Optimization

Muchang Bahng

Spring 2025

Contents

Optimization is such an important tool that it deserves a set of notes in itself. All problems in model training essentially stems from non-ideal optimization. Knowing the strengths and weaknesses of each optimizer allows you to diagnose which ones to use.

Generally, we (non-exclusively) categorize optimization algorithms as such:

1. *Convex*? Convex optimization is pretty easy to solve and has been studied extensively. For nonconvex optimization, none of the algorithms can guarantee that we will find the global minima, and this is one of the hardest problems in statistics.[1]
2. *Constrained*? Is the parameter space constrained to a certain manifold?
3. *Order*. Do we use derivatives at all? First-order derivatives (gradient)? Second-order (Hessian)?

These algorithms try to solve the following potential problems.

1. *Convergence*. Do we converge to some point?
2. *Optimality*. Is this point close to the true global minima?
3. *Efficiency*. Can we iterate efficiently?

As a benchmark test, the following function will be used a lot.

Definition 0.1 (Rosenbrock Function)

The **Rosenbrock function** is defined

$$f(x, y) = (a - x)^2 + b(y - x^2)^2 \tag{1}$$

which has a global minimum at (a, b).

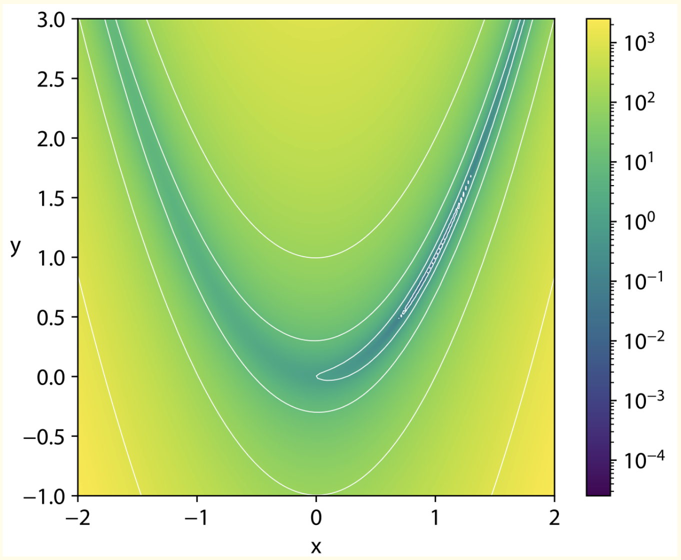


Figure 1: Typically, we set $a = 1, b = 100$.

[1] In practice, when we are doing high-dimensional nonconvex optimization, the best we can do is play around with some properties. In these cases, 0th order approximations are hopeless since the dimensions are too high, and second order approximations are hopeless either since computing the Hessian is too expensive for one run. Therefore, we must resort to some first order methods.

1 Gradient Methods

The first thing you learn about gradients in multivariate calculus is that they point in the step of steepest ascent. Generally, you can think of the function you're trying to minimize as a "landscape." This inspires a greedy approach to simply walk in this direction $\nabla f(x)$ to maximize a function (or walk in the opposite direction $-\nabla f(x)$). This gives the following.

Algorithm 1.1 (Gradient Descent)

Generally speaking, at every point you should point in the direction of steepest descent, and move in that direction. The only question that remains is: how far? This is manually adjusted by the *learning rate*.

```
Require: Function $f(x)$, initialization $x_0$, learning rate
1: procedure GradientDescent($f, x_0$)
2:     $x \leftarrow x_0$
3:     while not converged do
4:         $x \leftarrow x - \eta \nabla f(x)$
5:     end while
6:     return $x$
7: end procedure
```

1.1 Newton-Raphson Method for Root Finding

Before we talk more about gradient descent methods, let's take a look at one of the simplest numerical root-finders.[2]

Theorem 1.1 (Convergence)

Given a differentiable function $f : \mathbb{R} \to \mathbb{R}$, the sequence (x_n) defined

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} \tag{2}$$

for any $x_0 \in \mathbb{R}$, converges to a root of f.

