Google

Optimizing Flax NNX Models with Optax:
A PyTorch User's Guide

From Fundamentals to Advanced Strategies

Review - The JAX Ecosystem for PyTorch Users

- **Flax NNX**: A Pythonic and flexible API for defining neural networks in JAX. It offers a mutable, object-oriented approach that will feel familiar to PyTorch users.
- **Optax**: The de facto gradient processing and optimization library for JAX, known for its composability.
- **The Synergy**: `flax.nnx.Optimizer` acts as a bridge, allowing Flax NNX models to seamlessly use Optax optimizers.
- **Goal**: Combine JAX's high performance with an intuitive model definition style (like PyTorch's `nn.Module`) and a sophisticated optimization toolkit.

Optax Core Philosophy: Composability

- **Composability**: Optax provides small, well-tested building blocks (gradient transformations) that can be chained together.
- **Flexibility**: Easily construct custom optimizers or complex gradient processing pipelines by combining these blocks.
- **Readability & Extensibility**: Code often mirrors mathematical equations, and new ideas can be readily integrated.
- **PyTorch Parallel**: Think of Optax transformations as more granular components than PyTorch's often monolithic optimizers. Optax encourages a "mix-and-match" approach to building optimizer behavior, offering more fine-grained control.

Flax NNX Model & Parameter Handling

- **Models**: Defined by subclassing `flax.nnx.Module`.
- **Parameters**: Instances of `flax.nnx.Param`, defined as attributes. They are typically initialized eagerly when the module is created, if `flax.nnx.Rngs` (for random keys) are provided.
- **State Management**: Uses Python's reference semantics, allowing models to be regular Python objects holding their own state.
- **PyTorch Parallel**: Defining an NNX model is very much like defining a `torch.nn.Module`. Parameters are attributes, and the model object itself holds its state. A key difference is the explicit handling of random number generator seeds (Rngs) for parameter initialization in NNX.

Basic Optimizer Usage - Defining an NNX Model

```
class SimpleMLP(nnx.Module):
  def __init__(self, din: int, dmid: int, dout: int, *,
                rngs: nnx.Rngs):
    self.linear1 = nnx.Linear(din, dmid, rngs=rngs)
    self.relu = nnx.relu
    self.linear2 = nnx.Linear(dmid, dout, rngs=rngs)

  def __call__(self, x: jax.Array):
    x = self.linear1(x)
    x = self.relu(x)
    x = self.linear2(x)
    return x
```

Basic Optimizer Usage - Instantiating Model & Optimizer

```
# Example instantiation
key = jax.random.key(0)
model_rngs = nnx.Rngs(key)
model = SimpleMLP(din=10, dmid=20, dout=5, rngs=model_rngs)

# Optimizer Initialization
learning_rate = 1e-3
optax_opt = optax.adam(learning_rate=learning_rate) # Optax transform
optimizer = nnx.Optimizer(model, optax_opt, wrt=nnx.Param)
```

PyTorch Parallel (Optimizer):

NNX/Optax: optimizer_state = nnx.Optimizer(model, optax_opt, wrt=nnx.Param)
PyTorch: optimizer = torch.optim.Adam(pytorch_model.parameters(), lr=0.001)

Basic Optimizer Usage - Loss Function & Gradients

- **Loss Function**: A Python function that takes the model instance, input data, and targets, then returns a scalar loss value.
- **Gradient Calculation**: Use `flax.nnx.value_and_grad(loss_fn)` to get both the loss and the gradients w.r.t. model parameters.
- **PyTorch Parallel (Gradients)**:
  - **NNX/JAX:** `loss, grads = nnx.value_and_grad(loss_fn_closure)(model)` computes and returns new gradient values.
  - **PyTorch**: `loss.backward()` computes gradients and stores them in the .grad attribute of parameters. JAX's functional nature means no `optimizer.zero_grad()` is needed.

