f` would increment a counter or draw a random number. Would `f` draw the same or a different random number for each item in the vector? Would each item in the batch have its own counter or is the counter shared among the items? And in what order is the counter incremented if `f` is computed in parallel? The answer to all these questions is "it depends". The behavior is ambiguous and the functional constraint elegantly avoids this problem.

Flax introduces a safe way to have limited randomness and stateful variables in a JAX-compatible form. The reason why the state in Flax is not problematic is because it is local: inside a Flax `Module` there are variables and PRNG sequences, but on the outside there are only JAX Arrays and PRNG keys.

For most use cases, Flax is used to define models in a stateful way. Because a `Module` behaves like a pure function externally, we can fully utilize JAX with all of its transformations. There are, however, cases when we want to have the best of both worlds by using transformations and `Module` together. This design note explains how we extend JAX's functional transformation to work on `Module`s that have internal state and randomness.

Functionalization

Before we jump into the details let's consider a simple example where we would like to use `vmap` inside a `Module`.

First, we define a simple MLP without any transformations:

```
import jax
from jax import random, numpy as jnp
from flax import linen as nn

class MLP(nn.Module):
  @nn.compact
  def __call__(self, xs):
    h = nn.Dense(4, name='hidden')(xs)
    h = nn.relu(h)
    return nn.Dense(1, name='out')(h)
```

Now what if we want to have separate MLP parameters for each item in `xs`? If this were "vanilla JAX" we could imagine writing something like `jax.vmap(apply_mlp)(mlp_params, xs)`. But doing something like this in Linen will actually fail:

```
class NaiveVmapMLP(nn.Module):
  @nn.compact
  def __call__(self, xs):
    mlp = MLP()
    return jax.vmap(lambda mlp, x: mlp(x))(mlp, xs)  # fails
```

JAX will raise an error when `vmap` is used on `mlp` because it's not a JAX array or a simple container of arrays. We can not really blame JAX for refusing to perform this under-specified job. After all, it's not even clear what should happen here. The parameters inside the MLP are not even initialized yet and we will need a separate PRNG key for

each group of parameters. `jax.vmap` can only broadcast or map over an axis but it cannot automatically split an PRNG key. Therefore, we have to call `jax.random.split` manually.

We can fix this problem by first turning `MLP` into a pure init and apply function. Afterwards, we use the `param` method to store the parameters:

```
class ManualVmapMLP(nn.Module):
  @nn.compact
  def __call__(self, xs):
    mlp = MLP(parent=None)
    init_fn = lambda rng, xs: jax.vmap(mlp.init, in_axes=0)(random.split(rng, xs.
↪shape[0]), xs)['params']
    apply_fn = jax.vmap(mlp.apply, in_axes=0)
    mlp_params = self.param('mlp', init_fn, xs)
    return apply_fn({'params': mlp_params}, xs)

xs = jnp.ones((3, 4))
variables = ManualVmapMLP().init(random.PRNGKey(0), xs)
print(jax.tree_util.tree_map(jnp.shape, variables['params']))
"""==>
{
    mlp: {
        hidden: {
            bias: (3, 4),
            kernel: (3, 4, 4),
        },
        out: {
            bias: (3, 1),
            kernel: (3, 4, 1),
        },
    },
}
"""
```

Here, `MLP(parent=None)` creates a detached instance of `MLP`. This avoids reserving a name for the submodule inside the current module. Although not strictly necessary, this also ensures we cannot accidentally use the MLP instance in a stateful way and we are forced to use it through either `.init` or `.apply`.

This example is still relatively concise but it already takes a few extra "bookkeeping" statements to make it work. However, this implementation has a number of limitations:

1. During initialization, we call the submodule twice through `init_fn` and `apply_fn`. If the submodule used the same trick to do functional transformation we will end up executing a lot of code as the number of module calls grows like 2^d where d is the number of nested function transformations.
2. The implementation assumes the submodule only requires the parameter RNG sequence.
3. The implementation assumes we only create variables in the "params" collection during `init`. However, it does not support other variable collections and creating/updating variables in `apply`.

Point 3 in particular makes manual functionalization cumbersome. Feel free to try and extend the above example with a `nn.BatchNorm` layer in the `MLP` module. This will require dealing with some additional complexity like storing the updated batch stats and making sure the batch stats are not mutable inside `vmap` when it should be immutable (e.g.: eval mode).

We call the process of transforming a stateful Module into a pure function "functionalization". By temporarily turning a stateful `Module` into a function we make it compatible with JAX's functional transformations.

Lifting

Flax provides an alternative for manual functionalization which we call lifted transformation. Lifted transformations are defined in `flax.core.lift`. All the lifted JAX transformations are defined with a single generic lifting API called `pack`.

A number of decisions had to be made in order to define `pack`. The implementation of `pack` controls how variables and rngs are lifted and how fine-grained the user control is. It must also decide whether lifting decisions are made at variable or transformation definition.

Lifting granularity

With the Linen API, users can define arbitrary variable collections and PRNG sequences. Each variable in a collection is lifted in the same way.

Collections are typically given a semantically meaningful name like "params" or "batch_stats" rather than a general purpose name like "state". Because collections carry semantic meaning we can decide at the transformation level how each collection should be lifted. For example, we want to share all parameter variables when we add a batch dimension to a model.

At the same time we can write generic code that uses transformations without knowing exactly what kind of variables the submodules will create. Collections thus strike a balance between fine-grained control and generality. We also avoid brittle string matching code that loops over all variables and tries to split up collections in an ad-hoc way based on naming conventions like: target all variables with the name prefix "kernel". If more fine-grained control is necessary a user can simply split up a set of variables over multiple collections that should be handled differently.

Transformation vs variable control

Lifting behavior could be defined either at the transformation level or during variable definition. We use transformation level definitions of lifting behavior. The reason for this choice is that there are many different transformations with various behaviors. For example: `vmap` has broadcasted and vectorized arguments, while `scan` has scan, carry, and broadcast arguments. A variable would have to define its behavior for all these transformations otherwise a `Module` would not be compatible with these transformations. Alternatively, we would have to make default decisions for how transformations are handled. However, this could lead to silent bugs because the behavior might not actually be valid given the users intent.

The lift package also provides a general purpose `transform`, which allows an arbitrary function to transform a variable collection. For example, this can be used to tie the weights in a tied auto-encoder by transposing the weights. It is unclear whether a similar general purpose transform could be defined if lifting decisions were made at variable definition.

Linen

The lifting module does not know about the Linen `Module` API. Instead it operates directly on instances of `flax.core.Scope`. A `Scope` instance contains the variables and PRNG sequences of a `Module`. Each `Module` instance has a `Scope` instance in the `.scope` field if it has a parent or it was created using `init` or `apply`. Typically, the top-level `Module` instance — on which you call `init` or `apply` — is the only `Module` instance that does not have a `Scope` bound to it.

When a `Module` is transformed, we use the `flax.core.lift` APIs to lift the scope and use `Module.clone()` to create a new `Module` instance with the lifted scope bound to it.

`flax.linen.transforms` exposes wrappers for the transformations in `flax.core.lift`. The core lifting APIs operate on functions while the Linen wrappers can transform either a `Module` class or a `Module` method.

Thus, lifting is implemented independently from the Linen API. This separation of concern simplifies the implementation, while potentially allowing alternative `Module` abstractions to build upon a common core for lifting and state management.

Implementation

The `pack(fn, in_vars, out_vars, rngs)` API goes through the following stages:

1. *Scope de-duplication*

   This stage is only relevant if multiple Scopes are lifted together. In this case we must first find the set of root scopes. A scope is a root if none of its ancestors are in the set of scopes that need to be lifted.

   By only lifting roots we avoid lifting the same variables twice.

   For non-root scopes we store a reference to its ancestor scope and a path such that we can later reconstruct it (stage 4).

2. *Filter stage*

   Variables and PRNG sequences are split up into groups. This way `fn` can lift each group into the transformation separately. A group is defined by a filter specified as:

   - a list of collections/prng names
   - `True` (match everything)
   - `False` (match nothing)
   - `DenyList(filter)` (match everything but the specified collections (e.g.: `DenyList(['params'])` matches everything except the 'params' collection.)).

   A collection or PRNG sequence can only be put into a single group. If a collection matches multiple filters, it will be put into the first group with a matching filter. If a collection or PRNG sequence does not match any filter it will not be lifted. This means that it cannot be used inside the transformation and attempting to do this will cause an error to be raised. For example, `in_vars = (["params"], True)` will cause the "params" collection to be put in the first group and all other collection to be put in the second group.

   For each PRNG sequence that is matched we seed a new PRNG sequence by calling `make_rng`. This avoids the need to update the PRNG state after the lifted transformation is complete.

3. *Transform-specific lifting*

   `fn` is called with the variable and PRNG groups. JAX transforms have varying signatures and lifting options. Arguably the cleanest example is `vmap`. In the case of vmap the function arguments, PRNGs and variable collections are passed into a `jax.vmap` wrapped function.

4. *Scope reconstruction*

   Now that the variables and PRNGs are lifted inside the transformation, we want to recreate the lifted scopes. Pack calls `fn` with a `scope_fn` that takes the lifted variables and PRNGs and returns the reconstructed scopes with the lifted variables and rng sequences.

5. *Repack stage*

   After we have used the lifted scopes we have to retrieve the updated variables (PRNG sequences can simply be discarded). pack passes the `repack_fn` to support this. This stage is similar to stage 2 except that we only lift variables and immutable variables are ignored. Immutable variables cannot be updated. Therefore, they should not be returned from the transformed function.

6. *Commit stage*

pack expects fn to return a pair where the first item will simply be returned from pack and the second item should be the repacked variables. The updated variables are stored in the original/un-lifted scopes such that the mutations that happen inside the transformation survive after the transformation completes.

Using pack example

A minimal example of using pack to transpose each matrix in a variable collection:

```
from flax.core import lift
from flax.core import Scope, init, apply, nn as core_nn

def lift_transpose(fn, target='params', variables=True, rngs=True):
  # by default we transpose 'params' and simply pass through all other variables.
  def wrapper(scope_fn, repack_fn, variable_groups, rng_groups, *args):
    # normally we would first call into a JAX transformed function here...
    target, rest = variable_groups
    def trans(x):
      if x.ndim == 2:
        return x.T
      return x
    target = jax.tree_util.tree_map(trans, target)
    variable_groups = (target, rest)
    scope = scope_fn(variable_groups, rng_groups)
    y = fn(scope, *args)
    out_variables = repack_fn(scope)
    return y, out_variables
  return lift.pack(
      wrapper,
      in_variable_filters=(target, variables),
      out_variable_filters=(variables,),
      rng_filters=(rngs,))

x = jnp.ones((3, 2))
y, params = init(lift_transpose(core_nn.dense))(random.PRNGKey(0), x, 4)
```

NOTE that most users should not need to interact with pack directly. Please open a GitHub issue when you find a use case that is not supported yet by the existing lifted transformations.

Supported transformations

Jax Transform	Supported in Linen?	Comments
vmap		
scan		Carry variables cannot be initialized inside the scan body.
remat		
jit		Current implementation might cause unnecessary recompilation.
jvp		
vjp		
custom_vjp		
custom_jvp		
while_loop		Carry variables cannot be initialized inside the while_loop body.
cond		Variable initialization / mutation must structurally match across branches.
switch		Variable initialization / mutation must structurally match across branches.
pmap		
xmap		

References:

- Linen transforms documentation.
- Linen transforms source code
- Core lifting source code

Linen examples

Going back to our original example, we can now use `nn.vmap` to simplify our implementation:

```
class LinenVmapMLP(nn.Module):
  @nn.compact
  def __call__(self, xs):
    VmapMLP = nn.vmap(MLP, variable_axes={'params': 0}, split_rngs={'params': True}, in_
↪axes=0)
    return VmapMLP(name='mlp')(xs)

variables = LinenVmapMLP().init(random.PRNGKey(0), xs)
print(jax.tree_util.tree_map(jnp.shape, variables['params']))
"""==>
{
    mlp: {
```

(continues on next page)

(continued from previous page)

```
        Dense_0: {
            bias: (3, 4),
            kernel: (3, 2, 4),
        },
        Dense_1: {
            bias: (3, 1),
            kernel: (3, 4, 1),
        },
    },
}
"""
```

Here we use `variable_axes={'params': 0}` to indicate that parameters are vectorized rather than shared and `split_rngs={'params': True}` means each set of parameters is initialized independently.

We can also extend the example with some inner state by adding a `BatchNorm` layer:

```
class StatefulMLP(nn.Module):
  @nn.compact
  def __call__(self, x, *, train):
    h = nn.Dense(4, name='hidden')(x)
    h = nn.BatchNorm(axis_name='batch')(h, use_running_average=not train)
    h = nn.relu(h)
    return nn.Dense(1, name='out')(h)

class LinenStatefulVmapMLP(nn.Module):
  @nn.compact
  def __call__(self, xs, *, train):
    VmapMLP = nn.vmap(StatefulMLP, variable_axes={'params': 0, 'batch_stats': 0}, split_
↪rngs={'params': True}, in_axes=0)
    return VmapMLP(name='mlp')(xs, train=train)
variables = LinenStatefulVmapMLP().init(random.PRNGKey(0), xs)
```

All we had to add to `nn.vmap` is `'batch_stats': 0`, indicating that the batch stats are vectorized rather than shared along the first axis.

Alternatives

Other numerical computation frameworks consider variables a first-class citizen. An alternative to functionalization would be to use a variable system either integrated or on top of JAX. An advantage of this is that per-variable lifting becomes easier. If variables are part of the JAX IR (JAXPR), we could inspect which variables have to be lifted in a certain computation. Optionally, they could be annotated with a collection tag to decide on various lifting options.

The downside of this approach is that a variable system is more complicated. Variables are related references and break a core assumption of Functional Programming (see referential transparency) Other APIs that currently have a functional interface would probably require integration as well (e.g.: checkpointing and optimization APIs).

5.6 The Flax philosophy

In no particular order:

- Library code should be easy to read and understand.
- Prefer duplicating code over a bad abstraction.
- Generally, prefer duplicating code over adding options to functions.
- Comment-driven design: If it's hard to document your code, consider changing the design.
- Unit test-driven design: If it's hard to test your code, consider changing the design.
- People start projects by copying an existing implementation — make base implementations excellent.
- If we expose an abstraction to our developers, we own the mental overhead.
- Developer-facing functional programming abstractions confuse some users, expose them where the benefit is high.
- "Read the manual" is not an appropriate response to developer confusion. The framework should guide developers towards good solutions, such as through assertions and error messages.
- An unhelpful error message is a bug.
- "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." — Brian Kernighan

5.6.1 Design principles

Flax is a neural network library built on JAX that has been adopted by a growing set of users, most notably in the JAX submissions for the MLPerf 0.7 benchmark. Our experience over the last year (and many conversations with users and JAX core devs) has guided a redesign of the API called Linen (`flax.linen`) in response to the following basic design questions.

How does a neural network library benefit from being built on JAX and leverage JAX's unique strengths?

The world already has TensorFlow and PyTorch, and there's little need to build a clone of either. We believe that the composable function-transformation approach that JAX takes opens up new frontiers for making neural net code more maintainable, more scalable and more performant than existing libraries. While we strive to offer an API familiar to those experienced with Keras/Sonnet/PyTorch, Linen is fundamentally a functional system for defining neural nets in JAX. Just a few examples of what we believe a JAX-targeted library can enable:

- Write models as "single-example" code and introduce batching automatically with `jax.vmap`.
- Automatically handle ragged batches in NLP and other masking issues.
- Create efficient compile-time and runtime models by utilizing rematerialized `scan` for massive convolutional networks.
- Remove memory headaches by enabling easy rematerialization, reversibility, and model-parallel data sharding.

How does one interoperate with JAX transformations?

Arguably, the entire point of a neural net library is to offer an implicit variable management API to save the user from having to manually thread thousands of variables through a complex tree of functions. However, JAX operates on pure functions. To handle both current and future JAX transforms (configured and composed in any way), Linen Modules are directly "functionalized", that is, automatically cast in-place as explicit functions of the form:

$$f\left(v_{in}, x\right) \rightarrow v_{out}, y$$

Where v_{in} is the variable collections and PRNG state used by the model, v_{out} the mutated output variable collections, x the input data and y the output data. Applying JAX transformations then simply reduces to specifying any argument-specific transform options to the various variable collections and PRNG state. This unleashes the flexibility and strength of JAX transformations – for example, one can achieve either device-parallel training or per-device ensembling by using `jax.pmap` in different ways, without any explicit library support. Moreover, **within Modules**, we expose lightweight wrappers around the complex JAX transforms such as `jax.vmap` and `jax.lax.scan` that annotate how each variable collection is to be transformed by JAX. Importantly, we handle the nontrivial cases of creating new variables and transformed variables under mapping and loop transforms correctly for initialization and application.

How are parameters represented, and how do we handle general "differentiable algorithms" that update stateful variables?

We follow the JAX functional conventions of storing data in "pytrees": JAX arrays contained in nested tuples, lists, dictionaries. Because researchers inevitably manually interact with this data, we use nested dictionaries with meaningful default keys and offer several utilities (traversals, etc.) for handling them directly. Linen uses an accelerated version of a Python frozen dictionary that caches its JAX-flattened form to speed up `jit`ted function call overheads.

Flax generalizes the operation of a neural net by allowing models to accept collections of several different "kinds": parameters, batch-norm stats, autoregressive caches, debug information, fine-grained hyperparameters, etc. Each collection is stored in a nested dictionary of the same structure as the model. Importantly, we do *not* conflate these various kinds under the single vague rubric of "state", but keep different logical types of variables separate that can be treated differently under JAX transformations and under mutations (e.g. training vs prediction). Similarly, we allow for multiple separate named PRNG chains inside Modules for separate treatment of randomness for different applications such as initialization, dropout, sampling, etc.

At every stage the data associated with a neural net is not kept in a custom object hierarchy, but left in an explicit, Python and JAX native form that is easy to introspect and modify. Users have utilized this to map TF and PyTorch checkpoints to Flax, to implement submodel-specific loss terms, and to perform fast model surgery, etc. For saving this data, most Flax examples store these nested dictionaries via the efficient "msgpack" binary format – but as variables are simply Python dicts, you can use any (non-JAX-aware) serialization library directly.

How does one interoperate with purely functional JAX code?

To be broadly useful to the JAX ecosystem, users shouldn't need to heavily refactor their code in order to add "trainability" for a given numerical task. *"The library should not get in the way."* Utilizing purely functional code from within Linen is trivial: Module implementations are just JAX code with named variables. Using Linen Modules inside otherwise purely functional code can be as simple as using a single top-level Module transformation to allow initialization and pure application of any JAX program that might contain various trainable sections.

5.7 How to contribute

Everyone can contribute to Flax, and the Flax development team values everyone's contributions! You can contribute in many more ways than just writing code. Answering questions on the Flax GitHub Discussions page, helping each other, and improving Flax documentation are extremely valuable to the Flax ecosystem.

We also appreciate if you spread the word, for instance by starring the Flax GitHub repository, or referencing Flax in blog posts of projects that used it.

This project follows Google's Open Source Community Guidelines.

5.7.1 Ways to contribute

We welcome pull requests (PRs), in particular for those issues marked as PR-ready. For other proposals, you should first open a GitHub Issue or a GitHub Discussion to start a conversation about your planned contribution.

5.7.2 Contributing code using pull requests

The Flax development team performs all development using Git. To contribute, you should have basic knowledge of Git and GitHub. (You can learn how to set up Git by following Git's official Getting Started - First-Time Git Setup and GitHub's Set Up Git guides.)

To contribute code to Flax on GitHub, follow these steps:

To create a pull request from a fork

1. Using GitHub's web UI, fork the Flax repository by clicking the 'Fork' button on the `github.com/google/flax` repository page. This creates a fork (a copy) of the Flax repository in your own GitHub.

   Reference: Creating a pull request from a fork.

2. Install Python >=3.7.

3. (Optional) Create a virtual environment or a Docker container. See `dev/README.md` for details on how to set up a Docker Container. To set up a virtual environment, run the following:

   ```
   python3 -m virtualenv env
   . env/bin/activate
   ```

   This ensures all your dependencies are installed in this environment.

4. Clone your local forked Flax repo with `git clone`. Then, install the required packages with PyPi. This enables you to immediately test the code after modifying it:

   ```
   git clone https://github.com/YOUR_USERNAME/flax
   cd flax
   pip install -e .[all]
   pip install -e .[testing]
   pip install -r docs/requirements.txt
   ```

5. Set up pre-commit hooks, this will run some automated checks during each `git` commit and possibly update some files that require changes.

   ```
   pip install pre-commit
   pre-commit install
   ```

6. Add the Google Flax repo (not your fork) as an upstream remote, so you can use it to sync your changes.

```
git remote add upstream http://www.github.com/google/flax
```

7. Create a branch, such as `my_development_branch`, you will develop from:

```
git checkout -b my_development_branch
```

8. Implement your changes using your favorite editor (we recommend Visual Studio Code).

   Make sure the tests pass by running the following command from the top of the repository:

```
./tests/run_all_tests.sh
```

9. Once you finish making changes, don't forget to create commits (learn how to write a commit message):

```
git add file1.py file2.py ...
# or use `git add .` to add all changed files
git commit -m "Your commit message"
```

   Then sync your code with the main repository:

```
git rebase upstream/main
```

10. Finally, push your commit on your `my_development_branch`, and create a remote branch in your fork that you can use to create a pull request from:

```
git push --set-upstream origin my_development_branch
```

After running the command, you should get a GitHub link in your (VS Code) terminal output for creating a pull request. If you don't receive a link after `git push`, use the GitHub web UI to create a pull request.

11. Make sure your pull request passes the Flax PR checklist. If so, create a pull request from the Flax repository and send it for review. Consult GitHub Help for more information on using pull requests.

You can learn more in GitHub's Creating a pull request from a fork . documentation.

Updating Jupyter Notebooks

We use jupytext to maintain two synced copies of docs in `docs/notebooks`: one in the Jupyter Notebook (`.ipynb`) format, and one in Markdown (`.md`).

The former can be opened and executed directly in Google Colab. Markdown makes it easier to track changes/diffs within version control and, for example, GitHub web UI, since `.ipynb` files are based on JSON.

Editing Jupyter Notebooks (`.ipynb`)

For making large changes that substantially modify code and outputs, it's recommended to edit the notebooks in Jupyter or in Colab.

If you choose to work in Colab, go to **File** and click **Upload notebook**, then pick your file. After loading it into Colab and editing it, make sure you run the cells, and that there aren't any errors. Click on **Runtime**, then select **Run all**. After you finish, click **File > Download > Download ipynb**. You may also want to test that the file executes properly by using `sphinx-build`, as explained above.

After you make changes in your Jupyter Notebook, follow the steps *Syncing notebooks* below.

Editing Markdown files (`.md`)

For making smaller changes to the text content of the notebooks, it is easiest to edit the `.md` versions using a text editor.

After you make changes in your Markdown file, follow the steps *Syncing notebooks* below.

Syncing notebooks

After editing either the `.ipynb` or `.md` versions of the docs, sync the two versions using jupytext by running `jupytext --sync` on the updated notebooks.

First, make sure you have jupytext installed. The jupytext version should match the one specified in .pre-commit-config.yaml (currently, it is v1.13.8).

```
pip install jupytext==1.13.8
```

Then, after you have made your changes in the Jupyter Notebook, sync the contents with its Markdown-equivalent file by running the following command:

```
jupytext --sync path/to/the/file.ipynb
```

Similarly, to sync your Markdown file with its Jupyter Notebook version, run:

```
jupytext --sync path/to/the/file.md
```

Note that if you receive an error, and it is the first time you worked in a Jupyter Notebook, you may need to (re)create a synced copy of the document (which is explained in detail in *Creating new notebooks* section below):

```
jupytext --set-formats ipynb,md:myst path/to/the/notebook.ipynb
```

Once you're finished with syncing the `.md` and `.ipynb` files, you can check that they are properly synced using the pre-commit framework to perform the same checks used in the Flax GitHub CI:

```
git add docs -u  # pre-commit runs on files in git staging.
pre-commit run jupytext
```

Creating new notebooks

If you are adding a new Jupyter Notebook to the documentation, you can use `jupytext --set-formats`. It can set up both the Jupyter Notebook (`.ipynb`) and Markdown (`.md`) versions of the file:

```
jupytext --set-formats ipynb,md:myst path/to/the/notebook.ipynb
```

This works by adding a `"jupytext"` metadata field to the notebook file which specifies the desired formats. The `jupytext --sync` command can then recognize them when invoked.

After you make changes in your file(s), follow the steps from the *Syncing notebooks* section above to keep the contents of both Markdown and Jupyter Notebook files in sync.

Notebooks within the Sphinx build

Some of the notebooks are built automatically as part of the pre-submit checks and as part of the Read the Docs build. The build will fail if cells raise errors. If the errors are intentional, you can either catch them, or tag the cell with `raises-exceptions` metadata (example PR). You have to add this metadata by hand in the `.ipynb` file. It will be preserved when somebody else re-saves the notebook.

We exclude some notebooks from the build because, for example, they contain long computations. See `exclude_patterns` in `conf.py`.

Updating the pull request contents

Every pull request should ideally be limited to just one commit, so if you have multiple commits please squash them.

Assuming you now have only one commit in your pull request, and want to add changes requested during review:

1. Make the changes locally in your editor.
2. Run `git commit -a --amend`. This updates the commit contents and allows you to edit the commit message.
3. At this point, `git push` alone will result in an error. Instead, use `git push --force`.
4. Check that it's done: The changes to your commit should be immediately reflected in the Github web UI.

5.7.3 Troubleshooting

Too many commits in a pull request

If your PR has too many commits associated with it (for example, more than five), you need to squash them. Otherwise, the Flax docs build process may fail with an error message. This is because of the following reasons:

- There are more than five commits in your pull request; and
- The Flax source sync process fails when the commit tree is too large.

To squash your commits, you can rebase your branch to `main` and create a new commit containing all your changes, run the following command:

```
git rebase main && git reset --soft main && git commit
```

This will apply all your changes to the main branch. Note that if you had to resolve any conflicts while working on your change (for instance, you did a `pull upstream main` which led to conflict), then you will have to resolve these conflicts again.

After you have successfully rebased your branch, you should push your changes. And because you changed the commit history, you may have to use `git push --force`.

5.7.4 Contributor License Agreement

Contributions to this project must be accompanied by a Contributor License Agreement. You (or your employer) retain the copyright to your contribution; this simply gives us permission to use and redistribute your contributions as part of the project. Head over to https://cla.developers.google.com/ to see your current agreements on file or to sign a new one.

You generally only need to submit a CLA once, so if you've already submitted one (even if it was for a different project), you probably don't need to do it again.

5.8 API Reference

5.8.1 flax.config package

Global configuration options for Flax.

Now a wrapper over jax.config, in which all config vars have a 'flax_' prefix.

To modify a config value on run time, call: `flax.config.update('flax_<config_name>', <value>)`

`flax.configurations.`**`define_bool_state`**(*name*, *default*, *help*)

Set up a boolean flag using JAX's config system.

The flag will actually be stored as an environment variable of 'FLAX_<UPPERCASE_NAME>'. JAX config ensures that the flag can be overwritten on runtime with *flax.config.update('flax_<config_name>', <value>)*.

`flax.configurations.`**`static_bool_env`**(*varname*, *default*)

Read an environment variable and interpret it as a boolean.

This is deprecated. Please use define_bool_state() unless your flag will be used in a static method and does not require runtime updates.

True values are (case insensitive): 'y', 'yes', 't', 'true', 'on', and '1'; false values are 'n', 'no', 'f', 'false', 'off', and '0'. :param varname: the name of the variable :param default: the default boolean value

Returns
boolean return value derived from defaults and environment.

Raises: ValueError if the environment variable is anything else.

5.8.2 flax.core.frozen_dict package

class `flax.core.frozen_dict.`**`FrozenDict`**(**args*, *__unsafe_skip_copy__=False*, ***kwargs*)

An immutable variant of the Python dict.

`copy`(*add_or_replace*)

Create a new FrozenDict with additional or replaced entries.

`pop`(*key*)

Create a new FrozenDict where one entry is removed.

Example:

```
state, params = variables.pop('params')
```

Parameters
key – the key to remove from the dict

Returns
A pair with the new FrozenDict and the removed value.

`pretty_repr`(*num_spaces=4*)

Returns an indented representation of the nested dictionary.

`unfreeze`()

Unfreeze this FrozenDict.

Returns
An unfrozen version of this FrozenDict instance.

`flax.core.frozen_dict.freeze`(*xs*)

Freeze a nested dict.

Makes a nested *dict* immutable by transforming it into *FrozenDict*.

Parameters
xs – Dictionary to freeze (a regualr Python dict).

Returns
The frozen dictionary.

`flax.core.frozen_dict.unfreeze`(*x*)

Unfreeze a FrozenDict.

Makes a mutable copy of a *FrozenDict* mutable by transforming it into (nested) dict.

Parameters
x – Frozen dictionary to unfreeze.

Returns
The unfrozen dictionary (a regular Python dict).

`flax.core.frozen_dict.copy`(*x*, *add_or_replace*)

Create a new dict with additional and/or replaced entries. This is a utility function that can act on either a FrozenDict or regular dict and mimics the behavior of *FrozenDict.copy*.

Example:

new_variables = copy(variables, {'additional_entries': 1})

Parameters

- **x** – the dictionary to be copied and updated
- **add_or_replace** – dictionary of key-value pairs to add or replace in the dict x

Returns
A new dict with the additional and/or replaced entries.

`flax.core.frozen_dict.pop`(*x*, *key*)

Create a new dict where one entry is removed. This is a utility function that can act on either a FrozenDict or regular dict and mimics the behavior of *FrozenDict.pop*.

