Checkpointing Flax NNX Models with Orbax

Saving and Restoring Your JAX/NNX Training Progress

Why Checkpoint & Intro to NNX/Orbax

- **Goal**: Save training progress (model parameters, optimizer state) to resume later or analyze results. Essential for long training runs.
- **Flax NNX**: A newer Flax API. Modules are stateful Python classes, holding their own parameters and state directly – similar feel to PyTorch modules.
- **Orbax**: The recommended JAX library for robust, scalable checkpointing, handling complex scenarios like distributed training.
- **Focus**: Using Orbax to save/restore the state managed by Flax NNX modules.

Understanding NNX State (1/4): nnx.Module

- NNX Modules (**`nnx.Module`**) are Python classes holding their own state (parameters, etc.).
- State components are defined as attributes, often using specific **`nnx.Variable`** types.
- Initialization is typically eager (state created with the module instance).
- Feels object-oriented, state is part of the module object itself.

Understanding NNX State (2/4): nnx.Module Example

```python
# Example NNX Module Definition
class SimpleLinear(nnx.Module):
  def __init__(self, din: int, dout: int, *, rngs: nnx.Rngs):
    # Parameters defined using nnx.Param (a type of nnx.Variable)
    self.weight = nnx.Param(jax.random.uniform(rngs.params(), (din, dout)))
    self.bias = nnx.Param(jnp.zeros((dout,)))

  def __call__(self, x: jax.Array) -> jax.Array:
    # Parameters used directly via self.weight, self.bias
    return x @ self.weight + self.bias

# Instantiate
key = nnx.Rngs(params=0) # NNX requires explicit RNG management
linear_layer = SimpleLinear(din=10, dout=5, rngs=key)
print(linear_layer.weight.value.shape) # Access parameter value: (10, 5)
```

Understanding NNX State (3/4): nnx.Variable & nnx.State

- **`nnx.Variable`**: Base class for all dynamic state (parameters, batch stats, custom state).
- **NNX Modules are now native JAX Pytrees**: As of v0.11, the module object itself can be used with JAX transformations.
- **`nnx.state(module)`**: A utility to extract a *filtered* Pytree containing only the dynamic state (all **`nnx.Variables`**).
- This filtered **`nnx.State`** Pytree is typically what Orbax saves and restores to ensure only dynamic state is checkpointed.

Understanding NNX State (4/4): The Functional Bridge

NNX provides functions to move between the Python module and its Pytree state:

1. **`nnx.split(module)`**: Separates module into static structure (**`GraphDef`**) and dynamic state (**`nnx.State`**). Needed to get the state for saving/JAX transforms.
2. **`nnx.merge(graphdef, state)`**: Reconstructs a module instance from its structure and state. Used after restoring state from a checkpoint.
3. **`nnx.update(module, state)`**: Updates an existing module instance in-place with data from a state object. Also used after restoring.

These are essential for interacting with JAX and Orbax.

Basic Checkpointing: Orbax Core Components

- `orbax.checkpoint.Checkpointer`: Base object for saving/restoring specific types. `StandardCheckpointer` handles generic Pytrees like `nnx.State`.
- `orbax.checkpoint.CheckpointManager`: Higher-level utility for managing checkpoints over a training run.
  - Handles versions (steps), saving, restoring.
  - Manages policies (e.g., keep latest N checkpoints).
  - Uses a Checkpointer internally.
- **Recommendation**: Use `CheckpointManager` for typical training loops.

Basic Checkpointing: Saving `nnx.State` Workflow

1. **Instantiate `CheckpointManager`**: Specify checkpoint directory and options (e.g., max_to_keep).
2. **Split the Model**: Use `graphdef, state = nnx.split(model)` to get the `nnx.State` Pytree.
3. **Save**: Call `manager.save(step, args=...)` passing the state wrapped in `ocp.args.StandardSave(...)` or the generated args.
4. **Wait**: Call `manager.wait_until_finished()` if saving asynchronously.

Basic Checkpointing: Saving nnx.State Code Example (1/2)

```python
# Assume 'model' is an initialized nnx.Module instance
# e.g., model = SimpleLinear(din=10, dout=5, rngs=nnx.Rngs(0))

ckpt_dir = '/tmp/my_nnx_checkpoints'

# 1. Instantiate CheckpointManager
options = ocp.CheckpointManagerOptions(max_to_keep=3)
mngr = ocp.CheckpointManager(ckpt_dir, options=options)

# 2. Split the model to get the state Pytree
_graphdef, state_to_save = nnx.split(model)
# Alternatively: state_to_save = nnx.state(model)
```

Basic Checkpointing: Saving nnx.State Code Example (2/2)

```
# (Continuing from previous slide)

# 3. Save the state at a specific step
step = 100
mngr.save(step, args=ocp.args.StandardSave(state_to_save))
mngr.wait_until_finished() # Ensure save completes if async

print(f"Checkpoint saved for step {step}.")
mngr.close() # Clean up resources
```