Basic Optimizer Usage - Loss & Gradient Code

```
def mse_loss(model_instance: SimpleMLP,
             x_batch: jax.Array, y_batch: jax.Array):
  predictions = model_instance(x_batch) # Forward pass
  loss = jnp.mean((predictions - y_batch) ** 2)
  return loss

# Inside a training step, using a closure for x_batch, y_batch:
def loss_fn_for_grad(mdl):
    return mse_loss(mdl, x_batch_static, y_batch_static)

loss_val, grads = nnx.value_and_grad(loss_fn_for_grad)(optimizer_state.model)
```

Google

Basic Optimizer Usage - Parameter Updates & Training Step

- **Parameter Update**: The `optimizer_state.update(model, grads)` method applies the computed gradients to the model's parameters (in-place within optimizer_state) and updates the Optax optimizer's internal state.
- **Training Step Function**: Encapsulate loss calculation, gradient computation, and parameter update within a single function.
- **`@nnx.jit`**: Decorate the training step function with **`@nnx.jit`** for JAX's Just-In-Time compilation. This is crucial for performance and correctly handles state updates in NNX objects.
- **PyTorch Parallel (Updates)**:
  - NNX/Optax: `optimizer_state.update(model, grads)`
  - PyTorch: `optimizer.step()`

Basic Optimizer Usage - Training Step Code (Part 1)

```
@nnx.jit
def train_step(model: SimpleMLP,
               optimizer: nnx.Optimizer,
               x_batch: jax.Array,
               y_batch: jax.Array):

  # Define loss_fn to capture x_batch, y_batch
  def loss_fn_for_grad(model_to_train: SimpleMLP):
    return mse_loss(model_to_train, x_batch, y_batch)

  loss_value, grads = nnx.value_and_grad(loss_fn_for_grad)(model)
  # ... (update on next slide)
```

Basic Optimizer Usage - Training Step Code (Part 2)

```
 # ... (continued from previous slide)
  optimizer.update(model, grads) # Updates model params
  return loss_value

# Dummy data for example
key_data, key_loop = jax.random.split(jax.random.key(1))
x_dummy = jax.random.normal(key_data, (32, 10))
y_dummy = jax.random.normal(key_data, (32, 5))
```

Basic Optimizer Usage - Training Loop Code

```
# optimizer was initialized earlier
print("Starting basic training loop...")

for i in range(100):
  loss = train_step(model, optimizer, x_dummy, y_dummy)
  if i % 10 == 0:
    print(f"Step {optimizer.step.value}, Loss: {loss}")

print("Basic training loop finished.")
```

Advanced Optax - Gradient Transformations

- **Core Idea**: Optax transformations are functions that take gradients (and potentially optimizer state/parameters) and produce modified gradients.
- **`optax.chain`**: The primary tool to combine multiple transformations sequentially, creating sophisticated optimization pipelines.
- Common Transformations:
    - **`optax.clip_by_global_norm`**: Gradient clipping.
    - **`optax.scale_by_adam`**: Adam's adaptive scaling.
    - **`optax.add_decayed_weights`**: Weight decay.
- **PyTorch Parallel**: Optax's chain allows more explicit and composable optimizer construction than relying on built-in features of a single PyTorch optimizer. You're essentially building your optimizer's behavior step-by-step.

Google

Advanced Optax - Chained Transformations Code (Part 1)

```
learning_rate_chained = 1e-3
max_grad_norm = 1.0
momentum_decay = 0.9

# Example 1: Adding gradient clipping to Adam
opt_adam_with_clipping = optax.chain(
    optax.clip_by_global_norm(max_grad_norm),
    optax.adam(learning_rate=learning_rate_chained)
)
# model is an existing NNX model instance
optimizer_adam_clipped = nnx.Optimizer(model, opt_adam_with_clipping, wrt=nnx.Param)
```

Advanced Optax - Chained Transformations Code (Part 2)

```python
# Example 2: Building SGD with momentum and clipping from scratch
opt_sgd_manual = optax.chain(
    optax.clip_by_global_norm(max_grad_norm),
    optax.trace(decay=momentum_decay, nesterov=False), # Momentum
    optax.scale(-learning_rate_chained) # Scale by -LR
)
optimizer_sgd_manual = nnx.Optimizer(model, opt_sgd_manual, wrt=nnx.Param)

# Training loop (simplified for SGD example)
print("Starting training with manually chained SGD...")
for i in range(50):
  optimizer_sgd_manual, loss = train_step(optimizer_sgd_manual, x_dummy, y_dummy)
  if i % 10 == 0:
    print(f"Step {optimizer_sgd_manual.step.value}, Loss: {loss}")
```

Advanced Optax - Learning Rate Scheduling

- **Dynamic LR**: Adjusting the learning rate during training is vital.
  Two options:
  - **Optax Schedules**: These are functions that take the current training step count and return the learning rate for that step (e.g., **`optax.cosine_decay_schedule`**).
  - **`optax.inject_hyperparams`**: This higher-order function wraps an Optax transformation (like optax.adam) and allows its hyperparameters (e.g., learning_rate) to be dynamically controlled by a schedule function.