Example:

```
state, params = pop(variables, 'params')
```

Parameters

- **x** – the dictionary to remove the entry from
- **key** – the key to remove from the dict

Returns
A pair with the new dict and the removed value.

flax.core.frozen_dict.**pretty_repr**(*x*, *num_spaces=4*)

Returns an indented representation of the nested dictionary. This is a utility function that can act on either a FrozenDict or regular dict and mimics the behavior of *FrozenDict.pretty_repr*. If x is any other dtype, this function will return *repr(x)*.

Parameters

- **x** – the dictionary to be represented
- **num_spaces** – the number of space characters in each indentation level

Returns
An indented string representation of the nested dictionary.

5.8.3 flax.error package

Flax has the following classes of errors.

exception flax.errors.**AlreadyExistsError**(*path*)

Attempting to overwrite a file via copy.

You can pass `overwrite=True` to disable this behavior and overwrite existing files in.

exception flax.errors.**ApplyModuleInvalidMethodError**(*method*)

When calling *Module.apply()*, you can specify

the method to apply using parameter `method`. This error is thrown if the provided parameter is not a method in the Module and not a function with at least one argument.

Learn more on the reference docs for *Module.apply()*.

exception flax.errors.**ApplyScopeInvalidVariablesStructureError**(*variables*)

This error is thrown when the dict passed as *variables* to apply() has an

extra 'params' layer, i.e. {'params': {'params': ...}}. For more explanation on variable dicts, please see *flax.core.variables*.

exception flax.errors.**ApplyScopeInvalidVariablesTypeError**

When calling *Module.apply()*, the first

argument should be a variable dict. For more explanation on variable dicts, please see *flax.core.variables*.

exception flax.errors.**AssignSubModuleError**(*cls*)

You are only allowed to create submodules in two places:

1. If your Module is noncompact: inside *Module.setup()*.
2. If your Module is compact: inside the method wrapped in *nn.compact()*.

For instance, the following code throws this error, because `nn.Conv` is created in `__call__`, which is not marked as compact:

```
class Foo(nn.Module):
  def setup(self):
    pass

  def __call__(self, x):
    conv = nn.Conv(features=3, kernel_size=3)

Foo().init(random.PRNGKey(0), jnp.zeros((1,)))
```

Note that this error is also thrown if you partially defined a Module inside setup:

```
class Foo(nn.Module):
  def setup(self):
    self.conv = functools.partial(nn.Conv, features=3)

  def __call__(self, x):
    x = self.conv(kernel_size=4)(x)
    return x

Foo().init(random.PRNGKey(0), jnp.zeros((1,)))
```

In this case, `self.conv(kernel_size=4)` is called from `__call__`, which is disallowed because it's neither within `setup` nor a method wrapped in x``nn.compact``.

exception flax.errors.**CallCompactUnboundModuleError**

This error occurs when you are trying to call a Module directly, rather than

through *Module.apply()*. For instance, the error will be raised when trying to run this code:

```
from flax import linen as nn
import jax.numpy as jnp

test_dense = nn.Dense(10)
test_dense(jnp.ones((5,5)))
```

Instead, you should pass the variables (parameters and other state) via *Module.apply()* (or use *Module.init()* to get initial variables):

```
from jax import random
variables = test_dense.init(random.PRNGKey(0), jnp.ones((5,5)))

y = test_dense.apply(variables, jnp.ones((5,5)))
```

exception flax.errors.**CallSetupUnboundModuleError**

This error occurs when you are trying to call *.setup()* directly.

For instance, the error will be raised when trying to run this code:

```
from flax import linen as nn
import jax.numpy as jnp

class MyModule(nn.Module):
  def setup(self):
    self.submodule = MySubModule()
```

(continues on next page)

(continued from previous page)

```
module = MyModule()
module.setup() # <-- ERROR!
submodule = module.submodule
```

In general you shouldn't call *.setup()* yourself, if you need to get access to a field or submodule defined inside *setup* you can instead create a function to extract it and pass it to *nn.apply*:

```
# setup() will be called automatically by `nn.apply`
def get_submodule(module):
  return module.submodule.clone() # avoid leaking the Scope

empty_variables = {} # you can also use the real variables
submodule = nn.apply(get_submodule, module)(empty_variables)
```

exception flax.errors.**CallUnbindOnUnboundModuleError**

This error occurs when you are trying to call `.unbind()` on an unbound Module. For instance, when you try running the following example, an error will be raised:

```
from flax import linen as nn

class MyModule(nn.Module):
  @nn.compact
  def __call__(self, x):
    return nn.Dense(features=10)(x)

module = MyModule()
module.unbind() # <-- ERROR!
```

Instead, you should `bind` the Module to a variable collection before calling `.unbind()`:

```
bound_module = module.bind(variables)
... # do something with bound_module
module = bound_module.unbind() # <-- OK!
```

exception flax.errors.**DescriptorAttributeError**

This error occurs when you are trying to access a property that is accessing a non-existent attribute.

For example, the error will be raised when trying to run this code:

```
class Foo(nn.Module):
    @property
    def prop(self):
        return self.non_existent_field # ERROR!
    def __call__(self, x):
        return self.prop

foo = Foo()
variables = foo.init(jax.random.PRNGKey(0), jnp.ones(shape=(1, 8)))
```

exception flax.errors.**IncorrectPostInitOverrideError**

This error occurs when you overrode *.__post_init__()* without calling *super().__post_init__()*.

For example, the error will be raised when trying to run this code:

```
from flax import linen as nn
import jax.numpy as jnp
import jax
class A(nn.Module):
  x: float
  def __post_init__(self):
    self.x_square = self.x ** 2
    # super().__post_init__() <-- forgot to add this line
  @nn.compact
  def __call__(self, input):
    return input + 3

r = A(x=3)
r.init(jax.random.PRNGKey(2), jnp.ones(3))
```

exception flax.errors.**InvalidCheckpointError**(*path*, *step*)

A checkpoint cannot be stored in a directory that already has

a checkpoint at the current or a later step.

You can pass `overwrite=True` to disable this behavior and overwrite existing checkpoints in the target directory.

exception flax.errors.**InvalidFilterError**(*filter_like*)

A filter should be either a boolean, a string or a container object.

exception flax.errors.**InvalidInstanceModuleError**

This error occurs when you are trying to call *.init()*, *.init_with_output()*, *.apply() or `.bind()*

on the Module class itself, instead of an instance of the Module class. For example, the error will be raised when trying to run this code:

```
class B(nn.Module):
  @nn.compact
  def __call__(self, x):
    return x

k = random.PRNGKey(0)
x = random.uniform(random.PRNGKey(1), (2,))
B.init(k, x)   # B is module class, not B() a module instance
B.apply(vs, x)   # similar issue with apply called on class instead of
instance.
```

exception flax.errors.**InvalidRngError**(*msg*)

All rngs used in a Module should be passed to

Module.init() and *Module.apply()* appropriately. We explain both separately using the following example:

```
class Bar(nn.Module):
  @nn.compact
  def __call__(self, x):
    some_param = self.param('some_param', nn.initializers.zeros_init(), (1, ))
    dropout_rng = self.make_rng('dropout')
    x = nn.Dense(features=4)(x)
    ...
```

(continues on next page)

(continued from previous page)

```
class Foo(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = Bar()(x)
    ...
```

PRNGs for Module.init()

In this example, two rngs are used:

- `params` is used for initializing the parameters of the model. This rng is used to initialize the `some_params` parameter, and for initializing the weights of the `Dense` Module used in `Bar`.
- `dropout` is used for the dropout rng that is used in `Bar`.

So, `Foo` is initialized as follows:

```
init_rngs = {'params': random.PRNGKey(0), 'dropout': random.PRNGKey(1)}
variables = Foo().init(init_rngs, init_inputs)
```

If a Module only requires an rng for `params`, you can use:

```
SomeModule().init(rng, ...)  # Shorthand for {'params': rng}
```

PRNGs for Module.apply()

When applying `Foo`, only the rng for `dropout` is needed, because `params` is only used for initializing the Module parameters:

```
Foo().apply(variables, inputs, rngs={'dropout': random.PRNGKey(2)})
```

If a Module only requires an rng for `params`, you don't have to provide rngs for apply at all:

```
SomeModule().apply(variables, inputs)  # rngs=None
```

exception flax.errors.**InvalidScopeError**(*scope_name*)

A temporary Scope is only valid within the context in which it is created:

with Scope(variables, rngs=rngs).temporary() as root:
y = fn(root, *args, **kwargs) # Here root is valid.

Here root is invalid.

exception flax.errors.**JaxTransformError**

JAX transforms and Flax modules cannot be mixed.

JAX's functional transformations expect pure function. When you want to use JAX transformations **inside** Flax models, you should make use of the Flax transformation wrappers (e.g.: `flax.linen.vmap`, `flax.linen.scan`, etc.).

exception flax.errors.**LazyInitError**(*partial_val*)

Lazy Init function has uncomputable return values.

This happens when passing an argument to lazy_init with `jax.ShapeDtypeStruct` that affects the initialized variables. Make sure the init function only uses the shape and dtype or pass an actual JAX array if this is impossible.

Example:

```
class Foo(nn.Module):
  @compact
  def __call__(self, x):
    # This parameter depends on the input x
    # this causes an error when using lazy_init.
    k = self.param("kernel", lambda _: x)
    return x * k
Foo().lazy_init(random.PRNGKey(0), jax.ShapeDtypeStruct((8, 4), jnp.float32))
```

exception flax.errors.**MPACheckpointingRequiredError**(*path*, *step*)

To optimally save and restore a multiprocess array (GDA or jax Array outputted from pjit), use GlobalAsyncCheckpointManager.

You can create an GlobalAsyncCheckpointManager at top-level and pass it as argument:

```
from jax.experimental.gda_serialization import serialization as gdas
gda_manager = gdas.GlobalAsyncCheckpointManager()
save_checkpoint(..., gda_manager=gda_manager)
```

exception flax.errors.**MPARestoreDataCorruptedError**(*step*, *path*)

A multiprocess array stored in Google Cloud Storage doesn't contain a "commit_success.txt" file, which should be written at the end of the save.

Failure of finding it could indicate a corruption of your saved GDA data.

exception flax.errors.**MPARestoreTargetRequiredError**(*path*, *step*, *key=None*)

Provide a valid target when restoring a checkpoint with a multiprocess array.

Multiprocess arrays need a sharding (global meshes and partition specs) to be initialized. Therefore, to restore a checkpoint that contains a multiprocess array, make sure the `target` you passed contains valid multiprocess arrays at the corresponding tree structure location. If you cannot provide a full valid `target`, consider `allow_partial_mpa_restoration=True`.

exception flax.errors.**MPARestoreTypeNotMatchError**(*step*, *gda_path*)

Make sure the multiprocess array type you use matches your configuration in jax.config.jax_array.

If you turned *jax.config.jax_array* on, you should use *jax.experimental.array.Array* everywhere, instead of using *GlobalDeviceArray*. Otherwise, avoid using jax.experimental.array to restore your checkpoint.

exception flax.errors.**ModifyScopeVariableError**(*col*, *variable_name*, *scope_path*)

You cannot update a variable if the collection it belongs to is immutable.

When you are applying a Module, you should specify which variable collections are mutable:

```
class MyModule(nn.Module):
  @nn.compact
  def __call__(self, x):
    ...
    var = self.variable('batch_stats', 'mean', ...)
    var.value = ...
    ...

v = MyModule.init(...)
...
logits = MyModule.apply(v, batch)  # This throws an error.
logits = MyModule.apply(v, batch, mutable=['batch_stats'])  # This works.
```

exception flax.errors.**MultipleMethodsCompactError**

The `@compact` decorator may only be added to at most one method in a Flax

module. In order to resolve this, you can:

- remove `@compact` and define submodules and variables using *Module.setup()*.
- Use two separate modules that both have a unique `@compact` method.

TODO(marcvanzee): Link to a design note explaining the motivation behind this. There is no need for an equivalent to *hk.transparent* and it makes submodules much more sane because there is no need to prefix the method names.

exception flax.errors.**NameInUseError**(*key_type*, *value*, *module_name*)

This error is raised when trying to create a submodule, param, or variable

with an existing name. They are all considered to be in the same namespace.

Sharing Submodules

This is the wrong pattern for sharing submodules:

```
y = nn.Dense(feature=3, name='bar')(x)
z = nn.Dense(feature=3, name='bar')(x+epsilon)
```

Instead, modules should be shared by instance:

```
dense = nn.Dense(feature=3, name='bar')
y = dense(x)
z = dense(x+epsilon)
```

If submodules are not provided with a name, a unique name will be given to them automatically:

```
class MyModule(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = MySubModule()(x)
    x = MySubModule()(x)  # This is fine.
    return x
```

Parameters and Variables

A parameter name can collide with a submodule or variable, since they are all stored in the same variable dict:

```
class Foo(nn.Module):
  @nn.compact
  def __call__(self, x):
    bar = self.param('bar', nn.initializers.zeros_init(), (1, ))
    embed = nn.Embed(num_embeddings=2, features=5, name='bar')  # <-- ERROR!
```

Variables should also have unique names, even if they have their own collection:

```
class Foo(nn.Module):
  @nn.compact
  def __call__(self, inputs):
    _ = self.param('mean', initializers.lecun_normal(), (2, 2))
    _ = self.variable('stats', 'mean', initializers.zeros_init(), (2, 2))
```

exception flax.errors.**PartitioningUnspecifiedError**(*target*)

This error is raised when trying to add an axis to a Partitioned variable by

using a transformation (e.g.: `scan`, `vmap`) without specifying the "partition_name" in the `metadata_params` dict.

exception flax.errors.**ReservedModuleAttributeError**(*annotations*)

This error is thrown when creating a Module that is using reserved attributes.

The following attributes are reserved:

- `parent`: The parent Module of this Module.
- `name`: The name of this Module.

exception flax.errors.**ScopeCollectionNotFound**(*col_name*, *var_name*, *scope_path*)

This error is thrown when trying to access a variable from an empty collection.

There are two common causes: 1. | The collection was not passed to `apply` correctly.

For example, you might have used `module.apply(params, ...)` instead of `module.apply({'params': params}, ...)`.

2. The collection is empty because the variables need to be initialized.
   In this case, you should have made the collection mutable during
   apply (e.g.: `module.apply(variables, ..., mutable=['state'])`.

exception flax.errors.**ScopeParamNotFoundError**(*param_name*, *scope_path*)

This error is thrown when trying to access a parameter that does not exist.

For instance, in the code below, the initialized embedding name 'embedding' does not match the apply name 'embed':

```
class Embed(nn.Module):
  num_embeddings: int
  features: int

  @nn.compact
  def __call__(self, inputs, embed_name='embedding'):
    inputs = inputs.astype('int32')
    embedding = self.param(embed_name,
                           jax.nn.initializers.lecun_normal(),
                           (self.num_embeddings, self.features))
    return embedding[inputs]

model = Embed(4, 8)
variables = model.init(random.PRNGKey(0), jnp.ones((5, 5, 1)))
_ = model.apply(variables, jnp.ones((5, 5, 1)), 'embed')
```

exception flax.errors.**ScopeParamShapeError**(*param_name*, *scope_path*, *value_shape*, *init_shape*)

This error is thrown when the shape of an existing parameter is different from

the shape of the return value of the `init_fn`. This can happen when the shape provided during *Module.apply()* is different from the one used when initializing the module.

For instance, the following code throws this error because the apply shape (`(5, 5, 1)`) is different from the init shape (`(5, 5)`. As a result, the shape of the kernel during `init` is `(1, 8)`, and the shape during `apply` is `(5, 8)`, which results in this error.:

```
class NoBiasDense(nn.Module):
  features: int = 8

  @nn.compact
  def __call__(self, x):
    kernel = self.param('kernel',
                        lecun_normal(),
                        (x.shape[-1], self.features))  # <--- ERROR
    y = lax.dot_general(x, kernel,
                        (((x.ndim - 1,), (0,)), ((), ())))
    return y

variables = NoBiasDense().init(random.PRNGKey(0), jnp.ones((5, 5, 1)))
_ = NoBiasDense().apply(variables, jnp.ones((5, 5)))
```

exception flax.errors.**ScopeVariableNotFoundError**(*name*, *col*, *scope_path*)

This error is thrown when trying to use a variable in a Scope in a collection

that is immutable. In order to create this variable, mark the collection as mutable explicitly using the `mutable` keyword in *Module.apply()*.

exception flax.errors.**SetAttributeFrozenModuleError**(*module_cls*, *attr_name*, *attr_val*)

You can only assign Module attributes to `self` inside

Module.setup(). Outside of that method, the Module instance is frozen (i.e., immutable). This behavior is similar to frozen Python dataclasses.

For instance, this error is raised in the following case:

```
class SomeModule(nn.Module):
  @nn.compact
  def __call__(self, x, num_features=10):
    self.num_features = num_features  # <-- ERROR!
    x = nn.Dense(self.num_features)(x)
    return x

s = SomeModule().init(random.PRNGKey(0), jnp.ones((5, 5)))
```

Similarly, the error is raised when trying to modify a submodule's attributes after constructing it, even if this is done in the `setup()` method of the parent module:

```
class Foo(nn.Module):
    def setup(self):
      self.dense = nn.Dense(features=10)
      self.dense.features = 20  # <--- This is not allowed

    def __call__(self, x):
      return self.dense(x)
```

exception flax.errors.**SetAttributeInModuleSetupError**

You are not allowed to modify Module class attributes in

Module.setup():

```
class Foo(nn.Module):
  features: int = 6

  def setup(self):
    self.features = 3  # <-- ERROR

  def __call__(self, x):
    return nn.Dense(self.features)(x)

variables = SomeModule().init(random.PRNGKey(0), jnp.ones((1, )))
```

Instead, these attributes should be set when initializing the Module:

```
class Foo(nn.Module):
  features: int = 6

  @nn.compact
  def __call__(self, x):
    return nn.Dense(self.features)(x)

variables = SomeModule(features=3).init(random.PRNGKey(0), jnp.ones((1, )))
```

TODO(marcvanzee): Link to a design note explaining why it's necessary for modules to stay frozen (otherwise we can't safely clone them, which we use for lifted transformations).

exception flax.errors.**TransformTargetError**(*target*)

Linen transformations must be applied to Modules classes or functions taking a Module instance as the first argument.

This error occurs when passing an invalid target to a linen transform (nn.vmap, nn.scan, etc.). This occurs for example when trying to transform a Module instance:

```
nn.vmap(nn.Dense(features))(x)  # raises TransformTargetError
```

You can transform the *nn.Dense* class directly instead:

```
nn.vmap(nn.Dense)(features)(x)
```

Or you can create a function that takes the module instance as the first argument:

```
class BatchDense(nn.Module):
  @nn.compact
  def __call__(self, x):
    return nn.vmap(
        lambda mdl, x: mdl(x),
        variable_axes={'params': 0}, split_rngs={'params':
        True})(nn.Dense(3), x)
```

exception flax.errors.**TransformedMethodReturnValueError**(*name*)

Transformed Module methods cannot return other Modules or Variables.

5.8.4 flax.jax_utils package

Utilities we could consider upstreaming to Jax.

flax.jax_utils.**partial_eval_by_shape**(*fn*, *input_spec*, **args*, ***kwargs*)

Lazily evaluate a function by using the shapes of the inputs.

This function is similar to *jax.eval_shape* with the key difference that function outputs that can be computed without a concrete value of the inputs are returned as is instead of only the shape. See for example *module.init_by_shape* where this functionality is used to initialize a model without using input data lr computation.

Parameters

- **fn** – the function to be lazily evaluated.
- **input_spec** – an iterable of shapes or (shape, dtype) tuples specifying the shape and type of the inputs. If unspecified the dtype is float32.
- ***args** – other arguments passed to the module's apply function
- ****kwargs** – keyword arguments passed to the module's apply function

Returns

A pair consisting of the model output and an instance of Model

Multi device utilities

flax.jax_utils.**replicate**(*tree*, *devices=None*)

Replicates arrays to multiple devices.

Parameters

- **tree** – a pytree containing the arrays that should be replicated.
- **devices** – the devices the data is replicated to (default: same order as expected by *jax.pmap()*).

Returns

A new pytree containing the replicated arrays.

flax.jax_utils.**unreplicate**(*tree*)

Returns a single instance of a replicated array.

flax.jax_utils.**prefetch_to_device**(*iterator*, *size*, *devices=None*)

Shard and prefetch batches on device.

This utility takes an iterator and returns a new iterator which fills an on device prefetch buffer. Eager prefetching can improve the performance of training loops significantly by overlapping compute and data transfer.

This utility is mostly useful for GPUs, for TPUs and CPUs it should not be necessary – the TPU & CPU memory allocators (normally) don't pick a memory location that isn't free yet so they don't block. Instead those allocators OOM.

Parameters

- **iterator** – an iterator that yields a pytree of ndarrays where the first dimension is sharded across devices.
- **size** – the size of the prefetch buffer.

  If you're training on GPUs, 2 is generally the best choice because this guarantees that you can overlap a training step on GPU with a data prefetch step on CPU.

- **devices** – the list of devices to which the arrays should be prefetched.

  Defaults to the order of devices expected by *jax.pmap*.

Yields

The original items from the iterator where each ndarray is now sharded to the specified devices.

`flax.jax_utils.`**pmean**(*xs*, *axis_name*)

`flax.jax_utils.`**pad_shard_unpad**(*wrapped*, *static_argnums=(0,)*, *static_argnames=()*, *static_return=False*)

Wraps a function with code that pads, shards, then un-shards, un-pads.

Parameters

- **wrapped** – the function to be wrapped. Signature is *params, *args, *kwargs*.
- **static_argnums** – indices of arguments to *wrapped* that should _not_ be padded and sharded, but instead be forwarded as-is. The default is (0,) because by far the most common use-case is to pass *params* first.
- **static_argnames** – names of kwargs to *wrapped* that should _not_ be padded and sharded, but instead be forwarded as-is.
- **static_return** – whether not to un-shard, and un-pad the return value; static return values are typically used with eval steps that compute metrics

Returns

A new function that pads and shards its arguments before passing them to the wrapped function, and un-shards and un-pads the returned pytree.

This is useful for calling a pmap'ed function with inputs that aren't divisible by the number of devices. A typical use is:

@pad_shard_unpad @jax.pmap def forward(params, x): ...

Notes

The padding is done in host-memory before being passed to the function, and the values returned by the function are transferred back to host memory.

The returned function is augmented with a new keyword-only argument *min_device_batch* that, if specified, forces padding inputs to at least this size per device. This can be useful to avoid recompiles for the last batch and reduce memory fragmentation.

For more information refer to https://flax.readthedocs.io/en/latest/guides/full_eval.html

5.8.5 flax.linen package

Linen is the Flax Module system. Read more about our design goals in the Linen README.

Module

class flax.linen.**Module**

Base class for all neural network modules. Layers and models should subclass this class.

All Flax Modules are Python 3.7 dataclasses. Since dataclasses take over `__init__`, you should instead override *setup()*, which is automatically called to initialize the module.

Modules can contain submodules, and in this way can be nested in a tree structure. Submodels can be assigned as regular attributes inside the *setup()* method.

You can define arbitrary "forward pass" methods on your Module subclass. While no methods are special-cased, `__call__` is a popular choice because it allows you to use module instances as if they are functions:

```
from flax import linen as nn

class Module(nn.Module):
  features: Tuple[int, ...] = (16, 4)

  def setup(self):
    self.dense1 = Dense(self.features[0])
    self.dense2 = Dense(self.features[1])

  def __call__(self, x):
    return self.dense2(nn.relu(self.dense1(x)))
```

Optionally, for more concise module implementations where submodules definitions are co-located with their usage, you can use the *compact()* wrapper.

__setattr__(*name*, *val*)

Sets an attribute on this Module.

We overload setattr solely to support pythonic naming via assignment of submodules in the special *setup()* function:

```
self.submodule_name = MyModule(...)
```

We also support lists and other general pytrees, e.g.:

```
self.submodules = [MyModule0(..), MyModule1(..), ...]
```

Parameters

- **name** – Attribute to set.
- **val** – Value of the attribute.

apply(*variables*, **args*, *rngs=None*, *method=None*, *mutable=False*, *capture_intermediates=False*, ***kwargs*)

Applies a module method to variables and returns output and modified variables.

Note that *method* should be set if one would like to call *apply* on a different class method than `__call__`. For instance, suppose a Transformer modules has a method called *encode*, then the following calls *apply* on that method:

```
model = Transformer()
encoded = model.apply({'params': params}, x, method=Transformer.encode)
```

If a function instance is provided, the unbound function is used. For instance, the example below is equivalent to the one above:

```
encoded = model.apply({'params': params}, x, method=model.encode)
```

You can also pass a string to a callable attribute of the module. For example, the previous can be written as:

```
encoded = model.apply({'params': params}, x, method='encode')
```

Note `method` can also be a function that is not defined in `Transformer`. In that case, the function should have at least one argument representing an instance of the Module class:

```
def other_fn(instance, ...):
  instance.some_module_attr(...)
  ...

model.apply({'params': params}, x, method=other_fn)
```

Parameters

- **variables** – A dictionary containing variables keyed by variable collections. See *flax.core.variables* for more details about variables.
- ***args** – Named arguments passed to the specified apply method.
- **rngs** – a dict of PRNGKeys to initialize the PRNG sequences. The "params" PRNG sequence is used to initialize parameters.
- **method** – A function to call apply on. This is generally a function in the module. If provided, applies this method. If not provided, applies the `__call__` method of the module. A string can also be provided to specify a method by name.
- **mutable** – Can be bool, str, or list. Specifies which collections should be treated as mutable: `bool`: all/no collections are mutable. `str`: The name of a single mutable collection. `list`: A list of names of mutable collections.
- **capture_intermediates** – If *True*, captures intermediate return values of all Modules inside the "intermediates" collection. By default only the return values of all `__call__` methods are stored. A function can be passed to change the filter behavior. The filter function takes the Module instance and method name and returns a bool indicating whether the output of that method invocation should be stored.
- ****kwargs** – Keyword arguments passed to the specified apply method.

Returns

If `mutable` is False, returns output. If any collections are mutable, returns `(output, vars)`, where `vars` are is a dict of the modified collections.

bind(*variables*, **args*, *rngs=None*, *mutable=False*)

Creates an interactive Module instance by binding variables and RNGs.

`bind` provides an "interactive" instance of a Module directly without transforming a function with `apply`. This is particularly useful for debugging and interactive use cases like notebooks where a function would limit the ability to split up code into different cells.

Once the variables (and optionally RNGs) are bound to a `Module` it becomes a stateful object. Note that idiomatic JAX is functional and therefore an interactive instance does not mix well with vanilla JAX APIs.

`bind()` should only be used for interactive experimentation, and in all other cases we strongly encourage users to use `apply()` instead.

Example:

```
import jax
import jax.numpy as jnp
import flax.linen as nn

class AutoEncoder(nn.Module):
  def setup(self):
    self.encoder = nn.Dense(3)
    self.decoder = nn.Dense(5)

  def __call__(self, x):
    return self.decoder(self.encoder(x))

x = jnp.ones((16, 9))
ae = AutoEncoder()
variables = ae.init(jax.random.PRNGKey(0), x)
model = ae.bind(variables)
z = model.encoder(x)
x_reconstructed = model.decoder(z)
```

Parameters

- **variables** – A dictionary containing variables keyed by variable collections. See *flax.core.variables* for more details about variables.
- ***args** – Named arguments (not used).
- **rngs** – a dict of PRNGKeys to initialize the PRNG sequences.
- **mutable** – Can be bool, str, or list. Specifies which collections should be treated as mutable:

  `bool`: all/no collections are mutable. `str`: The name of a single mutable collection. `list`: A list of names of mutable collections.

Returns

A copy of this instance with bound variables and RNGs.

init(*rngs*, **args*, *method=None*, *mutable=DenyList(deny='intermediates')*, *capture_intermediates=False*, ***kwargs*)

Initializes a module method with variables and returns modified variables.

`init` takes as first argument either a single `PRNGKey`, or a dictionary mapping variable collections names to their `PRNGKeys`, and will call `method` (which is the module's `__call__` function by default) passing `*args` and `**kwargs`, and returns a dictionary of initialized variables.

Example:

```
>>> import flax.linen as nn
>>> import jax.numpy as jnp
>>> import jax
...
>>> class Foo(nn.Module):
```

(continues on next page)

(continued from previous page)

```
...   @nn.compact
...   def __call__(self, x, train):
...     x = nn.Dense(16)(x)
...     x = nn.BatchNorm(use_running_average=not train)(x)
...     x = nn.relu(x)
...     return nn.Dense(1)(x)
...
>>> module = Foo()
>>> key = jax.random.PRNGKey(0)
>>> variables = module.init(key, jnp.empty((1, 7)), train=False)
```

If you pass a single `PRNGKey`, Flax will use it to feed the `'params'` RNG stream. If you want to use a different RNG stream or need to use multiple streams, you must pass a dictionary mapping each RNG stream name to its corresponding `PRNGKey` to `init`.

Example:

```
>>> class Foo(nn.Module):
...   @nn.compact
...   def __call__(self, x, train):
...     x = nn.Dense(16)(x)
...     x = nn.BatchNorm(use_running_average=not train)(x)
...     x = nn.relu(x)
...
...     # Add gaussian noise
...     noise_key = self.make_rng('noise')
...     x = x + jax.random.normal(noise_key, x.shape)
...
...     return nn.Dense(1)(x)
...
>>> module = Foo()
>>> rngs = {'params': jax.random.PRNGKey(0), 'noise': jax.random.PRNGKey(1)}
>>> variables = module.init(rngs, jnp.empty((1, 7)), train=False)
```

Jitting *init* initializes a model lazily using only the shapes of the provided arguments, and avoids computing the forward pass with actual values. Example:

```
>>> module = nn.Dense(1)
>>> init_jit = jax.jit(module.init)
>>> variables = init_jit(jax.random.PRNGKey(0), jnp.empty((1, 7)))
```

`init` is a light wrapper over `apply`, so other `apply` arguments like `method`, `mutable`, and `capture_intermediates` are also available.

