Save and load checkpoints

Contents

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

> **Note:** The Flax team does not actively maintain a library for saving and loading model checkpoints to disk. Therefore, it is recommended you use external libraries like Orbax to do it.

In this guide you will learn how to:

- Save checkpoints.
- Restore checkpoints.
- Restore checkpoints if checkpoint structures differ.
- Perform multi-process checkpointing.

The Orbax API examples used throughout the guide are for demonstration purposes, and for the most up-to-date recommended APIs refer to the Orbax website.

> **Note:** The Flax team recommends using Orbax for saving and loading checkpoints to disk, as we do not actively maintain a library for these functionalities.

> **Note:** If you are looking for Flax Linen's legacy `flax.training.checkpoints` package, it was deprecated in 2023 in favor of Orbax. The documentation resides on the Flax Linen site.

Setup

Import the necessary dependencies, set up a checkpoint directory and an example Flax NNX model - `TwoLayerMLP` - by subclassing `nnx.Module`.

```
from flax import nnx
import orbax.checkpoint as ocp
import jax
from jax import numpy as jnp
import numpy as np

ckpt_dir = ocp.test_utils.erase_and_create_empty('/tmp/my-checkpoints/')
```

```
class TwoLayerMLP(nnx.Module):
  def __init__(self, dim, rngs: nnx.Rngs):
    self.linear1 = nnx.Linear(dim, dim, rngs=rngs, use_bias=False)
    self.linear2 = nnx.Linear(dim, dim, rngs=rngs, use_bias=False)

  def __call__(self, x):
    x = self.linear1(x)
    return self.linear2(x)

# Instantiate the model and show we can run it.
model = TwoLayerMLP(4, rngs=nnx.Rngs(0))
x = jax.random.normal(jax.random.key(42), (3, 4))
assert model(x).shape == (3, 4)
```

Save checkpoints

JAX checkpointing libraries, such as Orbax, can save and load any given JAX pytree, which is a pure, possibly nested container of `jax.Array`s) (or, "tensors" as some other frameworks would put it). In the context of machine learning, the checkpoint is usually a pytree of model parameters and other data, such as optimizer states.

In Flax NNX, you can obtain such a pytree from an `nnx.Module` by calling `nnx.split`, and picking up the returned `nnx.State`.

```
_, state = nnx.split(model)
nnx.display(state)

checkpointer = ocp.StandardCheckpointer()
checkpointer.save(ckpt_dir / 'state', state)
```

```
▼State({
    'linear1': ▼{
      'bias': ▶VariableState(type=Param, value=None),
      'kernel': ▶VariableState(type=Param, value=<jax.Array float32(4, 4) ≈−0.
    },
    'linear2': ▼{
      'bias': ▶VariableState(type=Param, value=None),
      'kernel': ▶VariableState(type=Param, value=<jax.Array float32(4, 4) ≈−0.
    },
  })
```

```
▼State({
    'linear1': ▼{
      'bias': ▶VariableState(type=Param, value=None),
      'kernel': ▶VariableState(type=Param, value=<jax.Array float32(4, 4) ≈−0.08 ±0.6 [≥−0.94, ≤1.1] nonzero:1
    },
    'linear2': ▼{
      'bias': ▶VariableState(type=Param, value=None),
      'kernel': ▶VariableState(type=Param, value=<jax.Array float32(4, 4) ≈−0.031 ±0.58 [≥−0.85, ≤1.1] nonzero
    },
  })
```

Restore checkpoints

Note that you saved the checkpoint as a Flax class of `nnx.State`, which is also nested with `nnx.VariableState` and `nnx.Param` classes.

At checkpoint restoration time, you need to have these classes ready in your runtime, and instruct the checkpointing library (Orbax) to restore your pytree back to that structure. This can be achieved as follows:

- First, create an abstract Flax NNX model (without allocating any memory for arrays), and show its

abstract variable state to the checkpointing library.