Basic Checkpointing: Restoring `nnx.State` Workflow

1. **Create Abstract Model**: Instantiate model using `nnx.eval_shape`. Replaces arrays with ShapeDtypeStruct (structure without data/memory). Crucial step!
2. **Split Abstract Model**: Get `graphdef` and `abstract_state` from the abstract model via `nnx.split`. `abstract_state` acts as a template for Orbax.
3. **Instantiate `CheckpointManager`**: Point to the checkpoint directory.
4. **Restore**: Call `manager.restore(step, args=...)` providing the `abstract_state` wrapped in `ocp.args.StandardRestore(...)` or generated args.
5. **Reconstruct/Update Model**: Use `restored_state` with `nnx.merge(graphdef, restored_state)` or `nnx.update(existing_model, restored_state)`.

Basic Checkpointing: Restoring Code Example (1/2)

```python
# Assume SimpleLinear class and ckpt_dir from saving exist
mngr = ocp.CheckpointManager(ckpt_dir) # Re-open manager

# 1. Create abstract model using nnx.eval_shape
def create_abstract_model():
    # Use dummy RNG key/inputs for abstract creation
    return SimpleLinear(din=10, dout=5, rngs=nnx.Rngs(0))

abstract_model = nnx.eval_shape(create_abstract_model)

# 2. Split abstract model to get abstract state structure
graphdef, abstract_state = nnx.split(abstract_model)
# abstract_state now contains ShapeDtypeStruct leaves
```

Basic Checkpointing: Restoring Code Example (2/2)

```
# (Continuing from previous slide)
# 3. Restore the state for the latest step
step_to_restore = mngr.latest_step()
if step_to_restore is not None:
    restored_state = mngr.restore(step_to_restore,
        args=ocp.args.StandardRestore(abstract_state))

    # 4. Reconstruct the model using graphdef and restored state
    restored_model = nnx.merge(graphdef, restored_state)
    print(f"Model restored from step {step_to_restore}.")
    # Now 'restored_model' is ready to use
    # print(restored_model.bias.value) # Can check values
else:
    print("No checkpoint found.")

mngr.close()
```

Google

Checkpointing Optimizer State

- Training involves model parameters and optimizer state (e.g., momentum).
- Flax NNX provides **`nnx.Optimizer`**, which wraps a model and an Optax optimizer (**`optax.GradientTransformation`**).
- **`nnx.Optimizer`** itself is an NNX structure holding its state (Optax state, step count) as **`nnx.Variables`**.
- Can extract optimizer state using **`nnx.state(optimizer)`**.
- Typically save model params and optimizer state together in one step.

Saving Model & Optimizer State

- Use `ocp.args.Composite` to save multiple named items in one checkpoint step.
- Extract model parameters (e.g., using `nnx.split(model, nnx.Param)`).
- Extract optimizer state (`nnx.state(optimizer)`).
- Pass both to `manager.save` within `ocp.args.Composite`.

Saving Model & Optimizer State (1/2)

```python
# Assume 'model' is initialized, tx is an Optax transformer
# optimizer = nnx.Optimizer(model, tx)  (Simulate some training steps on optimizer...)

ckpt_dir_comp = '/tmp/my_nnx_composite_checkpoints'
mngr_comp = ocp.CheckpointManager(ckpt_dir_comp,
                  options=ocp.CheckpointManagerOptions(max_to_keep=3))

# Extract states
_graphdef, params_state = nnx.split(optimizer.model, nnx.Param)
optimizer_state_tree = nnx.state(optimizer)
```

Google

Saving Model & Optimizer State (2/2)

```python
# (Continuing from previous slide)

step = optimizer.step.value # Get current step from optimizer

# Save using Composite args
save_items = {
    'params': ocp.args.StandardSave(params_state),
    'optimizer': ocp.args.StandardSave(optimizer_state_tree)
}
# Can generate args per item using orbax_utils too

mngr_comp.save(step, args=ocp.args.Composite(**save_items))
mngr_comp.wait_until_finished()
print(f"Composite checkpoint saved for step {step}.")
mngr_comp.close()
```

Restoring Model & Optimizer State

- Follows the same pattern: Abstract -> Restore -> Update.
- Create abstract versions of both model and optimizer using **`nnx.eval_shape`**.
- Get abstract state templates for both parameter state and optimizer state.
- Use **`ocp.args.Composite`** with **`ocp.args.StandardRestore`** for restoring.

