More on Optimization using JAX

Machine Learning Fundamentals for Economists

Jesse Perla

jesse.perla@ubc.ca

University of British Columbia

Table of contents

Linear Regression with Raw JAX

Packages

- `optax` is a common package for ML optimization methods

```python
import jax
import jax.numpy as jnp
from jax import grad, jit, value_and_grad, vmap
from jax import random
import optax
from flax import nnx
```

Simulate Data

- Few differences here, except for manual use of the `key`
- Remember that if you use the same `key` you get the same value.
- See JAX docs for more details

```python
N = 500  # samples
M = 2
sigma = 0.001
key = random.PRNGKey(42)
# Pattern: split before using key, replace name "key"
key, *subkey = random.split(key, num=4)
theta = random.normal(subkey[0], (M,))
X = random.normal(subkey[1], (N, M))
Y = X @ theta + sigma * random.normal(subkey[2], (N,))  # Adding noise
```

Dataloaders Provide Batches

- For more complicated data (e.g. images, text) JAX can use other packages, but it doesn't have a canonical dataloader at this point
- But in this case we can manually create this, using `yield`

```python
def data_loader(key, X, Y, batch_size):
    N = X.shape[0]
    assert N == Y.shape[0]
    indices = jnp.arange(N)
    indices = random.permutation(key, indices)
    # Loop over batches and yield
    for i in range(0, N, batch_size):
        b_indices = indices[i:i + batch_size]
        yield X[b_indices], Y[b_indices]
# e.g. iterate and get first element
dl_test = data_loader(key, X, Y, 4)
print(next(iter(dl_test)))
```

```
(Array([[-0.92034245, -0.7187076 ],
       [-0.6151726 ,  0.47314   ],
       [-0.35952824, -0.8299562 ],
       [ 0.88198936, -0.3076048 ]], dtype=float32),
Array([-1.1311196 ,  0.0050716 , -0.88230723,
0.28763232], dtype=float32))
```

Hypothesis Class

- The "Hypothesis Class" for our ERM approximation is linear in this case
- JAX is functional and non-mutating, so you must write stateless code
- We will move towards a more general class with the Flax NNX package, but for now we will implement the model with the parameters directly
- The underlying parameters will have a random initialization, which becomes **crucial** with overparameterized models (but wouldn't be important here)

```python
def predict(theta, X):
    return jnp.matmul(X, theta) #or jnp.dot(X, theta)

# Need to randomize our own theta_0 parameters
key, subkey = random.split(key)
theta_0 = random.normal(subkey, (M,))
print(f"theta_0 = {theta_0}, theta = {theta}")
```

```
theta_0 = [-0.21089035 -1.3627948 ], theta = [0.60576403 0.7990441 ]
```

Loss Function for Gradient Descent

- Reminder: need to provide AD-able functions which give a gradient estimate, not necessarily the objective itself!
- In particular, for LLS we simply can find the MSE between the prediction and the data for the batch itself
- For now, we are passing the `params` rather than the `model` itself

```python
def vectorized_residuals(params, X, Y):
    Y_hat = predict(params, X)
    return jnp.mean((Y_hat - Y) ** 2)
```

Optimizer

- The `optimizer.init(theta_0)` provides the initial state for the iterations
- With SGD it is empty, but with momentum/etc. it will have internal state

```python
lr = 0.001
batch_size = 16
num_epochs = 201

# optax.adam(lr) is worse here
optimizer = optax.sgd(lr)
opt_state = optimizer.init(theta_0)
print(f"Optimizer state:{opt_state}")
params = theta_0 # initial condition
```

```
Optimizer state:(EmptyState(), EmptyState())
```

Using Optimizer for a Step

- Here we write a (compiled) utility function which:
    1. Calculates the loss and gradient estimates for the batch
    2. Updates the optimizer state
    3. Applies the updates to the parameters
    4. Returns the updated parameters, optimizer state, and loss
- The reason to set this up as a function is to maintain JAXs “pure” style

```python
@jax.jit
def make_step(params, opt_state, X, Y):
  loss_value, grads = jax.value_and_grad(vectorized_residuals)(params, X, Y)
  updates, opt_state = optimizer.update(grads, opt_state, params)
  params = optax.apply_updates(params, updates)
  return params, opt_state, loss_value
```

Training Loop Version 1

- Note that unlike Pytorch the gradients are passed as parameters

```python
for epoch in range(num_epochs):
    key, subkey = random.split(key) # changing key for shuffling each epoch
    train_loader = data_loader(subkey, X, Y, batch_size)
    for X_batch, Y_batch in train_loader:
        params, opt_state, train_loss = make_step(params, opt_state, X_batch, Y_batch)
    if epoch % 100 == 0:
        print(f"Epoch {epoch},||theta - theta_hat|| = {jnp.linalg.norm(theta - params)}")

print(f"||theta - theta_hat|| = {jnp.linalg.norm(theta - params)}")
```

```
Epoch 0,||theta - theta_hat|| = 2.1659655570983887
Epoch 100,||theta - theta_hat|| = 0.0036812787875533104
Epoch 200,||theta - theta_hat|| = 6.539194873766974e-05
||theta - theta_hat|| = 6.539194873766974e-05
```