- Once you have the state, use `nnx.merge` to obtain your Flax NNX model, and use it as usual.

```
# Restore the checkpoint back to its `nnx.State` structure - need an abstract referenc
abstract_model = nnx.eval_shape(lambda: TwoLayerMLP(4, rngs=nnx.Rngs(0)))
graphdef, abstract_state = nnx.split(abstract_model)
print('The abstract NNX state (all leaves are abstract arrays):')
nnx.display(abstract_state)

state_restored = checkpointer.restore(ckpt_dir / 'state', abstract_state)
jax.tree.map(np.testing.assert_array_equal, state, state_restored)
print('NNX State restored: ')
nnx.display(state_restored)

# The model is now good to use!
model = nnx.merge(graphdef, state_restored)
assert model(x).shape == (3, 4)
```

```
The abstract NNX state (all leaves are abstract arrays):
```

```
▼State({
  'linear1': ▼{
    'bias': ▶VariableState(type=Param, value=None),
    'kernel': ▼VariableState(
      type=▶Param,
      value=▶ShapeDtypeStruct(shape=(4, 4), dtype=dtype('float32')),
    ),
  },
  'linear2': ▼{
    'bias': ▶VariableState(type=Param, value=None),
    'kernel': ▼VariableState(
      type=▶Param,
      value=▶ShapeDtypeStruct(shape=(4, 4), dtype=dtype('float32')),
    ),
  },
})
```

```
NNX State restored:
```

```
/Users/ivyzheng/envs/flax-head/lib/python3.11/site-packages/orbax/checkpoint/type_handl
  warnings.warn(
```

```
▼State({
  'linear1': ▼{
    'bias': ▶VariableState(type=Param, value=None),
    'kernel': ▶VariableState(type=Param, value=<jax.Array float32(4, 4) ≈-0.
  },
  'linear2': ▼{
    'bias': ▶VariableState(type=Param, value=None),
    'kernel': ▶VariableState(type=Param, value=<jax.Array float32(4, 4) ≈-0.
  },
})
```

```
The abstract NNX state (all leaves are abstract arrays):
```

```
▼State({
  'linear1': ▼{
    'bias': ▶VariableState(type=Param, value=None),
    'kernel': ▼VariableState(
      type=▶Param,
      value=▶ShapeDtypeStruct(shape=(4, 4), dtype=dtype('float32')),
    ),
  },
```

```
  'linear2': ▼{
    'bias': ▶VariableState(type=Param, value=None),
    'kernel': ▼VariableState(
      type=▶Param,
      value=▶ShapeDtypeStruct(shape=(4, 4), dtype=dtype('float32')),
    ),
  },
})
```

```
NNX State restored:


/Users/ivyzheng/envs/flax-head/lib/python3.11/site-packages/orbax/checkpoint/type_handl
  warnings.warn(
```

```
▼State({
  'linear1': ▼{
    'bias': ▶VariableState(type=Param, value=None),
    'kernel': ▶VariableState(type=Param, value=<jax.Array float32(4, 4) ≈-0.08 ±0.6 [≥-0.94, ≤1.1] nonzero:1
  },
  'linear2': ▼{
    'bias': ▶VariableState(type=Param, value=None),
    'kernel': ▶VariableState(type=Param, value=<jax.Array float32(4, 4) ≈-0.031 ±0.58 [≥-0.85, ≤1.1] nonzero
  },
})
```

Save and restore as pure dictionaries

When interacting with checkpoint libraries (like Orbax), you may prefer to work with Python built-in container types. In this case, you can use the `nnx.State.to_pure_dict` and `nnx.State.replace_by_pure_dict` API to convert an `nnx.State` to and from pure nested dictionaries.

```
# Save as pure dict
pure_dict_state = nnx.to_pure_dict(state)
nnx.display(pure_dict_state)
checkpointer.save(ckpt_dir / 'pure_dict', pure_dict_state)

# Restore as a pure dictionary.
restored_pure_dict = checkpointer.restore(ckpt_dir / 'pure_dict')
abstract_model = nnx.eval_shape(lambda: TwoLayerMLP(4, rngs=nnx.Rngs(0)))
graphdef, abstract_state = nnx.split(abstract_model)
nnx.replace_by_pure_dict(abstract_state, restored_pure_dict)
model = nnx.merge(graphdef, abstract_state)
assert model(x).shape == (3, 4)  # The model still works!
```

```
▼{
  'linear1': ▶{'bias': None, 'kernel': <jax.Array float32(4, 4) ≈-0.08 ±0.6
  'linear2': ▶{'bias': None, 'kernel': <jax.Array float32(4, 4) ≈-0.031 ±0.5
}
```

```
WARNING:absl:`StandardCheckpointHandler` expects a target tree to be provided for restc
```

```
▼{
  'linear1': ▶{'bias': None, 'kernel': <jax.Array float32(4, 4) ≈-0.08 ±0.6 [≥-0.94, ≤1.1] nonzero:16>},
  'linear2': ▶{'bias': None, 'kernel': <jax.Array float32(4, 4) ≈-0.031 ±0.58 [≥-0.85, ≤1.1] nonzero:16>},
}
```

```
WARNING:absl:`StandardCheckpointHandler` expects a target tree to be provided for restc
```

Restore when checkpoint structures differ

The ability to load a checkpoint as a pure nested dictionary can come in handy when you want to load some outdated checkpoints that no longer match with your current model code. Check out this simple example below.

