Speeding up MLE with Jax (and JIT)

Richard Chen

1 Introduction

We would like to find the global minima of a convex function

$$\min_{w \in \chi} f(\mathbf{w}) \tag{1}$$

We assume the function $f(\mathbf{w})$ is twice differentiable and convex. Convex functions are nice because a local minimum is guaranteed to be a global minimum. A sufficient condition to prove a point is a local minimum is that the gradient must be 0, and the Hessian must be positive semi-definite in a neighborhood.

Proposition: Suppose f is twice continuously differentiable with $\nabla^2 f$ positive semi-definite in a neighborhood of w^*, and that $\nabla f(w^*) = 0$. Then w^* is a local minimum of f.

2 Optimization Methods

There are a lot of solvers for minimizing Equation 1 including (but not limited to) Gradient Descent based ones (SGD, ADAM, AdaGrad) which are especially popular in ML communities. These algorithms find the optimum by repeatedly moving in a direction opposite of the gradient.

$$w_{t+1} = w_t - \eta \nabla f(w_t) \tag{2}$$

The update rules usually look like Equation 2 where η is the step size.

Another class of solvers are Quasi-Newton methods. To determine the optimal step size, these algorithms use an approximation of the Hessian matrix. An approximation is used because

1. Calculating the Hessian is computationally expensive ($O(n^3)$)
2. If the objective function isn't perfectly convex, the Hessian won't necessarily be PSD or invertible

Popular Quasi-Newton methods include BFGS, SR1, and Powell's. These algorithms follow the same general iterative update of their estimates.

$$w_{t+1} = w_k - \gamma_k P(w_t)^{-1} \nabla_w f(w_t) \tag{3}$$

For Newton's method, the matrix $P(\theta_k)$ is just the Hessian, with a step size $\gamma_k = 1$. BFGS uses a P matrix which approximates the Hessian. Note that Equation 3 reduces to a gradient descent solver when P is replaced with the identity matrix.

We focus on quasi-newtonian methods simply because those are the default solvers in the commonly used python function

```
scipy.optimize.minimize
```

3 What is Jax?

Jax is an open source python library from Google that offers several attractive features when it comes to optimization. The 2 we'll focus on are

1. Automatic Differentiation
2. Jit

Automatic Differentiation There are many open source python libraries that offer auto-differentiation (sometimes referred to as autograd or autodiff). Depending on the complexity of the function, these can provide a significant speed up compared to finite-difference approaches [1].

Jit stands for Just-In-Time compilation. Every programming language is converted to machine code (binary) at some point. Some languages (C, Go, Fortran) are *compiled* languages. These are "lower" level languages which are first compiled to "translate" them to machine code, and are then executed. Interpreted languages (Python, R) don't perform this translation until run time. Beneath the hood, these higher level languages are implemented in a lower level language. E.g. Python is written in C, which performs the translation into machine code for Python. One big advantage of interpreted languages like Python is it's easier to interpret for humans. So when one write a line of python code and executes it, the computer must first translate it into machine code, and then execute the code.
This creates an opportunity for JIT compilation. Executing code in a JIT way creates a longer start up time, but very fast subsequent calls. This is ideal when performing optimizations for gradient descent. The code translates all of the necessary mathematical structure in machine code with arbitrary values for the variables which will be filled in at run time. Now each time the function is called, the code simply fills in the placeholders with the called function values, providing a large compuatationl speed-up.

3.1 How to use JIT

Jax's official documentation page here is very helpful. If you would like to JIT a block of code, the main things to keep in mind are

1. Array shapes and types must be static
2. Control flow is annoying (if else statements)

3.1.1 JIT Example

Suppose we have the following python code

```
def loglikelihood(theta):
    ...
init_guess = np.array([...])
opt = scipy.optimize.minimize(loglikelihood, init_guess)
```

This can easily be sped up using Jax by putting a Jit wrapper around the loglikelihood function.

```
def loglikelihood(theta):
    ...
init_guess = np.array([...])
loglikelihood_jit = jax.jit(loglikelihood)
# Initial call to translate to machine code
loglikelihood_jit(init_guess)
opt = scipy.optimize.minimize(loglikelihood_jit, init_guess)
```

[1] Check appendix section A for more details

3.2 Jax Autograd Functions

Jax has 3 different functions you can call to calculate a gradient.

