Solving Optimization Problems with JAX

Mazeyar Moeini Feizabadi

May 25, 2020

1 Introduction

What is JAX? As described by the main JAX webpage, JAX is Autograd and XLA, brought together for high-performance machine learning research. JAX essentially augments the numpy library to create a nouvelle library with Autograd, Vector Mapping (vmap), Just In Time compilation (JIT), all compiled with Accelerated Linear Algebra (XLA) with TPU support and much more. With all of these features, problems that depend on linear algebra and matrix methods can be solved more efficiently. The purpose of this document is to show that indeed, these features can be used to solve a range of simple to complex optimization problems with matrix methods, and to provide an intuitive understanding of the mathematics and implementation behind the code.

Firstly we will be required to import the JAX libraries and nanargmin and nanargmax from numpy, as they are not implemented in JAX yet. If you are using Google Colab, there is no installation of JAX required, as JAX is open-sourced and maintained by Google.

```
import jax.numpy as np
from jax import grad, jit, vmap
from jax import random
from jax import jacfwd, jacrev
from jax.numpy import linalg

from numpy import nanargmin,nanargmax

key = random.PRNGKey(42)

```

Importing the Libraries

2 Grad, Jacobians and Vmap

Grad is best used for taking the automatic derivative of a function. It creates a function that evaluates the gradient of a given function. If we called $grad(grad(f))$, this would be the second derivative.

```
def f(x) : return 3*x[0]**2
gradf = grad(f)
gradf(np.array([2.0]))
>> 12.0

```

Jacobian is best used for taking the automatic derivative of a function with a vector input. We can see that it returns the expected vector from a circle function given that $\nabla circle = [2x, 2y]$

```
def circle(x): return x[0]**2 + x[1]**2
J = jacfwd(circle)
```

```
J(np.array([1.0,2.0]))
>> [2.0, 4.0]

```

Even more interesting is how we can compute the Hessian of a function by computing the Jacobian twice; this is what makes JAX so powerful! We see that the function *hessian* takes in a function and return a function as well.

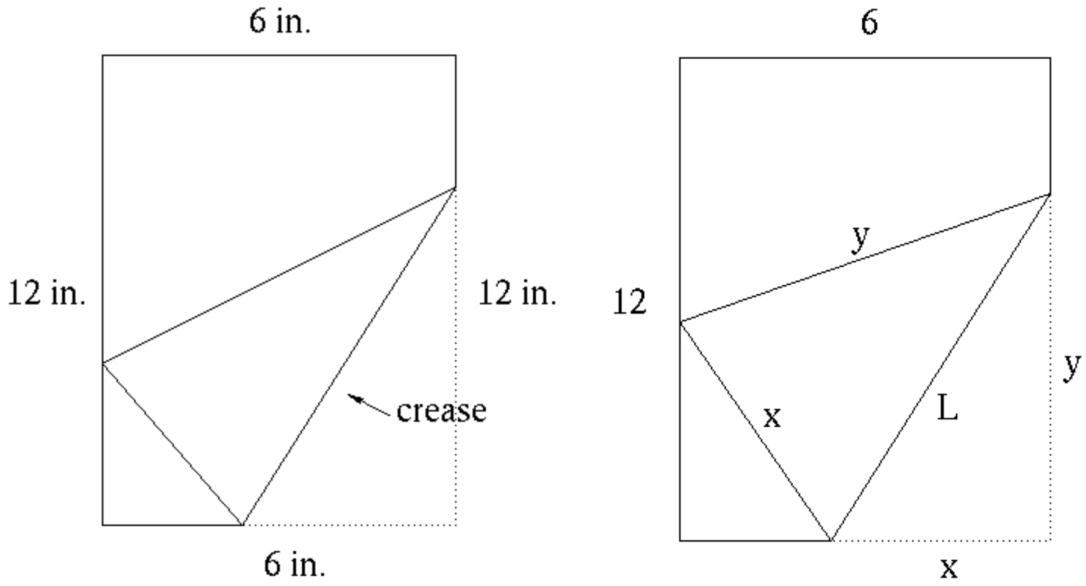
```
def circle(x): return x[0]**2 + x[1] **2
def hessian(f): return jacfwd(jacrev(f))
H = hessian(circle)
H(np.array([1.0,2.0]))
>> [[2.0, 0.0],
[0.0, 2.0]]

```

It should be noted that the gradients are computed with automatic differentiation, which is much more accurate and efficient compared to finite differences.

