You don't know JAX

Colin Raffel
January 16th, 2019
colinraffel.com/blog

This brief tutorial covers the basics of JAX. JAX is a Python library which augments `numpy` and Python code with *function transformations* which make it trivial to perform operations common in machine learning programs. Concretely, this makes it simple to write standard Python/`numpy` code and immediately be able to

- Compute the derivative of a function via a successor to autograd
- Just-in-time compile a function to run efficiently on an accelerator via XLA
- Automagically vectorize a function, so that e.g. you can process a "batch" of data in parallel

In this tutorial, we'll cover each of these transformations in turn by demonstrating their use on one of the core problems of AGI: learning the Exclusive OR (XOR) function with a neural network.

Note - this blog post is available as an interactive Jupyter notebook here.

1 JAX is just `numpy` (mostly)

At its core, you can think of JAX as augmenting `numpy` with the machinery required to perform the aforementioned transformations. JAX's augmented numpy lives at `jax.numpy`. With a few exceptions, you can think of `jax.numpy` as directly interchangeable with `numpy`. As a general rule, you should use `jax.numpy` whenever you plan to use any of JAX's transformations (like computing gradients or just-in-time compiling code) and whenever you want the code to run on an accelerator. You only ever *need* to use `numpy` when you're computing something which is not supported by `jax.numpy`.

```
import random
import itertools

import jax
import jax.numpy as np
# Current convention is to import original numpy as "onp"
import numpy as onp

from __future__ import print_function
```

2 Background

As previously mentioned, we will be learning the XOR function with a small neural network. The XOR function takes as input two binary numbers and outputs a binary number, like so:

In 1	In 2	Out
0	0	0
0	1	1
1	0	1
1	1	0

We'll use a neural network with a single hidden layer with 3 neurons and a hyperbolic tangent nonlinearity, trained with the cross-entropy loss via stochastic gradient descent. Let's implement this model and loss function. Note that the code is exactly as you'd write in standard `numpy`.

```
# Sigmoid nonlinearity
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Computes our network's output
def net(params, x):
    w1, b1, w2, b2 = params
    hidden = np.tanh(np.dot(w1, x) + b1)
    return sigmoid(np.dot(w2, hidden) + b2)

# Cross-entropy loss
def loss(params, x, y):
    out = net(params, x)
    cross_entropy = -y * np.log(out) - (1 - y)*np.log(1 - out)
    return cross_entropy

# Utility function for testing whether the net produces the correct
# output for all possible inputs
def test_all_inputs(inputs, params):
    predictions = [int(net(params, inp) > 0.5) for inp in inputs]
    for inp, out in zip(inputs, predictions):
        print(inp, '->', out)
    return (predictions == [onp.bitwise_xor(*inp) for inp in inputs])
```

As mentioned above, there are some places where we want to use standard `numpy` rather than `jax.numpy`. One of those places is with parameter initialization. We'd like to initialize our parameters randomly before we train our network, which is not an operation for which we need derivatives or compilation. JAX uses its own `jax.random` library instead of `numpy.random` which provides better support for reproducibility (seeding) across different transformations. Since we don't need to transform the initialization of parameters in any way, it's simplest just to use standard `numpy.random` instead of `jax.random` here.

```
def initial_params():
    return [
        onp.random.randn(3, 2),  # w1
        onp.random.randn(3),  # b1
        onp.random.randn(3),  # w2
        onp.random.randn(),  #b2
    ]
```

3 `jax.grad`

The first transformation we'll use is `jax.grad`. `jax.grad` takes a function and returns a new function which computes the gradient of the original function. By default, the gradient is taken with respect to the first argument; this can be controlled via the `argnums` argument to `jax.grad`.

To use gradient descent, we want to be able to compute the gradient of our loss function with respect to our neural network's parameters. For this, we'll simply use `jax.grad(loss)` which will give us a function we can call to get these gradients.

```
loss_grad = jax.grad(loss)

# Stochastic gradient descent learning rate
learning_rate = 1.
# All possible inputs
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

# Initialize parameters randomly
params = initial_params()

for n in itertools.count():
    # Grab a single random input
    x = inputs[onp.random.choice(inputs.shape[0])]
    # Compute the target output
    y = onp.bitwise_xor(*x)
    # Get the gradient of the loss for this input/output pair
    grads = loss_grad(params, x, y)
    # Update parameters via gradient descent
    params = [param - learning_rate * grad
              for param, grad in zip(params, grads)]
    # Every 100 iterations, check whether we've solved XOR
    if not n % 100:
        print('Iteration {}'.format(n))
        if test_all_inputs(inputs, params):
            break
```

```
Iteration 0
[0 0] -> 1
[0 1] -> 0
[1 0] -> 1
[1 1] -> 1
Iteration 100
[0 0] -> 0
[0 1] -> 1
[1 0] -> 1
[1 1] -> 0
```