1. jax.grad
2. jax.jacfwd
3. jax.jacrev

jax.jacfwd uses forward-propagation while jax.jacrev and jax.grad use backwards-propagation. jax.jacrev is more efficient for "wide" Jacobian matrices. A wide Jacobian will have a lot of inputs, which will benefit from the speed-up advantage backwards propagation gives when traversing the graph backwards. Conversely, jax.jacfwd is more efficient for "tall" Jacobian matrices. Per jax documentation, "jacfwd probably has an edge over jacrev." for near-square matrices. One can then daisy chain these in whatever order is convenient to efficiently compute the hessian.

4 Empirical Results

We solve a toy example where data is generated from the following utility function (the actual parameters are known):

$$u_{ij} = \beta_0 + \beta_1 \cdot \text{logsquarefeet}_j + \beta_2 \cdot \text{bathrooms}_j + \beta_3 \cdot \text{bathrooms}_j \cdot \text{family size}_i + \gamma_i \cdot \text{outdoor space}_j + \epsilon_{ij} \quad (4)$$

γ_i is drawn from a normal distribution. Consumer i chooses from a set of apartments j. We observe the choice each consumer makes, as well as all characteristics of each apartment. We assume the outside option is not buying an apartment.

We solve this problem using the python *scipy.optimize.minimize* library[2], passing in functions for the loglikelihood and jacobian.

Table 1 shows the results for 3 different gradient methods when varying what function we jit. The results show that convergence is fastest when using an auto differentiator, and we "jit" both the loglikelihood and gradient functions.

Tables 2 and 3 show the results of the three jacobian methods along with the *xlogit* library. We vary 3 sets of variables: # of consumers, # of products, and # of random draws. In all 3 cases, convergence was achieved most quickly and most accurately using the auto differentiator.

Table 4 fixes the data dimensions and compares results when varying the solver used in *scipy.optimize.minimize*. All solvers converged to the same value, but L-BFGS-B converged the fastest. Again, autograd provided the fastest convergence out of all three jacobian functions used.

[2]Results are from running on a 2015 iMac with a 4 GHz Quad-Core Intel Core i7 processor. Obviously these results aren't systematic, but should give a sense on what methods are generally faster or slower.

Data type	Gradient	Jit(log likelihood)	Jit(Grad)	Time (s)
Pandas	Finite Differences			Very slow
jnp	Finite Differences			10.36
jnp	Finite Differences	✓		4.45
jnp	Analytic			10.10
jnp	Analytic	✓		9.84
jnp	Analytic		✓	2.74
jnp	Analytic	✓	✓	1.73
jnp	autograd			1.83
jnp	autograd	✓		1.18
jnp	autograd		✓	1.59
jnp	autograd	✓	✓	0.96

Table 1: i=10k, j=21, random draws=12

# individuals (i)	autograd (s)	analytic (s)	finite differences (s)	xlogit (s)
10,000	0.96	1.73	10.59	2.28
50,000	10.32	21.96	94.42	13.25
100,000	16.14	23.21	81.69	27.74
j=21; random draws=12. **Winner:** Autograd				
# options (j)	autograd (s)	analytic (s)	finite differences (s)	xlogit (s)
20	0.96	1.73	10.59	2.28
100	5.37	10.54	42.68	11.64
500	30.27	OOM	186.53	no convergence
i=10,000; random draws=12. **Winner:** Autograd				
# random draws	autograd (s)	analytic (s)	finite differences (s)	xlogit (s)
12	1.02	1.67	10.22	2.36
50	4.52	8.92	28.37	5.38
100	8.95	no convergence	53.62	10.06
i=10,000; j=21. **Winner:** Autograd				