Restoring Model & Optimizer State (1/2)

```
# Assume ExampleModule class, ckpt_dir_comp exist
mngr_comp = ocp.CheckpointManager(ckpt_dir_comp)

# 1. Create abstract model and optimizer
def create_abstracts():
    rngs = nnx.Rngs(0)
    abs_model = nnx.eval_shape(lambda: ExampleModule(rngs=rngs))
    abs_opt = nnx.eval_shape(lambda: nnx.Optimizer(abs_model, optax.adam(1e-3)))
    return abs_model, abs_opt

abs_model, abs_optimizer = create_abstracts()

# 2. Get abstract states
graphdef, abs_params_state = nnx.split(abs_model, nnx.Param)
abs_optimizer_state = nnx.state(abs_optimizer)
```

Restoring Model & Optimizer State (2/2)

```
# (Continuing from previous slide)
step = mngr_comp.latest_step()
if step is not None:
    restore_targets = {  # 3. Restore using Composite args
        'params': ocp.args.StandardRestore(abs_params_state),
        'optimizer': ocp.args.StandardRestore(abs_optimizer_state)
    }
    restored_items = mngr_comp.restore(step, args=ocp.args.Composite(**restore_targets))

    # 4. Instantiate concrete model/optimizer and update
    model = ExampleModule(rngs=nnx.Rngs(1)) # Fresh instance
    optimizer = nnx.Optimizer(model, optax.adam(1e-3))

    nnx.update(model, restored_items['params'])
    nnx.update(optimizer, restored_items['optimizer'])
    print(f"Restored step: {optimizer.step.value}")
mngr_comp.close()
```

Google

Distributed Checkpointing: Context

- Large models often use SPMD (Single Program, Multiple Data) via JAX for training across multiple GPUs/TPUs.
- Data (parameters, activations) is sharded (split) across devices defined in a `jax.sharding.Mesh`.
- Flax NNX allows attaching sharding annotations (e.g., `PartitionSpec`) directly to `nnx.Variable` metadata, often using `nnx.spmd.with_partitioning`.
- These annotations guide JAX on how to distribute the array data.

Distributed Checkpointing

- Orbax is designed for sharded JAX arrays. It saves/restores individual shards efficiently.
- **Crucial**: For restoration, Orbax needs the target sharding information if the topology has changed.
- Otherwise:
    - If using `StandardRestore`, Orbax will handle it correctly.
    - If using `PyTreeRestore`, you should use `ocp.checkpoint_utils.construct_restore_args`.

Distributed Checkpointing: Saving Sharded State

- The live state (after JAX transformations like `jax.jit` with sharding constraints) contains the actual `jax.Array` objects with sharding information.
- Orbax handles sharded-array saving under the hood, so it looks the same as regular saving.

Distributed Checkpointing: Restoring Sharded State

- Requires an abstract state Pytree that includes sharding specifications.
- Create abstract model (**`nnx.eval_shape`**), get its state.
- Extract partition specs (**`nnx.get_partition_spec`**).
- Apply specs to abstract state using **`jax.lax.with_sharding_constraint`** (often within **`jax.jit`** and **`Mesh`** context) to create the sharding-aware template.

Distributed Checkpointing: Restoring Sharded State

```
# Conceptual - Creating the abstract target with sharding
def create_abstract_sharded_state():
   abstract_model = nnx.eval_shape(...)
   _graphdef, abstract_state = nnx.split(abstract_model)
   sharding_specs = nnx.get_partition_spec(abstract_state)
   # Apply constraints to embed sharding in the abstract state
   abs_state_with_sharding = jax.lax.with_sharding_constraint(
       abstract_state, sharding_specs)
   return abs_state_with_sharding

with mesh:
   abstract_target = jax.jit(create_abstract_sharded_state)()
   restored_sharded_state = mngr_sharded.restore(step,
                                              args=StandardRestore(abstract_target))
   model = nnx.merge(graphdef_restore, restored_sharded_state)
```

Optimizations & Advanced Orbax Features

- **Asynchronous Checkpointing**: `manager.save` can return immediately while saving happens in the background (configure via options). Use `manager.wait_until_finished()` before exit or needing the checkpoint. Improves training throughput.
- **Atomicity**: `CheckpointManager` ensures checkpoints are saved atomically – no corrupted checkpoints if a crash occurs mid-save.
- **Non-Pytree Data**: Save metadata (like dataset iterators) alongside Pytrees using `ocp.args.JsonSave` within `ocp.args.Composite`. Restore with `ocp.args.JsonRestore`.
- **TensorStore Backend**: For very large arrays / cloud storage, Orbax can use TensorStore for efficient, potentially parallel I/O (often transparent).

Google

Conclusion & Key Takeaways

- Flax NNX provides a stateful, Pythonic way to define models in JAX.
- Orbax is the standard for checkpointing NNX state (`nnx.State` Pytrees).
- **Workflow**: nnx.split -> Save; `nnx.eval_shape` -> Abstract State -> Restore -> `nnx.merge/update`.
- `CheckpointManager` simplifies managing checkpoints over training.
- Use `ocp.args.Composite` for saving multiple items (e.g., model + optimizer).
- For sharded data, `flax.training.orbax_utils` are crucial for handling sharding info via live (save) and abstract (restore) states with the mesh context.

Learning Resources

Code Exercises, Quick References, and Slides

- https://goo.gle/learning-jax

Learn JAX

Community and Docs

Community:

- https://goo.gle/jax-community

Docs

- JAX AI Stack: https://jaxstack.ai
- JAX: https://jax.dev
- Flax NNX: https://flax.readthedocs.io