Auto-Vectorizing

- In the above case the `vectorized_residuals` was able to use a directly vectorized function.
- However in many cases it will be more convenient to write code for a single element of the finite-sum objectives
- Now we will rewrite our objective to demonstrate how to use `vmap`

```python
def residual(theta, x, y):
    y_hat = predict(theta, x)
    return (y_hat - y) ** 2

@jit
def residuals(theta, X, Y):
    # Use vmap, fixing the 1st argument
    batched_residuals = jax.vmap(residual, in_axes=(None, 0, 0))
    return jnp.mean(batched_residuals(theta, X, Y))
print(residual(theta_0, X[0], Y[0]))
print(residuals(theta_0, X, Y))
```

```
2.6319637
5.4140573
```

New Step and Initialization

- This simply changes the function used for the `value_and_grad` call to use the new `residuals` function and resets our optimizer

```
@jax.jit
def make_step(params, opt_state, X, Y):
  loss_value, grads = jax.value_and_grad(residuals)(params, X, Y)
  updates, opt_state = optimizer.update(grads, opt_state, params)
  params = optax.apply_updates(params, updates)
  return params, opt_state, loss_value
optimizer = optax.sgd(lr) # better than optax.adam here
opt_state = optimizer.init(theta_0)
params = theta_0
```

Training Loop Version 2

- Otherwise the training loop is the same

```
for epoch in range(num_epochs):
    key, subkey = random.split(key) # changing key for shuffling each epoch
    train_loader = data_loader(subkey, X, Y, batch_size)
    for X_batch, Y_batch in train_loader:
        params, opt_state, train_loss = make_step(params, opt_state, X_batch, Y_batch)
    if epoch % 100 == 0:
        print(f"Epoch {epoch},||theta - theta_hat|| = {jnp.linalg.norm(theta - params)}")

print(f"||theta - theta_hat|| = {jnp.linalg.norm(theta - params)}")
```

```
Epoch 0,||theta - theta_hat|| = 2.167938232421875
Epoch 100,||theta - theta_hat|| = 0.003675078274682164
Epoch 200,||theta - theta_hat|| = 6.522066541947424e-05
||theta - theta_hat|| = 6.522066541947424e-05
```

JAX Examples

- See examples/linear_regression_jax_sgd.py
  - → This implements the inline code above without the vmap
- See examples/linear_regression_jax_vmap.py
  - → This implements the `vmap` as above
  - → This also adds in an learning rate schedule
- See examples/linear_regression_jax_nnx.py and examples/linear_regression_jax_nnx_split.py for ones using the Flax NNX

Linear Regression with Flax

Flax NNX

- While it seems convenient to work in a functional style, when we move towards nested, deep approximations it can become cumbersome to manage the parameters
- Flax is a package which provides flexible ways to define and work with function approximations
  - → There is a newer (NNX) and older (Linen) interface. Use NNX.
- We will also introduce a DataLoader class to remove boilerplate

Hypothesis Class

- We are moving towards Neural Networks, which are a very broad class of approximations.
- Here lets just use a linear approximation with no constant term
- As always, the initial randomization will become increasingly important

```python
N, M, sigma = 500, 2, 0.001
rngs = nnx.Rngs(42)
model = nnx.Linear(M, 1, use_bias=False, rngs=rngs)
print(model.kernel) # the initial parameters
```

```
Param( # 2 (8 B)

  value=Array([[ 0.05825231],

         [-0.37180716]], dtype=float32)

)
```

Residuals Using the “Model”

- The model now contains all of the, potentially nested, parameters for the approximation class
- It provides call notation to evaluate the function with those parameters

```python
def residual(model, x, y):
    y_hat = model(x)
    return (y_hat - y) ** 2

def residuals_loss(model, X, Y):
    return jnp.mean(jax.vmap(residual, in_axes=(None, 0, 0))(model, X, Y))
theta = random.normal(rngs(), (M,))
X = random.normal(rngs(), (N, M))
Y = X @ theta + sigma * random.normal(rngs(), (N,))
```

Gradients of Models

- As discussed, we can find the gradients of richer objects than just arrays
- Optimizer updates use perturbations of the underlying PyTree
- Updates can be applied because the type of the gradients matches the underlying PyTree

```python
grads = nnx.grad(residuals_loss)(model, X, Y)
print(grads)
```

```
State({

  'kernel': Param( # 2 (8 B)

    value=Array([[-1.1906744],

                 [-2.351897 ]], dtype=float32)

  )

})
```

Setup Optimizer and Training Step

- Note the `@nnx.jit` which replaces `@jax.jit`

```python
@nnx.jit
def train_step(model, optimizer, X, Y):
    def loss_fn(model):
        return residuals_loss(model, X, Y)
    loss, grads = nnx.value_and_grad(loss_fn)(model)
    optimizer.update(model, grads)
    return loss
optimizer = nnx.Optimizer(model, optax.sgd(0.001), wrt=nnx.Param)
```