Parameters

- **rngs** – The rngs for the variable collections.
- ***args** – Named arguments passed to the init function.
- **method** – An optional method. If provided, applies this method. If not provided, applies the `__call__` method. A string can also be provided to specify a method by name.
- **mutable** – Can be bool, str, or list. Specifies which collections should be treated as mutable: `bool`: all/no collections are mutable. `str`: The name of a single mutable collection.

`list`: A list of names of mutable collections. By default all collections except "intermediates" are mutable.

- **capture_intermediates** – If *True*, captures intermediate return values of all Modules inside the "intermediates" collection. By default only the return values of all `__call__` methods are stored. A function can be passed to change the filter behavior. The filter function takes the Module instance and method name and returns a bool indicating whether the output of that method invocation should be stored.
- ****kwargs** – Keyword arguments passed to the init function.

Returns
The initialized variable dict.

init_with_output(*rngs*, **args*, *method=None*, *mutable=DenyList(deny='intermediates')*, *capture_intermediates=False*, ***kwargs*)

Initializes a module method with variables and returns output and modified variables.

Parameters

- **rngs** – The rngs for the variable collections.
- ***args** – Named arguments passed to the init function.
- **method** – An optional method. If provided, applies this method. If not provided, applies the `__call__` method. A string can also be' provided to specify a method by name.
- **mutable** – Can be bool, str, or list. Specifies which collections should be treated as mutable: `bool`: all/no collections are mutable. `str`: The name of a single mutable collection. `list`: A list of names of mutable collections. By default all collections except "intermediates" are mutable.
- **capture_intermediates** – If *True*, captures intermediate return values of all Modules inside the "intermediates" collection. By default only the return values of all `__call__` methods are stored. A function can be passed to change the filter behavior. The filter function takes the Module instance and method name and returns a bool indicating whether the output of that method invocation should be stored.
- ****kwargs** – Keyword arguments passed to the init function.

Returns
*(output, vars)`*, where `vars` are is a dict of the modified collections.

is_initializing()

Returns True if running under self.init(…) or nn.init(…)().

This is a helper method to handle the common case of simple initialization where we wish to have setup logic occur when only called under `module.init` or `nn.init`. For more complicated multi-phase initialization scenarios it is better to test for the mutability of particular variable collections or for the presence of particular variables that potentially need to be initialized.

make_rng(*name*)

Returns a new RNG key from a given RNG sequence for this Module.

The new RNG key is split from the previous one. Thus, every call to *make_rng* returns a new RNG key, while still guaranteeing full reproducibility.

TODO: Link to Flax RNG design note.

Parameters
name – The RNG sequence name.

Returns
The newly generated RNG key.

param(*name*, *init_fn*, **init_args*, *unbox=True*)

Declares and returns a parameter in this Module.

Parameters are read-only variables in the collection named "params". See *flax.core.variables* for more details on variables.

The first argument of *init_fn* is assumed to be a PRNG key, which is provided automatically and does not have to be passed using *init_args*:

```
mean = self.param('mean', lecun_normal(), (2, 2))
```

In the example above, the function *lecun_normal* expects two arguments: *key* and *shape*, but only *shape* has to be provided explicitly; *key* is set automatically using the PRNG for *params* that is passed when initializing the module using *init()*.

Parameters

- **name** – The parameter name.
- **init_fn** – The function that will be called to compute the initial value of this variable. This function will only be called the first time this parameter is used in this module.
- ***init_args** – The arguments to pass to init_fn.
- **unbox** – If True, `AxisMetadata` instances are replaced by their unboxed value, see `flax.nn.meta.unbox` (default: True).

Returns
The value of the initialized parameter. Throws an error if the parameter exists already.

perturb(*name*, *value*, *collection='perturbations'*)

Add an zero-value variable ('perturbation') to the intermediate value.

The gradient of *value* would be the same as the gradient of this perturbation variable. Therefore, if you define your loss function with both params and perturbations as standalone arguments, you can get the intermediate gradients of *value* by running *jax.grad* on the perturbation argument.

Note: this is an experimental API and may be tweaked later for better performance and usability. At its current stage, it creates extra dummy variables that occupies extra memory space. Use it only to debug gradients in training.

Example:

```
import jax
import jax.numpy as jnp
import flax.linen as nn

class Foo(nn.Module):
    @nn.compact
    def __call__(self, x):
        x = nn.Dense(3)(x)
        x = self.perturb('dense3', x)
        return nn.Dense(2)(x)

def loss(params, perturbations, inputs, targets):
  variables = {'params': params, 'perturbations': perturbations}
  preds = model.apply(variables, inputs)
```

(continues on next page)

(continued from previous page)

```
  return jnp.square(preds - targets).mean()

x = jnp.ones((2, 9))
y = jnp.ones((2, 2))
model = Foo()
variables = model.init(jax.random.PRNGKey(0), x)
intm_grads = jax.grad(loss, argnums=1)(variables['params'], variables[
↪'perturbations'], x, y)
print(intm_grads['dense3']) # ==> [[-1.456924   -0.44332537  0.02422847]
                            #      [-1.456924   -0.44332537  0.02422847]]
```

If perturbations are not passed to *apply*, *perturb* behaves like a no-op so you can easily disable the behavior when not needed:

```
model.apply({'params': params, 'perturbations': perturbations}, x) # works as␣
↪expected
model.apply({'params': params}, x) # behaves like a no-op
```

setup()

Initializes a Module lazily (similar to a lazy `__init__`).

`setup` is called once lazily on a module instance when a module is bound, immediately before any other methods like `__call__` are invoked, or before a `setup`-defined attribute on *self* is accessed.

This can happen in three cases:

1. Immediately when invoking *apply()*, *init()* or `init_and_output()`.
2. Once the module is given a name by being assigned to an attribute of another module inside the other module's `setup` method (see *__setattr__()*):

```
class MyModule(nn.Module):
  def setup(self):
    submodule = Conv(...)

    # Accessing `submodule` attributes does not yet work here.

    # The following line invokes `self.__setattr__`, which gives
    # `submodule` the name "conv1".
    self.conv1 = submodule

    # Accessing `submodule` attributes or methods is now safe and
    # either causes setup() to be called once.
```

3. Once a module is constructed inside a method wrapped with *compact()*, immediately before another method is called or `setup` defined attribute is accessed.

sow(*col*, *name*, *value*, *reduce_fn=<function <lambda>>*, *init_fn=<function <lambda>>*)

Stores a value in a collection.

Collections can be used to collect intermediate values without the overhead of explicitly passing a container through each Module call.

If the target collection is not mutable *sow* behaves like a no-op and returns *False*.

Example:

```
import jax
import jax.numpy as jnp
import flax.linen as nn

class Foo(nn.Module):
  @nn.compact
  def __call__(self, x):
    h = nn.Dense(4)(x)
    self.sow('intermediates', 'h', h)
    return nn.Dense(2)(h)

x = jnp.ones((16, 9))
model = Foo()
variables = model.init(jax.random.PRNGKey(0), x)
y, state = model.apply(variables, x, mutable=['intermediates'])
print(state['intermediates'])  # {'h': (...,)}
```

By default the values are stored in a tuple and each stored value is appended at the end. This way all intermediates can be tracked when the same module is called multiple times. Alternatively, a custom init/reduce function can be passed:

```
class Foo2(nn.Module):
  @nn.compact
  def __call__(self, x):
    init_fn = lambda: 0
    reduce_fn = lambda a, b: a + b
    self.sow('intermediates', 'h', x,
             init_fn=init_fn, reduce_fn=reduce_fn)
    self.sow('intermediates', 'h', x * 2,
             init_fn=init_fn, reduce_fn=reduce_fn)
    return x

model = Foo2()
variables = model.init(jax.random.PRNGKey(0), x)
y, state = model.apply(variables, jnp.ones((1, 1)), mutable=['intermediates'])
print(state['intermediates'])  # ==> {'h': [[3.]]}
```

Parameters

- **col** – The name of the variable collection.
- **name** – The name of the variable.
- **value** – The value of the variable.
- **reduce_fn** – The function used to combine the existing value with the new value. The default is to append the value to a tuple.
- **init_fn** – For the first value stored, *reduce_fn* will be passed the result of *init_fn* together with the value to be stored. The default is an empty tuple.

Returns

True if the value has been stored successfully, *False* otherwise.

tabulate(*rngs*, **args*, *depth=None*, *show_repeated=False*, *mutable=True*, *console_kwargs=None*, ***kwargs*)

Creates a summary of the Module represented as a table.

This method has the same signature and internally calls *Module.init*, but instead of returning the variables, it returns the string summarizing the Module in a table. *tabulate* uses *jax.eval_shape* to run the forward computation without consuming any FLOPs or allocating memory.

Additional arguments can be passed into the *console_kwargs* argument, for example, *{'width': 120}*. For a full list of *console_kwargs* arguments, see: https://rich.readthedocs.io/en/stable/reference/console.html#rich.console.Console

Example:

```
import jax
import jax.numpy as jnp
import flax.linen as nn

class Foo(nn.Module):
    @nn.compact
    def __call__(self, x):
        h = nn.Dense(4)(x)
        return nn.Dense(2)(h)

x = jnp.ones((16, 9))

print(Foo().tabulate(jax.random.PRNGKey(0), x))
```

This gives the following output:

```
                                Foo Summary
 path     module  inputs          outputs        params
┏━━━━━━━━━┳━━━━━━━━┳━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┓
┃         ┃ Foo    ┃ float32[16,9] ┃ float32[16,2] ┃                        ┃
┡━━━━━━━━━╇━━━━━━━━╇━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━┩
│ Dense_0 │ Dense  │ float32[16,9] │ float32[16,4] │ bias: float32[4]       │
│         │        │               │               │ kernel: float32[9,4]   │
│         │        │               │               │                        │
│         │        │               │               │ 40 (160 B)             │
├─────────┼────────┼───────────────┼───────────────┼────────────────────────┤
│ Dense_1 │ Dense  │ float32[16,4] │ float32[16,2] │ bias: float32[2]       │
│         │        │               │               │ kernel: float32[4,2]   │
│         │        │               │               │                        │
│         │        │               │               │ 10 (40 B)              │
└─────────┴────────┴───────────────┴───────────────┴────────────────────────┘
                                                 Total 50 (200 B)

                     Total Parameters: 50 (200 B)
```

Note: rows order in the table does not represent execution order, instead it aligns with the order of keys in *variables* which are sorted alphabetically.

Parameters

- **rngs** – The rngs for the variable collections as passed to *Module.init*.

- ***args** – The arguments to the forward computation.
- **depth** – controls how many submodule deep the summary can go. By default its *None* which means no limit. If a submodule is not shown because of the depth limit, its parameter count and bytes will be added to the row of its first shown ancestor such that the sum of all rows always adds up to the total number of parameters of the Module.
- **show_repeated** – If *True*, repeated calls to the same module will be shown in the table, otherwise only the first call will be shown. Default is *False*.
- **mutable** – Can be bool, str, or list. Specifies which collections should be treated as mutable: `bool`: all/no collections are mutable. `str`: The name of a single mutable collection. `list`: A list of names of mutable collections. By default all collections except 'intermediates' are mutable.
- **console_kwargs** – An optional dictionary with additional keyword arguments that are passed to *rich.console.Console* when rendering the table. Default arguments are *{'force_terminal': True, 'force_jupyter': False}*.
- ****kwargs** – keyword arguments to pass to the forward computation.

Returns
A string summarizing the Module.

unbind()

Returns an unbound copy of a Module and its variables.

`unbind` helps create a stateless version of a bound Module.

An example of a common use case: to extract a sub-Module defined inside `setup()` and its corresponding variables: 1) temporarily `bind` the parent Module; and then 2) `unbind` the desired sub-Module. (Recall that `setup()` is only called when the Module is bound.):

```
class AutoEncoder(nn.Module):
  def setup(self):
    self.encoder = Encoder()
    self.decoder = Decoder()

  def __call__(self, x):
    return self.decoder(self.encoder(x))

module = AutoEncoder()
variables = module.init(jax.random.PRNGKey(0), jnp.ones((1, 784)))
...
# Extract the Encoder sub-Module and its variables
encoder, encoder_vars = module.bind(variables).encoder.unbind()
```

Returns
A tuple with an unbound copy of this Module and its variables.

variable(*col*, *name*, *init_fn=None*, **init_args*, *unbox=True*)

Declares and returns a variable in this Module.

See *flax.core.variables* for more information. See also *param()* for a shorthand way to define read-only variables in the "params" collection.

Contrary to *param()*, all arguments passing using *init_fn* should be passed on explicitly:

```
key = self.make_rng('stats')
mean = self.variable('stats', 'mean', lecun_normal(), key, (2, 2))
```

In the example above, the function *lecun_normal* expects two arguments: *key* and *shape*, and both have to be passed on. The PRNG for *stats* has to be provided explicitly when calling *init()* and *apply()*.

Parameters

- **col** – The variable collection name.
- **name** – The variable name.
- **init_fn** – The function that will be called to compute the initial value of this variable. This function will only be called the first time this variable is used in this module. If None, the variable must already be initialized otherwise an error is raised.
- ***init_args** – The arguments to pass to init_fn.
- **unbox** – If True, `AxisMetadata` instances are replaced by their unboxed value, see `flax.nn.meta.unbox` (default: True).

Returns

A *flax.core.variables.Variable* that can be read or set via ".value" attribute. Throws an error if the variable exists already.

property variables

Returns the variables in this module.

Init/Apply

flax.linen.**apply**(*fn*, *module*, *mutable=False*, *capture_intermediates=False*)

Creates an apply function to call `fn` with a bound module.

Unlike `Module.apply` this function returns a new function with the signature `(variables, *args, rngs=None, **kwargs) -> T` where *T* is the return type of `fn`. If `mutable` is not `False` the return type is a tuple where the second item is a `FrozenDict` with the mutated variables.

The apply function that is returned can be directly composed with JAX transformations like `jax.jit`:

```
def f(foo, x):
  z = foo.encode(x)
  y = foo.decode(z)
  # ...
  return y

foo = Foo()
f_jitted = jax.jit(nn.apply(f, foo))
f_jitted(variables, x)
```

Parameters

- **fn** – The function that should be applied. The first argument passed will be an module instance of the `module` with variables and RNGs bound to it.
- **module** – The `Module` that will be used to bind variables and RNGs to. The `Module` passed as the first argument to `fn` will be a clone of module.

- **mutable** – Can be bool, str, or list. Specifies which collections should be treated as mutable: `bool`: all/no collections are mutable. `str`: The name of a single mutable collection. `list`: A list of names of mutable collections.
- **capture_intermediates** – If *True*, captures intermediate return values of all Modules inside the "intermediates" collection. By default only the return values of all *__call__* methods are stored. A function can be passed to change the filter behavior. The filter function takes the Module instance and method name and returns a bool indicating whether the output of that method invocation should be stored.

Returns
The apply function wrapping `fn`.

`flax.linen.`**init**(*fn*, *module*, *mutable=DenyList(deny='intermediates')*, *capture_intermediates=False*)

Creates an init function to call `fn` with a bound module.

Unlike `Module.init` this function returns a new function with the signature `(rngs, *args, **kwargs) -> variables`. The rngs can be a dict of PRNGKeys or a single `PRNGKey which is equivalent to passing a dict with one PRNGKey with the name "params".

The init function that is returned can be directly composed with JAX transformations like `jax.jit`:

```
def f(foo, x):
  z = foo.encode(x)
  y = foo.decode(z)
  # ...
  return y

foo = Foo()
f_jitted = jax.jit(nn.init(f, foo))
variables = f_jitted(rng, x)
```

Parameters

- **fn** – The function that should be applied. The first argument passed will be an module instance of the `module` with variables and RNGs bound to it.
- **module** – The `Module` that will be used to bind variables and RNGs to. The `Module` passed as the first argument to `fn` will be a clone of module.
- **mutable** – Can be bool, str, or list. Specifies which collections should be treated as mutable: `bool`: all/no collections are mutable. `str`: The name of a single mutable collection. `list`: A list of names of mutable collections. By default all collections except "intermediates" are mutable.
- **capture_intermediates** – If *True*, captures intermediate return values of all Modules inside the "intermediates" collection. By default only the return values of all *__call__* methods are stored. A function can be passed to change the filter behavior. The filter function takes the Module instance and method name and returns a bool indicating whether the output of that method invocation should be stored.

Returns
The init function wrapping `fn`.

`flax.linen.`**init_with_output**(*fn*, *module*, *mutable=DenyList(deny='intermediates')*, *capture_intermediates=False*)

Creates an init function to call `fn` with a bound module that also returns the function outputs.

Unlike `Module.init_with_output` this function returns a new function with the signature `(rngs, *args, **kwargs) -> (T, variables)` where *T* is the return type of `fn`. The rngs can be a dict of PRNGKeys or a single `PRNGKey which is equivalent to passing a dict with one PRNGKey with the name "params".

The init function that is returned can be directly composed with JAX transformations like `jax.jit`:

```
def f(foo, x):
  z = foo.encode(x)
  y = foo.decode(z)
  # ...
  return y

foo = Foo()
f_jitted = jax.jit(nn.init_with_output(f, foo))
y, variables = f_jitted(rng, x)
```

Parameters

- **fn** – The function that should be applied. The first argument passed will be an module instance of the `module` with variables and RNGs bound to it.
- **module** – The `Module` that will be used to bind variables and RNGs to. The `Module` passed as the first argument to `fn` will be a clone of module.
- **mutable** – Can be bool, str, or list. Specifies which collections should be treated as mutable: `bool`: all/no collections are mutable. `str`: The name of a single mutable collection. `list`: A list of names of mutable collections. By default all collections except "intermediates" are mutable.
- **capture_intermediates** – If *True*, captures intermediate return values of all Modules inside the "intermediates" collection. By default only the return values of all *__call__* methods are stored. A function can be passed to change the filter behavior. The filter function takes the Module instance and method name and returns a bool indicating whether the output of that method invocation should be stored.

Returns

The init function wrapping `fn`.

Variable dictionary

A variable dict is a normal Python dictionary, which is a container for one or more "variable collections", each of which are nested dictionaries whose leaves are `jax.numpy` arrays.

The different variable collections share the same nested tree structure.

For example, consider the following variable dictionary:

```
{
  "params": {
    "Conv1": { "weight": ..., "bias": ... },
    "BatchNorm1": { "scale": ..., "mean": ... },
    "Conv2": {...}
  },
  "batch_stats": {
    "BatchNorm1": { "moving_mean": ..., "moving_average": ...}
  }
}
```

In this case, the "BatchNorm1" key lives in both the "params" and `"batch_stats"" collections. This reflects the fact that the submodule named ""BatchNorm1"" has both trainable parameters (the "params" collection), as well as other non-trainable variables (the "batch_stats" collection)

TODO: Make "variable dict" design note, and link to it from here.

class flax.core.variables.**Variable**(*scope*, *collection*, *name*, *unbox*)

A Variable object allows mutable access to a variable in a VariableDict.

Variables are identified by a collection (e.g., "batch_stats") and a name (e.g., "moving_mean"). The value property gives access to the variable's content and can be assigned to for mutation.

Compact methods

flax.linen.**compact**(*fun*)

Marks the given module method allowing inlined submodules.

Methods wrapped in @compact can define submodules directly within the method.

For instance:

```
@compact
__call__(self, x, features):
  x = nn.Dense(features)(x)
  ...
```

At most one method in each Module may be wrapped with @compact.

Parameters
fun – The Module method to mark as compact.

Returns
The given function *fun* marked as compact.

No wrap methods

flax.linen.**nowrap**(*fun*)

Marks the given module method as a helper method that needn't be wrapped.

Methods wrapped in @nowrap are private helper methods that needn't be wrapped with the state handler or a separate named_call transform.

This is needed in several concrete instances:

- if you're subclassing a method like Module.param and don't want this overriden core function decorated with the state management wrapper.
- If you want a method to be callable from an unbound Module (e.g.: a function of construction of arguments that doesn't depend on params/RNGs)

For instance:

```
@nowrap
def _make_dense(self, num_features):
  return nn.Dense(num_features)

@compact
```

(continues on next page)

(continued from previous page)

```
def __call__(self, x):
  # now safe to use constructor helper even if using named_call
  dense = self._make_dense(self.num_features)
  return dense(x)
```

Parameters
fun – The Module method to mark as nowrap.

Returns
The given function *fun* marked as nowrap.

Profiling

The Flax Module system.

enable_named_call()	Enables named call wrapping for labelling profile traces.
disable_named_call()	Disables named call wrapping.
override_named_call([enable])	Returns a context manager that enables/disables named call wrapping.

flax.linen.enable_named_call

flax.linen.**enable_named_call**()

Enables named call wrapping for labelling profile traces.

When named call wrapping is enabled all JAX ops executed in a Module will be run under `jax.named_scope`. The `Module` class name will show up around the operations belonging to that Module in the Tensorboard profiling UI, simplifying the profiling process.

Note that `jax.named_scope` only works for compiled functions (e.g.: using jax.jit or jax.pmap).

flax.linen.disable_named_call

flax.linen.**disable_named_call**()

Disables named call wrapping.

See `enable_named_call`

flax.linen.override_named_call

flax.linen.**override_named_call**(*enable=True*)

Returns a context manager that enables/disables named call wrapping.

Parameters
enable – If true, enables named call wrapping for labelling profile traces. (see `enabled_named_call`).

Inspection

The Flax Module system.

tabulate(module, rngs[, depth, ...])	Returns a function that creates a summary of the Module represented as a table.

flax.linen.tabulate

flax.linen.**tabulate**(*module*, *rngs*, *depth=None*, *show_repeated=False*, *mutable=True*, *console_kwargs=None*, ***kwargs*)

Returns a function that creates a summary of the Module represented as a table.

This function accepts most of the same arguments and internally calls *Module.init*, except that it returns a function of the form *(*args, **kwargs) -> str* where **args* and ***kwargs* are passed to *method* (e.g. *__call__*) during the forward pass.

tabulate uses *jax.eval_shape* under the hood to run the forward computation without consuming any FLOPs or allocating memory.

Additional arguments can be passed into the *console_kwargs* argument, for example, *{'width': 120}*. For a full list of *console_kwargs* arguments, see: https://rich.readthedocs.io/en/stable/reference/console.html#rich.console.Console

Example:

```
import jax
import jax.numpy as jnp
import flax.linen as nn

class Foo(nn.Module):
  @nn.compact
  def __call__(self, x):
    h = nn.Dense(4)(x)
    return nn.Dense(2)(h)

x = jnp.ones((16, 9))
tabulate_fn = nn.tabulate(Foo(), jax.random.PRNGKey(0))

print(tabulate_fn(x))
```

This gives the following output:

```
                                Foo Summary

 path     module  inputs          outputs         params

│         │ Foo     │ float32[16,9] │ float32[16,2] │                       │
├─────────┼─────────┼───────────────┼───────────────┼───────────────────────┤
 Dense_0  Dense   float32[16,9]  float32[16,4]  bias: float32[4]
                                                 kernel: float32[9,4]

                                                 40 (160 B)
└─────────┴─────────┴───────────────┴───────────────┴───────────────────────┘
```

(continues on next page)

(continued from previous page)

```
┃ Dense_1 ┃ Dense  ┃ float32[16,4] ┃ float32[16,2] ┃ bias: float32[2]       ┃
┃         ┃        ┃               ┃               ┃ kernel: float32[4,2]  ┃
┃         ┃        ┃               ┃               ┃                       ┃
┃         ┃        ┃               ┃               ┃ 10 (40 B)             ┃
┡━━━━━━━━━╇━━━━━━━━╇━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━┩
┃         ┃        ┃               ┃         Total ┃ 50 (200 B)            ┃
┗━━━━━━━━━┻━━━━━━━━┻━━━━━━━━━━━━━━━┻━━━━━━━━━━━━━━━┻━━━━━━━━━━━━━━━━━━━━━━━┛

                    Total Parameters: 50 (200 B)
```

Note: rows order in the table does not represent execution order, instead it aligns with the order of keys in *variables* which are sorted alphabetically.

Parameters

- **module** – The module to tabulate.
- **rngs** – The rngs for the variable collections as passed to *Module.init*.
- **depth** – controls how many submodule deep the summary can go. By default its *None* which means no limit. If a submodule is not shown because of the depth limit, its parameter count and bytes will be added to the row of its first shown ancestor such that the sum of all rows always adds up to the total number of parameters of the Module.
- **mutable** – Can be bool, str, or list. Specifies which collections should be treated as mutable: `bool`: all/no collections are mutable. `str`: The name of a single mutable collection. `list`: A list of names of mutable collections. By default all collections except 'intermediates' are mutable.
- **show_repeated** – If *True*, repeated calls to the same module will be shown in the table, otherwise only the first call will be shown. Default is *False*.
- **console_kwargs** – An optional dictionary with additional keyword arguments that are passed to *rich.console.Console* when rendering the table. Default arguments are *{'force_terminal': True, 'force_jupyter': False}*.
- ****kwargs** – Additional arguments passed to *Module.init*.

Returns

A function that accepts the same **args* and ***kwargs* of the forward pass (*method*) and returns a string with a tabular representation of the Modules.

Transformations

JAX transformations on Modules.

Jax functional transformations operate on pure functions. Flax extends these transformations to also operate on Module's which have stateful variables and PRNG sequences. We refer to these extended versions as "lifted transformations".

A lifted transformation can be applied to a `Module` class or a function that takes a `Module` instance as its first argument.

vmap(target[, variable_axes, split_rngs, ...])	A lifted version of `jax.vmap`.
scan(target[, variable_axes, ...])	A lifted version of `jax.lax.scan`.
jit(target[, variables, rngs, ...])	Lifted version of `jax.jit`.
remat(target[, variables, rngs, concrete, ...])	Lifted version of `jax.checkpoint`.
remat_scan(target[, lengths, policy, ...])	Combines remat and scan for memory efficiency and constant time compilation.
map_variables(target[, mapped_collections, ...])	Map Variables inside a module.
jvp(fn, mdl, primals, tangents, ...[, ...])	A lifted version of `jax.jvp`.
vjp(fn, mdl, *primals[, has_aux, ...])	A lifted version of `jax.vjp`.
custom_vjp(fn, forward_fn, backward_fn[, ...])	Lifted version of *jax.custom_vjp*.
while_loop(cond_fn, body_fn, mdl, init[, ...])	Lifted version of jax.lax.while_loop.
cond(pred, true_fun, false_fun, mdl, *operands)	Lifted version of `jax.lax.cond`.
switch(index, branches, mdl, *operands[, ...])	Lifted version of `jax.lax.switch`.

flax.linen.vmap

flax.linen.**vmap**(*target*, *variable_axes=FrozenDict({})*, *split_rngs=FrozenDict({})*, *in_axes=0*, *out_axes=0*, *axis_size=None*, *axis_name=None*, *spmd_axis_name=None*, *metadata_params={}*, *methods=None*)

A lifted version of `jax.vmap`.

See `jax.vmap` for the unlifted batch transform in Jax.

`vmap` can be used to add a batch axis to a `Module`. For example we could create a version of `Dense` with a batch axis that does not share parameters:

```
BatchDense = nn.vmap(
    nn.Dense,
    in_axes=0, out_axes=0,
    variable_axes={'params': 0},
    split_rngs={'params': True})
```

By using `variable_axes={'params': 0}`, we indicate that the parameters themselves are mapped over and therefore not shared along the mapped axis. Consequently, we also split the 'params' RNG, otherwise the parameters would be initialized identically along the mapped axis.

Similarly, `vmap` could be use to add a batch axis with parameter sharing:

```
BatchFoo = nn.vmap(
    Foo,
    in_axes=0, out_axes=0,
    variable_axes={'params': None},
    split_rngs={'params': False})
```

Here we use `variable_axes={'params': None}` to indicate the parameter variables are shared along the mapped axis. Consequently, the 'params' RNG must also be shared.

Parameters

- **target** – a `Module` or a function taking a `Module` as its first argument.
- **variable_axes** – the variable collections that are lifted into the batching transformation. Use *None* to indicate a broadcasted collection or an integer to map over an axis.

- **split_rngs** – Split PRNG sequences will be different for each index of the batch dimension. Unsplit PRNGs will be broadcasted.
- **in_axes** – Specifies the mapping of the input arguments (see *jax.vmap*).
- **out_axes** – Specifies the mapping of the return value (see *jax.vmap*).
- **axis_size** – Specifies the size of the batch axis. This only needs to be specified if it cannot be derived from the input arguments.
- **axis_name** – Specifies a name for the batch axis. Can be used together with parallel reduction primitives (e.g. *jax.lax.pmean*, *jax.lax.ppermute*, etc.)
- **methods** – If *target* is a *Module*, the methods of *Module* to vmap over.
- **spmd_axis_name** – Axis name added to any pjit sharding constraints appearing in *fn*. See also https://github.com/google/flax/blob/main/flax/linen/partitioning.py.
- **metadata_params** – arguments dict passed to AxisMetadata instances in the variable tree.