Proof.

Need to verify this.

[2]Optimization and root finding are very similar, since to optimize f you should first find a root of f'.

Algorithm 1.2 (Newton-Raphson Method)

```
Require: Function $f(x)$, initial guess $x_0$
 1: procedure NewtonRaphson($f, f', x_0, \epsilon, N_{max}$)
 2:     $x \leftarrow x_0$
 3:     while not converged do
 4:         if $f'(x) = 0$ then
 5:             Retry with another $x_0$.
 6:         end if
 7:         $x \leftarrow \frac{f(x)}{f'(x)}$
 8:     end while
 9:     return $x$
10: end procedure
```

1.2 Stochastic Gradient Descent

Now let's go back to gradient descent. Usually, in machine learning, we are trying to optimize a parameteric model $\mathcal{P} = \{\mathbb{P}_\theta \mid \theta \in \Theta\}$ over a dataset $\mathcal{D} = \{x_i\}_{i=1}^n$., For example, our estimator in the maximum-likelihood approach will be

$$\delta(\mathcal{D}) = \underset{\theta}{\operatorname{argmax}}\, L(\theta \mid \mathcal{D}) = \log p(\mathcal{D} \mid \theta) \tag{3}$$

for some loss function L.

If we assume that the samples are iid, then we can decompose the gradient as

$$\nabla_\theta \log p(\mathcal{D} \mid \theta) = \sum_i \nabla_\theta \log p(d_i \mid \theta) \tag{4}$$

which scales linearly with the size of a dataset. This is not scalable with extremely large datasets, and so we must remove this $O(n)$ term. Intuitively, we can think of approximating this gradient by taking a minibatch b of $\mathcal{D}$ and computing the gradient only across that minibatch.

Theorem 1.2 (Minibatch Gradient is an Unbiased Estimator)

Let us take a minibatch of b samples $\mathcal{B} \subset \mathcal{D}$ without replacement, where $b << D$. Then, our approximation of the gradient of the log likelihood

$$\nabla_\theta \log p(\mathcal{B} \mid \theta) \coloneqq \frac{1}{b} \sum_{x \in \mathcal{B}} \nabla_\theta \log p(x \mid \theta) \tag{5}$$

is an unbiased estimator of the true gradient $\nabla_\theta \log p(\mathcal{D} \mid \theta)$. That is, setting $\mathcal{M}$ as a random variable of samples over $\mathcal{D}$, we have

$$\mathbb{E}_{\mathcal{M}}[\nabla \mathcal{L}_{\mathcal{M}}(\mathbf{w})] = \nabla \mathcal{L}(\mathbf{w}) \tag{6}$$

Proof.

We use linearity of expectation for all $\mathcal{M} \subset \mathcal{D}$ of size M.

This also has the additional advantage of saving memory. You don't have to load in the gradients for the whole dataset (which may be a few TB), and can allocate just enough memory for each batch (perhaps a few GB).

Algorithm 1.3 (Stochastic Gradient Descent)

```
Require: Function $L(\theta)$, initialization $\theta_0$, learning rate, batch size $b$, dataset $\mathcal{D}$
1: procedure StochasticGradientDescent($f, \theta_0, b$)
2:     $\theta \leftarrow \theta_0$
3:     while not converged do
4:         Sample minibatch $\mathcal{B} \subset \mathcal{D}$.
5:         $\theta \leftarrow \theta - \eta \nabla_\theta \log p(\mathcal{B} \mid \theta)$
6:     end while
7:     return $\theta$
8: end procedure
```

Example 1.1 (Linear Regression)

We have assumed knowledge of gradient descent in the back propagation step in the previous section, but let's revisit this by looking at linear regression. Given our dataset $\mathcal{D} = \{\mathbf{x}^(n), y^{(n)}\}$, we are fitting a linear model of the form

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{w}^T \mathbf{x} + b \tag{7}$$