Run Optimizer

- Run optimizer and extract the parameters in the `model`

```python
batch_size = 64
for epoch in range(500):
    key, subkey = random.split(key)
    train_loader = data_loader(subkey, X, Y, batch_size)
    for X_batch, Y_batch in train_loader:
        loss = train_step(model, optimizer, X_batch, Y_batch)

    if epoch % 100 == 0:
        norm_diff = jnp.linalg.norm(theta - jnp.squeeze(model.kernel.value))
        print(f"Epoch {epoch},||theta-theta_hat|| = {norm_diff}")
norm_diff = jnp.linalg.norm(theta - jnp.squeeze(model.kernel.value))
print(f"||theta - theta_hat|| = {norm_diff}")
```

```
Epoch 0,||theta-theta_hat|| = 1.2717349529266357

Epoch 100,||theta-theta_hat|| = 0.24903634190559387

Epoch 200,||theta-theta_hat|| = 0.04919437691569328

Epoch 300,||theta-theta_hat|| = 0.009857591241598l3

Epoch 400,||theta-theta_hat|| = 0.0020401095971465ll
||theta - theta_hat|| = 0.0004721158475149423
```

Define a Custom Type

- “Neural Networks” are custom types which nest parameterized function calls
- Nest calls to other `nnx.Module` or create/use differentiable `nnx.Param`

```python
class MyLinear(nnx.Module):
    def __init__(self, in_size, out_size, rngs):
        self.out_size = out_size
        self.in_size = in_size
        self.kernel = nnx.Param(jax.random.normal(rngs(), (self.out_size, self.in_size)))
    # Similar to Pytorch's forward
    def __call__(self, x):
        return self.kernel @ x

model = MyLinear(M, 1, rngs = rngs)
```

Same Optimization Loop

```python
optimizer = nnx.Optimizer(model, optax.sgd(0.001), wrt=nnx.Param)
for epoch in range(500):
    for X_batch, Y_batch in train_loader:
        loss = train_step(model, optimizer, X_batch, Y_batch)

    if epoch % 100 == 0:
        norm_diff = jnp.linalg.norm(theta - jnp.squeeze(model.kernel.value))
        print(f"Epoch {epoch},||theta-theta_hat|| = {norm_diff}")
norm_diff = jnp.linalg.norm(theta - jnp.squeeze(model.kernel.value))
print(f"||theta - theta_hat|| = {norm_diff}")
```

```
Epoch 0,||theta-theta_hat|| = 0.6275200247764587
Epoch 100,||theta-theta_hat|| = 0.6275200247764587
Epoch 200,||theta-theta_hat|| = 0.6275200247764587
Epoch 300,||theta-theta_hat|| = 0.6275200247764587
Epoch 400,||theta-theta_hat|| = 0.6275200247764587
||theta - theta_hat|| = 0.6275200247764587
```

Filtering Transformations

- Much of the NNX package is built around **filtering** members of the underlying python class
- Within an `nnx.Module` the `nnx.Param` are values which you might look to differentiate, others are fixed
- Since JAX code is (primarily) “pure” and functional, a key part of the package is to split and recombine parameters intended for gradients from those which are not

Splitting into Differentiable Parameters

- For our custom type, the fields are `out_size, in_size, kernel`. We only want to differentate the `kernel` since wrapped in `nnx.Param`
- To separate out parameters use `nnx.split` and to recombine use `nnx.merge`

```python
model = MyLinear(M, 1, rngs = rngs)
graphdef, state = nnx.split(model)
print(graphdef)
```

```
GraphDef(nodes=[NodeDef(
  type='MyLinear',
  index=0,
  outer_index=None,
  num_attributes=5,
  metadata=MyLinear
), NodeDef(
  type='GenericPytree',
  index=None,
  outer_index=None,
  num_attributes=0,
  metadata=({}, PyTreeDef(CustomNode(PytreeState[(False,
False)], [])))
), VariableDef(
  type='Param',
  index=1,
  outer_index=None,
  metadata=PrettyMapping({
```

Merging

- `graphdef` was the fixed structure, `state` is the differentiable
- Use `nnx.merge` to combine the fixed and differentiable parts

```python
print(state)
# Emulate a "gradient" update
def apply_fake_gradient(param):
    return param + 0.01
# Apply "gradient" update to tree
state_2 = jax.tree_util.tree_map(
                apply_fake_gradient, state)
# Combine to form a model
model_2 = nnx.merge(graphdef, state_2)
print(model_2)
```

```
State({
  'kernel': Param( # 2 (8 B)
    value=Array([[-0.2166012, -1.9878021]], dtype=float32)
  )
})
MyLinear( # Param: 2 (8 B)
  in_size=2,
  kernel=Param( # 2 (8 B)
    value=Array(shape=(1, 2), dtype=dtype('float32'))
  ),
  out_size=1
)
```

More Advanced Optimization Loops

- Filtering is often automated by replacing `jax` with `nnx` equivalents
  → `nnx.jit, nnx.value_and_grad` etc. automatically filter for Params
- This process provides some overhead, so for high-speed examples may manually split and merge