Some useful libraries in the JAX ecosystem

Vadim Bertrand — JAXATHON 2025

Overview

- Optax — https://optax.readthedocs.io/ (optimization)
- Optimistix — https://docs.kidger.site/optimistix/ (optimization)
- Equinox — https://docs.kidger.site/equinox/ (models)
- Diffrax — https://docs.kidger.site/diffrax/ (DE solvers)
- Flax — https://flax.readthedocs.io/en/latest/ (Neural Networks)
- Grain — https://google-grain.readthedocs.io/en/latest/index.html (datasets)
- Orbax — https://orbax.readthedocs.io/en/latest/index.html (training)

https://github.com/n2cholas/awesome-jax

Optax

```python
import functools

import jax.numpy as jnp
import jax
import optax


@functools.partial(jax.vmap, in_axes=(None, 0))
def network(params, x):
  return jnp.dot(params, x)

def compute_loss(params, x, y):
  y_pred = network(params, x)
  loss = jnp.mean(optax.l2_loss(y_pred, y))
  return loss
```

```python
target_params = 0.5

xs = jax.random.normal(jax.random.PRNGKey(0), (16, 2))
ys = jnp.sum(xs * target_params, axis=-1)

optimizer = optax.adam(start_learning_rate=1e-1)

params = jnp.array([0.0, 0.0])
opt_state = optimizer.init(params)

for _ in range(1000):
  grads = jax.grad(compute_loss)(params, xs, ys)
  updates, opt_state = optimizer.update(grads, opt_state)
  params = optax.apply_updates(params, updates)
```

- Predefined losses
- Several optimizers
- Gradients transformations (mask, clip, finite…)
- Chaining optimizers / transformations

Optimistix

```python
import jax.numpy as jnp
import optimistix as optx

# Let's solve the ODE dy/dt=tanh(y(t)) with the implicit Euler method.
# We need to find y1 s.t. y1 = y0 + tanh(y1)dt.

y0 = jnp.array(1.)
dt = jnp.array(0.1)

def fn(y, args):
    return y0 + jnp.tanh(y) * dt

solver = optx.Newton(rtol=1e-5, atol=1e-5)
sol = optx.fixed_point(fn, solver, y0)
y1 = sol.value  # satisfies y1 == fn(y1)
```

- Different classes of problems (minimization, root finding, fixed points)
- Several solvers (BFGS, GD, GaussNewton, …)

Equinox

```python
import equinox as eqx
import jax

class Linear(eqx.Module):
    weight: jax.Array
    bias: jax.Array

    def __init__(self, in_size, out_size, key):
        wkey, bkey = jax.random.split(key)
        self.weight = jax.random.normal(wkey, (out_size, in_size))
        self.bias = jax.random.normal(bkey, (out_size,))

    def __call__(self, x):
        return self.weight @ x + self.bias
```

```python
@jax.jit
@jax.grad
def loss_fn(model, x, y):
    pred_y = jax.vmap(model)(x)
    return jax.numpy.mean((y - pred_y) ** 2)

batch_size, in_size, out_size = 32, 2, 3
model = Linear(in_size, out_size, key=jax.random.PRNGKey(0))
x = jax.numpy.zeros((batch_size, in_size))
y = jax.numpy.zeros((batch_size, out_size))
grads = loss_fn(model, x, y)
```

- Register classes as PyTrees
- Natively compatible with jit, grad, vmap, etc…
- Utility functions to manipulate Pytrees (e.g. filtering)
- Neural network layers

Diffrax

```python
from diffrax import diffeqsolve, ODETerm, Dopri5
import jax.numpy as jnp

def f(t, y, args):
    return -y

term = ODETerm(f)
solver = Dopri5()
y0 = jnp.array([2., 3.])
solution = diffeqsolve(term, solver, t0=0, t1=1, dt0=0.1, y0=y0)
```

- ODE and SDE solvers
- Support for dense solutions
- Multiple adjoints methods (recursive checkpoint, forward, implicit)
- Support for forward or reverse automatic differentiation

