# Orbax & Flax NNX Checkpointing: Quick Reference

**Goal:** Save and restore Flax NNX model parameters, optimizer state, and other training artifacts using Orbax.
**Core Idea:** NNX Modules are stateful. **`nnx.state(module)`** extracts a JAX Pytree (**`nnx.State`**) that Orbax saves/restores.

## Core Orbax Components (aliased as ocp)

- **`ocp.CheckpointManager(directory, options=None)`**:
    - Manages checkpoint versions, saving, restoring, and cleanup policies.
    - Recommended for most training loops.
    - options: `ocp.CheckpointManagerOptions(max_to_keep=N, save_interval_steps=M, ...)`
- **`ocp.Checkpointer (e.g., StandardCheckpointer):`**
    - Handles serialization/deserialization of specific types (Pytrees like `nnx.State`).
    - Used internally by `CheckpointManager`.
- **`ocp.args`**: Namespace for specifying save/restore arguments.
    - **`ocp.args.StandardSave(pytree)`**: Argument for saving a standard Pytree.
    - **`ocp.args.StandardRestore(abstract_pytree)`**: Argument for restoring a standard Pytree, needs an abstract structure.
    - **`ocp.args.Composite(**kwargs)`**: For saving/restoring multiple named items.
        - E.g., `Composite(params=StandardSave(p_state), opt=StandardSave(o_state))`
    - **`ocp.args.JsonSave(data) / ocp.args.JsonRestore()`**: For non-Pytree JSON-serializable metadata.

## Essential NNX Helper Functions for Checkpointing

- **`nnx.state(module)`**: Extracts all `nnx.Variables` into an `nnx.State` Pytree. **This is what Orbax saves.**

- **`nnx.split(module, [filter_spec])`**: Returns (`GraphDef, nnx.State`). `GraphDef` is static structure, `nnx.State` is dynamic data.
    - `filter_spec` (e.g., `nnx.Param`) can select specific variable types.
- **`nnx.merge(graphdef, state)`**: Reconstructs a new module instance from `GraphDef` and `nnx.State`.
- **`nnx.update(module_instance, state)`**: Updates an existing module instance in-place with data from `nnx.State`.
- **`nnx.eval_shape(create_fn)`**: Creates an "abstract" module (arrays replaced by `ShapeDtypeStruct`) without allocating memory. Used to get target structure for restoration.
    - `create_fn`: A function that instantiates your model, e.g., `lambda: MyModel(rngs=...)`.
- **`nnx.Optimizer(model, optax_tx)`**: Wraps model and Optax optimizer; manages optimizer state as `nnx.Variables`. Its state can be extracted with `nnx.state(optimizer)`.
- **`nnx.get_partition_spec(pytree)`**: Extracts sharding `PartitionSpecs` from a Pytree of `nnx.Variables` (if they have sharding metadata).

# Basic Checkpointing Workflow: Model State

### Saving Model State

```python
# import orbax.checkpoint as ocp
# model: initialized nnx.Module
# mngr: ocp.CheckpointManager instance

_graphdef, state_to_save = nnx.split(model)
# OR: state_to_save = nnx.state(model)

mngr.save(step, args=ocp.args.StandardSave(state_to_save))
mngr.wait_until_finished() # Important for async saves
mngr.close()
```

### Restoring Model State

```python
# mngr: ocp.CheckpointManager instance for the ckpt_dir

# 1. Create abstract model & get abstract state for structure
abstract_model = nnx.eval_shape(lambda: YourModelClass(...))
graphdef, abstract_state = nnx.split(abstract_model)

# 2. Restore
step_to_restore = mngr.latest_step()
if step_to_restore is not None:
    restored_state_pytree = mngr.restore(
        step_to_restore,
        args=ocp.args.StandardRestore(abstract_state)
    )
    # 3. Reconstruct model
    restored_model = nnx.merge(graphdef, restored_state_pytree)
    # OR: update an existing model
    # existing_model = YourModelClass(...)
    # nnx.update(existing_model, restored_state_pytree)
mngr.close()
```