The squared loss function is

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2} \sum_{n=1}^N \big(y - f(\mathbf{x}; \mathbf{w}, b)\big)^2 = \frac{1}{2} \sum_{n=1}^N \big(y - (\mathbf{w}^T \mathbf{x} + b)\big)^2 \tag{8}$$

If we want to minimize this function, we can visualize it as a d-dimensional surface that we have to traverse. Recall from multivariate calculus that the gradient of an arbitrary function $\mathcal{L}$ points in the steepest direction in which $\mathcal{L}$ increases. Therefore, if we can compute the gradient of $\mathcal{L}$ and step in the *opposite direction*, then we would make the more efficient progress towards minimizing this function (at least locally). The gradient can be solved using chain rule. Let us solve it with respect to $\mathbf{w}$ and b separately first. Beginners might find it simpler to compute the gradient element-wise.

$$\begin{aligned}
\frac{\partial}{\partial w_j} \mathcal{L}(\mathbf{w}, b) &= \frac{\partial}{\partial w_j} \left(\frac{1}{2} \sum_{n=1}^N \Big(f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)} \Big)^2 \right) && (9) \\
&= \frac{1}{2} \sum_{n=1}^N \frac{\partial}{\partial w_j} \Big(f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)} \Big)^2 && (10) \\
&= \frac{1}{2} \sum_{n=1}^N 2 \Big(f(\mathbf{x}^{(n)}) - y^{(n)} \Big) \cdot \frac{\partial}{\partial w_j} \big(f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)} \big) && (11) \\
&= \frac{1}{2} \sum_{n=1}^N 2 \Big(f(\mathbf{x}^{(n)}) - y^{(n)} \Big) \cdot \frac{\partial}{\partial w_j} \big(\mathbf{w}^T \mathbf{x}^{(n)} + b - y^{(n)} \big) && (12) \\
&= \sum_{n=1}^N \big(f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)} \big) \cdot x_j^{(n)} \qquad (\text{for } j = 0, 1, \ldots, d) && (13)
\end{aligned}$$

As for getting the derivative w.r.t. b, we can redo the computation and get

$$\frac{\partial}{\partial w_j} \mathcal{L}(\mathbf{w}, b) = \sum_{n=1}^N \big(f(\mathbf{x}^{(n)}; \mathbf{w}, b) - y^{(n)} \big) \tag{14}$$

and in the vector form, setting $\boldsymbol{\theta} = (\mathbf{w}, b)$, we can set

$$\nabla \mathcal{L}(\mathbf{w}) = \mathbf{X}^T (\hat{\mathbf{y}} - \mathbf{y}) \tag{15}$$

$$\nabla \mathcal{L}(b) = (\hat{\mathbf{y}} - \mathbf{y}) \cdot \mathbf{1} \tag{16}$$

where $\hat{\mathbf{y}}_n = f(\mathbf{x}^{(n)}; \mathbf{w}, b)$ are the predictions under our current linear model and $\mathbf{X} \in \mathbb{R}^{n \times d}$ is our design matrix. This can easily be done on a computer using a package like `numpy`.
Rather than updating the vector $\boldsymbol{\theta}$ in batches, we can apply **stochastic gradient descent** that works incrementally by updating $\boldsymbol{\theta}$ with each term in the summation. That is, rather than updating as a batch by performing the entire matrix computation by multiplying over N dimensions,

$$\nabla \mathcal{L}(\mathbf{w}) = \underbrace{\mathbf{X}^T}_{D \times N} \underbrace{(\hat{\mathbf{y}} - \mathbf{y})}_{N \times 1} \tag{17}$$

we can reduce this load by choosing a smaller subset $\mathcal{M} \subset \mathcal{D}$ of $M < N$ elements, which gives

$$\nabla \mathcal{L}_{\mathcal{M}}(\mathbf{w}) = \underbrace{\mathbf{X}_{\mathcal{M}}^T}_{D \times M} \underbrace{(\hat{\mathbf{y}_{\mathcal{M}}} - \mathbf{y})}_{\mathcal{M}}{}_{M \times 1} \tag{18}$$

Even though these estimators are noisy, we get to do much more iterations and therefore have a faster net rate of convergence. But now we have an additional choice to make. What should our minibatch size be?