Returns

A batched/vectorized version of `target`, with the same arguments but with extra axes at positions indicated by `in_axes`, and the same return value, but with extra axes at positions indicated by `out_axes`.

flax.linen.scan

`flax.linen.scan`(*target*, *variable_axes=FrozenDict({})*, *variable_broadcast=False*, *variable_carry=False*, *split_rngs=FrozenDict({})*, *in_axes=0*, *out_axes=0*, *length=None*, *reverse=False*, *unroll=1*, *data_transform=None*, *metadata_params={}*, *methods=None*)

A lifted version of `jax.lax.scan`.

See `jax.lax.scan` for the unlifted scan in Jax.

To improve consistency with `vmap`, this version of scan uses `in_axes` and `out_axes` to determine which arguments are scanned over and along which axis.

`scan` distinguishes between 3 different types of values inside the loop:

1. **scan**: a value that is iterated over in a loop. All scan values must have the same size in the axis they are scanned over. Scanned outputs will be stacked along the scan axis.
2. **carry**: A carried value is updated at each loop iteration. It must have the same shape and dtype throughout the loop.
3. **broadcast**: a value that is closed over by the loop. When a variable is broadcasted they are typically initialized inside the loop body but independent of the loop variables.

The loop body should have the signature `(scope, body, carry, *xs) -> (carry, ys)`, where `xs` and `ys` are the scan values that go in and out of the loop.

Example:

```
>>> import flax.linen as nn
>>> import jax
>>> import jax.numpy as jnp
...
>>> class LSTM(nn.Module):
...   features: int
...
```

(continues on next page)

(continued from previous page)

```
...   @nn.compact
...   def __call__(self, x):
...     batch_size = x.shape[0]
...     ScanLSTMCell = nn.scan(
...       nn.LSTMCell, variable_broadcast="params",
...       split_rngs={"params": False}, in_axes=1, out_axes=1)
...
...     carry = nn.LSTMCell.initialize_carry(
...       jax.random.PRNGKey(0), (batch_size,), self.features)
...     carry, x = ScanLSTMCell()(carry, x)
...     return x
...
>>> x = jnp.ones((4, 12, 7))
>>> module = LSTM(features=32)
>>> y, variables = module.init_with_output(jax.random.PRNGKey(0), x)
```

Note that when providing a function to `nn.scan`, the scanning happens over all arguments starting from the third argument, as specified by `in_axes`. The previous example could also be written using the functional form as:

```
>>> class LSTM(nn.Module):
...   features: int
...
...   @nn.compact
...   def __call__(self, x):
...     batch_size = x.shape[0]
...
...     cell = nn.LSTMCell()
...     def body_fn(cell, carry, x):
...       carry, y = cell(carry, x)
...       return carry, y
...     scan = nn.scan(
...       body_fn, variable_broadcast="params",
...       split_rngs={"params": False}, in_axes=1, out_axes=1)
...
...     carry = nn.LSTMCell.initialize_carry(
...       jax.random.PRNGKey(0), (batch_size,), self.features)
...     carry, x = scan(cell, carry, x)
...     return x
...
>>> module = LSTM(features=32)
>>> variables = module.init(jax.random.PRNGKey(0), jnp.ones((4, 12, 7)))
```

You can also use `scan` to reduce the compilation time of your JAX program by merging multiple layers into a single scan loop, you can do this when you have a sequence of identical layers that you want to apply iteratively to an input. For example:

```
>>> class ResidualMLPBlock(nn.Module):
...   @nn.compact
...   def __call__(self, x, _):
...     h = nn.Dense(features=2)(x)
...     h = nn.relu(h)
...     return x + h, None
```

(continues on next page)

(continued from previous page)

```
...
>>> class ResidualMLP(nn.Module):
...   n_layers: int = 4
...
...   @nn.compact
...   def __call__(self, x):
...     ScanMLP = nn.scan(
...       ResidualMLPBlock, variable_axes={'params': 0},
...       variable_broadcast=False, split_rngs={'params': True},
...       length=self.n_layers)
...     x, _ = ScanMLP()(x, None)
...     return x
...
>>> model = ResidualMLP(n_layers=4)
>>> variables = model.init(jax.random.PRNGKey(42), jnp.ones((1, 2)))
```

To reduce both compilation and memory usage, you can use *remat_scan()* which will in addition checkpoint each layer in the scan loop.

Parameters

- **target** – a `Module` or a function taking a `Module` as its first argument.
- **variable_axes** – the variable collections that are scanned over.
- **variable_broadcast** – Specifies the broadcasted variable collections. A broadcasted variable should not depend on any computation that cannot be lifted out of the loop. This is typically used to define shared parameters inside the fn.
- **variable_carry** – Specifies the variable collections that are carried through the loop. Mutations to these variables are carried to the next iteration and will be preserved when the scan finishes.
- **split_rngs** – Split PRNG sequences will be different for each loop iterations. If split is False the PRNGs will be the same across iterations.
- **in_axes** – Specifies the axis to scan over for the arguments. Should be a prefix tree of the arguments. Use *flax.core.broadcast* to feed an entire input to each iteration of the scan body.
- **out_axes** – Specifies the axis to scan over for the return value. Should be a prefix tree of the return value.
- **length** – Specifies the number of loop iterations. This only needs to be specified if it cannot be derived from the scan arguments.
- **reverse** – If true, scan from end to start in reverse order.
- **unroll** – how many scan iterations to unroll within a single iteration of a loop (default: 1).
- **data_transform** – optional function to transform raw functional-core variable and rng groups inside lifted scan body_fn, intended for inline SPMD annotations.
- **metadata_params** – arguments dict passed to AxisMetadata instances in the variable tree.
- **methods** – If *target* is a *Module*, the methods of *Module* to scan over.

Returns

The scan function with the signature `(scope, carry, *xxs) -> (carry, yys)`, where `xxs` and `yys` are the scan values that go in and out of the loop.

flax.linen.jit

flax.linen.**jit**(*target*, *variables=True*, *rngs=True*, *static_argnums=()*, *donate_argnums=()*, *device=None*, *backend=None*, *methods=None*)

Lifted version of `jax.jit`.

Parameters

- **target** – a `Module` or a function taking a `Module` as its first argument.
- **variables** – The variable collections that are lifted. By default all collections are lifted.
- **rngs** – The PRNG sequences that are lifted. By default all PRNG sequences are lifted.
- **static_argnums** – An int or collection of ints specifying which positional arguments to treat as static (compile-time constant). Operations that only depend on static arguments will be constant-folded in Python (during tracing), and so the corresponding argument values can be any Python object. Static arguments should be hashable, meaning both `__hash__` and `__eq__` are implemented, and immutable. Calling the jitted function with different values for these constants will trigger recompilation. If the jitted function is called with fewer positional arguments than indicated by `static_argnums` then an error is raised. Arguments that are not arrays or containers thereof must be marked as static. Defaults to ().
- **donate_argnums** – Specify which arguments are "donated" to the computation. It is safe to donate arguments if you no longer need them once the computation has finished. In some cases XLA can make use of donated buffers to reduce the amount of memory needed to perform a computation, for example recycling one of your input buffers to store a result. You should not reuse buffers that you donate to a computation, JAX will raise an error if you try to.
- **device** – This is an experimental feature and the API is likely to change. Optional, the Device the jitted function will run on. (Available devices can be retrieved via `jax.devices()`.) The default is inherited from XLA's DeviceAssignment logic and is usually to use `jax.devices()[0]`.
- **backend** – a string representing the XLA backend: `'cpu'`, `'gpu'`, or `'tpu'`.
- **methods** – If *target* is a *Module*, the methods of *Module* to jit.

Returns

A wrapped version of target, set up for just-in-time compilation.

flax.linen.remat

flax.linen.**remat**(*target*, *variables=True*, *rngs=True*, *concrete=False*, *prevent_cse=True*, *static_argnums=()*, *policy=None*, *methods=None*)

Lifted version of `jax.checkpoint`.

Checkpointing is a technique for reducing memory usage by recomputing activations during backpropagation. When training large models, it can be helpful to checkpoint parts of the model to trade off memory usage for additional computation.

Example:

```
>>> import jax
>>> import jax.numpy as jnp
>>> import flax.linen as nn
```

(continues on next page)

(continued from previous page)

```
...
>>> class CheckpointedMLP(nn.Module):
...   @nn.compact
...   def __call__(self, x):
...     CheckpointDense = nn.checkpoint(nn.Dense)
...     x = CheckpointDense(128)(x)
...     x = nn.relu(x)
...     x = CheckpointDense(1)(x)
...     return x
...
>>> model = CheckpointedMLP()
>>> variables = model.init(jax.random.PRNGKey(0), jnp.ones((1, 16)))
```

This function is aliased to `remat` just like `jax.remat`.

Parameters

- **target** – a `Module` or a function taking a `Module` as its first argument. intermediate computations will be re-computed when computing gradients for the target.
- **variables** – The variable collections that are lifted. By default all collections are lifted.
- **rngs** – The PRNG sequences that are lifted. By default all PRNG sequences are lifted.
- **concrete** – Optional, boolean indicating whether `fun` may involve value-dependent Python control flow (default False). Support for such control flow is optional, and disabled by default, because in some edge-case compositions with `jax.jit()` it can lead to some extra computation.
- **prevent_cse** – Optional, boolean indicating whether to prevent common subexpression elimination (CSE) optimizations in the HLO generated from differentiation. This CSE prevention has costs because it can foil other optimizations, and because it can incur high overheads on some backends, especially GPU. The default is True because otherwise, under a `jit` or `pmap`, CSE can defeat the purpose of this decorator. But in some settings, like when used inside a `scan`, this CSE prevention mechanism is unnecessary, in which case `prevent_cse` should be set to False.
- **static_argnums** – Optional, int or sequence of ints, indicates which argument values on which to specialize for tracing and caching purposes. Specifying arguments as static can avoid ConcretizationTypeErrors when tracing, but at the cost of more retracing overheads.
- **policy** – Experimental checkpoint policy, see `jax.checkpoint`.
- **methods** – An optional list of method names that will be lifted, if *methods* is None (default) only the *__call__* method will be lifted. If *target* is a function, *methods* is ignored.

Returns

A wrapped version of `target`. When computing gradients intermediate computations will be re-computed on the backward pass.

flax.linen.remat_scan

flax.linen.**remat_scan**(*target*, *lengths=()*, *policy=None*, *variable_broadcast=False*, *variable_carry=False*, *variable_axes=FrozenDict({True: 0})*, *split_rngs=FrozenDict({True: True})*)

Combines remat and scan for memory efficiency and constant time compilation.

`remat_scan` allows for constant compile times and sublinear memory usage with respect to model depth. At a small constant penalty. This is typically beneficial for very deep models.

Example:

```
class BigModel(nn.Module):
  @nn.compact
  def __call__(self, x):
    DenseStack = nn.remat_scan(nn.Dense, lengths=(10, 10))
    # 100x dense with O(sqrt(N)) memory for gradient computation
    return DenseStack(8, name="dense_stack")(x)
```

Parameters

- **target** – a `Module` or a function taking a `Module` as its first argument.
- **lengths** – number of loop iterations at the given level. The total number of iterations *n = prod(lengths)*. each loop is rematerialized. This way the memory consumption is proportional to *n^(1 / d)* where *d = len(lengths)*. Minimal memory consumptions requires tuning the lengths such that the same amount of memory is consumed at each level of the nested loop.
- **policy** – Experimental checkpoint policy, see `jax.checkpoint`.
- **variable_broadcast** – Specifies the broadcasted variable collections. A broadcasted variable should not depend on any computation that cannot be lifted out of the loop. This is typically used to define shared parameters inside the fn.
- **variable_carry** – Specifies the variable collections that are carried through the loop. Mutations to these variables are carried to the next iteration and will be preserved when the scan finishes.
- **variable_axes** – the variable collections that are scanned over. Defaults to `{True: 0}`.
- **split_rngs** – Split PRNG sequences will be different for each loop iterations. If split is False the PRNGs will be the same across iterations. Defaults to `{True: True}`.

Returns

A wrapped version of `target` that repeats itself prod(lengths) times.

flax.linen.map_variables

flax.linen.**map_variables**(*target*, *mapped_collections=True*, *trans_in_fn=<function <lambda>>*, *trans_out_fn=<function <lambda>>*, *init=False*, *mutable=False*, *rngs=True*, *variables=True*, *methods=None*)

Map Variables inside a module.

`map_variables` can be used to transform the variables inside a module both before and after the module is applied. This is useful among other things for masking the weights of a module without having to modify the module itself.

Example::

```
>>> import jax
>>> import jax.numpy as jnp
>>> import flax.linen as nn
...
>>> class CausalDense(nn.Module):
...   '''A dense layer that masks the weights such that the output is
...   causal, i.e. output i only depends on input <= i.
...   '''
...   features: int
...
...   def apply_mask(self, variables):
...     return (jax.tree_map(jnp.triu, variables)
...             if not self.is_initializing() else variables)
...
...   def setup(self):
...     # temporary class
...     _CausalDense = nn.map_variables(
...       nn.Dense, 'params', self.apply_mask, init=self.is_initializing())
...     self.dense = _CausalDense(features=self.features, use_bias=False)
...
...   def __call__(self, x):
...     return self.dense(x)
...
>>> module = CausalDense(features=5)
>>> variables = module.init(jax.random.PRNGKey(0), jnp.ones((1, 5)))
```

Parameters

- **target** – the module or function to be transformed.
- **mapped_collections** – the collection(s) to be transformed.
- **trans_in_fn** – modifies the variables before applying the module or function.
- **trans_out_fn** – modifies the variables after applying the module or function, it is only applied if either `init` or `mutable` are not False.
- **init** – If True, variables are initialized before transformation.
- **mutable** – If True, the mapped variable collections will be mutable.
- **rngs** – PRNGSequences added to the transformed scope (default: all).
- **variables** – Additional Variable collections added to the transformed scope. Besides those specified by *target* (default: all).
- **methods** – If *target* is a *Module*, the methods of *Module* to map variables for.

Returns

a wrapped version of `target` that will map the specified collections.

flax.linen.jvp

flax.linen.**jvp**(*fn*, *mdl*, *primals*, *tangents*, *variable_tangents*, *variables=True*, *rngs=True*)

A lifted version of `jax.jvp`.

See `jax.jvp` for the unlifted Jacobian-vector product (forward gradient).

Note that no tangents are returned for variables. When variable tangents are required their value should be returned explicitly by *fn* using *Module.variables*:

```
class LearnScale(nn.Module):
  @nn.compact
  def __call__(self, x):
    p = self.param('test', nn.initializers._init(), ())
    return p * x

class Foo(nn.Module):
  @nn.compact
  def __call__(self, x):
    scale = LearnScale()
    vars_t = jax.tree_util.tree_map(jnp.ones_like,
                                    scale.variables.get('params', {}))
    _, out_t = nn.jvp(
        lambda mdl, x: mdl(x), scale, (x,), (jnp.zeros_like(x),),
        variable_tangents={'params': vars_t})
    return out_t
```

Example:

```
def learn_scale(scope, x):
  p = scope.param('scale', nn.initializers.zeros_init(), ())
  return p * x

def f(scope, x):
  vars_t = jax.tree_util.tree_map(jnp.ones_like, scope.variables().get('params', {}
↪))
  x, out_t = lift.jvp(
      learn_scale, scope, (x,), (jnp.zeros_like(x),),
      variable_tangents={'params': vars_t})
  return out_t
```

Parameters

- **fn** – Function to be differentiated. Its arguments should be arrays, scalars, or standard Python containers of arrays or scalars. It should return an array, scalar, or standard Python container of arrays or scalars. It will receive the scope and primals as arguments.
- **mdl** – The module of which the variables will be differentiated.
- **primals** – The primal values at which the Jacobian of `fun` should be evaluated. Should be either a tuple or a list of arguments, and its length should be equal to the number of positional parameters of `fun`.
- **tangents** – The tangent vector for which the Jacobian-vector product should be evaluated. Should be either a tuple or a list of tangents, with the same tree structure and array shapes as `primals`.

- **variable_tangents** – A dict or PyTree fo dicts with the same structure as scopes. Each entry in the dict specifies the tangents for a variable collection. Not specifying a collection in variable_tangents is equivalent to passing a zero vector as the tangent.
- **variables** – other variables collections that are available in *fn* but do not receive a tangent.
- **rngs** – the prngs that are available inside *fn*.

Returns

A `(primals_out, tangents_out)` pair, where `primals_out` is `fun(*primals)`, and `tangents_out` is the Jacobian-vector product of `function` evaluated at `primals` with `tangents`. The `tangents_out` value has the same Python tree structure and shapes as `primals_out`.

flax.linen.vjp

`flax.linen.vjp`(*fn*, *mdl*, **primals*, *has_aux=False*, *reduce_axes=()*, *vjp_variables='params'*, *variables=True*, *rngs=True*)

A lifted version of `jax.vjp`.

See `jax.vjp` for the unlifted vector-Jacobiam product (backward gradient).

Note that a gradient is returned for all variables in the collections specified by *vjp_variables*. However, the backward funtion only expects a cotangent for the return value of *fn*. If variables require a co-tangent as well they can be returned from *fn* using *Module.variables*.

Example:

```
class LearnScale(nn.Module):
  @nn.compact
  def __call__(self, x, y):
    p = self.param('scale', nn.initializers.zeros_init(), ())
    return p * x * y

class Foo(nn.Module):
  @nn.compact
  def __call__(self, x, y):
    z, bwd = nn.vjp(lambda mdl, x, y: mdl(x, y), LearnScale(), x, y)
    params_grad, x_grad, y_grad = bwd(jnp.ones(z.shape))
    return z, params_grad, x_grad, y_grad
```

Parameters

- **fn** – Function to be differentiated. Its arguments should be arrays, scalars, or standard Python containers of arrays or scalars. It should return an array, scalar, or standard Python container of arrays or scalars. It will receive the scope and primals as arguments.
- **mdl** – The module of which the variables will be differentiated.
- ***primals** – A sequence of primal values at which the Jacobian of `fn` should be evaluated. The length of `primals` should be equal to the number of positional parameters to `fn`. Each primal value should be a tuple of arrays, scalar, or standard Python containers thereof.
- **has_aux** – Optional, bool. Indicates whether `fn` returns a pair where the first element is considered the output of the mathematical function to be differentiated and the second element is auxiliary data. Default False.

- **reduce_axes** – Optional, tuple of axis names. If an axis is listed here, and `fn` implicitly broadcasts a value over that axis, the backward pass will perform a `psum` of the corresponding gradient. Otherwise, the VJP will be per-example over named axes. For example, if `'batch'` is a named batch axis, `vjp(f, *args, reduce_axes=('batch',))` will create a VJP function that sums over the batch while `vjp(f, *args)` will create a per-example VJP.
- **vjp_variables** – The vjpfun will return a cotangent vector for all variable collections specified by this filter.
- **variables** – other variables collections that are available inside *fn* but do not receive a cotangent.
- **rngs** – the prngs that are available inside *fn*.

Returns

If `has_aux` is `False`, returns a `(primals_out, vjpfun)` pair, where `primals_out` is `fn(*primals)`. `vjpfun` is a function from a cotangent vector with the same shape as `primals_out` to a tuple of cotangent vectors with the same shape as `primals`, representing the vector-Jacobian product of `fn` evaluated at `primals`. If `has_aux` is `True`, returns a `(primals_out, vjpfun, aux)` tuple where `aux` is the auxiliary data returned by `fn`.

flax.linen.custom_vjp

`flax.linen.`**custom_vjp**(*fn*, *forward_fn*, *backward_fn*, *grad_vars='params'*, *nondiff_argnums=()*)

Lifted version of *jax.custom_vjp*.

forward_fn and *backward_fn* together define a custom vjp for *fn*. The original *fn* will run in case a vjp (backward gradient) is not computed.

The *forward_fn* receives the same arguments as *fn* but is expected to return a tuple containing the output of *fn(mdl, *args)* and the residuals that are passed to *backward_fn*.

The *backward_fn* receives the nondiff arguments, residuals, and the output tangents. It should return a tuple containing the variable and input tangents.

Note that the vjp function returned by *nn.vjp* can be passed as residual and used in the *backward_fn*. The scope is unavailable during the backward pass. If the module is required in *backward_fn*, a snapshot of the variables can be taken and returned as a residual in the *forward_fn*.

Example:

```
class Foo(nn.Module):
  @nn.compact
  def __call__(self, x):
    def f(mdl, x):
      return mdl(x)

    def fwd(mdl, x):
      return nn.vjp(f, mdl, x)

    def bwd(vjp_fn, y_t):
      params_t, *inputs_t = vjp_fn(y_t)
      params_t = jax.tree_util.tree_map(jnp.sign, params_t)
      return (params_t, *inputs_t)

    sign_grad = nn.custom_vjp(
        f, forward_fn=fwd, backward_fn=bwd)
```

(continues on next page)

(continued from previous page)

```
    return sign_grad(nn.Dense(1), x).reshape(())

x = jnp.ones((2,))
variables = Foo().init(random.PRNGKey(0), x)
grad = jax.grad(Foo().apply)(variables, x)
```

Parameters

- **fn** – The function to define a custom_vjp for.
- **forward_fn** – A function with the same arguments as `fn` returning an tuple with the original output and the residuals that will be passsed to `backward_fn`.
- **backward_fn** – arguments are passed as `(*nondiff_args, residuals, tangents)` The function should return a tuple containing the tangents for the variable in the collections specified by *grad_vars* and the input arguments (except the module and nondiff args).
- **grad_vars** – The collections for which a vjp will be computed (default: "params").
- **nondiff_argnums** – arguments for which no vjp is computed.

Returns

A function with the same signature as *fn* with the custom vjp.

flax.linen.while_loop

flax.linen.**while_loop**(*cond_fn*, *body_fn*, *mdl*, *init*, *carry_variables=False*, *broadcast_variables=True*, *split_rngs=FrozenDict({})*)

Lifted version of jax.lax.while_loop.

The lifted scope is passed to *cond_fn* and *body_fn*. Broadcasted variables are immutable. The carry variable are mutable but cannot change shape and dtype. This also means you cannot initialize variables inside the body. Consider calling *body_fn* once manually before calling *while_loop* if variable initialization is required.

Example:

```
class WhileLoopExample(nn.Module):
  @nn.compact
  def __call__(self, x):
    def cond_fn(mdl, c):
      return mdl.variables['state']['acc'] < 10
    def body_fn(mdl, c):
      acc = mdl.variable('state', 'acc', lambda: jnp.array(0))
      acc.value += 1
      y = nn.Dense(c.shape[-1])(c)
      return y
    c = x
    if self.is_mutable_collection('params'):
      return body_fn(self, c)
    else:
      return nn.while_loop(cond_fn, body_fn, self, c,
                           carry_variables='state')

k = random.PRNGKey(0)
```

(continues on next page)

(continued from previous page)

```
x = jnp.ones((2, 2))
intial_vars = WhileLoopExample().init(k, x)
result, state = WhileLoopExample().apply(intial_vars, x, mutable=['state'])
```

Parameters

- **cond_fn** – Should return True as long as the loop should continue.
- **body_fn** – The body of the while loop.
- **mdl** – The Module which should be lifted into the loop.
- **init** – The initial state passed to the loop
- **carry_variables** – collections that are carried through the loop and are therefore mutable (default: none).
- **broadcast_variables** – collections that are closed over and are therefore read-only (default: all collections)
- **split_rngs** – Split PRNG sequences will be different for each loop iterations. If split is False the PRNGs will be the same across iterations.

Returns

The final state after executing the while loop.

flax.linen.cond

`flax.linen.`**cond**(*pred*, *true_fun*, *false_fun*, *mdl*, **operands*, *variables=True*, *rngs=True*)

Lifted version of `jax.lax.cond`.

The returned values from `true_fun` and `false_fun` must have the same Pytree structure, shapes, and dtypes. The variables created or updated inside the branches must also have the same structure. Note that this constraint is violated when creating variables or submodules in only one branch. Because initializing variables in just one branch causes the parameter structure to be different.

Example:

```
class CondExample(nn.Module):
  @nn.compact
  def __call__(self, x, pred):
    self.variable('state', 'true_count', lambda: 0)
    self.variable('state', 'false_count', lambda: 0)
    def true_fn(mdl, x):
      mdl.variable('state', 'true_count').value += 1
      return nn.Dense(2, name='dense')(x)
    def false_fn(mdl, x):
      mdl.variable('state', 'false_count').value += 1
      return -nn.Dense(2, name='dense')(x)
    return nn.cond(pred, true_fn, false_fn, self, x)
```

Parameters

- **pred** – determines if true_fun or false_fun is evaluated.
- **true_fun** – The function evalauted when `pred` is *True*. The signature is (module, *operands) -> T.

- **false_fun** – The function evalauted when `pred` is *False*. The signature is (module, *operands) -> T.
- **mdl** – A Module target to pass.
- ***operands** – The arguments passed to `true_fun` and `false_fun`
- **variables** – The variable collections passed to the conditional branches (default: all)
- **rngs** – The PRNG sequences passed to the conditionals (default: all)

Returns

The result of the evaluated branch (`true_fun` or `false_fun`).

flax.linen.switch

`flax.linen.`**switch**(*index*, *branches*, *mdl*, **operands*, *variables=True*, *rngs=True*)

Lifted version of `jax.lax.switch`.

The returned values from `branches` must have the same Pytree structure, shapes, and dtypes. The variables created or updated inside the branches must also have the same structure. Note that this constraint is violated when creating variables or submodules in only one branch. Because initializing variables in just one branch causes the parameter structure to be different.

Example:

```
class SwitchExample(nn.Module):
  @nn.compact
  def __call__(self, x, index):
    self.variable('state', 'a_count', lambda: 0)
    self.variable('state', 'b_count', lambda: 0)
    self.variable('state', 'c_count', lambda: 0)
    def a_fn(mdl, x):
      mdl.variable('state', 'a_count').value += 1
      return nn.Dense(2, name='dense')(x)
    def b_fn(mdl, x):
      mdl.variable('state', 'b_count').value += 1
      return -nn.Dense(2, name='dense')(x)
    def c_fn(mdl, x):
      mdl.variable('state', 'c_count').value += 1
      return nn.Dense(2, name='dense')(x)
    return nn.switch(index, [a_fn, b_fn, c_fn], self, x)
```

If you want to have a different parameter structure for each branch you should run all branches on initialization before calling switch:

```
class MultiHeadSwitchExample(nn.Module):
  def setup(self) -> None:
    self.heads = [
      nn.Sequential([nn.Dense(10), nn.Dense(7), nn.Dense(5)]),
      nn.Sequential([nn.Dense(11), nn.Dense(5)]),
      nn.Dense(5),
    ]

  @nn.compact
  def __call__(self, x, index):
```

(continues on next page)

(continued from previous page)

```
    def head_fn(i):
      return lambda mdl, x: mdl.heads[i](x)
    branches = [head_fn(i) for i in range(len(self.heads))]

    # run all branches on init
    if self.is_mutable_collection('params'):
      for branch in branches:
        _ = branch(self, x)

    return nn.switch(index, branches, self, x)
```

Parameters

- **index** – Integer scalar type, indicating which branch function to apply.
- **branches** – Sequence of functions to be applied based on index. The signature of each function is (module, *operands) -> T.
- **mdl** – A Module target to pass.
- ***operands** – The arguments passed to the branches.
- **variables** – The variable collections passed to the conditional branches (default: all)
- **rngs** – The PRNG sequences passed to the conditionals (default: all)

Returns

The result of the evaluated branch.

SPMD

Utilities for working with pjit and partitioned models.

This module introduces *axis_rules*, *logical_to_mesh_axes*, *logical_to_mesh*, *with_logical_constraint* for appyling pjit sharding constraints in terms of "logical named axes" rather than pjit's default mesh axes.

Additionally the *LogicallyPartitioned* metadata wrapper is defined as well as the initializer function wrapper *with_logical_partitioning* for introducing logical axis metadata into a model's variables.

Partitioned(value, names[, mesh])	Wrapper for partitioning metadata.
with_partitioning(fn, names[, mesh])	Wraps a function's return value with Partitioned.
get_partition_spec(tree)	Extracts a PartitionSpec tree from a PyTree containing `Partitioned` values.
get_sharding(tree, mesh)	Extracts a jax.sharding tree from a PyTree containing `Partitioned` values and a mesh.
LogicallyPartitioned(value, names[, mesh, rules])	
logical_axis_rules(rules)	Context manager for setting the logical to mesh axis bindings.
set_logical_axis_rules(rules)	Sets the global logical axis to mesh axis binding.
get_logical_axis_rules()	Returns the global logical axis to mesh axis binding.
logical_to_mesh_axes(array_dim_names[, rules])	Compute layout for an array.
logical_to_mesh(tree[, rules])	Applies logical_to_mesh_axes to pytrees of logical PartitionSpecs.
logical_to_mesh_sharding(tree, mesh[, rules])	Convert pytrees of logical PartitionSpecs to shardings.
with_logical_constraint(x, ...[, rules, ...])	Version of pjit's with_sharding_constraint that uses logical axis names.
with_logical_partitioning(fn, names[, mesh, ...])	Wraps a function's return value with LogicallyPartitioned.

flax.linen.Partitioned

class flax.linen.**Partitioned**(*value*, *names*, *mesh=None*)

Wrapper for partitioning metadata.

`Partitioned` is used to extend variables with partitioning information required for `jax.experimental.pjit`.

The easiest way to define Partitioned variables is by using the `with_partitioning` wrapper around the variable initializer.