Advanced Optax - Learning Rate Scheduling

- **PyTorch Parallel:**
  - **NNX/Optax**: The schedule is integrated into the Optax transformation definition. No explicit `scheduler.step()` call is needed in the training loop; Optax handles it internally.
  - **PyTorch**: Schedulers (e.g., `torch.optim.lr_scheduler.StepLR`) are separate objects that wrap an optimizer, and you call `scheduler.step()` typically once per epoch.

Advanced Optax - LR Scheduling Code (Part 1)

```
total_training_steps = 10000
warmup_fraction = 0.1
peak_learning_rate = 1e-3
final_learning_rate = 1e-5

# Define the learning rate schedule function
lr_schedule_fn = optax.warmup_cosine_decay_schedule(
    init_value=0.0,
    peak_value=peak_learning_rate,
    warmup_steps=int(total_training_steps * warmup_fraction),
    decay_steps=int(total_training_steps * (1.0 - warmup_fraction)),
    end_value=final_learning_rate
)
```

Advanced Optax - LR Scheduling Code (Part 2)

```python
# Adam with learning rate schedule
opt_adam_with_schedule = optax.adam(learning_rate=lr_schedule_fn)

# nnx.Optimizer uses this Optax transform with the scheduled LR
optimizer_scheduled_lr = nnx.Optimizer(model, opt_adam_with_schedule, wrt=nnx.Param)

# The train_step function remains the same.
# LR is computed internally by Optax at each step.
# Example of checking LR (conceptual):
# lr_at_step_50 = lr_schedule_fn(50)
```

Advanced Optax - Per-Parameter Optimization

- **Goal**: Apply different optimization settings (e.g., learning rates, weight decay) to different parts of a model. For example, biases vs. kernels.
- **Optax Tools**: **`optax.partition`** is the primary tool for applying different, complete Optax transformations to distinct parameter subsets. optax.masked can also be used for more targeted applications.
- **PyTorch Parallel**: This is conceptually similar to "parameter groups" in PyTorch, where you pass a list of dictionaries to the optimizer constructor, each specifying parameters and their options. In Optax, this logic is configured within the Optax transformation itself.

Advanced Optax - `optax.partition`

- **`optax.partition`**: Applies different Optax transformations to distinct, non-overlapping subsets of parameters.
- Requires:
  - **A dictionary mapping** string labels to Optax transformations (e.g., {'biases_group': optax.sgd(...), 'kernels_group': optax.adam(...) }).
  - **A param_labels PyTree**: This PyTree must have the same structure as your model's parameters (**`nnx.state(model, nnx.Param)`**). Each leaf in **`param_labels`** contains a string label from the dictionary, indicating which transform to apply to that parameter.

Advanced Optax - `optax.partition`

- **Challenge: Creating the param_labels PyTree.**
    - This usually involves traversing the parameter PyTree (e.g., using `jax.tree_util.tree_map_with_path`) and assigning labels based on parameter names or paths.

Advanced Optax - `optax.partition` (Part 1 - Label)

```python
# Assume 'model' is an instance of SimpleMLP
params_pytree_for_labels = nnx.state(model, nnx.Param)

def label_fn(path, leaf):
  """Assigns a label to a parameter based on its path."""
  param_name = path[-1].name
  if 'bias' in param_name:
    return 'biases_group'
  elif 'kernel' in param_name:
    return 'kernels_group'
  return 'default_group'
```

Google

Advanced Optax - `optax.partition` (Part 2 - Partition)

```
param_labels_pytree = jax.tree.map_with_path(
    label_fn, params_pytree_for_labels
)

partitioned_opt = optax.partition(
    transforms={
        'kernels_group': optax.adam(learning_rate=1e-4),
        'biases_group': optax.sgd(learning_rate=1e-3),
        'default_group': optax.adam(learning_rate=1e-5)
    },
    param_labels=param_labels_pytree
)
optimizer_partitioned = nnx.Optimizer(model, partitioned_opt, wrt=nnx.Param)
```