Table 2: Varying data dimensions. Convergence time in seconds

# individuals (i)	autograd (ll)	analytic (ll)	finite differences (ll)	xlogit (ll)
10,000	23812.98	23812.98	23812.98	24022.44
50,000	117379.22	117379.22	117379.22	118751.42
100,000	262236.97	262236.97	262236.97	264003.72
j=21; random draws=12. **Winner:** Autograd				
# options (j)	autograd (ll)	analytic (ll)	finite differences (ll)	xlogit (ll)
20	23812.98	23812.98	23812.98	24022.44
100	38418.69	38418.69	38418.69	38943.93
500	51974.99	OOM	51974.99	no convergence
i=10,000; random draws=12. **Winner:** Autograd				
# random draws	autograd (ll)	analytic (ll)	finite differences (ll)	xlogit (ll)
12	23812.98	23812.98	23812.98	24022.44
50	23809.98	23809.98	23809.98	23978.29
100	23809.97	no convergence	23809.97	23976.35
i=10,000; j=21. **Winner:** Autograd				

Table 3: Varying data dimensions. Loglikelihood

solver (i)	autograd (s)	analytic (s)	finite differences (s)
BFGS	0.96	1.73	10.59
CG	6.96	18.15	90.75
Newton-CG	2.69	6.75	47.84
L-BFGS-B	0.58	1.05	4.80
TNC	2.00	4.87	35.00
SLSQP	0.81	1.24	5.36
i=10,000; j=21; random draws=12. **Winner:** Autograd			

Table 4: Varying optimizer solvers. Loglikelihood

5 Takeaways

Use jax/jit operations whenever possible in optimization. Its implementation is quite easy as long as there aren't any control flow statements in loglikelihood functions. In general, it seems faster to use an autodifferentiator as opposed to an analytical derivative function. Do not ever use pandas dataframes in an optimizer. The calls make it extremely slow. Xlogit is not a reliable library to use. You cannot easily view jacobians, or set starting values.

A Calculating Gradients

There are different classes of methods in finding the gradient. We will focus on numerical and analytical methods.

A.1 Simple Review

Jacobians A jacobian is a matrix of the first-order partial derivatives for a vector function. Assume we have an n-dimensional vector, with parameters β

$$\hat{\mathbf{f}}(\beta) = \begin{bmatrix} f_1(\beta) \\ . \\ . \\ . \\ f_n(\beta) \end{bmatrix} \tag{5}$$

The jacobian matrix takes the gradient of every element in $\hat{\mathbf{f}}$. Specifically, one can define each element in the matrix as $\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j}$

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_1}{\partial x_m} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \tag{6}$$

A.2 Finite Differences

Finite differences is a numerical method in approximating the gradient. The approximation stems from the taylor expansion of a derivative. Let's say we want calculate the derivative at a point, x_0 for function $f(x)$.

$$f(x_0 + h) = f(x_0) + \frac{f'(x_0)}{1!}h + \frac{f''(x_0)}{2!}h^2 + ... + \frac{f^{(n)}(x_0)}{n!}h^n$$

We ignore $n \geq 2$ terms as residual terms.

$$f'(x_0)h = f(x_0 + h) - f(x_0)$$

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h}$$

As h approaches 0, this numerical approximation gets closer to the true derivative value, as the residual terms go to 0. This is essentially the limit definition of the derivative. It's important to note that when implemented in code, this remains an approximation and is not an exact value.

This is the default differentiation method for *scipy.optimize.minimize*

A.3 Automatic Differentiation

Automatic differentiation (also colloquially known as autograd or autodiff) takes a different approach.

Chain rule of composite functions Both forward/backward propagation are computing partial derivatives of composite functions.

$$y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$$

It is useful to define composite functions in terms of weights, denoted by w_n. This convenience will become more obvious later. Keeping with notation, the derivative of y is

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\frac{\partial w_1}{\partial x} \tag{7}$$

A.3.1 Forward Propagation

There are 2 different ways of calculating Eq 7. One could start with $\frac{\partial w_1}{\partial x}$ and work from right to left. Alternatively one could start with $\frac{\partial y}{\partial w_2}$ and work from left to right. The former describes the recursive process for forward propagation (the latter for backwards propagation), which we will now focus on[3].