Example:

```
class MLP(nn.Module):
  hidden_size: int
  @nn.compact
  def __call__(self, x):
    ki = nn.linear.default_kernel_init
    h = nn.Dense(
        self.hidden_size,
        kernel_init=nn.with_partitioning(ki, ('data', 'model')))(x)
    h = nn.relu(h)
    return nn.Dense(
        x.shape[-1],
        kernel_init=nn.with_partitioning(ki, ('model', 'data')))(h)
mlp = MLP(4096)
x = jnp.ones((8 * 1024, 1024))
# use eval_shape to get the Partitioned instances for the variables.
# this way we can determinte the PartitionSpecs for the init variables
# before we call the init fn.
var_spec = nn.get_partition_spec(
    jax.eval_shape(mlp.init, random.PRNGKey(0), x))
```

(continues on next page)

(continued from previous page)

```
init_fn = mesh(pjit(mlp.init,
                    (None, PartitionSpec("data", "model")), var_spec))
variables = init_fn(random.PRNGKey(0), x)
apply_fn = mesh(pjit(
    mlp.apply,
    (var_spec, PartitionSpec("data", "model")),
     PartitionSpec("data", "model")))
apply_fn(variables, x)
```

`Partitioned` values can gain additional axes when using transformations like `nn.vmap` and `nn.scan`. In this case you can specify the name of the new axis with the *metadata_params* args in vmap/scan:

```
class Model(nn.Module):
@nn.compact
def __call__(self, x):
  def body(mdl, c):
    c = MLP(4096)(c)
    return c, ()
  c, _ = nn.scan(
      body, variable_axes={"params": 0}, split_rngs={"params": 0}, length=8,
      metadata_params={nn.meta.PARTITION_NAME: "layers"})(self, x)
  return c
```

__init__(*value*, *names*, *mesh=None*)

Methods

__init__(value, names[, mesh])	

`add_axis`(index, params)	Adds a new axis to the axis metadata.
`get_partition_spec`()	Returns the `Partitionspec` for this partitioned value.
`get_sharding`(mesh)	Returns the `NamedSharding` for this partitioned value.
`remove_axis`(index, params)	Removes an axis from the axis metadata.
`replace`(**updates)	"Returns a new object replacing the specified fields with new values.
`replace_boxed`(val)	Replaces the boxed value with the provided value.
`unbox`([apply_constraint])	Returns the wrapped value with the partitioning applied as a sharding constraint.

Attributes

mesh
value
names

flax.linen.with_partitioning

flax.linen.**with_partitioning**(*fn*, *names*, *mesh=None*)

Wraps a function's return value with Partitioned.

Example:

```
kernel_init = with_partitioning(
    nn.initializers.lecun_normal, (None, "data"))
partitioned_dense = nn.Dense(features, kernel_init=kernel_init)
```

Parameters

- **fn** – The function to be wrapped. Typically this is an initializer.
- **names** – The logical axis passed to Partitioned.
- **mesh** – The mesh to use for the partitioning. If None, the global mesh resource is used if available.

Returns

A function wrapping fn that will return an instance of Partitioned.

flax.linen.get_partition_spec

flax.linen.**get_partition_spec**(*tree*)

Extracts a PartitionSpec tree from a PyTree containing Partitioned values.

flax.linen.get_sharding

flax.linen.**get_sharding**(*tree*, *mesh*)

Extracts a jax.sharding tree from a PyTree containing Partitioned values and a mesh.

flax.linen.LogicallyPartitioned

class flax.linen.**LogicallyPartitioned**(*value: Any*, *names: Tuple[Union[str, NoneType], ...]*, *mesh: Union[jax._src.mesh.Mesh, NoneType] = None*, *rules: Union[Sequence[Tuple[str, Union[str, Tuple[str], NoneType]]], NoneType] = None*)

__init__(*value*, *names*, *mesh=None*, *rules=None*)

Methods

__init__(value, names[, mesh, rules])

add_axis(index, params)	Adds a new axis to the axis metadata.
get_partition_spec()	Returns the Partitionspec for this partitioned value.
get_sharding(mesh)	Returns the NamedSharding for this partitioned value.
remove_axis(index, params)	Removes an axis from the axis metadata.
replace(**updates)	"Returns a new object replacing the specified fields with new values.
replace_boxed(val)	Replaces the boxed value with the provided value.
unbox([apply_constraint])	Returns the wrapped value with the partitioning constraint applied.

Attributes

mesh
rules

flax.linen.logical_axis_rules

flax.linen.**logical_axis_rules**(*rules*)

Context manager for setting the logical to mesh axis bindings.

flax.linen.set_logical_axis_rules

flax.linen.**set_logical_axis_rules**(*rules*)

Sets the global logical axis to mesh axis binding.

flax.linen.get_logical_axis_rules

flax.linen.**get_logical_axis_rules**()

Returns the global logical axis to mesh axis binding.

flax.linen.logical_to_mesh_axes

flax.linen.**logical_to_mesh_axes**(*array_dim_names*, *rules=None*)

Compute layout for an array.

The rules are in order of precedence, and consist of pairs: (ArrayDimensionName, MeshDimensionName), meaning that the given array dimension (if present and unused) should be sharded across the given mesh dimension (if present and unused).

A Layout of an Array is expressed as a tuple with one element for each dimension in the Array. The element is either None, or is the name of a mesh-dimension, meaning that this dimension of the array is sharded across this dimension of the mesh.

For example, given an array with

array_dim_names = ('batch', 'length', 'heads', 'features')

and the layout rules are:

rules = (('batch', 'X'),
('features', 'X'), ('heads', 'Y'), ('batch', 'Z'))

then this function will return

PartitionSpec('X', None, 'Y', None)

Parameters

- **array_dim_names** – Tuple of array dimension names or None.
- **rules** – Optional logical to mesh rules override. Defaults to using the rules defined in the dynamic context set from the *axis_rules* function.

Returns

PartitionSpec for the parameter.

flax.linen.logical_to_mesh

flax.linen.**logical_to_mesh**(*tree*, *rules=None*)

Applies logical_to_mesh_axes to pytrees of logical PartitionSpecs.

flax.linen.logical_to_mesh_sharding

flax.linen.**logical_to_mesh_sharding**(*tree*, *mesh*, *rules=None*)

Convert pytrees of logical PartitionSpecs to shardings.

flax.linen.with_logical_constraint

flax.linen.**with_logical_constraint**(*x*, *logical_axis_resources*, *rules=None*, *mesh=None*, *fallback=RulesFallback.AXIS_IS_UNSHARDED*)

Version of pjit's with_sharding_constraint that uses logical axis names.

flax.linen.with_logical_partitioning

flax.linen.**with_logical_partitioning**(*fn*, *names*, *mesh=None*, *rules=None*)

Wraps a function's return value with LogicallyPartitioned.

Example:

```
kernel_init = with_logical_partitioning(
    nn.initializers.lecun_normal, (None, "data"))
partitioned_dense = nn.Dense(features, kernel_init=kernel_init)
```

Parameters

- **fn** – The function to be wrapped. Typically this is an initializer.
- **names** – The logical axis passed to `LogicallyPartitioned`.
- **mesh** – The mesh to use for the partitioning. If None, the global mesh resource is used if available.
- **rules** – Optional logical to mesh rules use. If None, the global rules are used if available.

Returns

A function wrapping `fn` that will return an instance of `LogicallyPartitioned`.

Linear Modules

Dense(features[, use_bias, dtype, ...])	A linear transformation applied over the last dimension of the input.
DenseGeneral(features[, axis, batch_dims, ...])	A linear transformation with flexible axes.
Conv(features, kernel_size[, strides, ...])	Convolution Module wrapping *lax.conv_general_dilated*.
ConvTranspose(features, kernel_size[, ...])	Convolution Module wrapping lax.conv_transpose.
ConvLocal(features, kernel_size[, strides, ...])	Local convolution Module wrapping *lax.conv_general_dilated_local*.
Embed(num_embeddings, features[, dtype, ...])	Embedding Module.

flax.linen.Dense

class flax.linen.**Dense**(*features*, *use_bias=True*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *precision=None*, *kernel_init=<function variance_scaling.<locals>.init>*, *bias_init=<function zeros>*, *dot_general=<function dot_general>*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

A linear transformation applied over the last dimension of the input.

features

the number of output features.

Type
int

use_bias

whether to add a bias to the output (default: True).

Type
bool

dtype

the dtype of the computation (default: infer from input and params).

Type
Optional[Any]

param_dtype

the dtype passed to parameter initializers (default: float32).

Type
Any

precision

numerical precision of the computation see *jax.lax.Precision* for details.

Type
Union[None, str, jax._src.lax.lax.Precision, Tuple[str, str], Tuple[jax._src.lax.lax.Precision, jax._src.lax.lax.Precision]]

kernel_init

initializer function for the weight matrix.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

bias_init

initializer function for the bias.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

__call__(*inputs*)

Applies a linear transformation to the inputs along the last dimension.

Parameters
inputs – The nd-array to be transformed.

Returns
The transformed input.

Methods

flax.linen.DenseGeneral

class flax.linen.**DenseGeneral**(*features*, *axis=-1*, *batch_dims=()*, *use_bias=True*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *kernel_init=<function variance_scaling.<locals>.init>*, *bias_init=<function zeros>*, *precision=None*, *dot_general=<function dot_general>*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

A linear transformation with flexible axes.

features

int or tuple with number of output features.

Type
Union[int, Sequence[int]]

axis

int or tuple with axes to apply the transformation on. For instance, (-2, -1) will apply the transformation to the last two axes.

Type
Union[int, Sequence[int]]

batch_dims

tuple with batch axes.

Type
Sequence[int]

use_bias

whether to add a bias to the output (default: True).

Type
bool

dtype

the dtype of the computation (default: infer from input and params).

Type
Optional[Any]

param_dtype

the dtype passed to parameter initializers (default: float32).

Type
Any

kernel_init

initializer function for the weight matrix.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

`bias_init`

initializer function for the bias.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

`precision`

numerical precision of the computation see *jax.lax.Precision* for details.

Type
Union[None, str, jax._src.lax.lax.Precision, Tuple[str, str], Tuple[jax._src.lax.lax.Precision, jax._src.lax.lax.Precision]]

`__call__`(*inputs*)

Applies a linear transformation to the inputs along multiple dimensions.

Parameters
`inputs` – The nd-array to be transformed.

Returns
The transformed input.

Methods

flax.linen.Conv

`class flax.linen.Conv`(*features*, *kernel_size*, *strides=1*, *padding='SAME'*, *input_dilation=1*, *kernel_dilation=1*, *feature_group_count=1*, *use_bias=True*, *mask=None*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *precision=None*, *kernel_init=<function variance_scaling.<locals>.init>*, *bias_init=<function zeros>*, *conv_general_dilated=<function conv_general_dilated>*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Convolution Module wrapping *lax.conv_general_dilated*.

`features`

number of convolution filters.

Type
int

`kernel_size`

shape of the convolutional kernel. For 1D convolution, the kernel size can be passed as an integer. For all other cases, it must be a sequence of integers.

Type
Sequence[int]

`strides`

an integer or a sequence of *n* integers, representing the inter-window strides (default: 1).

Type
Union[None, int, Sequence[int]]

`padding`

either the string *'SAME'*, the string *'VALID'*, the string *'CIRCULAR'* (periodic boundary conditions), or a sequence of *n (low, high)* integer pairs that give the padding to apply before and after each spatial dimension. A single int is interpreted as applying the same padding in all dims and assign a single int in a sequence causes the same padding to be used on both sides. *'CAUSAL'* padding for a 1D convolution will left-pad the convolution axis, resulting in same-sized output.

Type
Union[str, int, Sequence[Union[int, Tuple[int, int]]]]

`input_dilation`

an integer or a sequence of *n* integers, giving the dilation factor to apply in each spatial dimension of *inputs* (default: 1). Convolution with input dilation *d* is equivalent to transposed convolution with stride *d*.

Type
Union[None, int, Sequence[int]]

`kernel_dilation`

an integer or a sequence of *n* integers, giving the dilation factor to apply in each spatial dimension of the convolution kernel (default: 1). Convolution with kernel dilation is also known as 'atrous convolution'.

Type
Union[None, int, Sequence[int]]

`feature_group_count`

integer, default 1. If specified divides the input features into groups.

Type
int

`use_bias`

whether to add a bias to the output (default: True).

Type
bool

`mask`

Optional mask for the weights during masked convolution. The mask must be the same shape as the convolution weight matrix.

Type
Optional[Any]

`dtype`

the dtype of the computation (default: infer from input and params).

Type
Optional[Any]

`param_dtype`

the dtype passed to parameter initializers (default: float32).

Type
Any

`precision`

numerical precision of the computation see *jax.lax.Precision* for details.

Type
Union[None, str, jax._src.lax.lax.Precision, Tuple[str, str], Tuple[jax._src.lax.lax.Precision, jax._src.lax.lax.Precision]]

kernel_init

initializer for the convolutional kernel.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

bias_init

initializer for the bias.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

__call__(*inputs*)

Applies a (potentially unshared) convolution to the inputs.

Parameters
inputs – input data with dimensions (*batch_dims, spatial_dims..., features). This is the channels-last convention, i.e. NHWC for a 2d convolution and NDHWC for a 3D convolution. Note: this is different from the input convention used by *lax.conv_general_dilated*, which puts the spatial dimensions last. Note: If the input has more than 1 batch dimension, all batch dimensions are flattened into a single dimension for the convolution and restored before returning. In some cases directly vmap'ing the layer may yield better performance than this default flattening approach. If the input lacks a batch dimension it will be added for the convolution and removed n return, an allowance made to enable writing single-example code.

Returns
The convolved data.

Methods

flax.linen.ConvTranspose

class flax.linen.**ConvTranspose**(*features*, *kernel_size*, *strides=None*, *padding='SAME'*, *kernel_dilation=None*, *use_bias=True*, *mask=None*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *precision=None*, *kernel_init=<function variance_scaling.<locals>.init>*, *bias_init=<function zeros>*, *transpose_kernel=False*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Convolution Module wrapping lax.conv_transpose.

features

number of convolution filters.

Type
int

kernel_size

shape of the convolutional kernel. For 1D convolution, the kernel size can be passed as an integer. For all other cases, it must be a sequence of integers.

Type
Union[int, Sequence[int]]

`strides`

a sequence of *n* integers, representing the inter-window strides.

Type
Optional[Sequence[int]]

`padding`

either the string *'SAME'*, the string *'VALID'*, the string *'CIRCULAR'* (periodic boundary conditions), or a sequence of *n (low, high)* integer pairs that give the padding to apply before and after each spatial dimension. A single int is interpreted as applying the same padding in all dims and assign a single int in a sequence causes the same padding to be used on both sides.

Type
Union[str, int, Sequence[Union[int, Tuple[int, int]]]]

`kernel_dilation`

None, or a sequence of *n* integers, giving the dilation factor to apply in each spatial dimension of the convolution kernel. Convolution with kernel dilation is also known as 'atrous convolution'.

Type
Optional[Sequence[int]]

`use_bias`

whether to add a bias to the output (default: True).

Type
bool

`mask`

Optional mask for the weights during masked convolution. The mask must be the same shape as the convolution weight matrix.

Type
Optional[Any]

`dtype`

the dtype of the computation (default: infer from input and params).

Type
Any

`param_dtype`

the dtype passed to parameter initializers (default: float32).

Type
Any

`precision`

numerical precision of the computation see *jax.lax.Precision* for details.

Type
Union[None, str, jax._src.lax.lax.Precision, Tuple[str, str], Tuple[jax._src.lax.lax.Precision, jax._src.lax.lax.Precision]]

`kernel_init`

initializer for the convolutional kernel.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

`bias_init`

initializer for the bias.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

`transpose_kernel`

if True flips spatial axes and swaps the input/output channel axes of the kernel.

Type
bool

`__call__`(*inputs*)

Applies a transposed convolution to the inputs.

Behaviour mirrors of *jax.lax.conv_transpose*.

Parameters
inputs – input data with dimensions (*batch_dims, spatial_dims..., features). This is the channels-last convention, i.e. NHWC for a 2d convolution and NDHWC for a 3D convolution. Note: this is different from the input convention used by *lax.conv_general_dilated*, which puts the spatial dimensions last. Note: If the input has more than 1 batch dimension, all batch dimensions are flattened into a single dimension for the convolution and restored before returning. In some cases directly vmap'ing the layer may yield better performance than this default flattening approach. If the input lacks a batch dimension it will be added for the convolution and removed n return, an allowance made to enable writing single-example code.

Returns
The convolved data.

Methods

flax.linen.ConvLocal

`class flax.linen.ConvLocal`(*features*, *kernel_size*, *strides=1*, *padding='SAME'*, *input_dilation=1*, *kernel_dilation=1*, *feature_group_count=1*, *use_bias=True*, *mask=None*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *precision=None*, *kernel_init=<function variance_scaling.<locals>.init>*, *bias_init=<function zeros>*, *conv_general_dilated=<function conv_general_dilated>*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Local convolution Module wrapping *lax.conv_general_dilated_local*.

`features`

number of convolution filters.

Type
int

`kernel_size`

shape of the convolutional kernel. For 1D convolution, the kernel size can be passed as an integer. For all other cases, it must be a sequence of integers.

Type
Sequence[int]

`strides`

an integer or a sequence of *n* integers, representing the inter-window strides (default: 1).

Type
Union[None, int, Sequence[int]]

`padding`

either the string *'SAME'*, the string *'VALID'*, the string *'CIRCULAR'* (periodic boundary conditions), or a sequence of *n (low, high)* integer pairs that give the padding to apply before and after each spatial dimension. A single int is interpreted as applying the same padding in all dims and assign a single int in a sequence causes the same padding to be used on both sides. *'CAUSAL'* padding for a 1D convolution will left-pad the convolution axis, resulting in same-sized output.

Type
Union[str, int, Sequence[Union[int, Tuple[int, int]]]]

`input_dilation`

an integer or a sequence of *n* integers, giving the dilation factor to apply in each spatial dimension of *inputs* (default: 1). Convolution with input dilation *d* is equivalent to transposed convolution with stride *d*.

Type
Union[None, int, Sequence[int]]

`kernel_dilation`

an integer or a sequence of *n* integers, giving the dilation factor to apply in each spatial dimension of the convolution kernel (default: 1). Convolution with kernel dilation is also known as 'atrous convolution'.

Type
Union[None, int, Sequence[int]]

`feature_group_count`

integer, default 1. If specified divides the input features into groups.

Type
int

`use_bias`

whether to add a bias to the output (default: True).

Type
bool

`mask`

Optional mask for the weights during masked convolution. The mask must be the same shape as the convolution weight matrix.

Type
Optional[Any]

`dtype`

the dtype of the computation (default: infer from input and params).

Type
Optional[Any]

`param_dtype`

the dtype passed to parameter initializers (default: float32).

Type
Any

precision

numerical precision of the computation see *jax.lax.Precision* for details.

Type
Union[None, str, jax._src.lax.lax.Precision, Tuple[str, str], Tuple[jax._src.lax.lax.Precision, jax._src.lax.lax.Precision]]

kernel_init

initializer for the convolutional kernel.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

bias_init

initializer for the bias.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

__call__(*inputs*)

Applies a (potentially unshared) convolution to the inputs.

Parameters
inputs – input data with dimensions (*batch_dims, spatial_dims..., features). This is the channels-last convention, i.e. NHWC for a 2d convolution and NDHWC for a 3D convolution. Note: this is different from the input convention used by *lax.conv_general_dilated*, which puts the spatial dimensions last. Note: If the input has more than 1 batch dimension, all batch dimensions are flattened into a single dimension for the convolution and restored before returning. In some cases directly vmap'ing the layer may yield better performance than this default flattening approach. If the input lacks a batch dimension it will be added for the convolution and removed n return, an allowance made to enable writing single-example code.

Returns
The convolved data.

Methods

flax.linen.Embed

class flax.linen.**Embed**(*num_embeddings*, *features*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *embedding_init=<function variance_scaling.<locals>.init>*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Embedding Module.

A parameterized function from integers [0, n) to d-dimensional vectors.

num_embeddings

number of embeddings.

Type
int

features

number of feature dimensions for each embedding.

Type
int

dtype

the dtype of the embedding vectors (default: same as embedding).

Type
Optional[Any]

param_dtype

the dtype passed to parameter initializers (default: float32).

Type
Any

embedding_init

embedding initializer.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

__call__(*inputs*)

Embeds the inputs along the last dimension.

Parameters
inputs – input data, all dimensions are considered batch dimensions.

Returns
Output which is embedded input data. The output shape follows the input, with an additional *features* dimension appended.

Methods

attend(query)	Attend over the embedding using a query array.
setup()	Initializes a Module lazily (similar to a lazy __init__).

Normalization

BatchNorm([use_running_average, axis, ...])	BatchNorm Module.
LayerNorm([epsilon, dtype, param_dtype, ...])	Layer normalization (https://arxiv.org/abs/1607.06450).
GroupNorm([num_groups, group_size, epsilon, ...])	Group normalization (arxiv.org/abs/1803.08494).

flax.linen.BatchNorm

class flax.linen.**BatchNorm**(*use_running_average=None*, *axis=-1*, *momentum=0.99*, *epsilon=1e-05*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *use_bias=True*, *use_scale=True*, *bias_init=<function zeros>*, *scale_init=<function ones>*, *axis_name=None*, *axis_index_groups=None*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

BatchNorm Module.

Usage Note: If we define a model with BatchNorm, for example:

```
BN = nn.BatchNorm(use_running_average=False, momentum=0.9, epsilon=1e-5,
                  dtype=jnp.float32)
```

The initialized variables dict will contain in addition to a 'params' collection a separate 'batch_stats' collection that will contain all the running statistics for all the BatchNorm layers in a model:

```
vars_initialized = BN.init(key, x)  # {'params': ..., 'batch_stats': ...}
```

We then update the batch_stats during training by specifying that the *batch_stats* collection is mutable in the *apply* method for our module.:

```
vars_in = {'params': params, 'batch_stats': old_batch_stats}
y, mutated_vars = BN.apply(vars_in, x, mutable=['batch_stats'])
new_batch_stats = mutated_vars['batch_stats']
```

During eval we would define BN with *use_running_average=True* and use the batch_stats collection from training to set the statistics. In this case we are not mutating the batch statistics collection, and needn't mark it mutable:

```
vars_in = {'params': params, 'batch_stats': training_batch_stats}
y = BN.apply(vars_in, x)
```

use_running_average

if True, the statistics stored in batch_stats will be used instead of computing the batch statistics on the input.

Type
Optional[bool]

axis

the feature or non-batch axis of the input.

Type
int

momentum

decay rate for the exponential moving average of the batch statistics.

Type
float

epsilon

a small float added to variance to avoid dividing by zero.

Type
float

`dtype`

the dtype of the result (default: infer from input and params).

Type
Optional[Any]

`param_dtype`

the dtype passed to parameter initializers (default: float32).

Type
Any

`use_bias`

if True, bias (beta) is added.

Type
bool

`use_scale`

if True, multiply by scale (gamma). When the next layer is linear (also e.g. nn.relu), this can be disabled since the scaling will be done by the next layer.

Type
bool

`bias_init`

initializer for bias, by default, zero.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

`scale_init`

initializer for scale, by default, one.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

`axis_name`

the axis name used to combine batch statistics from multiple devices. See *jax.pmap* for a description of axis names (default: None).

Type
Optional[str]

`axis_index_groups`

groups of axis indices within that named axis representing subsets of devices to reduce over (default: None). For example, *[[0, 1], [2, 3]]* would independently batch-normalize over the examples on the first two and last two devices. See *jax.lax.psum* for more details.

Type
Any

`__call__`(*x*, *use_running_average=None*)

Normalizes the input using batch statistics.

NOTE: During initialization (when *self.is_initializing()* is *True*) the running average of the batch statistics will not be updated. Therefore, the inputs fed during initialization don't need to match that of the actual input distribution and the reduction axis (set with *axis_name*) does not have to exist.

Parameters

- **x** – the input to be normalized.
- **use_running_average** – if true, the statistics stored in batch_stats will be used instead of computing the batch statistics on the input.

Returns
Normalized inputs (the same shape as inputs).

Methods

flax.linen.LayerNorm

class flax.linen.**LayerNorm**(*epsilon=1e-06*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *use_bias=True*, *use_scale=True*, *bias_init=<function zeros>*, *scale_init=<function ones>*, *reduction_axes=-1*, *feature_axes=-1*, *axis_name=None*, *axis_index_groups=None*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Layer normalization (https://arxiv.org/abs/1607.06450).

LayerNorm normalizes the activations of the layer for each given example in a batch independently, rather than across a batch like Batch Normalization. i.e. applies a transformation that maintains the mean activation within each example close to 0 and the activation standard deviation close to 1.

epsilon

A small float added to variance to avoid dividing by zero.

Type
float

dtype

the dtype of the result (default: infer from input and params).

Type
Optional[Any]

param_dtype

the dtype passed to parameter initializers (default: float32).

Type
Any

use_bias

If True, bias (beta) is added.

Type
bool

use_scale

If True, multiply by scale (gamma). When the next layer is linear (also e.g. nn.relu), this can be disabled since the scaling will be done by the next layer.

Type
bool

`bias_init`

Initializer for bias, by default, zero.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

`scale_init`

Initializer for scale, by default, one.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

`reduction_axes`

Axes for computing normalization statistics.

Type
Union[int, Any]

`feature_axes`

Feature axes for learned bias and scaling.

Type
Union[int, Any]

`axis_name`

the axis name used to combine batch statistics from multiple devices. See *jax.pmap* for a description of axis names (default: None). This is only needed if the model is subdivided across devices, i.e. the array being normalized is sharded across devices within a pmap.

Type
Optional[str]

`axis_index_groups`

groups of axis indices within that named axis representing subsets of devices to reduce over (default: None). For example, *[[0, 1], [2, 3]]* would independently batch-normalize over the examples on the first two and last two devices. See *jax.lax.psum* for more details.

Type
Any

`__call__`(*x*)

Applies layer normalization on the input.

Parameters
x – the inputs

Returns
Normalized inputs (the same shape as inputs).

Methods

flax.linen.GroupNorm

class flax.linen.**GroupNorm**(*num_groups=32*, *group_size=None*, *epsilon=1e-06*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *use_bias=True*, *use_scale=True*, *bias_init=<function zeros>*, *scale_init=<function ones>*, *axis_name=None*, *axis_index_groups=None*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Group normalization (arxiv.org/abs/1803.08494).

This op is similar to batch normalization, but statistics are shared across equally-sized groups of channels and not shared across batch dimension. Thus, group normalization does not depend on the batch composition and does not require maintaining internal state for storing statistics. The user should either specify the total number of channel groups or the number of channels per group.

num_groups

the total number of channel groups. The default value of 32 is proposed by the original group normalization paper.

Type
Optional[int]

group_size

the number of channels in a group.

Type
Optional[int]

epsilon

A small float added to variance to avoid dividing by zero.

Type
float

dtype

the dtype of the result (default: infer from input and params).

Type
Optional[Any]

param_dtype

the dtype passed to parameter initializers (default: float32).

Type
Any

use_bias

If True, bias (beta) is added.

Type
bool

use_scale

If True, multiply by scale (gamma). When the next layer is linear (also e.g. nn.relu), this can be disabled since the scaling will be done by the next layer.

Type
bool

bias_init

Initializer for bias, by default, zero.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

scale_init

Initializer for scale, by default, one.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

axis_name

the axis name used to combine batch statistics from multiple devices. See *jax.pmap* for a description of axis names (default: None). This is only needed if the model is subdivided across devices, i.e. the array being normalized is sharded across devices within a pmap.

Type
Optional[str]

axis_index_groups

groups of axis indices within that named axis representing subsets of devices to reduce over (default: None). For example, *[[0, 1], [2, 3]]* would independently batch-normalize over the examples on the first two and last two devices. See *jax.lax.psum* for more details.

Type
Any

__call__(*x*)

Applies group normalization to the input (arxiv.org/abs/1803.08494).

Parameters
x – the input of shape N...C, where N is a batch dimension and C is a channels dimensions. ... represents an arbitrary number of extra dimensions that are used to accumulate statistics over.

Returns
Normalized inputs (the same shape as inputs).

Methods

Pooling

max_pool(inputs, window_shape[, strides, ...])	Pools the input by taking the maximum of a window slice.
avg_pool(inputs, window_shape[, strides, ...])	Pools the input by taking the average over a window.
pool(inputs, init, reduce_fn, window_shape, ...)	Helper function to define pooling functions.

flax.linen.max_pool

flax.linen.**max_pool**(*inputs*, *window_shape*, *strides=None*, *padding='VALID'*)

Pools the input by taking the maximum of a window slice.

Parameters

- **inputs** – input data with dimensions (batch, window dims..., features).
- **window_shape** – a shape tuple defining the window to reduce over.
- **strides** – a sequence of *n* integers, representing the inter-window strides (default: *(1, ..., 1)*).
- **padding** – either the string *'SAME'*, the string *'VALID'*, or a sequence of *n (low, high)* integer pairs that give the padding to apply before and after each spatial dimension (default: *'VALID'*).

Returns

The maximum for each window slice.

flax.linen.avg_pool

flax.linen.**avg_pool**(*inputs*, *window_shape*, *strides=None*, *padding='VALID'*, *count_include_pad=True*)

Pools the input by taking the average over a window.

Parameters

- **inputs** – input data with dimensions (batch, window dims..., features).
- **window_shape** – a shape tuple defining the window to reduce over.
- **strides** – a sequence of *n* integers, representing the inter-window strides (default: *(1, ..., 1)*).
- **padding** – either the string *'SAME'*, the string *'VALID'*, or a sequence of *n (low, high)* integer pairs that give the padding to apply before and after each spatial dimension (default: *'VALID'*).
- **count_include_pad** – a boolean whether to include padded tokens in the average calculation (default: *True*).

Returns

The average for each window slice.

flax.linen.pool

flax.linen.**pool**(*inputs*, *init*, *reduce_fn*, *window_shape*, *strides*, *padding*)

Helper function to define pooling functions.

Pooling functions are implemented using the ReduceWindow XLA op. NOTE: Be aware that pooling is not generally differentiable. That means providing a reduce_fn that is differentiable does not imply that pool is differentiable.