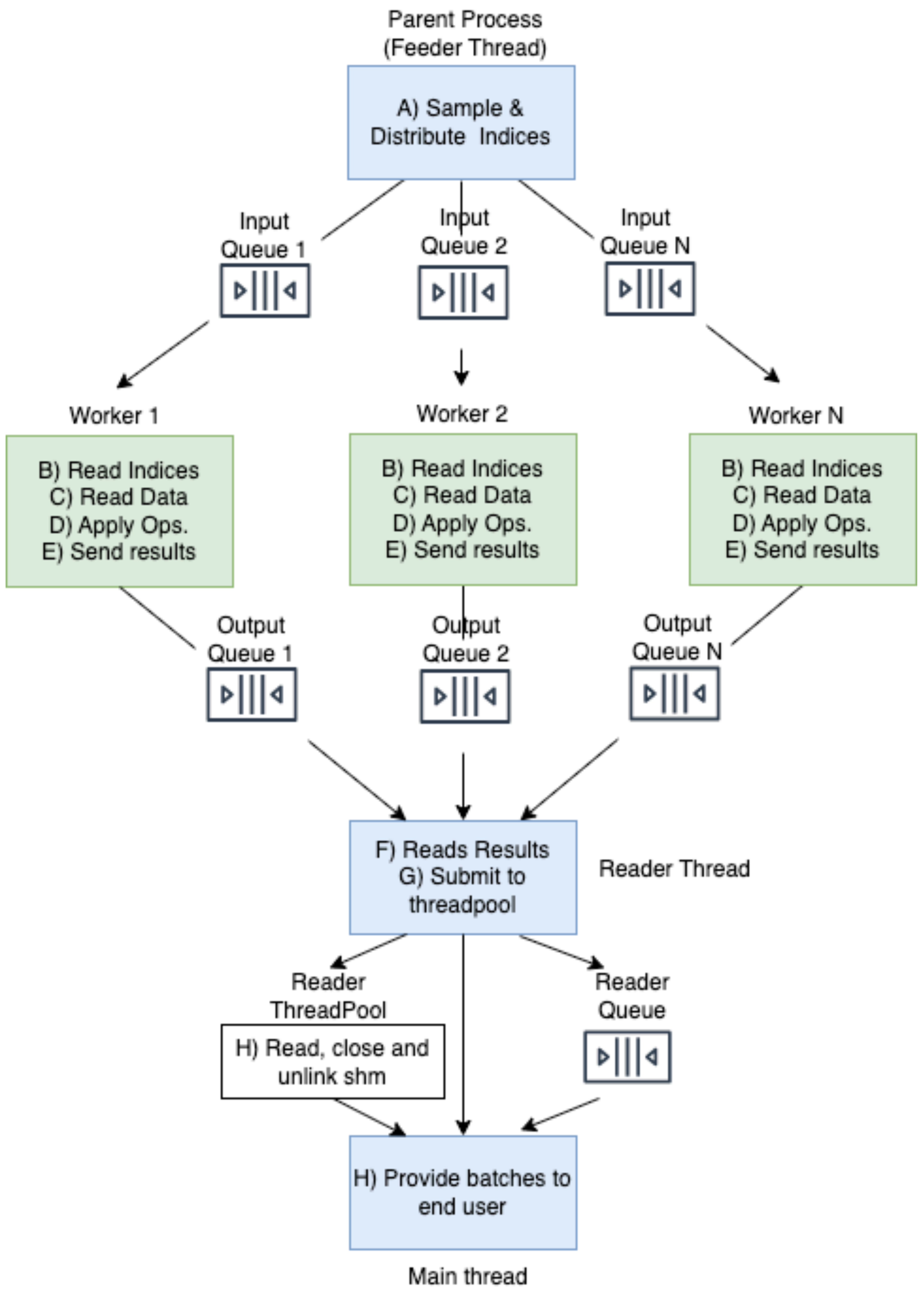
Flax

```python
from flax import nnx
import optax


class Model(nnx.Module):
  def __init__(self, din, dmid, dout, rngs: nnx.Rngs):
    self.linear = nnx.Linear(din, dmid, rngs=rngs)
    self.bn = nnx.BatchNorm(dmid, rngs=rngs)
    self.dropout = nnx.Dropout(0.2, rngs=rngs)
    self.linear_out = nnx.Linear(dmid, dout, rngs=rngs)

  def __call__(self, x):
    x = nnx.relu(self.dropout(self.bn(self.linear(x))))
    return self.linear_out(x)

model = Model(2, 64, 3, rngs=nnx.Rngs(0))  # eager initialization
optimizer = nnx.Optimizer(model, optax.adam(1e-3))  # reference sharing

@nnx.jit  # automatic state management for JAX transforms
def train_step(model, optimizer, x, y):
  def loss_fn(model):
    y_pred = model(x)  # call methods directly
    return ((y_pred - y) ** 2).mean()

  loss, grads = nnx.value_and_grad(loss_fn)(model)
  optimizer.update(grads)  # in-place updates

  return loss
```

Define and train
Neural Networks

Grain

Datasets batching



Orbax

```
checkpointer = ocp.StandardCheckpointer()
# 'checkpoint_name' must not already exist.
checkpointer.save(path / 'checkpoint_name', my_tree)
checkpointer.restore(
    path / 'checkpoint_name/',
    abstract_my_tree
)
```

Save and restore Pytrees (checkpointing)

Other librairies?

Different needs?