3 Single Variable Optimization

3.1 Gradient Descent

Let's imagine that we have the following optimization problem from UC Davis; A rectangular piece of paper is 12 inches in length and six inches wide. The lower right-hand corner is folded over so as to reach the leftmost edge of the paper, find the minimum length of the resulting crease where L is the length.



After doing some trigonometry, we can find the length of the crease in respect to the variable x to be:

$$L(x) = \sqrt{x^2 + \left(\frac{x\sqrt{12x-36}}{2(x-3)}\right)^2}$$

To find the minimum we would have to check all the critical points such that $L^{'} = 0$. However, although this is a relatively simple optimization problem, it would still lead to a

messy derivative that requires chain rule and quotient rule. Therefore, as these problems only become more complex, it would be wise to find numerical methods to solve them.

Jumping over to JAX, we can define the functions in python.

```
def y(x): return ((x * np.sqrt(12*x - 36 )) / (2*(x - 3)))
def L(x): return np.sqrt(x**2 + y(x)**2)

```

Then, using $grad(L)$ we can find the derivative of L and minimize this using step wise gradient descent.

$$x_{n+1} = x_n - 0.01L^{'}(x_n)$$

```
gradL = grad(L)

def minGD(x): return x - 0.01 * gradL(x)

domain = np.linspace(3.0, 5.0, num=50)

vfuncGD = vmap(minGD)
for epoch in range(50):
 domain = vfuncGD(domain)

minfunc = vmap(L)
minimums = minfunc(domain)

```

We can see how simple things become with JAX; the actual optimization happens with 6 lines of code! Notice how first *vmap* is used in each epoch to map the *minGD* function over the whole domain, then it's used to map the domain with the objective function L to find the objective minimum and argmin.

```
arglist = nanargmin(minimums)
argmin = domain[arglist]
minimum = minimums[arglist]

print("The minimum is {} the arg min is {}".format(minimum,argmin))

>> The minimum is 7.794247150421143 the argmin is 4.505752086639404

```

The numeric answer gives a 0.001851% error from the actual answer which is $9\frac{\sqrt{3}}{2}$, the error is acceptable given that the true value is an irrational number to begin with.

3.2 Newton's Method

The same problem can be solved using Newton's Method. Usually, Newton's Method is used for solving a function that is equal to zero such as $f(x) = x^2 - 2 = 0$, in the form of:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

This can easily be used for optimization given that we search for $f^{'}(x) = 0$

$$x_{n+1} = x_n - \frac{f^{'}(x_n)}{f^{''}(x_n)}$$

Newton's Method for optimization can easily be implemented with JAX.

```
1  gradL = grad(L)
2  gradL2 = grad(gradL)
3
4  def minNewton(x): return x - gradL(x)/gradL2(x)
5
6  domain = np.linspace(3.0, 5.0, num=50)
7  vfuncNT = vmap(minNewton)
8  for epoch in range(50):
9  domain = vfuncNT(domain)
10
11 minimums = minfunc(domain)
12
```

Notice how easily $L^{''}$ is calculated in line 2 of the code.

```
1  arglist = nanargmin(minimums)
2  argmin = domain[arglist]
3  minimum = minimums[arglist]
4
5  print("The minimum is {} the argmin is {}".format(minimum,argmin))
6
7  >> The minimum is 7.794229030609131 the argmin is 4.5
8
```

Newton's Method has the added advantage of the error being squared in each step.