## Checkpointing Multiple Items (e.g., Model Params & Optimizer State)

### Saving Composite State

```python
# optimizer: initialized nnx.Optimizer instance
_graphdef, params_state = nnx.split(optimizer.model, nnx.Param) # Just params
optimizer_state_pytree = nnx.state(optimizer) # Full optimizer state

save_items = {
    'params': ocp.args.StandardSave(params_state),
    'optimizer': ocp.args.StandardSave(optimizer_state_pytree)
}
mngr.save(optimizer.step.value, args=ocp.args.Composite(**save_items))
```

### Restoring Composite State

```python
# 1. Create abstract model & optimizer, get abstract states
abs_model = nnx.eval_shape(lambda: YourModelClass(rngs=nnx.Rngs(0)))
abs_opt = nnx.eval_shape(lambda: nnx.Optimizer(abs_model, optax.adam(1e-3)))

graphdef, abs_params_state = nnx.split(abs_model, nnx.Param)
abs_optimizer_state = nnx.state(abs_opt)

# 2. Define restore targets
restore_targets = {
    'params': ocp.args.StandardRestore(abs_params_state),
    'optimizer': ocp.args.StandardRestore(abs_optimizer_state)
}
# 3. Restore
restored_items_dict = mngr.restore(step,
args=ocp.args.Composite(**restore_targets))

# 4. Update concrete instances
model_instance = YourModelClass(rngs=nnx.Rngs(1)) # Fresh
optimizer_instance = nnx.Optimizer(model_instance, optax.adam(1e-3))

nnx.update(model_instance, restored_items_dict['params'])
nnx.update(optimizer_instance, restored_items_dict['optimizer'])
```

# Distributed Checkpointing (Sharded State)

- **Saving:** Looks same as basic saving. Orbax handles sharded `jax.Arrays` transparently if state is already sharded (e.g., from `jax.jit` within `Mesh`). `mngr.save(step, args=ocp.args.StandardSave(sharded_state_pytree))`

**Restoring:** Abstract state for `StandardRestore` **must** include target sharding info.

```python
# Conceptual: Inside a jax.jit function & Mesh context
def create_abstract_sharded_state_target():
    abstract_model = nnx.eval_shape(...)
    _graphdef_restore, abstract_state = nnx.split(abstract_model)
    sharding_specs = nnx.get_partition_spec(abstract_state) # Or define
manually
    # Embed sharding info into the abstract state structure
```

```
    return jax.lax.with_sharding_constraint(abstract_state, sharding_specs)

with mesh: # jax.sharding.Mesh
    abstract_target_with_sharding =
jax.jit(create_abstract_sharded_state_target)()

restored_sharded_state = mngr.restore(step,
    args=ocp.args.StandardRestore(abstract_target_with_sharding)
)
# Then nnx.merge or nnx.update
```

- Orbax needs target sharding if topology changes. `StandardRestore` uses sharding from the abstract target.
- For `PyTreeRestore` (less common with NNX `StandardRestore`), `ocp.checkpoint_utils.construct_restore_args` might be needed.

# Other Orbax Features

- **Asynchronous Checkpointing:** `CheckpointManager` can save in background (via options). Use `mngr.wait_until_finished()`.
- **Atomicity:** `CheckpointManager` ensures atomic saves (no corrupted checkpoints).
- **TensorStore Backend:** Orbax may use `TensorStore` for efficient I/O, especially for large arrays / cloud.

# More Information

- JAX AI Stack - [https://jaxstack.ai](https://jaxstack.ai)
- Orbax - [https://orbax.readthedocs.io](https://orbax.readthedocs.io)
- JAX - [https://jax.dev](https://jax.dev)
- Flax - [https://flax.readthedocs.io](https://flax.readthedocs.io)