Optax & Flax NNX Quick Reference

A cheat sheet for optimizing Flax NNX models with Optax, designed for users familiar with PyTorch.

Core Concepts & Initialization

Concept	Description	Example Code
Flax NNX Model	A mutable, object-oriented API for defining neural networks, similar to torch.nn.Module.	class MyModel(nnx.Module): ...
Optax Transform	A function or chain of functions that defines the optimization algorithm (e.g., adam, sgd, clipping).	opt = optax.adam(learning_rate=1e-3)
NNX Optimizer	The bridge that connects an NNX model with an Optax transform, holding the optimizer state (e.g., momentum). It is initialized with respect to a model's parameters.	# The `wrt` argument specifies what to differentiate with respect to. `nnx.Param` is the standard choice for trainable parameters. optimizer = nnx.Optimizer(model, opt, wrt=nnx.Param)

The Standard Training Workflow

The core training loop involves calculating gradients and updating the model parameters. This entire process should be wrapped in `@nnx.jit` for performance.

Python

```
@nnx.jit
def train_step(model: MyModel, optimizer: nnx.Optimizer, x_batch, y_batch):
  # 2. Define a loss function for grad calculation
  def loss_fn_for_grad(model_to_train):
    return mse_loss(model_to_train, x_batch, y_batch)
```

```python
    # 3. Calculate loss and gradients simultaneously
    loss, grads = nnx.value_and_grad(loss_fn_for_grad)(model)

    # 4. Apply gradients to update the model and optimizer state
    optimizer.update(model, grads)

    return model, optimizer, loss

# --- In the main training loop ---
for x, y in data_loader:
    model, optimizer, loss_val = train_step(model, optimizer, x, y)
```

Advanced Optax: Building Custom Optimizers

Optax's power comes from its composability. Use `optax.chain` to combine modular gradient transformations into a sophisticated optimizer.

Transformation	Description	Example Usage
optax.chain()	Sequentially combines multiple transformations.	opt = optax.chain(...)
optax.clip_by_global_norm()	Clips gradients if their global norm exceeds a threshold.	optax.clip_by_global_norm(1.0)
optax.adam()	The Adam optimizer. Can be combined with other steps.	optax.adam(learning_rate=...)
optax.add_decayed_weights()	Adds weight decay.	optax.add_decayed_weights(1e-4)
optax.scale()	Scales gradients by a constant factor.	optax.scale(-learning_rate)

Example: Adam with Gradient Clipping

Python

```python
opt_adam_with_clipping = optax.chain(
    optax.clip_by_global_norm(1.0),  # First, clip gradients
```

```
    optax.adam(learning_rate=1e-3)   # Then, apply Adam update
)
optimizer = nnx.Optimizer(model, opt_adam_with_clipping, wrt=nnx.Param)
```

Learning Rate Scheduling

Schedules are functions that return a learning rate based on the current step. They are integrated directly into the Optax transform. No separate scheduler.step() is needed.

Python

```
# 1. Define a schedule function
lr_schedule = optax.warmup_cosine_decay_schedule(
    init_value=0.0,
    peak_value=1e-3,
    warmup_steps=1000,
    decay_steps=9000
)

# 2. Pass the schedule function as the learning_rate
opt_with_schedule = optax.adam(learning_rate=lr_schedule)
optimizer = nnx.Optimizer(model, opt_with_schedule, wrt=nnx.Param)
```

Per-Parameter Optimization

Apply different optimization rules (e.g., learning rates) to different parameter groups (e.g., biases vs. kernels) using optax.partition.

Steps:

1. **Create a param_labels PyTree:** A PyTree with the same structure as your model's parameters, where each leaf is a string label (e.g., 'biases', 'kernels').
2. **Define a dictionary of transforms:** Map each label to a specific Optax transform.
3. **Use optax.partition:** Combine the transforms and labels.

```python
# 1. Create the param_labels PyTree
# (Requires a function to traverse the model state and assign labels based on
parameter names)
def label_fn(path, param):
    return 'biases' if 'bias' in path else 'kernels'

params = nnx.state(model, nnx.Param)
param_labels = jax.tree_util.tree_map_with_path(label_fn, params)


# 2. & 3. Define transforms and create the partitioned optimizer
partitioned_opt = optax.partition(
    transforms={
        'kernels': optax.adam(learning_rate=1e-4),
        'biases': optax.sgd(learning_rate=1e-3)
    },
    param_labels=param_labels
)
optimizer = nnx.Optimizer(model, partitioned_opt, wrt=nnx.Param)
```

Distributed Training: Sharding the Optimizer State

When sharding a model for distributed training, the optimizer's internal state (e.g., momentum) must also be sharded to match the parameters.

1. **Filter for Optimizer State:** Use nnx.state(optimizer, nnx.optimizer.OptState) to get only the optimizer's internal state, not the model parameters it contains.
2. **Generate Sharding Specs:** Use nnx.spmd.get_partition_spec on the filtered optimizer state.
3. **Apply Sharding:** Use jax.lax.with_sharding_constraint to apply the sharding.

```python
# Inside a jax.sharding.Mesh context
# shard optimizer state
optimizer_internal_state = nnx.state(optimizer, nnx.optimizer.OptState)
optimizer_shardings = nnx.spmd.get_partition_spec(optimizer_internal_state,
mesh)
sharded_opt_state = jax.lax.with_sharding_constraint(
    optimizer_internal_state, optimizer_shardings
)
```

```
nnx.update(optimizer, sharded_opt_state) # Update the optimizer with sharded
state
```

PyTorch vs. Optax/NNX Comparison

Feature	PyTorch (torch.optim)	Optax / Flax NNX
Optimizer Init	optim.Adam(model.parameters(), ...)	nnx.Optimizer(model, optax.adam(...), wrt=nnx.Param)
Parameter Groups	List of dicts in constructor	optax.partition(...)
LR Scheduling	scheduler = StepLR(opt, ...) then sched.step()	Schedule fn passed directly to optax.adam()
Gradient Calc	loss.backward() (in-place mutation of .grad)	loss, grads = nnx.value_and_grad(fn)(...)
Gradient Clear	optimizer.zero_grad()	Implicit (new grads are returned each step)
Update Step	optimizer.step()	optimizer.update(model, grads)