4 `jax.jit`

While carefully-written `numpy` code can be reasonably performant, for modern machine learning we want our code to run as fast as possible. This often involves running our code on different "accelerators" like GPUs or TPUs. JAX provides a JIT (just-in-time) compiler which takes a standard Python/`numpy` function and compiles it to run efficiently on an accelerator. Compiling a

function also avoids the overhead of the Python interpreter, which helps whether or not you're using an accelerator. In total, `jax.jit` can dramatically speed-up your code with essentially no coding overhead - you just ask JAX to compile the function for you. Even our tiny neural network can see a pretty dramatic speedup when using `jax.jit`:

```
# Time the original gradient function
%timeit loss_grad(params, x, y)
loss_grad = jax.jit(jax.grad(loss))
# Run once to trigger JIT compilation
loss_grad(params, x, y)
%timeit loss_grad(params, x, y)
```

```
10 loops, best of 3: 13.1 ms per loop
1000 loops, best of 3: 862 µs per loop
```

Note that JAX allows us to aribtrarily chain together transformations - first, we took the gradient of `loss` using `jax.grad`, then we just-in-time compiled it using `jax.jit`. This is one of the things that makes JAX extra powerful — apart from chaining `jax.jit` and `jax.grad`, we could also e.g. apply `jax.grad` multiple times to get higher-order derivatives. To make sure that training the neural network still works after compilation, let's train it again. Note that the code for training has not changed whatsoever.

```
params = initial_params()

for n in itertools.count():
    x = inputs[onp.random.choice(inputs.shape[0])]
    y = onp.bitwise_xor(*x)
    grads = loss_grad(params, x, y)
    params = [param - learning_rate * grad
              for param, grad in zip(params, grads)]
    if not n % 100:
        print('Iteration {}'.format(n))
        if test_all_inputs(inputs, params):
            break
```

```
Iteration 0
[0 0] -> 1
[0 1] -> 1
[1 0] -> 1
[1 1] -> 1
Iteration 100
[0 0] -> 0
[0 1] -> 1
[1 0] -> 1
[1 1] -> 0
```

5 `jax.vmap`

An astute reader may have noticed that we have been training our neural network on a single example at a time. This is “true” stochastic gradient descent; in practice, when training modern machine learning models we perform “minibatch” gradient descent where we average the loss gradients over a mini-batch of examples at each step of gradient descent. JAX provides `jax.vmap`, which is a transformation which automatically “vectorizes” a function. What this means is that it allows you to compute the output of a function in parallel over some axis of the input. For us, this means we can apply the `jax.vmap` function transformation and immediately get a version of our loss function gradient which is amenable to using a minibatch of examples.

`jax.vmap` takes in additional arguments:

- `in_axes` is a tuple or integer which tells JAX over which axes the function's arguments should be parallelized. The tuple should have the same length as the number of arguments of the function being `vmap`'d, or should be an integer when there is only one argument. In our example, we'll use `(None, 0, 0)`, meaning “don't parallelize over the first argument (`params`), and parallelize over the first (zeroth) dimension of the second and third arguments (`x` and `y`)".
- `out_axes` is analogous to `in_axes`, except it specifies which axes of the function's output to parallelize over. In our case, we'll use `0`, meaning to parallelize over the first (zeroth) dimension of the function's sole output (the loss gradients).

Note that we will have to change the training code a little bit - we need to grab a batch of data instead of a single example at a time, and we need to average the gradients over the batch before applying them to update the parameters.

```
loss_grad = jax.jit(jax.vmap(jax.grad(loss), in_axes=(None, 0, 0), out_axes=0))

params = initial_params()

batch_size = 100

for n in itertools.count():
    # Generate a batch of inputs
    x = inputs[onp.random.choice(inputs.shape[0], size=batch_size)]
    y = onp.bitwise_xor(x[:, 0], x[:, 1])
    # The call to loss_grad remains the same!
    grads = loss_grad(params, x, y)
    # Note that we now need to average gradients over the batch
    params = [param - learning_rate * np.mean(grad, axis=0)
              for param, grad in zip(params, grads)]
    if not n % 100:
        print('Iteration {}'.format(n))
        if test_all_inputs(inputs, params):
            break
```

```
Iteration 0
[0 0] -> 0
[0 1] -> 0
[1 0] -> 0
[1 1] -> 0
Iteration 100
[0 0] -> 0
[0 1] -> 1
[1 0] -> 1
[1 1] -> 0
```

6 Pointers

That's all we'll be covering in this short tutorial, but this actually covers a great deal of JAX. Since JAX is mostly `numpy` and Python, you can leverage your existing knowledge instead of having to learn a fundamentally new framework or paradigm. For additional resources, check the notebooks and examples directories on JAX's GitHub.

formatted by Markdeep 1.03