4 Multivariable Optimization

4.1 The Jacobian

In multivariable problems we define functions such that $f(X), X = [x_0, x_1, x_2, \ldots, x_n]$. When the number of variables increases, we can no longer use the normal derivative; it requires the Jacobian also written as ∇f.

$$\nabla f(X) = \left[\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]$$

A Jacobian is a derivative of multivariable function, therefore, it captures how each variable affects a function. Since these are the first derivatives, we can again use these to optimize a multivariable function.

$$X_{n+1} = X_n - 0.01\nabla f(X_n)$$

Now, to implement this with JAX is just as simple as the single variable case. We will optimize:

$$f(x_0, x_1) = (x_0 x_1 - 2)^2 + (x_1 - 3)^2$$

Notice again how easily JAX allows us to calculate the Jacobian.

```
def paraboloid(x): return (x[0]*x[1]-2)**2 + (x[1]-3)**2
minfunc = vmap(paraboloid)

J = jacfwd(paraboloid)

```

Similar to last time, once we have the optimization function we can run it through a loop.

```
def minJacobian(x): return x - 0.1*J(x)

domain = random.uniform(key, shape=(50,2),dtype='float32',minval=-5.0,maxval
    =5.0)

vfuncHS = vmap(minJacobian)
for epoch in range(150):
domain = vfuncHS(domain)


minimums = minfunc(domain)

```

Then we check for the results.

```
arglist = nanargmin(minimums)
argmin = domain[arglist]
minimum = minimums[arglist]

print("The minimum is {} the arg min is ({},{})".format(minimum,argmin[0],argmin
    [1]))

>> The minimum is 0.0 the argmin is (0.6666666865348816,3.0)

```

4.2 The Hessian

In multivariable problems we define functions such that $f(X), X = [x_0, x_1, x_2, \ldots, x_n]$. Previously we defined the Jacobian (∇f). The Hessian is just $(\nabla(\nabla f))$ or $\nabla^2 f$ which requires the differentiation of each function in the Jacobian to all variables, thus increasing the dimension.

$$\nabla f(X) = \left[\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]$$

$$\nabla^2 f(X) = H(X) = \begin{bmatrix} \frac{\partial f}{\partial x_0^2} & \frac{\partial f}{\partial x_0 x_1} & \cdots & \frac{\partial f}{\partial x_0 x_n} \\ \frac{\partial f}{\partial x_1 x_0} & \frac{\partial f}{\partial x_2^2} & \cdots & \frac{\partial f}{\partial x_1 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial x_n x_0} & \frac{\partial f}{\partial x_n x_1} & \cdots & \frac{\partial f}{\partial x_n^2} \end{bmatrix}$$

To use the Hessian in optimization, it is really similar to Newton's Method. In fact, it is analogous.

$$x_{n+1} = x_n - \frac{f^{'}(x_n)}{f^{''}(x_n)}$$
$$X_{n+1} = X_n - H^{-1}(X_n)\nabla f(X_n)$$

We can observe, where it's not possible to divide by a matrix, we multiply by its inverse. There is a mathematical explanation for this using the quadratic term of a Taylor expansion, however, it is too lengthy to explain.

Again using the autograd library it is incredibly easy to calculate the Hessian.

```
def hessian(f):
return jacfwd(jacrev(f))

H = hessian(paraboloid)

```

Then it's just the same iterative loop as before. However, this time we use the minHessian function to optimize.

```
def minHessian(x): return x - 0.1*linalg.inv(H(x)) @ J(x)


domain = random.uniform(key, shape=(50,2), dtype='float32', minval=-5.0, maxval
    =5.0)

vfuncHS = vmap(minHessian)
for epoch in range(150):
domain = vfuncHS(domain)


minimums = minfunc(domain)

```

After running the loop, we look for the argmin and the objective minimum

```
arglist = nanargmin(minimums)
argmin = domain[arglist]
minimum = minimums[arglist]

print("The minimum is {} the arg min is ({},{})".format(minimum,argmin[0],argmin
    [1]))

>> The minimum is 9.094947017729282e-13 the argmin is
    (0.6666664481163025,3.0000009536743164)

```

5 Multivariable Constrained Optimization

Multivariable constrained optimization uses the same techniques as before but requires a different way of framing the problem. For constrained optimization we will use Lagrangian multipliers. The classic Lagrange equation requires solving for $\nabla f = \lambda \nabla g$. However, computers have no way of symbolically solving this. Rather, we can rewrite the equation as $\nabla f - \lambda \nabla g = 0$ which is now an unconstrained optimization problem.