Define the derivative of variable w_i as

$$\dot{w_i} = \frac{\partial w_i}{\partial x} \tag{8}$$

This is equivalent to the recursive definition below.

$$\dot{w_i} = \sum_{j \in parentofi} \frac{\partial w_i}{\partial w_j} \dot{w_j} \tag{9}$$

Consider the function

$$y = f(x_1, x_2) = x_1 x_2 + sin(x_1) = w_1 w_2 + sin(w_1) = w_3 + w_4 = w_5 \tag{10}$$

If we want to find $\frac{\partial f}{\partial x_1}$, we calculate the following.

$$\frac{\partial f}{\partial x_1} = \frac{\partial w_5}{\partial x} = \dot{w_5}$$

$$\dot{w_5} = \frac{\partial w_5}{\partial w_4} \dot{w_4} + \frac{\partial w_5}{\partial w_3} \dot{w_3}$$

$$\dot{w_5} = \frac{\partial w_5}{\partial w_4} (\frac{\partial w_4}{\partial w_1} \dot{w_1}) + \frac{\partial w_5}{\partial w_3} (\frac{\partial w_3}{\partial w_2} \dot{w_2} + \frac{\partial w_3}{\partial w_1} \dot{w_1})$$

We are finding the partial derivative wrt x_1 so trivially $\dot{w_1} = 1$, and similarly $\dot{w_2} = 0$. If we now want to find the gradient of $f(x_1, x_2)$, a total of 3 passes are necessary. The first pass populates the fields with values of x_1, x_2, and symbolically represents partial derivatives such as $\frac{\partial w_4}{\partial w_1}$. The second pass finds $\frac{\partial f}{\partial x_1}$, using seed values of $\dot{w_1} = 1$, and similarly $\dot{w_2} = 0$. The third pass finds $\frac{\partial f}{\partial x_2}$, using seed values of $\dot{w_1} = 0$, and similarly $\dot{w_2} = 1$. For each input parameter, a pass must be made through the graph to recursively calculate these derivatives.

A.3.2 Backward Propagation

The back-propagation algorithm has become increasingly popular because of it's computational advantages in training Neural Networks. Neural networks often have many input parameters compared to its output parameters. This creates an advantage to Forward Propagation, which has a time complexity that scales with input parameters. The process is similar, but slightly different which improves run time.

We take Eq 7 and now first calculate $\frac{\partial y}{\partial w_2}$ and move from left to right. We can define a term $\bar{w_i}$ as

$$\bar{w_i} = \frac{\partial y}{\partial w_i} \tag{11}$$

which is equivalent to the recursive definition below.

$$\bar{w_i} = \sum_{j \in childofi} \frac{\partial w_j}{\partial w_i} \bar{w_j} \tag{12}$$

We return to Eq 10. If we want to find the gradient of $f(x_1, x_2)$, only one sweep is necessary.

[3]Derivation heavily inspired by wikipedia

$$\frac{\partial f}{\partial x_1} = \frac{\partial w_5}{\partial x_1} = \bar{w_1}$$

$$\bar{w_1} = \frac{\partial w_3}{\partial w_1}\bar{w_3} + \frac{\partial w_4}{\partial w_1}\bar{w_4} = \frac{\partial w_3}{\partial w_1}(\frac{\partial w_5}{\partial w_3}\bar{w_5}) + \frac{\partial w_4}{\partial w_1}(\frac{\partial w_5}{\partial w_4}\bar{w_5})$$

The key distinction is these intermediate partials don't have to be recalculated.

$$\frac{\partial f}{\partial x_2} = \frac{\partial w_5}{\partial x_2} = \bar{w_2}$$

$$\bar{w_2} = \frac{\partial w_3}{\partial w_2}\bar{w_3} = \frac{\partial w_3}{\partial w_2}(\frac{\partial w_5}{\partial w_3}\bar{w_5})$$

The key idea is that term $\frac{\partial w_5}{\partial w_3}\bar{w_5}$ has already been calculated once for the first partial derivative, and now we can use that in the second partial derivative as well. This is because $\bar{w_5}$ isn't changing values, we can reuse it. This illustrates how we only have to passthrough the entire graph once to get all of the partial derivatives. In contrast, for forward propagation, those seed values change based on what you're calculating, so you have to do multiple sweeps.