This pattern also works if you save the checkpoint as an `nnx.State` instead of a pure dictionary. Check out the Checkpoint surgery section of the Model Surgery guide for an example with code. The only difference is you need to reprocess your raw dictionary a bit before calling `nnx.State.replace_by_pure_dict`.

```
class ModifiedTwoLayerMLP(nnx.Module):
  """A modified version of TwoLayerMLP, which requires bias arrays."""
  def __init__(self, dim, rngs: nnx.Rngs):
    self.linear1 = nnx.Linear(dim, dim, rngs=rngs, use_bias=True)  # We need bias now!
    self.linear2 = nnx.Linear(dim, dim, rngs=rngs, use_bias=True)  # We need bias now!

  def __call__(self, x):
    x = self.linear1(x)
    return self.linear2(x)

# Accommodate your old checkpoint to the new code.
restored_pure_dict = checkpointer.restore(ckpt_dir / 'pure_dict')
restored_pure_dict['linear1']['bias'] = jnp.zeros((4,))
restored_pure_dict['linear2']['bias'] = jnp.zeros((4,))

# Same restore code as above.
abstract_model = nnx.eval_shape(lambda: ModifiedTwoLayerMLP(4, rngs=nnx.Rngs(0)))
graphdef, abstract_state = nnx.split(abstract_model)
nnx.replace_by_pure_dict(abstract_state, restored_pure_dict)
model = nnx.merge(graphdef, abstract_state)
assert model(x).shape == (3, 4)  # The new model works!

nnx.display(model.linear1)
```

```
WARNING:absl:`StandardCheckpointHandler` expects a target tree to be provided for restc
```

```
Linear(
  bias=Param(value=<jax.Array float32(4,) ≈0.0 ±0.0 [≥0.0, ≤0.0] zero:4>),
  bias_init=zeros,
  dot_general=jax.lax.dot_general,
  dtype=None,
  in_features=4,
  kernel=Param(value=<jax.Array float32(4, 4) ≈-0.08 ±0.6 [≥-0.94, ≤1.1] nc
  kernel_init=<function variance_scaling.<locals>.init at 0x177cd79c0>,
  out_features=4,
  param_dtype=jax.numpy.float32,
  precision=None,
  use_bias=True,
)
```

```
WARNING:absl:`StandardCheckpointHandler` expects a target tree to be provided for restc
```

```
Linear(
  bias=Param(value=<jax.Array float32(4,) ≈0.0 ±0.0 [≥0.0, ≤0.0] zero:4>),
  bias_init=zeros,
  dot_general=jax.lax.dot_general,
  dtype=None,
  in_features=4,
  kernel=Param(value=<jax.Array float32(4, 4) ≈-0.08 ±0.6 [≥-0.94, ≤1.1] nonzero:16>),
  kernel_init=<function variance_scaling.<locals>.init at 0x177cd79c0>,
  out_features=4,
  param_dtype=jax.numpy.float32,
  precision=None,
  use_bias=True,
```

```
)
```

Multi-process checkpointing

In a multi-host/multi-process environment, you would want to restore your checkpoint as sharded across multiple devices. Check out the Load sharded model from a checkpoint section in the Flax Scale up on multiple devices guide to learn how to derive a sharding pytree and use it to load your checkpoint.

> **Note:** JAX provides several 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). Check out JAX's Introduction to parallel programming, Using JAX in multi-host and multi-process environments, Distributed arrays and automatic parallelization, and Manual parallelism with `shard_map`.

Other checkpointing features

This guide only uses the simplest `orbax.checkpoint.StandardCheckpointer` API to show how to save and load on the Flax modeling side. Feel free to use other tools or libraries as you see fit.

In addition, check out the Orbax website for other commonly used features, such as:

- `CheckpointManager` to track checkpoints from different steps.
- Asynchronous checkpointing.
- Orbax transformations: A way to modify pytree structure during loading time, instead of after loading time, which is demonstrated in this guide.