$$\begin{aligned}\nabla f &= \lambda \nabla g \\ \nabla f - \lambda \nabla g &= 0 \\ \nabla (f - \lambda g) &= 0 \\ \nabla (\mathcal{L}) &= 0\end{aligned}$$

Just like the other optimization problems, we have a function that needs to be solved at zero $\nabla \mathcal{L} = 0$. Note the solving $\nabla \mathcal{L} = 0$ is no different than solving for systems of nonlinear equations. Our final iterative equation will look similar.

$$X_{n+1} = X_n - \nabla^2 \mathcal{L}^{-1}(X_n) \nabla \mathcal{L}(X_n)$$

The reason for the Hessian being involved again is due to minimizing $\mathcal{L}$ and solving for $\nabla \mathcal{L} = 0$ being the same statement. Also, when using Lagrangian multipliers we have to introduce a new variable λ in the code, $\mathcal{L}(X)$ will take in X where $X = [x_0, x_1, \lambda]$.

Let's say we have the objective function $f(X)$ and the constraint $g(X)$, in the code λ is $l[3]$.

$$\begin{aligned}f(X) &= 4x_0^2 x_1 \\ g(X) &= x_0^2 + x_1^2 - 3\end{aligned}$$

```
def f(x): return 4*(x[0]**2)*x[1]
def g(x): return x[0]**2 + x[1]**2 - 3

minfunc = vmap(f)

def Lagrange(l): return f(l[0:2]) - l[3]*g(l[0:2])

L = jacfwd(Lagrange)
gradL = jacfwd(L)

```

Using the same loop as before.

```
def solveLagrangian(l): return l - linalg.inv(gradL(l)) @ L(l)


domain = random.uniform(key, shape=(50,3), dtype='float32', minval=-5.0, maxval
    =5.0)

vfuncsLAG = vmap(solveLagrangian)
for epoch in range(150):
domain = vfuncsLAG(domain)

minimums = minfunc(domain)

```

Finding the argmin and objective minimum.

```
arglist = nanargmin(minimums)
argmin = domain[arglist]
minimum = minimums[arglist]
```

```
print("The minimum is {}, the arg min is ({},{}), the lagrangian is {}".format(
    minimum, argmin[0], argmin[1], argmin[2]))

>> The minimum is -7.999999523162842, the arg min is (-1.4142135381698608,-1.0),
    the lagrangian is -4.0
```

The correct minimum is -8, the argmin should be $(\sqrt{2}, -1)$, and since we included the λ in our calculation we find the Lagrangian multiplier is -4.0.

6 Three Variable Multivariable Constrained Optimization

Problems in real life usually have more than two variables to be optimized and optimization hyperparameters need to be fine tuned. As the complexity of optimization problems increases, other methods should be considered. For now we can use the models from the previous section and just increase the number of variables. Luckily, JAX will automatically adjust for this, we just need to adjust the L function in the code.

Let's attempt to solve a problem with real life applications found from Paul's Online Notes; Find the dimensions of the box with the largest volume if the total surface area is $64cm^2$. Our objective function is $f(x) = x_0x_1x_2$ the constraint is $g(x) = 2x_0x_1 + 2x_1x_2 + 2x_0x_2 - 64$. First we have to define the functions, then the only thing that we have to change is the index of the list feeding into *Lagrange*.

```
def f(x): return x[0]*x[1]*x[2]
def g(x): return 2*x[0]*x[1] + 2*x[1]*x[2] + 2*x[0]*x[2] - 64

minfunc = vmap(f)

def Lagrange(l): return f(l[0:3]) - l[3]*g(l[0:3])

L = jacfwd(Lagrange)
gradL = jacfwd(L)
```

This part of the code stays exactly the same except we add a learning rate of 0.1 to gain greater accuracy. We might also have to increase the total epochs.

$$X_{n+1} = X_n - 0.1\nabla^2\mathcal{L}^{-1}(X_n)\nabla\mathcal{L}(X_n)$$

```
def solveLagrangian(l): return l - 0.1*linalg.inv(gradL(l)) @ L(l)

domain = random.uniform(key, shape=(50,4), dtype='float32', minval=0, maxval=10)

vfuncsLAG = vmap(solveLagrangian)
for epoch in range(200):
domain = vfuncsLAG(domain)


maximums = minfunc(domain)
```