Optax/Flax NNX vs. PyTorch - Key API Differences

Feature	PyTorch (torch.optim)	Optax / Flax
Optimizer Init	optim.Adam(model.parameters(), ...)	nnx.Optimizer(model, optax.adam(...), wrt=nnx.Param)
Param Groups	List of dicts in optimizer constructor	optax.partition configured in Optax transform
LR Scheduling	scheduler = StepLR(opt, ...) sched.step()	Ex: optax.warmup_cosine_decay_schedule(...)
Gradient Calc	loss.backward() (modifies .grad)	loss, grads = nnx.value_and_grad(loss_fn)(...)
Gradient Clearing	optimizer.zero_grad()	Implicit (new grads returned each time)
Parameter Update	optimizer.step()	optimizer.update(grads)

Distributed Training - JAX Sharding Fundamentals Review

- **`jax.sharding.Mesh`:** Defines a logical grid of your physical devices (e.g., GPUs/TPUs), with named axes (like 'data', 'model').
- **`jax.sharding.PartitionSpec (P)`:** A tuple specifying how each dimension of a JAX array (tensor) is sharded (or replicated).
- **`jax.sharding.NamedSharding`:** A pairing of a Mesh and a `PartitionSpec`, fully describing how an array should be distributed.
- **PyTorch Parallel:** JAX uses a unified sharding system instead of model wrappers like DDP and FSDP in Pytorch.

Review Distributed Training - Sharding NNX Model Parameters

```python
# Define sharding helper 'NS':
def NS(*names: str | None) -> NamedSharding:
  return NamedSharding(mesh, P(*names))

class SimpleMLP(nnx.Module): # (Partial definition)
  def __init__(self, din: int, dmid: int, dout: int, *, rngs: nnx.Rngs):
    self.dense1 = nnx.Linear(din, dmid, rngs=rngs)
    # ...
    if 'model' in mesh.axis_names:
      # Shard 2nd dim of kernel along 'model' axis
      self.dense1.kernel.sharding = NS(None, 'model')
      self.dense1.bias.sharding = NS('model') # Shard bias
    else: # Fallback: replicate if 'model' axis not present
      self.dense1.kernel.sharding = NS()
      self.dense1.bias.sharding = NS()
```

Distributed Training - Sharding `nnx.Optimizer`

- **Goal**: The optimizer's internal state (e.g., Adam's momentum `m` and variance `v` vectors) should be sharded identically to the model parameters they correspond to.
- **Process**: Exactly like model sharding, with one difference. Instead of:

```
nnx.state(model)
```

You use:

```
nnx.state(optimizer, nnx.optimizer.OptState)
```

Google

Distributed Training - Sharding (Part 1)

```
@nnx.jit()
def create_model_and_optimizer():
  model = SimpleLinear()
  optimizer = nnx.Optimizer(model, optax.adamw(1e-3), wrt=nnx.Param)

  # shard model state
  model_state = nnx.state(model)
  model_shardings = nnx.spmd.get_partition_spec(model_state, mesh)
  model_sharded_state = jax.lax.with_sharding_constraint(
    model_state, model_shardings
  )
  nnx.update(model, model_sharded_state)

  # Continued on next slide …
```

Distributed Training - Sharding (Part 2)

```
  # … Continued from previous slide
  # shard optimizer state
  optimizer_state = nnx.state(optimizer, nnx.optimizer.OptState) # select only the
optimizer state
  optimizer_shardings = nnx.spmd.get_partition_spec(optimizer_state, mesh)
  optimizer_sharded_state = jax.lax.with_sharding_constraint(
    optimizer_state, optimizer_shardings
  )
  nnx.update(optimizer, optimizer_sharded_state)

  return model, optimizer

with mesh: # Run inside a Mesh context
  model, optimizer = create_model_and_optimizer()
```

Conclusion & Best Practices

- **Optax + Flax NNX**: A highly flexible and powerful optimization framework within JAX, offering a PyTorch-like feel for model definition with NNX.
- **Recommendations for PyTorch Users:**
    - Define Optax transformations (opt) separately for clarity, especially for complex chains or schedules.
    - Always use `@nnx.jit` for training step functions for performance and correct state handling.
    - For per-parameter optimization rules, `optax.partition` is generally preferred. Master creating the `param_labels` PyTree.
    - Distributed training in JAX involves explicit sharding. While requiring careful setup, it provides fine-grained control.

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