Parameters

- **inputs** – input data with dimensions (batch, window dims..., features).
- **init** – the initial value for the reduction

- **reduce_fn** – a reduce function of the form *(T, T) -> T*.
- **window_shape** – a shape tuple defining the window to reduce over.
- **strides** – a sequence of *n* integers, representing the inter-window strides (default: *(1, …, 1)*).
- **padding** – either the string *'SAME'*, the string *'VALID'*, or a sequence of *n (low, high)* integer pairs that give the padding to apply before and after each spatial dimension.

Returns
The output of the reduction for each window slice.

Activation functions

Activation functions.

PReLU([param_dtype, negative_slope_init, ...])	Parametric Rectified Linear Unit (PReLU) activation function.
celu(x[, alpha])	Continuously-differentiable exponential linear unit activation.
elu(x[, alpha])	Exponential linear unit activation function.
gelu(x[, approximate])	Gaussian error linear unit activation function.
glu(x[, axis])	Gated linear unit activation function.
hard_sigmoid(x)	Hard Sigmoid activation function.
hard_silu(x)	Hard SiLU activation function
hard_swish(x)	Hard SiLU activation function
hard_tanh(x)	Hard $\tanh$ activation function.
leaky_relu(x[, negative_slope])	Leaky rectified linear unit activation function.
log_sigmoid(x)	Log-sigmoid activation function.
log_softmax(x[, axis, where, initial])	Log-Softmax function.
logsumexp(a[, axis, b, keepdims, return_sign])	Log-sum-exp reduction.
one_hot(x, num_classes, *[, dtype, axis])	One-hot encodes the given indices.
relu	Rectified linear unit activation function.
relu6	Rectified Linear Unit 6 activation function.
selu(x)	Scaled exponential linear unit activation.
sigmoid(x)	Sigmoid activation function.
silu(x)	SiLU activation function.
soft_sign(x)	Soft-sign activation function.
softmax(x[, axis, where, initial])	Softmax function.
softplus(x)	Softplus activation function.
standardize(x[, axis, mean, variance, ...])	Normalizes an array by subtracting `mean` and dividing by $\sqrt{\text{variance}}$.
swish(x)	SiLU activation function.
tanh(x, /)	Compute hyperbolic tangent element-wise.

flax.linen.activation.PReLU

class flax.linen.activation.**PReLU**(*param_dtype=<class 'jax.numpy.float32'>*, *negative_slope_init=0.01*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Parametric Rectified Linear Unit (PReLU) activation function.

param_dtype

the dtype passed to parameter initializers (default: float32).

Type
Any

negative_slope_init

the value to initialize the negative slope (default 0.01).

Type
float

__init__(*param_dtype=<class 'jax.numpy.float32'>*, *negative_slope_init=0.01*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Methods

__init__([param_dtype, negative_slope_init, ...])	
apply(variables, *args[, rngs, method, ...])	Applies a module method to variables and returns output and modified variables.
bind(variables, *args[, rngs, mutable])	Creates an interactive Module instance by binding variables and RNGs.
clone(*[, parent])	Creates a clone of this Module, with optionally updated arguments.
get_variable(col, name[, default])	Retrieves the value of a Variable.
has_rng(name)	Returns true if a PRNGSequence with name *name* exists.
has_variable(col, name)	Checks if a variable of given collection and name exists in this Module.
init(rngs, *args[, method, mutable, ...])	Initializes a module method with variables and returns modified variables.
init_with_output(rngs, *args[, method, ...])	Initializes a module method with variables and returns output and modified variables.
is_initializing()	Returns True if running under self.init(...) or nn.init(...)().
is_mutable_collection(col)	Returns true if the collection *col* is mutable.
lazy_init(rngs, *args[, method, mutable])	Initializes a module without computing on an actual input.
make_rng(name)	Returns a new RNG key from a given RNG sequence for this Module.
param(name, init_fn, *init_args[, unbox])	Declares and returns a parameter in this Module.
perturb(name, value[, collection])	Add an zero-value variable ('perturbation') to the intermediate value.
put_variable(col, name, value)	Updates the value of the given variable if it is mutable, or an error otherwise.
setup()	Initializes a Module lazily (similar to a lazy `__init__`).
sow(col, name, value[, reduce_fn, init_fn])	Stores a value in a collection.
tabulate(rngs, *args[, depth, ...])	Creates a summary of the Module represented as a table.
unbind()	Returns an unbound copy of a Module and its variables.
variable(col, name[, init_fn, unbox])	Declares and returns a variable in this Module.

Attributes

name	
negative_slope_init	
parent	
scope	
variables	Returns the variables in this module.

flax.linen.activation.celu

`flax.linen.activation.`**`celu`**(*x*, *alpha=1.0*)

Continuously-differentiable exponential linear unit activation.

Computes the element-wise function:

$$\mathrm{celu}(x) = \begin{cases} x, & x > 0 \\ \alpha\left(\exp(\frac{x}{\alpha}) - 1\right), & x \leq 0 \end{cases}$$

For more information, see Continuously Differentiable Exponential Linear Units.

Parameters

- **`x`** – input array
- **`alpha`** – array or scalar (default: 1.0)

flax.linen.activation.elu

`flax.linen.activation.`**`elu`**(*x*, *alpha=1.0*)

Exponential linear unit activation function.

Computes the element-wise function:

$$\mathrm{elu}(x) = \begin{cases} x, & x > 0 \\ \alpha\left(\exp(x) - 1\right), & x \leq 0 \end{cases}$$

Parameters

- **`x`** – input array
- **`alpha`** – scalar or array of alpha values (default: 1.0)

flax.linen.activation.gelu

`flax.linen.activation.`**`gelu`**(*x*, *approximate=True*)

Gaussian error linear unit activation function.

If `approximate=False`, computes the element-wise function:

$$\mathrm{gelu}(x) = \frac{x}{2}\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$$

If `approximate=True`, uses the approximate formulation of GELU:

$$\mathrm{gelu}(x) = \frac{x}{2}\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right)$$

For more information, see Gaussian Error Linear Units (GELUs), section 2.

Parameters

- **`x`** – input array
- **`approximate`** – whether to use the approximate or exact formulation.

flax.linen.activation.glu

flax.linen.activation.**glu**(*x*, *axis=-1*)

Gated linear unit activation function.

Parameters

- **x** – input array
- **axis** – the axis along which the split should be computed (default: -1)

flax.linen.activation.hard_sigmoid

flax.linen.activation.**hard_sigmoid**(*x*)

Hard Sigmoid activation function.

Computes the element-wise function

$$\text{hard\_sigmoid}(x) = \frac{\text{relu6}(x+3)}{6}$$

Parameters

x – input array

flax.linen.activation.hard_silu

flax.linen.activation.**hard_silu**(*x*)

Hard SiLU activation function

Computes the element-wise function

$$\text{hard\_silu}(x) = x \cdot \text{hard\_sigmoid}(x)$$

Parameters

x – input array

flax.linen.activation.hard_swish

flax.linen.activation.**hard_swish**(*x*)

Hard SiLU activation function

Computes the element-wise function

$$\text{hard\_silu}(x) = x \cdot \text{hard\_sigmoid}(x)$$

Parameters

x – input array

flax.linen.activation.hard_tanh

`flax.linen.activation.`**`hard_tanh`**`(`x`)`

Hard $\tanh$ activation function.

Computes the element-wise function:

$$\text{hard\_tanh}(x) = \begin{cases} -1, & x < -1 \\ x, & -1 \le x \le 1 \\ 1, & 1 < x \end{cases}$$

Parameters
x – input array

flax.linen.activation.leaky_relu

`flax.linen.activation.`**`leaky_relu`**`(`*x*, *negative_slope=0.01*`)`

Leaky rectified linear unit activation function.

Computes the element-wise function:

$$\text{leaky\_relu}(x) = \begin{cases} x, & x \ge 0 \\ \alpha x, & x < 0 \end{cases}$$

where α = `negative_slope`.

Parameters

- **x** – input array
- **negative_slope** – array or scalar specifying the negative slope (default: 0.01)

flax.linen.activation.log_sigmoid

`flax.linen.activation.`**`log_sigmoid`**`(`x`)`

Log-sigmoid activation function.

Computes the element-wise function:

$$\text{log\_sigmoid}(x) = \log(\text{sigmoid}(x)) = -\log(1 + e^{-x})$$

Parameters
x – input array

flax.linen.activation.log_softmax

`flax.linen.activation.`**`log_softmax`**`(`*x*, *axis=-1*, *where=None*, *initial=None*`)`

Log-Softmax function.

Computes the logarithm of the `softmax` function, which rescales elements to the range $[-\infty, 0)$.

$$\text{log\_softmax}(x)_i = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

Parameters

- **x** – input array
- **axis** – the axis or axes along which the `log_softmax` should be computed. Either an integer or a tuple of integers.
- **where** – Elements to include in the `log_softmax`.
- **initial** – The minimum value used to shift the input array. Must be present when `where` is not None.

flax.linen.activation.logsumexp

`flax.linen.activation.`**`logsumexp`**(*a*, *axis=None*, *b=None*, *keepdims=False*, *return_sign=False*)

Log-sum-exp reduction.

Computes

$$\text{logsumexp}(a) = \log \sum_j b \cdot \exp(a_{ij})$$

where the j indices range over one or more dimensions to be reduced.

Parameters

- **a** – the input array
- **axis** – the axis or axes over which to reduce. May be either `None`, an int, or a tuple of ints.
- **b** – scaling factors for $\exp(a)$. Must be broadcastable to the shape of a.
- **keepdims** – If `True`, the axes that are reduced are left in the output as dimensions of size 1.
- **return_sign** – If `True`, the output will be a `(result, sign)` pair, where `sign` is the sign of the sums and `result` contains the logarithms of their absolute values. If `False` only `result` is returned and it will contain NaN values if the sums are negative.

Returns

Either an array `result` or a pair of arrays `(result, sign)`, depending on the value of the `return_sign` argument.

flax.linen.activation.one_hot

`flax.linen.activation.`**`one_hot`**(*x*, *num_classes*, ***, *dtype=<class 'jax.numpy.float64'>*, *axis=-1*)

One-hot encodes the given indices.

Each index in the input `x` is encoded as a vector of zeros of length `num_classes` with the element at `index` set to one:

```
>>> jax.nn.one_hot(jnp.array([0, 1, 2]), 3)
Array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]], dtype=float32)
```

Indices outside the range [0, num_classes) will be encoded as zeros:

```
>>> jax.nn.one_hot(jnp.array([-1, 3]), 3)
Array([[0., 0., 0.],
       [0., 0., 0.]], dtype=float32)
```

Parameters

- **x** – A tensor of indices.
- **num_classes** – Number of classes in the one-hot dimension.
- **dtype** – optional, a float dtype for the returned values (default `jnp.float_`).
- **axis** – the axis or axes along which the function should be computed.

flax.linen.activation.relu

`flax.linen.activation.`**relu = <jax._src.custom_derivatives.custom_jvp object>**

Rectified linear unit activation function.

Computes the element-wise function:

$$\mathrm{relu}(x) = \max(x, 0)$$

except under differentiation, we take:

$$\nabla \mathrm{relu}(0) = 0$$

For more information see Numerical influence of ReLU'(0) on backpropagation.

Parameters

x – input array

flax.linen.activation.relu6

`flax.linen.activation.`**relu6 = <jax._src.custom_derivatives.custom_jvp object>**

Rectified Linear Unit 6 activation function.

Computes the element-wise function

$$\mathrm{relu6}(x) = \min(\max(x, 0), 6)$$

except under differentiation, we take:

$$\nabla \mathrm{relu}(0) = 0$$

and

$$\nabla \mathrm{relu}(6) = 0$$

Parameters

x – input array

flax.linen.activation.selu

`flax.linen.activation.selu(x)`

Scaled exponential linear unit activation.

Computes the element-wise function:

$$\text{selu}(x) = \lambda \begin{cases} x, & x > 0 \\ \alpha e^x - \alpha, & x \leq 0 \end{cases}$$

where $\lambda = 1.0507009873554804934193349852946$ and $\alpha = 1.6732632423543772848170429916717$.

For more information, see Self-Normalizing Neural Networks.

Parameters
x – input array

flax.linen.activation.sigmoid

`flax.linen.activation.sigmoid(x)`

Sigmoid activation function.

Computes the element-wise function:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Parameters
x – input array

flax.linen.activation.silu

`flax.linen.activation.silu(x)`

SiLU activation function.

Computes the element-wise function:

$$\text{silu}(x) = x \cdot \text{sigmoid}(x) = \frac{x}{1 + e^{-x}}$$

Parameters
x – input array

flax.linen.activation.soft_sign

`flax.linen.activation.soft_sign(x)`

Soft-sign activation function.

Computes the element-wise function

$$\text{soft\_sign}(x) = \frac{x}{|x| + 1}$$

Parameters
x – input array

flax.linen.activation.softmax

`flax.linen.activation.`**`softmax`**(*x*, *axis=-1*, *where=None*, *initial=None*)

Softmax function.

Computes the function which rescales elements to the range $[0, 1]$ such that the elements along `axis` sum to 1.

$$\text{softmax}(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Parameters

- **`x`** – input array
- **`axis`** – the axis or axes along which the softmax should be computed. The softmax output summed across these dimensions should sum to 1. Either an integer or a tuple of integers.
- **`where`** – Elements to include in the `softmax`.
- **`initial`** – The minimum value used to shift the input array. Must be present when `where` is not None.

flax.linen.activation.softplus

`flax.linen.activation.`**`softplus`**(*x*)

Softplus activation function.

Computes the element-wise function

$$\text{softplus}(x) = \log(1 + e^x)$$

Parameters

`x` – input array

flax.linen.activation.standardize

`flax.linen.activation.`**`standardize`**(*x*, *axis=-1*, *mean=None*, *variance=None*, *epsilon=1e-05*, *where=None*)

Normalizes an array by subtracting `mean` and dividing by $\sqrt{\text{variance}}$.

flax.linen.activation.swish

`flax.linen.activation.`**`swish`**(*x*)

SiLU activation function.

Computes the element-wise function:

$$\text{silu}(x) = x \cdot \text{sigmoid}(x) = \frac{x}{1 + e^{-x}}$$

Parameters

`x` – input array

flax.linen.activation.tanh

flax.linen.activation.**tanh**(*x*, /)

Compute hyperbolic tangent element-wise.

LAX-backend implementation of `numpy.tanh()`.

Original docstring below.

Equivalent to `np.sinh(x)/np.cosh(x)` or `-1j * np.tan(1j*x)`.

Parameters
x (*array_like*) – Input array.

Returns
y – The corresponding hyperbolic tangent values. This is a scalar if *x* is a scalar.

Return type
ndarray

References

Initializers

Initializers for Flax.

constant(value[, dtype])	Builds an initializer that returns arrays full of a constant value.
delta_orthogonal([scale, column_axis, dtype])	Builds an initializer for delta orthogonal kernels.
glorot_normal([in_axis, out_axis, ...])	Builds a Glorot normal initializer (aka Xavier normal initializer).
glorot_uniform([in_axis, out_axis, ...])	Builds a Glorot uniform initializer (aka Xavier uniform initializer).
he_normal([in_axis, out_axis, batch_axis, dtype])	Builds a He normal initializer (aka Kaiming normal initializer).
he_uniform([in_axis, out_axis, batch_axis, ...])	Builds a He uniform initializer (aka Kaiming uniform initializer).
kaiming_normal([in_axis, out_axis, ...])	Builds a He normal initializer (aka Kaiming normal initializer).
kaiming_uniform([in_axis, out_axis, ...])	Builds a He uniform initializer (aka Kaiming uniform initializer).
lecun_normal([in_axis, out_axis, ...])	Builds a Lecun normal initializer.
lecun_uniform([in_axis, out_axis, ...])	Builds a Lecun uniform initializer.
normal([stddev, dtype])	Builds an initializer that returns real normally-distributed random arrays.
ones(key, shape[, dtype])	An initializer that returns a constant array full of ones.
ones_init()	Builds an initializer that returns a constant array full of ones.
orthogonal([scale, column_axis, dtype])	Builds an initializer that returns uniformly distributed orthogonal matrices.
uniform([scale, dtype])	Builds an initializer that returns real uniformly-distributed random arrays.
variance_scaling(scale, mode, distribution)	Initializer that adapts its scale to the shape of the weights tensor.
xavier_normal([in_axis, out_axis, ...])	Builds a Glorot normal initializer (aka Xavier normal initializer).
xavier_uniform([in_axis, out_axis, ...])	Builds a Glorot uniform initializer (aka Xavier uniform initializer).
zeros(key, shape[, dtype])	An initializer that returns a constant array full of zeros.
zeros_init()	Builds an initializer that returns a constant array full of zeros.

flax.linen.initializers.constant

flax.linen.initializers.**constant**(*value*, *dtype=<class 'jax.numpy.float64'>*)

Builds an initializer that returns arrays full of a constant `value`.

Parameters

- **value** – the constant value with which to fill the initializer.
- **dtype** – optional; the initializer's default dtype.

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.constant(-7)
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[-7., -7., -7.],
       [-7., -7., -7.]], dtype=float32)
```

flax.linen.initializers.delta_orthogonal

flax.linen.initializers.**delta_orthogonal**(*scale=1.0*, *column_axis=-1*, *dtype=<class 'jax.numpy.float64'>*)

Builds an initializer for delta orthogonal kernels.

Parameters

- **scale** – the upper bound of the uniform distribution.
- **column_axis** – the axis that contains the columns that should be orthogonal.
- **dtype** – the default dtype of the weights.

Returns

A delta orthogonal initializer. The shape passed to the initializer must be 3D, 4D, or 5D.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.delta_orthogonal()
>>> initializer(jax.random.PRNGKey(42), (3, 3, 3), jnp.float32)
Array([[[ 0.        ,  0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        ]],

       [[ 0.27858758, -0.7949833 , -0.53887904],
        [ 0.9120717 ,  0.04322892,  0.40774566],
        [-0.30085585, -0.6050892 ,  0.73712474]],

       [[ 0.        ,  0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        ],
        [ 0.        ,  0.        ,  0.        ]]], dtype=float32)
```

flax.linen.initializers.glorot_normal

flax.linen.initializers.**glorot_normal**(*in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Builds a Glorot normal initializer (aka Xavier normal initializer).

A Glorot normal initializer is a specialization of `jax.nn.initializers.variance_scaling()` where `scale = 1.0`, `mode="fan_avg"`, and `distribution="truncated_normal"`.

Parameters

- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.
- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.
- **dtype** – the dtype of the weights.

Returns

An initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.glorot_normal()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 0.41770416,  0.75262755,  0.7619329 ],
       [-0.5516644 , -0.6028657 ,  0.08661086]], dtype=float32)
```

flax.linen.initializers.glorot_uniform

flax.linen.initializers.**glorot_uniform**(*in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Builds a Glorot uniform initializer (aka Xavier uniform initializer).

A Glorot uniform initializer is a specialization of `jax.nn.initializers.variance_scaling()` where `scale = 1.0`, `mode="fan_avg"`, and `distribution="uniform"`.

Parameters

- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.
- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.
- **dtype** – the dtype of the weights.

Returns

An initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.glorot_uniform()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 0.50350785,  0.8088631 ,  0.81566876],
       [-0.6393332 , -0.6865721 ,  0.11003882]], dtype=float32)
```

flax.linen.initializers.he_normal

flax.linen.initializers.**he_normal**(*in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Builds a He normal initializer (aka Kaiming normal initializer).

A He normal initializer is a specialization of `jax.nn.initializers.variance_scaling()` where `scale = 2.0`, `mode="fan_in"`, and `distribution="truncated_normal"`.

Parameters

- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.
- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.
- **dtype** – the dtype of the weights.

Returns

An initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.kaiming_normal()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 0.6604483 ,  1.1900088 ,  1.2047218 ],
       [-0.87225807, -0.95321447,  0.1369438 ]], dtype=float32)
```

flax.linen.initializers.he_uniform

flax.linen.initializers.**he_uniform**(*in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Builds a He uniform initializer (aka Kaiming uniform initializer).

A He uniform initializer is a specialization of `jax.nn.initializers.variance_scaling()` where `scale = 2.0`, `mode="fan_in"`, and `distribution="uniform"`.

Parameters

- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.
- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.
- **dtype** – the dtype of the weights.

Returns

An initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.kaiming_uniform()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 0.79611576,  1.2789248 ,  1.2896855 ],
       [-1.0108745 , -1.0855657 ,  0.17398663]], dtype=float32)
```

flax.linen.initializers.kaiming_normal

flax.linen.initializers.**kaiming_normal**(*in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Builds a He normal initializer (aka Kaiming normal initializer).

A He normal initializer is a specialization of `jax.nn.initializers.variance_scaling()` where `scale = 2.0`, `mode="fan_in"`, and `distribution="truncated_normal"`.

Parameters

- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.
- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.
- **dtype** – the dtype of the weights.

Returns
An initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.kaiming_normal()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 0.6604483 ,  1.1900088 ,  1.2047218 ],
       [-0.87225807, -0.95321447,  0.1369438 ]], dtype=float32)
```

flax.linen.initializers.kaiming_uniform

flax.linen.initializers.**kaiming_uniform**(*in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Builds a He uniform initializer (aka Kaiming uniform initializer).

A He uniform initializer is a specialization of `jax.nn.initializers.variance_scaling()` where `scale = 2.0`, `mode="fan_in"`, and `distribution="uniform"`.

Parameters

- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.
- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.
- **dtype** – the dtype of the weights.

Returns
An initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.kaiming_uniform()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 0.79611576,  1.2789248 ,  1.2896855 ],
       [-1.0108745 , -1.0855657 ,  0.17398663]], dtype=float32)
```

flax.linen.initializers.lecun_normal

flax.linen.initializers.**lecun_normal**(*in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Builds a Lecun normal initializer.

A Lecun normal initializer is a specialization of `jax.nn.initializers.variance_scaling()` where `scale = 1.0`, `mode="fan_in"`, and `distribution="truncated_normal"`.

Parameters

- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.
- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.

- **dtype** – the dtype of the weights.

Returns
An initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.lecun_normal()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 0.46700746,  0.8414632 ,  0.8518669 ],
       [-0.61677957, -0.67402434,  0.09683388]], dtype=float32)
```

flax.linen.initializers.lecun_uniform

flax.linen.initializers.**lecun_uniform**(*in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Builds a Lecun uniform initializer.

A Lecun uniform initializer is a specialization of `jax.nn.initializers.variance_scaling()` where `scale = 1.0`, `mode="fan_in"`, and `distribution="uniform"`.

Parameters

- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.
- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.
- **dtype** – the dtype of the weights.

Returns
An initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.lecun_uniform()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 0.56293887,  0.90433645,  0.9119454 ],
       [-0.71479625, -0.7676109 ,  0.12302713]], dtype=float32)
```

flax.linen.initializers.normal

flax.linen.initializers.**normal**(*stddev=0.01*, *dtype=<class 'jax.numpy.float64'>*)

Builds an initializer that returns real normally-distributed random arrays.

Parameters

- **stddev** – optional; the standard deviation of the distribution.
- **dtype** – optional; the initializer's default dtype.

Returns
An initializer that returns arrays whose values are normally distributed with mean `0` and standard deviation `stddev`.

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.normal(5.0)
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 3.0613258 ,  5.6129413 ,  5.6866574 ],
       [-4.063663  , -4.4520254 ,  0.63115686]], dtype=float32)
```

flax.linen.initializers.ones

flax.linen.initializers.**ones**(*key*, *shape*, *dtype=<class 'jax.numpy.float64'>*)

An initializer that returns a constant array full of ones.

The key argument is ignored.

```
>>> import jax, jax.numpy as jnp
>>> jax.nn.initializers.ones(jax.random.PRNGKey(42), (3, 2), jnp.float32)
Array([[1., 1.],
       [1., 1.],
       [1., 1.]], dtype=float32)
```

flax.linen.initializers.ones_init

flax.linen.initializers.**ones_init**()

Builds an initializer that returns a constant array full of ones.

```
>>> import jax, jax.numpy as jnp
>>> from flax.linen.initializers import ones_init
>>> ones_initializer = ones_init()
>>> ones_initializer(jax.random.PRNGKey(42), (3, 2), jnp.float32)
Array([[1., 1.],
       [1., 1.],
       [1., 1.]], dtype=float32)
```

flax.linen.initializers.orthogonal

flax.linen.initializers.**orthogonal**(*scale=1.0*, *column_axis=-1*, *dtype=<class 'jax.numpy.float64'>*)

Builds an initializer that returns uniformly distributed orthogonal matrices.

If the shape is not square, the matrices will have orthonormal rows or columns depending on which side is smaller.

Parameters

- **scale** – the upper bound of the uniform distribution.
- **column_axis** – the axis that contains the columns that should be orthogonal.
- **dtype** – the default dtype of the weights.

Returns

An orthogonal initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.orthogonal()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 3.9026976e-01,  7.2495741e-01, -5.6756169e-01],
       [ 8.8047469e-01, -4.7409311e-01, -1.3157725e-04]],            dtype=float32)
```

flax.linen.initializers.uniform

flax.linen.initializers.**uniform**(*scale=0.01*, *dtype=<class 'jax.numpy.float64'>*)

Builds an initializer that returns real uniformly-distributed random arrays.

Parameters

- **scale** – optional; the upper bound of the random distribution.
- **dtype** – optional; the initializer's default dtype.

Returns

An initializer that returns arrays whose values are uniformly distributed in the range `[0, scale)`.

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.uniform(10.0)
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[7.298188 , 8.691938 , 8.7230015],
       [2.0818567, 1.8662417, 5.5022564]], dtype=float32)
```

flax.linen.initializers.variance_scaling

flax.linen.initializers.**variance_scaling**(*scale*, *mode*, *distribution*, *in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Initializer that adapts its scale to the shape of the weights tensor.

With `distribution="truncated_normal"` or `distribution="normal"`, samples are drawn from a (truncated) normal distribution with a mean of zero and a standard deviation (after truncation, if applicable) of $\sqrt{\frac{scale}{n}}$, where n is:

- the number of input units in the weights tensor, if `mode="fan_in"`,
- the number of output units, if `mode="fan_out"`, or
- the average of the numbers of input and output units, if `mode="fan_avg"`.

This initializer can be configured with `in_axis`, `out_axis`, and `batch_axis` to work with general convolutional or dense layers; axes that are not in any of those arguments are assumed to be the "receptive field" (convolution kernel spatial axes).

With `distribution="truncated_normal"`, the absolute values of the samples are truncated at 2 standard deviations before scaling.

With `distribution="uniform"`, samples are drawn from:

- a uniform interval, if *dtype* is real, or
- a uniform disk, if *dtype* is complex,

with a mean of zero and a standard deviation of $\sqrt{\frac{scale}{n}}$ where n is defined above.

Parameters

- **scale** – scaling factor (positive float).
- **mode** – one of "fan_in", "fan_out", and "fan_avg".
- **distribution** – random distribution to use. One of "truncated_normal", "normal" and "uniform".
- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.
- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.
- **dtype** – the dtype of the weights.

flax.linen.initializers.xavier_normal

flax.linen.initializers.**xavier_normal**(*in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Builds a Glorot normal initializer (aka Xavier normal initializer).

A Glorot normal initializer is a specialization of jax.nn.initializers.variance_scaling() where scale = 1.0, mode="fan_avg", and distribution="truncated_normal".

Parameters

- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.
- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.
- **dtype** – the dtype of the weights.

Returns

An initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.glorot_normal()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 0.41770416,  0.75262755,  0.7619329 ],
       [-0.5516644 , -0.6028657 ,  0.08661086]], dtype=float32)
```

flax.linen.initializers.xavier_uniform

flax.linen.initializers.**xavier_uniform**(*in_axis=-2*, *out_axis=-1*, *batch_axis=()*, *dtype=<class 'jax.numpy.float64'>*)

Builds a Glorot uniform initializer (aka Xavier uniform initializer).

A Glorot uniform initializer is a specialization of jax.nn.initializers.variance_scaling() where scale = 1.0, mode="fan_avg", and distribution="uniform".

Parameters

- **in_axis** – axis or sequence of axes of the input dimension in the weights array.
- **out_axis** – axis or sequence of axes of the output dimension in the weights array.

- **batch_axis** – axis or sequence of axes in the weight array that should be ignored.
- **dtype** – the dtype of the weights.

Returns
An initializer.

Example:

```
>>> import jax, jax.numpy as jnp
>>> initializer = jax.nn.initializers.glorot_uniform()
>>> initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[ 0.50350785,  0.8088631 ,  0.81566876],
       [-0.6393332 , -0.6865721 ,  0.11003882]], dtype=float32)
```

flax.linen.initializers.zeros

`flax.linen.initializers.`**zeros**(*key*, *shape*, *dtype=<class 'jax.numpy.float64'>*)

An initializer that returns a constant array full of zeros.

The `key` argument is ignored.

```
>>> import jax, jax.numpy as jnp
>>> jax.nn.initializers.zeros(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[0., 0., 0.],
       [0., 0., 0.]], dtype=float32)
```

flax.linen.initializers.zeros_init

`flax.linen.initializers.`**zeros_init**()

Builds an initializer that returns a constant array full of zeros.

```
>>> import jax, jax.numpy as jnp
>>> from flax.linen.initializers import zeros_init
>>> zeros_initializer = zeros_init()
>>> zeros_initializer(jax.random.PRNGKey(42), (2, 3), jnp.float32)
Array([[0., 0., 0.],
       [0., 0., 0.]], dtype=float32)
```

Combinators

Sequential(layers[, parent, name])	Applies a linear chain of Modules.

flax.linen.Sequential

class flax.linen.**Sequential**(*layers*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Applies a linear chain of Modules.

