JAX Sharding & Parallelism with Flax NNX

Quick Reference

This guide summarizes the core concepts, primitives, and workflows for distributed training using JAX's explicit sharding capabilities with Flax NNX.

Parallelism Strategies

Strategy	Description	Use Case
Data Parallelism	Replicate the model on each device; shard the input data. Gradients are synced and averaged.	The most common strategy for accelerating training when the model fits on a single device.
FSDP	Shard everything: model parameters, gradients, and optimizer state across devices.	Reduces memory usage significantly, allowing for larger models that don't fit on one device.
Tensor Parallelism	Split individual large layers (tensors) across multiple devices.	Enables layers that are too massive for a single device's memory.

Core JAX Sharding Primitives

Primitive	Description	Example Usage
`jax.sharding.Mesh`	A logical grid mapped to physical devices, with named axes.	`mesh = Mesh(devices, ('data', 'model'))`

Primitive	Description	Example Usage
`jax.sharding.PartitionSpec`	(Alias `P`) A tuple that defines how a tensor's dimensions map to `Mesh` axes. `None` means replicate.	`P('data', None)`
`jax.sharding.NamedSharding`	Binds a `Mesh` and a `PartitionSpec` into a concrete sharding strategy.	`data_sharding = NamedSharding(mesh, P('data', None))`
`jax.device_put`	Explicitly places a tensor onto devices according to a specified `NamedSharding`.	`sharded_batch = jax.device_put(numpy_array, data_sharding)`
`jax.lax.with_sharding_constraint`	Inside a `jit` function, asserts or enforces a `PartitionSpec` on an intermediate value.	`x = with_sharding_constraint(x, P('data', 'model'))`

Sharding with Flax NNX

The key is to embed sharding information as metadata directly within your `nnx.Module` definitions. This metadata acts as a hint for the JAX compiler.

1. Annotating Parameters

Use `nnx.with_metadata` during initialization to attach a `sharding` PartitionSpec to a parameter.

```python
from flax import nnx
from jax.sharding import PartitionSpec as P

# Inside an nnx.Module's __init__ method:
self.kernel = nnx.Param(
  nnx.with_metadata(
      nnx.initializers.lecun_normal(),
      sharding=('model', None) # Shard dim 0 on 'model',
replicate dim 1
  )(rng_key, shape)
)
```

2. Sharded Initialization (To Avoid OOM)

To prevent out-of-memory errors, initialize large models inside a jitted function where sharding is applied before the full model is materialized on a single device.

```python
@nnx.jit
def create_sharded_model(rngs):
  # 1. Instantiate the model (still unsharded, but with metadata)
  model = MyLargeModel(rngs=rngs)
  # 2. Extract functional State and PartitionSpec PyTrees
  state = nnx.state(model)
  pspecs = nnx.spmd.get_partition_spec(state)
  # 3. Apply sharding constraints to the state
  sharded_state = jax.lax.with_sharding_constraint(state, pspecs)
  # 4. Update the module with the now-sharded state
  nnx.update(model, sharded_state)
  return model
# --- To execute ---
```

```
with mesh: # Must be called within a Mesh context
  sharded_model = create_sharded_model(rngs)
```

Building the Training Loop

1. Shard Input Data

In each step of your training loop, explicitly place your input data onto the devices using `jax.device_put`.

Python
```python
# In training loop:
input_sharding = NamedSharding(mesh, P('data', None))
sharded_batch = jax.device_put(numpy_batch, input_sharding)
```

2. Compile the Training Step

Wrap your entire training step—forward pass, loss calculation, gradient computation, and optimizer update—in a function decorated with `@nnx.jit`. JAX will automatically handle the necessary communication (like gradient all-reduce) based on the sharding of the inputs and model parameters.

Python
```python
@nnx.jit
def train_step(model, optimizer, batch, labels):
  def loss_fn(model_stateful):
    logits = model_stateful(batch)
    return calculate_loss(logits, labels) # Your loss calculation

  loss, grads = nnx.value_and_grad(loss_fn)(model)
  optimizer.update(model, grads) # Optimizer updates sharded state
  return loss
```

Sharded Checkpointing with Orbax

To save and load huge sharded models without OOM errors, use a library like Orbax that handles individual shards.

Note on NNX v0.11+: The structure of checkpoints has changed, particularly for models containing RNG state (e.g., from Dropout or BatchNorm). While the code below correctly demonstrates how to pass sharding information to Orbax for a new model, migrating a checkpoint saved with NNX v0.10 requires a special process to handle the updated RNG structure.

```python
import orbax.checkpoint as ocp

# Get the PyTree of NamedSharding objects from the model
target_shardings = nnx.spmd.get_named_sharding(sharded_model,
mesh)

# Orbax uses these target shardings to restore correctly
checkpointer = ocp.PyTreeCheckpointer()
checkpointer.restore(path,
args=ocp.args.StandardRestore(target_shardings))
```