# Common Libraries / Functionalities in PyTorch vs. JAX

| | |
|---|---|
| torch.Tensor | jax.numpy |
| torch.optim | optax |
| torch.nn | flax.linen |
| torch.utils.data | tensorflow_datasets |
| torch.distributed | pmap |

# Model Definition

## PyTorch

```python
class MLP(nn.Module):
    def __init__(self, input_dim: int, hidden_sizes=(1024, 512), num_classes=10):
        super().__init__()
        self.hidden_sizes = hidden_sizes
        self.num_classes = num_classes
        layers = []
        prev_dim = input_dim
        for h in hidden_sizes:
            layers.append(nn.Linear(prev_dim, h))
            layers.append(nn.ReLU())
            prev_dim = h
        layers.append(nn.Linear(prev_dim, num_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

## JAX

```python
class MLP(nn.Module):
    hidden_sizes: Tuple[int, ...] = (1024, 512)
    num_classes: int = 10

    @nn.compact
    def __call__(self, x):
        for h in self.hidden_sizes:
            x = nn.Dense(h)(x)
            x = nn.relu(x)
        x = nn.Dense(self.num_classes)(x)
        return x
```

# Model Definition

## PyTorch

```python
class MLP(nn.Module):
    def __init__(self, input_dim: int, hidden_sizes=(1024, 512), num_classes=10):
        super().__init__()
        self.hidden_sizes = hidden_sizes
        self.num_classes = num_classes
        layers = []
        prev_dim = input_dim
        for h in hidden_sizes:
            layers.append(nn.Linear(prev_dim, h))
            layers.append(nn.ReLU())
            prev_dim = h
        layers.append(nn.Linear(prev_dim, num_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

## JAX

```python
class MLP(nn.Module):
    hidden_sizes: Tuple[int, ...] = (1024, 512)
    num_classes: int = 10

    @nn.compact
    def __call__(self, x):
        for h in self.hidden_sizes:
            x = nn.Dense(h)(x)
            x = nn.relu(x)
        x = nn.Dense(self.num_classes)(x)
        return x
```

# Model Definition

## PyTorch

```
class MLP(nn.Module):
    def __init__(self, input_dim: int, hidden_sizes=(1024, 512), num_classes=10):
```

```
        for h in hidden_sizes:
            layers.append(nn.Linear(prev_dim, h))
            layers.append(nn.ReLU())
            prev_dim = h
        layers.append(nn.Linear(prev_dim, num_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        return self.net(x)
```

## JAX

```
class MLP(nn.Module):
```

```
    def __call__(self, x):
        for h in self.hidden_sizes:
            x = nn.Dense(h)(x)
            x = nn.relu(x)
        x = nn.Dense(self.num_classes)(x)
        return x
```

PyTorch builds and stores layers upfront
JAX builds them dynamically on first call

# Model Initialization

## PyTorch

```python
model = MLP(28 * 28)
```

## JAX

```python
rng = jax.random.PRNGKey(args.seed)

x = jnp.zeros((1, 28 * 28), jnp.float32)

params = model.init({"params": rng},
x)["params"]
```

# Model Initialization

## PyTorch

```python
model = MLP(28 * 28)
```

## JAX

```python
rng = jax.random.PRNGKey(args.seed)

x = jnp.zeros((1, 28 * 28), jnp.float32)

params = model.init({"params": rng},
x)["params"]
```

# Model Initialization

PyTorch

```python
model = MLP(28 * 28)
```

JAX

```python
rng = jax.random.PRNGKey(args.seed)
x = jnp.zeros((1, 28 * 28), jnp.float32)
```

PyTorch uses global RNG
JAX needs explicit PRNG key

# Model Initialization

## PyTorch

```
model = MLP(28 * 28
```

## JAX

```
rng = jax.random.PRNGKey(args.seed)

x = jnp.zeros((1, 28 * 28), jnp.float32)

params = model.init({"params": rng},
x)["params"]
```

# Model Initialization

PyTorch

```python
model = MLP(28 * 28
```

JAX

```python
rng = jax.random.PRNGKey(args.seed)
```

PyTorch uses explicit model configuration

JAX infers model configuration from example input

# Model Initialization

PyTorch

```python
model = MLP(28 * 28
```

JAX

```python
rng = jax.random.PRNGKey(args.seed)

x = jnp.zeros((1, 28 * 28), jnp.float32)

params = model.init({"params": rng},
x)["params"]
```

# Model Initialization

PyTorch

```
model = MLP(28 * 28
```

JAX

```
rng = jax.random.PRNGKey(args.seed)
```

PyTorch stores weights in model
JAX stores them externally

# Model Training

JAX

```python
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["label"]).mean()
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = train_step(params_repl, opt_state_repl, batch)
        loss = float(jax.device_get(loss)[0])
```

# Model Training

JAX

```
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["label"]).mean()
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = train_step(params_repl, opt_state_repl, batch)
        loss = float(jax.device_get(loss)[0])
```

explicitly replicates parameters and optimizer states across devices

# Model Training

JAX

```
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = train_step(params_repl, opt_state_repl, batch)
        loss = float(jax.device_get(loss)[0])
```

compiles the function on the first call
reuses on all devices in parallel

# Model Training

JAX

```python
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["label"]).mean()
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = train_step(params_repl, opt_state_repl, batch)
        loss = float(jax.device_get(loss)[0])
```

gradient syncing is explicit rather than automatic as in PyTorch DDP

# Model Training

## JAX

```
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["label"]).mean()
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = trai
        loss = float(jax.device_get(loss)[0])
```

use shard to split data to different devices, instead of using sampler

# Model Training

JAX

```
tx = optax.adamw(args.learning_rate)
opt_state_cpu = tx.init(params_cpu)
params_repl = flax.jax_utils.replicate(params_cpu)
opt_state_repl = flax.jax_utils.replicate(opt_state_cpu)

@functools.partial(jax.pmap, axis_name="data")
def train_step(params, opt_state, batch):
    def loss_fn(p):
        logits = model.apply({"params": p}, batch["image"])
        loss = optax.softmax_cross_entropy_with_integer_labels(logits, batch["label"]).mean()
        return loss

    loss, grads = jax.value_and_grad(loss_fn)(params)
    loss = jax.lax.pmean(loss, axis_name="data")
    grads = jax.lax.pmean(grads, axis_name="data")
    updates, opt_state = tx.update(grads, opt_state, params)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss

for epoch in range(args.num_epochs):
    for _ in range(args.steps_per_epoch):
        batch = next(train_iter)
        batch = shard(batch)
        params_repl, opt_state_repl, loss = train_step(params_repl, opt_state_repl, batch)
        loss = float(jax.device_get(loss)[0])
```

variables in JAX are immutable; thus, they are overwritten instead of updated

# Should I Switch to JAX?

Pro

- Potentially faster; JAX's JIT is more performant than torch.compile

Con

- Lack of libraries / community support

Suggestion: use JAX if any of the following cases apply

- Have access to more TPU resources than GPU resources
- Heavy array computations are needed outside of training (e.g., RL envs)
- Prefer stateless computation (e.g., meta-learning that needs access to grad)