Meant to be used only for the simple case of fusing together callables where the input of a particular module/op is the output of the previous one.

Modules will be applied in the order that they are passed in the constructor.

The apply() method of Sequential accepts any input and forwards it to the first module it contains. It chains the output sequentially to the input of the next module and returns the output of the final module.

Example usage:

```
class Foo(nn.Module):
  feature_sizes: Sequence[int]

  @nn.compact
  def __call__(self, x):
    return nn.Sequential([nn.Dense(4),
                          nn.relu,
                          nn.Dense(2),
                          nn.log_softmax])(x)
```

This combinator supports also layers that return multiple outputs if returned as a tuple or a dictionary.

Example usage:

```
class CrossAttentionBlock(nn.Module):
  num_heads: int = 2
  qkv_features: int = 16

  @nn.compact
  def __call__(self, query, key_value):
    output = nn.MultiHeadDotProductAttention(
      num_heads=self.num_heads, qkv_features=self.qkv_features)(query,
                                                                key_value)
    output = nn.Dense(self.qkv_features)(output)
    return dict(query=output, key_value=key_value)  # also works for tuples

class CrossAttentionNetwork(nn.Module):
  num_layers: Sequence[int]

  @nn.compact
  def __call__(self, x):
    return nn.Sequential([CrossAttentionBlock() for _ in
                          range(self.num_layers)])(query, key_value)
```

__call__(**args*, ***kwargs*)

Call self as a function.

Methods

Attention primitives

dot_product_attention_weights(query, key[, ...])	Computes dot-product attention weights given query and key.
dot_product_attention(query, key, value[, ...])	Computes dot-product attention given query, key, and value.
make_attention_mask(query_input, key_input)	Mask-making helper for attention weights.
make_causal_mask(x[, extra_batch_dims, dtype])	Make a causal mask for self-attention.

flax.linen.dot_product_attention_weights

flax.linen.**dot_product_attention_weights**(*query*, *key*, *bias=None*, *mask=None*, *broadcast_dropout=True*, *dropout_rng=None*, *dropout_rate=0.0*, *deterministic=False*, *dtype=None*, *precision=None*)

Computes dot-product attention weights given query and key.

Used by *dot_product_attention()*, which is what you'll most likely use. But if you want access to the attention weights for introspection, then you can directly call this function and call einsum yourself.

Parameters

- **query** – queries for calculating attention with shape of *[batch..., q_length, num_heads, qk_depth_per_head]*.
- **key** – keys for calculating attention with shape of *[batch..., kv_length, num_heads, qk_depth_per_head]*.
- **bias** – bias for the attention weights. This should be broadcastable to the shape *[batch..., num_heads, q_length, kv_length]*. This can be used for incorporating causal masks, padding masks, proximity bias, etc.
- **mask** – mask for the attention weights. This should be broadcastable to the shape *[batch..., num_heads, q_length, kv_length]*. This can be used for incorporating causal masks. Attention weights are masked out if their corresponding mask value is *False*.
- **broadcast_dropout** – bool: use a broadcasted dropout along batch dims.
- **dropout_rng** – JAX PRNGKey: to be used for dropout
- **dropout_rate** – dropout rate
- **deterministic** – bool, deterministic or not (to apply dropout)
- **dtype** – the dtype of the computation (default: infer from inputs and params)
- **precision** – numerical precision of the computation see *jax.lax.Precision* for details.

Returns

Output of shape *[batch..., num_heads, q_length, kv_length]*.

flax.linen.dot_product_attention

flax.linen.**dot_product_attention**(*query*, *key*, *value*, *bias=None*, *mask=None*, *broadcast_dropout=True*, *dropout_rng=None*, *dropout_rate=0.0*, *deterministic=False*, *dtype=None*, *precision=None*)

Computes dot-product attention given query, key, and value.

This is the core function for applying attention based on https://arxiv.org/abs/1706.03762. It calculates the attention weights given query and key and combines the values using the attention weights.

Note: query, key, value needn't have any batch dimensions.

Parameters

- **query** – queries for calculating attention with shape of *[batch..., q_length, num_heads, qk_depth_per_head]*.
- **key** – keys for calculating attention with shape of *[batch..., kv_length, num_heads, qk_depth_per_head]*.
- **value** – values to be used in attention with shape of *[batch..., kv_length, num_heads, v_depth_per_head]*.
- **bias** – bias for the attention weights. This should be broadcastable to the shape *[batch..., num_heads, q_length, kv_length]*. This can be used for incorporating causal masks, padding masks, proximity bias, etc.
- **mask** – mask for the attention weights. This should be broadcastable to the shape *[batch..., num_heads, q_length, kv_length]*. This can be used for incorporating causal masks. Attention weights are masked out if their corresponding mask value is *False*.
- **broadcast_dropout** – bool: use a broadcasted dropout along batch dims.
- **dropout_rng** – JAX PRNGKey: to be used for dropout
- **dropout_rate** – dropout rate
- **deterministic** – bool, deterministic or not (to apply dropout)
- **dtype** – the dtype of the computation (default: infer from inputs)
- **precision** – numerical precision of the computation see *jax.lax.Precision* for details.

Returns

Output of shape *[batch..., q_length, num_heads, v_depth_per_head]*.

flax.linen.make_attention_mask

flax.linen.**make_attention_mask**(*query_input*, *key_input*, *pairwise_fn=<PjitFunction of <function jax.numpy.multiply>>*, *extra_batch_dims=0*, *dtype=<class 'jax.numpy.float32'>*)

Mask-making helper for attention weights.

In case of 1d inputs (i.e., *[batch..., len_q]*, *[batch..., len_kv]*, the attention weights will be *[batch..., heads, len_q, len_kv]* and this function will produce *[batch..., 1, len_q, len_kv]*.

Parameters

- **query_input** – a batched, flat input of query_length size
- **key_input** – a batched, flat input of key_length size

- **pairwise_fn** – broadcasting elementwise comparison function
- **extra_batch_dims** – number of extra batch dims to add singleton axes for, none by default
- **dtype** – mask return dtype

Returns
A *[batch..., 1, len_q, len_kv]* shaped mask for 1d attention.

flax.linen.make_causal_mask

flax.linen.**make_causal_mask**(*x*, *extra_batch_dims=0*, *dtype=<class 'jax.numpy.float32'>*)

Make a causal mask for self-attention.

In case of 1d inputs (i.e., *[batch..., len]*, the self-attention weights will be *[batch..., heads, len, len]* and this function will produce a causal mask of shape *[batch..., 1, len, len]*.

Parameters

- **x** – input array of shape *[batch..., len]*
- **extra_batch_dims** – number of batch dims to add singleton axes for, none by default
- **dtype** – mask return dtype

Returns
A *[batch..., 1, len, len]* shaped causal mask for 1d attention.

SelfAttention(num_heads[, dtype, ...])	Self-attention special case of multi-head dot-product attention.
MultiHeadDotProductAttention(num_heads[, ...])	Multi-head dot-product attention.

flax.linen.SelfAttention

class flax.linen.**SelfAttention**(*num_heads*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *qkv_features=None*, *out_features=None*, *broadcast_dropout=True*, *dropout_rate=0.0*, *deterministic=None*, *precision=None*, *kernel_init=<function variance_scaling.<locals>.init>*, *bias_init=<function zeros>*, *use_bias=True*, *attention_fn=<function dot_product_attention>*, *decode=False*, *qkv_dot_general=<function dot_general>*, *out_dot_general=<function dot_general>*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Self-attention special case of multi-head dot-product attention.

__call__(*inputs_q*, *mask=None*, *deterministic=None*)

Applies multi-head dot product self-attention on the input data.

Projects the inputs into multi-headed query, key, and value vectors, applies dot-product attention and project the results to an output vector.

Parameters

- **inputs_q** – input queries of shape *[batch_sizes..., length, features]*.
- **mask** – attention mask of shape *[batch_sizes..., num_heads, query_length, key/value_length]*. Attention weights are masked out if their corresponding mask value is *False*.

- **deterministic** – if false, the attention weight is masked randomly using dropout, whereas if true, the attention weights are deterministic.

Returns
output of shape *[batch_sizes..., length, features]*.

Methods

flax.linen.MultiHeadDotProductAttention

class flax.linen.**MultiHeadDotProductAttention**(*num_heads*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *qkv_features=None*, *out_features=None*, *broadcast_dropout=True*, *dropout_rate=0.0*, *deterministic=None*, *precision=None*, *kernel_init=<function variance_scaling.<locals>.init>*, *bias_init=<function zeros>*, *use_bias=True*, *attention_fn=<function dot_product_attention>*, *decode=False*, *qkv_dot_general=<function dot_general>*, *out_dot_general=<function dot_general>*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Multi-head dot-product attention.

num_heads
number of attention heads. Features (i.e. inputs_q.shape[-1]) should be divisible by the number of heads.

Type
int

dtype
the dtype of the computation (default: infer from inputs and params)

Type
Optional[Any]

param_dtype
the dtype passed to parameter initializers (default: float32)

Type
Any

qkv_features
dimension of the key, query, and value.

Type
Optional[int]

out_features
dimension of the last projection

Type
Optional[int]

`broadcast_dropout`
bool: use a broadcasted dropout along batch dims.

Type
bool

`dropout_rate`
dropout rate

Type
float

`deterministic`
if false, the attention weight is masked randomly using dropout, whereas if true, the attention weights are deterministic.

Type
Optional[bool]

`precision`
numerical precision of the computation see *jax.lax.Precision* for details.

Type
Union[None, str, jax._src.lax.lax.Precision, Tuple[str, str], Tuple[jax._src.lax.lax.Precision, jax._src.lax.lax.Precision]]

`kernel_init`
initializer for the kernel of the Dense layers.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

`bias_init`
initializer for the bias of the Dense layers.

Type
Callable[[Any, Tuple[int, ...], Any], Any]

`use_bias`
bool: whether pointwise QKVO dense transforms use bias.

Type
bool

`attention_fn`
dot_product_attention or compatible function. Accepts query, key, value, and returns output of shape *[bs, dim1, dim2, ..., dimN,, num_heads, value_channels]`*

Type
Callable[[...], Any]

`decode`
whether to prepare and use an autoregressive cache.

Type
bool

`__call__`(*inputs_q*, *inputs_kv*, *mask=None*, *deterministic=None*)
Applies multi-head dot product attention on the input data.

Projects the inputs into multi-headed query, key, and value vectors, applies dot-product attention and project the results to an output vector.

Parameters

- **inputs_q** – input queries of shape *[batch_sizes..., length, features]*.
- **inputs_kv** – key/values of shape *[batch_sizes..., length, features]*.
- **mask** – attention mask of shape *[batch_sizes..., num_heads, query_length, key/value_length]*. Attention weights are masked out if their corresponding mask value is *False*.
- **deterministic** – if false, the attention weight is masked randomly using dropout, whereas if true, the attention weights are deterministic.

Returns

output of shape *[batch_sizes..., length, features]*.

Methods

Stochastic

Dropout(rate[, broadcast_dims, ...])	Create a dropout layer.

flax.linen.Dropout

class flax.linen.**Dropout**(*rate*, *broadcast_dims=()*, *deterministic=None*, *rng_collection='dropout'*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Create a dropout layer.

Note: When using *Module.apply()*, make sure to include an RNG seed named *'dropout'*. For example:

```
model.apply({'params': params}, inputs=inputs, train=True, rngs={'dropout': dropout_
↪rng})`
```

rate

the dropout probability. (_not_ the keep rate!)

Type
float

broadcast_dims

dimensions that will share the same dropout mask

Type
Sequence[int]

deterministic

if false the inputs are scaled by *1 / (1 - rate)* and masked, whereas if true, no mask is applied and the inputs are returned as is.

Type
Optional[bool]

rng_collection
the rng collection name to use when requesting an rng key.

Type
str

__call__(*inputs*, *deterministic=None*)
Applies a random dropout mask to the input.

Parameters

- **inputs** – the inputs that should be randomly masked.
- **deterministic** – if false the inputs are scaled by *1 / (1 - rate)* and masked, whereas if true, no mask is applied and the inputs are returned as is.

Returns
The masked inputs reweighted to preserve mean.

Methods

RNN primitives

LSTMCell([gate_fn, activation_fn, ...])	LSTM cell.
OptimizedLSTMCell([gate_fn, activation_fn, ...])	More efficient LSTM Cell that concatenates state components before matmul.
GRUCell([gate_fn, activation_fn, ...])	GRU cell.
RNNCellBase([parent, name])	RNN cell base class.
RNN(cell, cell_size[, time_major, ...])	The RNN module takes any *RNNCellBase* instance and applies it over a sequence using *flax.linen.scan()*.
Bidirectional(forward_rnn, backward_rnn[, ...])	Processes the input in both directions and merges the results.

flax.linen.LSTMCell

class flax.linen.**LSTMCell**(*gate_fn=<PjitFunction of <function sigmoid>>*, *activation_fn=<PjitFunction of <function jax.numpy.tanh>>*, *kernel_init=<function variance_scaling.<locals>.init>*, *recurrent_kernel_init=<function orthogonal.<locals>.init>*, *bias_init=<function zeros>*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

LSTM cell.

The mathematical definition of the cell is as follows

$$
\begin{aligned}
i &= \sigma(W_{ii}x + W_{hi}h + b_{hi}) \\
f &= \sigma(W_{if}x + W_{hf}h + b_{hf}) \\
g &= \tanh(W_{ig}x + W_{hg}h + b_{hg}) \\
o &= \sigma(W_{io}x + W_{ho}h + b_{ho}) \\
c' &= f * c + i * g \\
h' &= o * \tanh(c')
\end{aligned}
$$

where x is the input, h is the output of the previous time step, and c is the memory.

`gate_fn`

activation function used for gates (default: sigmoid)

Type
Callable[[...], Any]

`activation_fn`

activation function used for output and memory update (default: tanh).

Type
Callable[[...], Any]

`kernel_init`

initializer function for the kernels that transform the input (default: lecun_normal).

Type
Callable[[Any, Tuple[int, ...], Any], jax.Array]

`recurrent_kernel_init`

initializer function for the kernels that transform the hidden state (default: initializers.orthogonal()).

Type
Callable[[Any, Tuple[int, ...], Any], jax.Array]

`bias_init`

initializer for the bias parameters (default: initializers.zeros_init())

Type
Callable[[Any, Tuple[int, ...], Any], jax.Array]

`dtype`

the dtype of the computation (default: infer from inputs and params).

Type
Optional[Any]

`param_dtype`

the dtype passed to parameter initializers (default: float32).

Type
Any

`__call__`(*carry*, *inputs*)

A long short-term memory (LSTM) cell.

Parameters

- **`carry`** – the hidden state of the LSTM cell, initialized using *LSTMCell.initialize_carry*.
- **`inputs`** – an ndarray with the input for the current time step. All dimensions except the final are considered batch dimensions.

Returns
A tuple with the new carry and the output.

Methods

`initialize_carry`(rng, batch_dims, size[, ...])	Initialize the RNN cell carry.

flax.linen.OptimizedLSTMCell

class flax.linen.**OptimizedLSTMCell**(*gate_fn=<PjitFunction of <function sigmoid>>*, *activation_fn=<PjitFunction of <function jax.numpy.tanh>>*, *kernel_init=<function variance_scaling.<locals>.init>*, *recurrent_kernel_init=<function orthogonal.<locals>.init>*, *bias_init=<function zeros>*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

More efficient LSTM Cell that concatenates state components before matmul.

The parameters are compatible with *LSTMCell*. Note that this cell is often faster than *LSTMCell* as long as the hidden size is roughly <= 2048 units.

The mathematical definition of the cell is the same as *LSTMCell* and as follows

$$
\begin{aligned}
i &= \sigma(W_{ii}x + W_{hi}h + b_{hi}) \\
f &= \sigma(W_{if}x + W_{hf}h + b_{hf}) \\
g &= \tanh(W_{ig}x + W_{hg}h + b_{hg}) \\
o &= \sigma(W_{io}x + W_{ho}h + b_{ho}) \\
c' &= f * c + i * g \\
h' &= o * \tanh(c')
\end{aligned}
$$

where x is the input, h is the output of the previous time step, and c is the memory.

gate_fn

activation function used for gates (default: sigmoid).

Type
Callable[[...], Any]

activation_fn

activation function used for output and memory update (default: tanh).

Type
Callable[[...], Any]

kernel_init

initializer function for the kernels that transform the input (default: lecun_normal).

Type
Callable[[Any, Tuple[int, ...], Any], jax.Array]

recurrent_kernel_init

initializer function for the kernels that transform the hidden state (default: initializers.orthogonal()).

Type
Callable[[Any, Tuple[int, ...], Any], jax.Array]

bias_init

initializer for the bias parameters (default: initializers.zeros_init()).

Type
Callable[[Any, Tuple[int, ...], Any], jax.Array]

dtype

the dtype of the computation (default: infer from inputs and params).

Type
Optional[Any]

param_dtype

the dtype passed to parameter initializers (default: float32).

Type
Any

__call__(*carry*, *inputs*)

An optimized long short-term memory (LSTM) cell.

Parameters

- **carry** – the hidden state of the LSTM cell, initialized using *LSTMCell.initialize_carry*.
- **inputs** – an ndarray with the input for the current time step. All dimensions except the final are considered batch dimensions.

Returns
A tuple with the new carry and the output.

Methods

initialize_carry(rng, batch_dims, size[, ...])	Initialize the RNN cell carry.

flax.linen.GRUCell

class flax.linen.**GRUCell**(*gate_fn=<PjitFunction of <function sigmoid>>*, *activation_fn=<PjitFunction of <function jax.numpy.tanh>>*, *kernel_init=<function variance_scaling.<locals>.init>*, *recurrent_kernel_init=<function orthogonal.<locals>.init>*, *bias_init=<function zeros>*, *dtype=None*, *param_dtype=<class 'jax.numpy.float32'>*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

GRU cell.

The mathematical definition of the cell is as follows

$$
\begin{aligned}
r &= \sigma(W_{ir}x + W_{hr}h + b_{hr}) \\
z &= \sigma(W_{iz}x + W_{hz}h + b_{hz}) \\
n &= \tanh(W_{in}x + b_{in} + r * (W_{hn}h + b_{hn})) \\
h' &= (1 - z) * n + z * h
\end{aligned}
$$

where x is the input and h, is the output of the previous time step.

gate_fn

activation function used for gates (default: sigmoid)

Type
Callable[[...], Any]

`activation_fn`

activation function used for output and memory update (default: tanh).

Type
Callable[[...], Any]

`kernel_init`

initializer function for the kernels that transform the input (default: lecun_normal).

Type
Callable[[Any, Tuple[int, ...], Any], jax.Array]

`recurrent_kernel_init`

initializer function for the kernels that transform the hidden state (default: initializers.orthogonal()).

Type
Callable[[Any, Tuple[int, ...], Any], jax.Array]

`bias_init`

initializer for the bias parameters (default: initializers.zeros_init())

Type
Callable[[Any, Tuple[int, ...], Any], jax.Array]

`dtype`

the dtype of the computation (default: None).

Type
Optional[Any]

`param_dtype`

the dtype passed to parameter initializers (default: float32).

Type
Any

`__call__`(*carry*, *inputs*)

Gated recurrent unit (GRU) cell.

Parameters

- **`carry`** – the hidden state of the GRU cell, initialized using *GRUCell.initialize_carry*.
- **`inputs`** – an ndarray with the input for the current time step. All dimensions except the final are considered batch dimensions.

Returns
A tuple with the new carry and the output.

Methods

`initialize_carry`(rng, batch_dims, size[, ...])	Initialize the RNN cell carry.

flax.linen.RNNCellBase

class flax.linen.**RNNCellBase**(*parent=<flax.linen.module._Sentinel object>*, *name=None*)

RNN cell base class.

__call__(***kwargs*)

Call self as a function.

Methods

initialize_carry(rng, batch_dims, size[, ...])	Initialize the RNN cell carry.

flax.linen.RNN

class flax.linen.**RNN**(*cell*, *cell_size*, *time_major=False*, *return_carry=False*, *reverse=False*, *keep_order=False*, *unroll=1*, *variable_axes=FrozenDict({})*, *variable_broadcast='params'*, *variable_carry=False*, *split_rngs=FrozenDict({ params: False, })*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

The RNN module takes any *RNNCellBase* instance and applies it over a sequence using *flax.linen.scan()*.

Example:

```
>>> import jax.numpy as jnp
>>> import jax
>>> import flax.linen as nn
...
>>> x = jnp.ones((10, 50, 32)) # (batch, time, features)
>>> lstm = nn.RNN(nn.LSTMCell(), cell_size=64)
>>> variables = lstm.init(jax.random.PRNGKey(0), x)
>>> y = lstm.apply(variables, x)
>>> y.shape # (batch, time, cell_size)
(10, 50, 64)
```

As shown above, RNN uses the `cell_size` argument to set the `size` argument for the cell's `initialize_carry` method, in practice this is typically the number of hidden units you want for the cell. However, this may vary depending on the cell you are using, for example the `ConvLSTMCell` requires a `size` argument of the form `(kernel_height, kernel_width, features)`:

```
>>> x = jnp.ones((10, 50, 32, 32, 3)) # (batch, time, height, width, features)
>>> conv_lstm = nn.RNN(nn.ConvLSTMCell(64, kernel_size=(3, 3)), cell_size=(32, 32,␣
↪64))
>>> y, variables = conv_lstm.init_with_output(jax.random.PRNGKey(0), x)
>>> y.shape # (batch, time, height, width, features)
(10, 50, 32, 32, 64)
```

By default RNN expect the time dimension after the batch dimension (`(*batch, time, *features)`), if you set `time_major=True` RNN will instead expect the time dimesion to be at the beginning (`(time, *batch, *features)`):

```
>>> x = jnp.ones((50, 10, 32)) # (time, batch, features)
>>> lstm = nn.RNN(nn.LSTMCell(), cell_size=64, time_major=True)
```

(continues on next page)

(continued from previous page)

```
>>> variables = lstm.init(jax.random.PRNGKey(0), x)
>>> y = lstm.apply(variables, x)
>>> y.shape # (time, batch, cell_size)
(50, 10, 64)
```

The output is an array of shape `(*batch, time, *cell_size)` by default (typically), however if you set `return_carry=True` it will instead return a tuple of the final carry and the output:

```
>>> x = jnp.ones((10, 50, 32)) # (batch, time, features)
>>> lstm = nn.RNN(nn.LSTMCell(), cell_size=64, return_carry=True)
>>> variables = lstm.init(jax.random.PRNGKey(0), x)
>>> carry, y = lstm.apply(variables, x)
>>> jax.tree_map(jnp.shape, carry) # ((batch, cell_size), (batch, cell_size))
((10, 64), (10, 64))
>>> y.shape # (batch, time, cell_size)
(10, 50, 64)
```

To support variable length sequences, you can pass a `seq_lengths` which is an integer array of shape `(*batch)` where each element is the length of the sequence in the batch. For example:

```
>>> seq_lengths = jnp.array([3, 2, 5])
```

The output elements corresponding to padding elements are NOT zeroed out. If `return_carry` is set to `True` the carry will be the state of the last valid element of each sequence.

RNN also accepts some of the arguments of *flax.linen.scan()*, by default they are set to work with cells like *LSTMCell* and *GRUCell* but they can be overriden as needed. Overriding default values to scan looks like this:

```
>>> lstm = nn.RNN(
...   nn.LSTMCell(), cell_size=64,
...   unroll=1, variable_axes={}, variable_broadcast='params',
...   variable_carry=False, split_rngs={'params': False})
```

cell

an instance of *RNNCellBase*.

Type
flax.linen.recurrent.RNNCellBase

cell_size

the size of the cell as requested by `RNNCellBase.initialize_carry()`, it can be an integer or a tuple of integers.

Type
Union[int, Tuple[int, ...]]

time_major

if `time_major=False` (default) it will expect inputs with shape `(*batch, time, *features)`, else it will expect inputs with shape `(time, *batch, *features)`.

Type
bool

return_carry

if `return_carry=False` (default) only the output sequence is returned, else it will return a tuple of the final carry and the output sequence.

Type
bool

`reverse`

if `reverse=False` (default) the sequence is processed from left to right and returned in the original order, else it will be processed from right to left, and returned in reverse order. If `seq_lengths` is passed, padding will always remain at the end of the sequence.

Type
bool

`keep_order`

if `keep_order=True`, when `reverse=True` the output will be reversed back to the original order after processing, this is useful to align sequences in bidirectional RNNs. If `keep_order=False` (default), the output will remain in the order specified by `reverse`.

Type
bool

`unroll`

how many scan iterations to unroll within a single iteration of a loop, defaults to 1. This argument will be passed to *nn.scan*.

Type
int

`variable_axes`

a dictionary mapping each collection to either an integer *i* (meaning we scan over dimension *i*) or *None* (replicate rather than scan). This argument is forwarded to *nn.scan*.

Type
Mapping[Union[bool, str, Collection[str], DenyList], Union[int, flax.core.lift.In[int], flax.core.lift.Out[int]]]

`variable_broadcast`

Specifies the broadcasted variable collections. A broadcasted variable should not depend on any computation that cannot be lifted out of the loop. This is typically used to define shared parameters inside the fn. This argument is forwarded to *nn.scan*.

Type
Union[bool, str, Collection[str], DenyList]

`variable_carry`

Specifies the variable collections that are carried through the loop. Mutations to these variables are carried to the next iteration and will be preserved when the scan finishes. This argument is forwarded to *nn.scan*.

Type
Union[bool, str, Collection[str], DenyList]

`split_rngs`

a mapping from PRNGSequenceFilter to bool specifying whether a collection's PRNG key should be split such that its values are different at each step, or replicated such that its values remain the same at each step. This argument is forwarded to *nn.scan*.

Type
Mapping[Union[bool, str, Collection[str], DenyList], bool]

`__call__`(*inputs*, *, *initial_carry=None*, *init_key=None*, *seq_lengths=None*, *return_carry=None*, *time_major=None*, *reverse=None*, *keep_order=None*)

Applies the RNN to the inputs.

`__call__` allows you to optionally override some attributes like `return_carry` and `time_major` defined in the constructor.

Parameters

- **inputs** – the input sequence.
- **initial_carry** – the initial carry, if not provided it will be initialized using the cell's `RNNCellBase.initialize_carry()` method.
- **init_key** – a PRNG key used to initialize the carry, if not provided `jax.random.PRNGKey(0)` will be used. Most cells will ignore this argument.
- **seq_lengths** – an optional integer array of shape `(*batch)` indicating the length of each sequence, elements whose index in the time dimension is greater than the corresponding length will be considered padding and will be ignored.
- **return_carry** – if `return_carry=False` (default) only the output sequence is returned, else it will return a tuple of the final carry and the output sequence.
- **time_major** – if `time_major=False` (default) it will expect inputs with shape `(*batch, time, *features)`, else it will expect inputs with shape `(time, *batch, *features)`.
- **reverse** – overrides the `reverse` attribute, if `reverse=False` (default) the sequence is processed from left to right and returned in the original order, else it will be processed from right to left, and returned in reverse order. If `seq_lengths` is passed, padding will always remain at the end of the sequence.
- **keep_order** – overrides the `keep_order` attribute, if `keep_order=True`, when `reverse=True` the output will be reversed back to the original order after processing, this is useful to align sequences in bidirectional RNNs. If `keep_order=False` (default), the output will remain in the order specified by `reverse`.

Returns

if `return_carry=False` (default) only the output sequence is returned, else it will return a tuple of the final carry and the output sequence.

Methods

flax.linen.Bidirectional

class flax.linen.**Bidirectional**(*forward_rnn*, *backward_rnn*, *merge_fn=<function _concatenate>*, *time_major=False*, *return_carry=False*, *parent=<flax.linen.module._Sentinel object>*, *name=None*)

Processes the input in both directions and merges the results.

__call__(*inputs*, *, *initial_carry=None*, *init_key=None*, *seq_lengths=None*, *return_carry=None*, *time_major=None*, *reverse=None*, *keep_order=None*)

Call self as a function.

Methods

5.8.6 flax.serialization package

Serialization utilities for Jax.

All Flax classes that carry state (e.g., Optimizer) can be turned into a state dict of numpy arrays for easy serialization.

State dicts

flax.serialization.**from_state_dict**(*target*, *state*, *name='.'*)

Restores the state of the given target using a state dict.

This function takes the current target as an argument. This lets us know the exact structure of the target, as well as lets us add assertions that shapes and dtypes don't change.

In practice, none of the leaf values in *target* are actually used. Only the tree structure, shapes and dtypes.

Parameters

- **target** – the object of which the state should be restored.
- **state** – a dictionary generated by *to_state_dict* with the desired new state for *target*.
- **name** – name of branch taken, used to improve deserialization error messages.

Returns

A copy of the object with the restored state.

flax.serialization.**to_state_dict**(*target*)

Returns a dictionary with the state of the given target.

flax.serialization.**register_serialization_state**(*ty*, *ty_to_state_dict*, *ty_from_state_dict*, *override=False*)

Register a type for serialization.

Parameters

- **ty** – the type to be registered
- **ty_to_state_dict** – a function that takes an instance of ty and returns its state as a dictionary.
- **ty_from_state_dict** – a function that takes an instance of ty and a state dict, and returns a copy of the instance with the restored state.
- **override** – override a previously registered serialization handler (default: False).

Serialization with MessagePack

flax.serialization.**msgpack_serialize**(*pytree*, *in_place=False*)

Save data structure to bytes in msgpack format.

Low-level function that only supports python trees with array leaves, for custom objects use *to_bytes*. It splits arrays above MAX_CHUNK_SIZE into multiple chunks.