This time we are using *nanargmax* because the is maximization problem.

```
arglist = nanargmax(maximums)
argmin = domain[arglist]
```

```
minimum = maximums[arglist]

print("The minimum is {}, the argmin is ({},{},{}), the lagrangian is {}".format
    (minimum, argmin[0], argmin[1], argmin[2], argmin[3]))

>>   The minimum is 34.83720, the argmin is (3.265987,3.265985,3.265987), the
    lagrangian is 0.8164968

```

The real answer is $\sqrt{\frac{32}{3}}^3 \approx 34.837187$ the length of each side should be $\sqrt{\frac{32}{3}} \approx 3.265986$ the calculation is almost perfect as the errors are negligible in real life. It's important to note that without the learning rate, optimization is unlikely and accuracy was increased by doubling the number of epochs. Hopefully it is now obvious how more variables can be included in the optimization model.

7 Multivariable MultiConstrained Optimization

In the final part of this tutorial we will look at one of the most advanced types of optimization problems, multivariable multiconstrained optimization problems. Some of the problems in the beginning are admittedly better solved by hand. However as complexity increases, other numerical methods might be needed. Gradient Descent, no matter how many epochs and hyperparameters, can never 100% guarantee the best result but it always better than a random guess.

Let's start by trying to maximize the object function $f(x_0, x_1)$ with the constraints $g(x_0, x_1)$ and $h(x_0, x_1)$.

$$
\begin{aligned}
f(x_0, x_1) &= 13x_0^2 + 10x_0x_1 + 7x_1^2 + x_0 + x_1 + 2 \\
g(x_0, x_1) &= 2x_0 - 5x_1 - 2 \\
h(x_0, x_1) &= x_0 + x_1 - 1
\end{aligned}
$$

More problems like this can be found at Duke University. The general form of the Lagrangian function can be written such that the Jacobian of the objective minus each constraint function Jacobian multiplied by a respective lambda is equal to zero.

$$\mathcal{L}(X) = \nabla f(X) - \sum_{n=1}^{M} \lambda_n \nabla g_n(X) = 0$$

$$\mathcal{L}(X) = f(X) - \lambda_1 g(X) - \lambda_2 h(X) = 0$$

$$\mathcal{L}(X) = 13x_0^2 + 10x_0x_1 + 7x_1^2 + x_0 + x_1 + 2 - \lambda_1(2x_0 - 5x_1 - 2) - \lambda_2(x_0 + x_1 - 1) = 0$$

We see now that we have to define all three functions. Note that $l[2] = \lambda_1$ and $l[3] = \lambda_2$.

```
def f(x) : return 13*x[0]**2 + 10*x[0]*x[1] + 7*x[1]**2 + x[0] + x[1]
def g(x) : return 2*x[0]-5*x[1]-2
def h(x) : return x[0] + x[1] -1

minfunc = vmap(f)

def Lagrange(l): return f(l[0:2]) - l[2]*g(l[0:2]) - l[3]*h(l[0:2])

L = jacfwd(Lagrange)
gradL = jacfwd(L)

```

The loop still remains the same.

```
def solveLagrangian(l): return l - 0.1*linalg.inv(gradL(l)) @ L(l)


domain = random.uniform(key, shape=(300,4), dtype='float32', minval=-4, maxval
    =1)


vfuncsLAG = vmap(solveLagrangian)
for epoch in range(300):
domain = vfuncsLAG(domain)


maximums = minfunc(domain)

```

Notice that the only thing that needs to be printed is the number of new parameters.

```
arglist = nanargmin(maximums)
argmin = domain[arglist]
minimum = maximums[arglist]

print("The minimum is {}, the argmin is ({},{}), the lagrangians are {} and {}".
    format(
minimum,
argmin[0],
argmin[1],
argmin[2],
argmin[3]))

The minimum is 13.999992, the argmin is (0.999999701,-1.18244605e-08), the
    lagrangians are 2.28571343 and 22.42856407

```