Parameters

- **pytree** – python tree of dict, list, tuple with python primitives and array leaves.
- **in_place** – boolean specifying if pytree should be modified in place.

Returns

msgpack-encoded bytes of pytree.

flax.serialization.**msgpack_restore**(*encoded_pytree*)

Restore data structure from bytes in msgpack format.

Low-level function that only supports python trees with array leaves, for custom objects use *from_bytes*.

Parameters

encoded_pytree – msgpack-encoded bytes of python tree.

Returns

Python tree of dict, list, tuple with python primitive and array leaves.

flax.serialization.**to_bytes**(*target*)

Save optimizer or other object as msgpack-serialized state-dict.

Parameters

target – template object with state-dict registrations to be serialized to msgpack format. Typically a flax model or optimizer.

Returns

Bytes of msgpack-encoded state-dict of *target* object.

flax.serialization.**from_bytes**(*target*, *encoded_bytes*)

Restore optimizer or other object from msgpack-serialized state-dict.

Parameters

- **target** – template object with state-dict registrations that matches the structure being deserialized from *encoded_bytes*.
- **encoded_bytes** – msgpack serialized object structurally isomorphic to *target*. Typically a flax model or optimizer.

Returns

A new object structurally isomorphic to *target* containing the updated leaf data from saved data.

5.8.7 flax.struct package

Utilities for defining custom classes that can be used with jax transformations.

`flax.struct.dataclass(clz)`

Create a class which can be passed to functional transformations.

NOTE: Inherit from `PyTreeNode` instead to avoid type checking issues when using PyType.

Jax transformations such as *jax.jit* and *jax.grad* require objects that are immutable and can be mapped over using the *jax.tree_util* methods. The *dataclass* decorator makes it easy to define custom classes that can be passed safely to Jax. For example:

```
from flax import struct

@struct.dataclass
class Model:
  params: Any
  # use pytree_node=False to indicate an attribute should not be touched
  # by Jax transformations.
  apply_fn: FunctionType = struct.field(pytree_node=False)

  def __apply__(self, *args):
    return self.apply_fn(*args)

model = Model(params, apply_fn)

model.params = params_b  # Model is immutable. This will raise an error.
model_b = model.replace(params=params_b)  # Use the replace method instead.

# This class can now be used safely in Jax to compute gradients w.r.t. the
# parameters.
model = Model(params, apply_fn)
model_grad = jax.grad(some_loss_fn)(model)
```

Note that dataclasses have an auto-generated `__init__` where the arguments of the constructor and the attributes of the created instance match 1:1. This correspondence is what makes these objects valid containers that work with JAX transformations and more generally the *jax.tree_util* library.

Sometimes a "smart constructor" is desired, for example because some of the attributes can be (optionally) derived from others. The way to do this with Flax dataclasses is to make a static or class method that provides the smart constructor. This way the simple constructor used by *jax.tree_util* is preserved. Consider the following example:

```
@struct.dataclass
class DirectionAndScaleKernel:
  direction: Array
  scale: Array

  @classmethod
  def create(cls, kernel):
    scale = jax.numpy.linalg.norm(kernel, axis=0, keepdims=True)
    directin = direction / scale
    return cls(direction, scale)
```

Parameters
clz – the class that will be transformed by the decorator.

Returns
The new class.

`class flax.struct.PyTreeNode(*args, **kwargs)`

Base class for dataclasses that should act like a JAX pytree node.

See `flax.struct.dataclass` for the `jax.tree_util` behavior. This base class additionally avoids type checking errors when using PyType.

Example:

```
from flax import struct

class Model(struct.PyTreeNode):
  params: Any
  # use pytree_node=False to indicate an attribute should not be touched
  # by Jax transformations.
  apply_fn: FunctionType = struct.field(pytree_node=False)

  def __apply__(self, *args):
    return self.apply_fn(*args)

model = Model(params, apply_fn)

model.params = params_b  # Model is immutable. This will raise an error.
model_b = model.replace(params=params_b)  # Use the replace method instead.

# This class can now be used safely in Jax to compute gradients w.r.t. the
# parameters.
model = Model(params, apply_fn)
model_grad = jax.grad(some_loss_fn)(model)
```

5.8.8 flax.traceback_util package

Flax specific traceback_util functions.

Traceback filtering utils

`flax.traceback_util.hide_flax_in_tracebacks()`

Hides Flax internal stack frames in tracebacks.

`flax.traceback_util.show_flax_in_tracebacks()`

Shows Flax internal stack frames in tracebacks.

5.8.9 flax.training package

Checkpoints

Checkpointing helper functions.

Handles saving and restoring optimizer checkpoints based on step-number or other numerical metric in filename. Cleans up older / worse-performing checkpoint files.

flax.training.checkpoints.**save_checkpoint**(*ckpt_dir*, *target*, *step*, *prefix='checkpoint_'*, *keep=1*, *overwrite=False*, *keep_every_n_steps=None*, *async_manager=None*, *orbax_checkpointer=None*)

Save a checkpoint of the model. Suitable for single-host.

In this method, every JAX process saves the checkpoint on its own. Do not use it if you have multiple processes and you intend for them to save data to a common directory (e.g., a GCloud bucket). To save multi-process checkpoints to a shared storage or to save *GlobalDeviceArray`s, use `save_checkpoint_multiprocess()* instead.

Pre-emption safe by writing to temporary before a final rename and cleanup of past files. However, if async_manager is used, the final commit will happen inside an async callback, which can be explicitly waited by calling *async_manager.wait_previous_save()*.

Parameters

- **ckpt_dir** – str or pathlib-like path to store checkpoint files in.
- **target** – serializable flax object, usually a flax optimizer.
- **step** – int or float: training step number or other metric number.
- **prefix** – str: checkpoint file name prefix.
- **keep** – number of past checkpoint files to keep.
- **overwrite** – overwrite existing checkpoint files if a checkpoint at the current or a later step already exits (default: False).
- **keep_every_n_steps** – if defined, keep every checkpoints every n steps (in addition to keeping the last 'keep' checkpoints).
- **async_manager** – if defined, the save will run without blocking the main thread. Only works for single host. Note that an ongoing save will still block subsequent saves, to make sure overwrite/keep logic works correctly.
- **orbax_checkpointer** – if defined, the save will be done by Orbax. In the future, all Flax checkpointing features will be migrated to Orbax, and starting to use an *orbax_checkpointer* is recommended. Please check out the checkpointing guide (https://flax.readthedocs.io/en/latest/guides/use_checkpointing.html#save-checkpoints) for how to use Orbax checkpointers.

Returns

Filename of saved checkpoint.

flax.training.checkpoints.**save_checkpoint_multiprocess**(*ckpt_dir*, *target*, *step*, *prefix='checkpoint_'*, *keep=1*, *overwrite=False*, *keep_every_n_steps=None*, *async_manager=None*, *gda_manager=None*, *orbax_checkpointer=None*)

Save a checkpoint of the model in multi-process environment.

Use this method to save `GlobalDeviceArray`s, or to save data to a common directory. Only process 0 will save the main checkpoint file and remove old checkpoint files.

Pre-emption safe by writing to temporary before a final rename and cleanup of past files. However, if async_manager or gda_manager is used, the final commit will happen inside an async callback, which can be explicitly waited by calling *async_manager.wait_previous_save()* or *gda_manager.wait_until_finished()*.

Parameters

- **ckpt_dir** – str or pathlib-like path to store checkpoint files in.
- **target** – serializable flax object, usually a flax optimizer.
- **step** – int or float: training step number or other metric number.
- **prefix** – str: checkpoint file name prefix.
- **keep** – number of past checkpoint files to keep.
- **overwrite** – overwrite existing checkpoint files if a checkpoint at the current or a later step already exits (default: False).
- **keep_every_n_steps** – if defined, keep every checkpoints every n steps (in addition to keeping the last 'keep' checkpoints).
- **async_manager** – if defined, the save will run without blocking the main thread. Only works for single host. Note that an ongoing save will still block subsequent saves, to make sure overwrite/keep logic works correctly.
- **gda_manager** – required if target contains a JAX GlobalDeviceArray. Type should be GlobalAsyncCheckpointManager (needs Tensorstore to be imported correctly). Will save the GDAs to a separate subdirectory with postfix "_gda" asynchronously. Same as async_manager, this will block subsequent saves.
- **orbax_checkpointer** – if defined, the save will be done by Orbax. In the future, all Flax checkpointing features will be migrated to Orbax, and starting to use an *orbax_checkpointer* is recommended. Please check out the checkpointing guide (https://flax.readthedocs.io/en/latest/guides/use_checkpointing.html#save-checkpoints) for how to use Orbax checkpointers.

Returns

Filename of saved checkpoint.

`flax.training.checkpoints.`**latest_checkpoint**(*ckpt_dir*, *prefix='checkpoint_'*)

Retrieve the path of the latest checkpoint in a directory.

Parameters

- **ckpt_dir** – str: directory of checkpoints to restore from.
- **prefix** – str: name prefix of checkpoint files.

Returns

The latest checkpoint path or None if no checkpoints were found.

`flax.training.checkpoints.`**restore_checkpoint**(*ckpt_dir*, *target*, *step=None*, *prefix='checkpoint_'*, *parallel=True*, *gda_manager=None*, *allow_partial_mpa_restoration=False*, *orbax_checkpointer=None*, *orbax_transforms=None*)

Restore last/best checkpoint from checkpoints in path.

Sorts the checkpoint files naturally, returning the highest-valued file, e.g.:

- `ckpt_1, ckpt_2, ckpt_3 --> ckpt_3`
- `ckpt_0.01, ckpt_0.1, ckpt_0.001 --> ckpt_0.1`

- `ckpt_-1.0, ckpt_1.0, ckpt_1e5 --> ckpt_1e5`

Parameters

- **ckpt_dir** – str: checkpoint file or directory of checkpoints to restore from.
- **target** – matching object to rebuild via deserialized state-dict. If None, the deserialized state-dict is returned as-is.
- **step** – int or float: step number to load or None to load latest. If specified, ckpt_dir must be a directory.
- **prefix** – str: name prefix of checkpoint files.
- **parallel** – bool: whether to load seekable checkpoints in parallel, for speed.
- **gda_manager** – required if checkpoint contains a multiprocess array (GlobalDeviceArray or jax Array from pjit). Type should be GlobalAsyncCheckpointManager (needs Tensorstore to be imported correctly). Will read the arrays from the separate subdirectory with postfix "_gda".
- **allow_partial_mpa_restoration** – If true, the given *target* doesn't have to contain all valid multiprocess arrays. As a result, the restored Pytree may have some MPAs not restored correctly. Use this if you cannot provide a fully valid `target` and don't need all the MPAs in the checkpoint to be restored.
- **orbax_checkpointer** – the *Orbax.Checkpointer* that handles the underlying restore, if the given checkpoint is saved with Orbax.
- **orbax_transforms** – the Orbax transformations that will be passed into *orbax_checkpointer.restore()* call.

Returns

Restored *target* updated from checkpoint file, or if no step specified and no checkpoint files present, returns the passed-in *target* unchanged. If a file path is specified and is not found, the passed-in *target* will be returned. This is to match the behavior of the case where a directory path is specified but the directory has not yet been created.

`flax.training.checkpoints.`**convert_pre_linen**(*params*)

Converts a pre-Linen parameter pytree.

In pre-Linen API submodules were numbered incrementally, independent of the submodule class. With Linen this behavior has changed to keep separate submodule counts per module class.

Consider the following module:

```
class Model(nn.Module):
  @nn.compact
  def __call__(self, x):
    x = nn.Conv(1, 1)(x)
    x = nn.Dense(1)(x)
    return x
```

In pre-Linen the resulting params would have had the structure:

```
{'Conv_0': { ... }, 'Dense_1': { ... } }
```

With Linen the resulting params would instead have had the structure:

```
{'Conv_0': { ... }, 'Dense_0': { ... } }
```

To convert from pre-Linen format to Linen simply call:

```
params = convert_pre_linen(pre_linen_params)
```

Note that you can also use this utility to convert pre-Linen collections because they're following the same module naming. Note though that collections were "flat" in pre-Linen and first need to be unflattened before they can be used with this function:

```
batch_stats = convert_pre_linen(flax.traverse_util.unflatten_dict({
    tuple(k.split('/')[1:]): v
    for k, v in pre_linen_model_state.as_dict().items()
}))
```

Then Linen variables can be defined from these converted collections:

```
variables = {'params': params, 'batch_stats': batch_stats}
```

Parameters

params – Parameter pytree in pre-Linen format. If the pytree is already in Linen format, then the returned pytree is unchanged (i.e. this function can safely be called on any loaded checkpoint for use with Linen).

Returns

Parameter pytree with Linen submodule naming.

Learning rate schedules

Learning rate schedules used in FLAX image classification examples.

Note that with FLIP #1009 learning rate schedules in `flax.training` are **effectively deprecated** in favor of Optax schedules. Please refer to Optimizer Schedules for more information.

`flax.training.lr_schedule.`**`create_constant_learning_rate_schedule`**(*base_learning_rate*, *steps_per_epoch*, *warmup_length=0.0*)

Create a constant learning rate schedule with optional warmup.

Note that with FLIP #1009 learning rate schedules in `flax.training` are **effectively deprecated** in favor of Optax schedules. Please refer to Optimizer Schedules for more information.

Holds the learning rate constant. This function also offers a learing rate warmup as per https://arxiv.org/abs/1706.02677, for the purpose of training with large mini-batches.

Parameters

- **base_learning_rate** – the base learning rate
- **steps_per_epoch** – the number of iterations per epoch
- **warmup_length** – if > 0, the learning rate will be modulated by a warmup factor that will linearly ramp-up from 0 to 1 over the first *warmup_length* epochs

Returns

Function *f(step) -> lr* that computes the learning rate for a given step.

`flax.training.lr_schedule.`**`create_stepped_learning_rate_schedule`**(*base_learning_rate*, *steps_per_epoch*, *lr_sched_steps*, *warmup_length=0.0*)

Create a stepped learning rate schedule with optional warmup.

Note that with FLIP #1009 learning rate schedules in `flax.training` are **effectively deprecated** in favor of Optax schedules. Please refer to Optimizer Schedules for more information.

A stepped learning rate schedule decreases the learning rate by specified amounts at specified epochs. The steps are given as the *lr_sched_steps* parameter. A common ImageNet schedule decays the learning rate by a factor of 0.1 at epochs 30, 60 and 80. This would be specified as:

```
[
  [30, 0.1],
  [60, 0.01],
  [80, 0.001]
]
```

This function also offers a learing rate warmup as per https://arxiv.org/abs/1706.02677, for the purpose of training with large mini-batches.

Parameters

- **base_learning_rate** – the base learning rate
- **steps_per_epoch** – the number of iterations per epoch
- **lr_sched_steps** – the schedule as a list of steps, each of which is a *[epoch, lr_factor]* pair; the step occurs at epoch *epoch* and sets the learning rate to *base_learning_rage * lr_factor*
- **warmup_length** – if > 0, the learning rate will be modulated by a warmup factor that will linearly ramp-up from 0 to 1 over the first *warmup_length* epochs

Returns

Function *f(step) -> lr* that computes the learning rate for a given step.

`flax.training.lr_schedule.`**create_cosine_learning_rate_schedule**(*base_learning_rate*, *steps_per_epoch*, *halfcos_epochs*, *warmup_length=0.0*)

Create a cosine learning rate schedule with optional warmup.

Note that with FLIP #1009 learning rate schedules in `flax.training` are **effectively deprecated** in favor of Optax schedules. Please refer to Optimizer Schedules for more information.

A cosine learning rate schedule modules the learning rate with half a cosine wave, gradually scaling it to 0 at the end of training.

This function also offers a learing rate warmup as per https://arxiv.org/abs/1706.02677, for the purpose of training with large mini-batches.

Parameters

- **base_learning_rate** – the base learning rate
- **steps_per_epoch** – the number of iterations per epoch
- **halfcos_epochs** – the number of epochs to complete half a cosine wave; normally the number of epochs used for training
- **warmup_length** – if > 0, the learning rate will be modulated by a warmup factor that will linearly ramp-up from 0 to 1 over the first *warmup_length* epochs

Returns

Function *f(step) -> lr* that computes the learning rate for a given step.

Train state

class flax.training.train_state.**TrainState**(*step*, *apply_fn*, *params*, *tx*, *opt_state*)

Simple train state for the common case with a single Optax optimizer.

Synopsis:

```
state = TrainState.create(
    apply_fn=model.apply,
    params=variables['params'],
    tx=tx)
grad_fn = jax.grad(make_loss_fn(state.apply_fn))
for batch in data:
  grads = grad_fn(state.params, batch)
  state = state.apply_gradients(grads=grads)
```

Note that you can easily extend this dataclass by subclassing it for storing additional data (e.g. additional variable collections).

For more exotic usecases (e.g. multiple optimizers) it's probably best to fork the class and modify it.

Parameters

- **step** – Counter starts at 0 and is incremented by every call to *.apply_gradients()*.
- **apply_fn** – Usually set to *model.apply()*. Kept in this dataclass for convenience to have a shorter params list for the *train_step()* function in your training loop.
- **params** – The parameters to be updated by *tx* and used by *apply_fn*.
- **tx** – An Optax gradient transformation.
- **opt_state** – The state for *tx*.

apply_gradients(*, *grads*, ***kwargs*)

Updates *step*, *params*, *opt_state* and ***kwargs* in return value.

Note that internally this function calls *.tx.update()* followed by a call to *optax.apply_updates()* to update *params* and *opt_state*.

Parameters

- **grads** – Gradients that have the same pytree structure as *.params*.
- ****kwargs** – Additional dataclass attributes that should be *.replace()*-ed.

Returns

An updated instance of *self* with *step* incremented by one, *params* and *opt_state* updated by applying *grads*, and additional attributes replaced as specified by *kwargs*.

classmethod create(*, *apply_fn*, *params*, *tx*, ***kwargs*)

Creates a new instance with *step=0* and initialized *opt_state*.

Early Stopping

class flax.training.early_stopping.**EarlyStopping**(*min_delta=0*, *patience=0*, *best_metric=inf*, *patience_count=0*, *should_stop=False*)

Early stopping to avoid overfitting during training.

The following example stops training early if the difference between losses recorded in the current epoch and previous epoch is less than 1e-3 consecutively for 2 times:

```
early_stop = EarlyStopping(min_delta=1e-3, patience=2)
for epoch in range(1, num_epochs+1):
  rng, input_rng = jax.random.split(rng)
  optimizer, train_metrics = train_epoch(
      optimizer, train_ds, config.batch_size, epoch, input_rng)
  _, early_stop = early_stop.update(train_metrics['loss'])
  if early_stop.should_stop:
    print('Met early stopping criteria, breaking...')
    break
```

min_delta

Minimum delta between updates to be considered an improvement.

Type
float

patience

Number of steps of no improvement before stopping.

Type
int

best_metric

Current best metric value.

Type
float

patience_count

Number of steps since last improving update.

Type
int

should_stop

Whether the training loop should stop to avoid overfitting.

Type
bool

update(*metric*)

Update the state based on metric.

Returns
A pair (has_improved, early_stop), where *has_improved* is True when there was an improvement greater than *min_delta* from the previous *best_metric* and *early_stop* is the updated *EarlyStop* object.

Common Utilities

flax.training.common_utils.**shard**(*xs*)

Helper for pmap to shard a pytree of arrays by local_device_count.

Parameters
xs – a pytree of arrays.

Returns
A matching pytree with arrays' leading dimensions sharded by the local device count.

flax.training.common_utils.**shard_prng_key**(*prng_key*)

Helper to shard (aka split) a PRNGKey for use with pmap'd functions.

PRNG keys can be used at train time to drive stochastic modules e.g. Dropout. We would like a different PRNG key for each local device so that we end up with different random numbers on each one, hence we split our PRNG key.

Parameters
prng_key – JAX PRNGKey

Returns
A new array of PRNGKeys with leading dimension equal to local device count.

flax.training.common_utils.**stack_forest**(*forest*)

Helper function to stack the leaves of a sequence of pytrees.

Parameters
forest – a sequence of pytrees (e.g tuple or list) of matching structure whose leaves are arrays with individually matching shapes.

Returns

A single pytree of the same structure whose leaves are individually
stacked arrays.

flax.training.common_utils.**get_metrics**(*device_metrics*)

Helper utility for pmap, gathering replicated timeseries metric data.

Parameters
device_metrics – replicated, device-resident pytree of metric data, whose leaves are presumed to be a sequence of arrays recorded over time.

Returns
A pytree of unreplicated, host-resident, stacked-over-time arrays useful for computing host-local statistics and logging.

flax.training.common_utils.**onehot**(*labels*, *num_classes*, *on_value=1.0*, *off_value=0.0*)

Create a dense one-hot version of an indexed array.

NB: consider using the more standard *jax.nn.one_hot* instead.

Parameters

- **labels** – an n-dim JAX array whose last dimension contains integer indices.
- **num_classes** – the maximum possible index.
- **on_value** – the "on" value for the one-hot array, defaults to 1.0.
- **off_value** – the "off" value for the one-hot array, defaults to 0.0.

Returns
A (n+1)-dim array whose last dimension contains one-hot vectors of length num_classes.

5.8.10 flax.traverse_util package

A utility for traversing immutable datastructures.

A Traversal can be used to iterate and update complex data structures. Traversals take in an object and return a subset of its contents. For example, a Traversal could select an attribute of an object:

```
x = Foo(foo=1)
traverse_util.TraverseAttr('foo').iterate(x) # [1]
```

More complex traversals can be constructed using composition. It is often useful to start from the identity traversal and use a method chain to construct the intended Traversal:

```
data = [{'foo': 1, 'bar': 2}, {'foo': 3, 'bar': 4}]
traversal = traverse_util.t_identity.each()['foo']
traversal.iterate(data) # [1, 3]
```

Traversals can also be used to make changes using the *update* method:

```
data = {'foo': Foo(bar=2)}
traversal = traverse_util.t_identity['foo'].bar
traversal.update(lambda x: x + x, data) # {'foo': Foo(bar=4)}
```

Traversals never mutate the original data. Therefore, an update essentially returns a copy of the data including the provided updates.

Traversal objects

class flax.traverse_util.**Traversal**(**args*, ***kwargs*)

Base class for all traversals.

compose(*other*)

Compose two traversals.

each()

Traverse each item in the selected containers.

filter(*fn*)

Filter the selected values.

abstract iterate(*inputs*)

Iterate over the values selected by this *Traversal*.

Parameters
inputs – the object that should be traversed.

Returns
An iterator over the traversed values.

merge(**traversals*)

Compose an arbitrary number of traversals and merge the results.

`set`(*values*, *inputs*)

Overrides the values selected by the *Traversal*.

Parameters

- **values** – a list containing the new values.
- **inputs** – the object that should be traversed.

Returns
A new object with the updated values.

`tree`()

Traverse each item in a pytree.

`abstract update`(*fn*, *inputs*)

Update the focused items.

Parameters

- **fn** – the callback function that maps each traversed item to its updated value.
- **inputs** – the object that should be traversed.

Returns
A new object with the updated values.

`class flax.traverse_util.TraverseId`(**args*, ***kwargs*)

The identity Traversal.

`iterate`(*inputs*)

Iterate over the values selected by this *Traversal*.

Parameters
inputs – the object that should be traversed.

Returns
An iterator over the traversed values.

`update`(*fn*, *inputs*)

Update the focused items.

Parameters

- **fn** – the callback function that maps each traversed item to its updated value.
- **inputs** – the object that should be traversed.

Returns
A new object with the updated values.

`class flax.traverse_util.TraverseMerge`(**args*, ***kwargs*)

Merges the selection from a set of traversals.

`iterate`(*inputs*)

Iterate over the values selected by this *Traversal*.

Parameters
inputs – the object that should be traversed.

Returns
An iterator over the traversed values.

update(*fn*, *inputs*)

Update the focused items.

Parameters

- **fn** – the callback function that maps each traversed item to its updated value.
- **inputs** – the object that should be traversed.

Returns
A new object with the updated values.

class flax.traverse_util.**TraverseCompose**(**args*, ***kwargs*)

Compose two traversals.

iterate(*inputs*)

Iterate over the values selected by this *Traversal*.

Parameters
inputs – the object that should be traversed.

Returns
An iterator over the traversed values.

update(*fn*, *inputs*)

Update the focused items.

Parameters

- **fn** – the callback function that maps each traversed item to its updated value.
- **inputs** – the object that should be traversed.

Returns
A new object with the updated values.

class flax.traverse_util.**TraverseFilter**(**args*, ***kwargs*)

Filter selected values based on a predicate.

iterate(*inputs*)

Iterate over the values selected by this *Traversal*.

Parameters
inputs – the object that should be traversed.

Returns
An iterator over the traversed values.

update(*fn*, *inputs*)

Update the focused items.

Parameters

- **fn** – the callback function that maps each traversed item to its updated value.
- **inputs** – the object that should be traversed.

Returns
A new object with the updated values.

class flax.traverse_util.**TraverseAttr**(**args*, ***kwargs*)

Traverse the attribute of an object.

`iterate`(*inputs*)

Iterate over the values selected by this *Traversal*.

Parameters
inputs – the object that should be traversed.

Returns
An iterator over the traversed values.

`update`(*fn*, *inputs*)

Update the focused items.

Parameters

- **fn** – the callback function that maps each traversed item to its updated value.
- **inputs** – the object that should be traversed.

Returns
A new object with the updated values.

`class flax.traverse_util.TraverseItem`(**args*, ***kwargs*)

Traverse the item of an object.

`iterate`(*inputs*)

Iterate over the values selected by this *Traversal*.

Parameters
inputs – the object that should be traversed.

Returns
An iterator over the traversed values.

`update`(*fn*, *inputs*)

Update the focused items.

Parameters

- **fn** – the callback function that maps each traversed item to its updated value.
- **inputs** – the object that should be traversed.

Returns
A new object with the updated values.

`class flax.traverse_util.TraverseEach`(**args*, ***kwargs*)

Traverse each item of a container.

`iterate`(*inputs*)

Iterate over the values selected by this *Traversal*.

Parameters
inputs – the object that should be traversed.

Returns
An iterator over the traversed values.

`update`(*fn*, *inputs*)

Update the focused items.

Parameters

- **fn** – the callback function that maps each traversed item to its updated value.

- **inputs** – the object that should be traversed.

Returns
A new object with the updated values.

class flax.traverse_util.**TraverseTree**(**args*, ***kwargs*)

Traverse every item in a pytree.

iterate(*inputs*)

Iterate over the values selected by this *Traversal*.

Parameters
inputs – the object that should be traversed.

Returns
An iterator over the traversed values.

update(*fn*, *inputs*)

Update the focused items.

Parameters

- **fn** – the callback function that maps each traversed item to its updated value.
- **inputs** – the object that should be traversed.

Returns
A new object with the updated values.

Dict utils

flax.traverse_util.**flatten_dict**(*xs*, *keep_empty_nodes=False*, *is_leaf=None*, *sep=None*)

Flatten a nested dictionary.

The nested keys are flattened to a tuple. See *unflatten_dict* on how to restore the nested dictionary structure.

Example:

```
xs = {'foo': 1, 'bar': {'a': 2, 'b': {}}}
flat_xs = flatten_dict(xs)
print(flat_xs)
# {
#   ('foo',): 1,
#   ('bar', 'a'): 2,
# }
```

Note that empty dictionaries are ignored and will not be restored by *unflatten_dict*.

Parameters

- **xs** – a nested dictionary
- **keep_empty_nodes** – replaces empty dictionaries with *traverse_util.empty_node*.
- **is_leaf** – an optional function that takes the next nested dictionary and nested keys and returns True if the nested dictionary is a leaf (i.e., should not be flattened further).
- **sep** – if specified, then the keys of the returned dictionary will be *sep*-joined strings (if *None*, then keys will be tuples).

Returns
The flattened dictionary.

flax.traverse_util.**unflatten_dict**(*xs*, *sep=None*)

Unflatten a dictionary.

See *flatten_dict*

Example:

```
flat_xs = {
  ('foo',): 1,
  ('bar', 'a'): 2,
}
xs = unflatten_dict(flat_xs)
print(xs)
# {
#   'foo': 1
#   'bar': {'a': 2}
# }
```

Parameters

- **xs** – a flattened dictionary
- **sep** – separator (same as used with *flatten_dict()*).

Returns

The nested dictionary.

flax.traverse_util.**path_aware_map**(*f*, *nested_dict*)

A map function that operates over nested dictionary structures while taking the path to each leaf into account.

Example:

```
>>> import jax.numpy as jnp
>>> from flax import traverse_util
...
>>> params = {'a': {'x': 10, 'y': 3}, 'b': {'x': 20}}
>>> f = lambda path, x: x + 5 if 'x' in path else -x
>>> traverse_util.path_aware_map(f, params)
{'a': {'x': 15, 'y': -3}, 'b': {'x': 25}}
```

Parameters

- **f** – A callable that takes in `(path, value)` arguments and maps them to a new value. Here `path` is a tuple of strings.
- **nested_dict** – A nested dictionary structure.

Returns

A new nested dictionary structure with the mapped values.

Model parameter traversal

class flax.traverse_util.**ModelParamTraversal**(**args*, ***kwargs*)

Select model parameters using a name filter.

This traversal operates on a nested dictionary of parameters and selects a subset based on the *filter_fn* argument.

See `flax.optim.MultiOptimizer` for an example of how to use *ModelParamTraversal* to update subsets of the parameter tree with a specific optimizer.

__init__(*filter_fn*)

Constructor a new ModelParamTraversal.

Parameters

filter_fn – a function that takes a parameter's full name and its value and returns whether this parameter should be selected or not. The name of a parameter is determined by the module hierarchy and the parameter name (for example: '/module/sub_module/parameter_name').

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z