Heuristic 1.1 (Choosing Batch Size)

It really depends on your hardware, but generally,

1. A smaller batch size might mean more noisy estimates, and therefore may not converge. However, it tends to escape local minima better.
2. A high batch size means more exact estimates, but it tends to get stuck in local minima.

You want to make sure that the batches fit into memory.

1.3 Learning Rates and Schedulers

The algorithm may not converge if α (the step size) is too high, since it may overshoot. This can be solved by reducing the α with each step, using *schedulers*.

Ideally, we would want to have a variable step size $h(t)$ so that $h \to 0$ as $t \to +\infty$.

Algorithm 1.4 (Decay on Plateau Learning Rate)

Basically, this says that if the loss doesn't decrease for the past p epochs, then decrease the learning rate $\eta \leftarrow \gamma \cdot \eta$.

```
Require: Patience $p$, decay rate $0 < \gamma < 1$, initial $\theta_0$, loss $L$
procedure LRDecayOnPlateau($p, \gamma, \theta, \eta$)
    best_loss $\leftarrow L(\theta)$
    bad_epochs $\leftarrow 0$
    while not converged do
        $\theta \leftarrow \theta - \eta \nabla L(\theta)$
        if $L(\theta) <$ best_loss then
            best_loss $\leftarrow L(\theta)$
            bad_epochs $\leftarrow 0$
        else
            bad_epochs $\leftarrow$ bad_epochs $+ 1$
        end if
        if bad_epochs $\geq p$ then
            $\eta \leftarrow \eta \cdot \gamma$
            bad_epochs $\leftarrow 0$
        end if
    end while
    return $\theta, \eta$
end procedure
```

1.4 Momentum and Nesterov

Algorithm 1.5 (Stochastic Gradient Descent with Momentum)

This modifies vanilla SGD by keeping a running velocity term that accumulates past gradients, which smooths updates and helps escape sharp local minima.

```
Require: Learning rate $\eta$, momentum $0 \leq \mu < 1$, initial parameters $\theta_0$, loss $L$
procedure SGDMomentum($\eta, \mu, \theta$)
    $v \leftarrow 0$                                   ▷Initialize velocity
    while not converged do
        $v \leftarrow \mu v + \nabla L(\theta)$        ▷Decay prev. velocity and add in gradient (acceleration)
        $\theta \leftarrow \theta - \eta v$
    end while
    return $\theta$
end procedure
```

Algorithm 1.6 (Stochastic Gradient Descent with Nesterov Momentum)

Nesterov momentum modifies standard momentum by computing the gradient at the *lookahead* position $\theta - \eta\mu v$, leading to faster convergence in practice.

```
Require: Learning rate $\eta$, momentum $0 \leq \mu < 1$, initial parameters $\theta_0$, loss $L$
1: procedure SGDNesterov($\eta, \mu, \theta$)
2:     $v \leftarrow 0$                                  ▷Initialize velocity
3:     while not converged do
4:         $g \leftarrow \nabla L(\theta - \eta\mu v)$    ▷Lookahead gradient
5:         $v \leftarrow \mu v + g$
6:         $\theta \leftarrow \theta - \eta v$
7:     end while
8:     return $\theta$
9: end procedure
```

1.5 Block Coordinate Descent

Algorithm 1.7 (Block Gradient Descent)

Block Gradient Descent partitions the parameter vector θ into m disjoint blocks. At each iteration, it selects one block (cyclically or randomly) and updates only that block's parameters using the gradient, while keeping the other blocks fixed.

```
Require: Learning rate $\eta$, number of blocks $m$, partition $\theta = (\theta^{(1)}, \ldots, \theta^{(m)})$, loss $L$
1: procedure BlockGradientDescent($\eta, \{\theta^{(j)}\}_{j=1}^{m}$)
2:     $t \leftarrow 0$
3:     while not converged do
4:         $j \leftarrow \text{SelectBlock}(t, m)$    ▷e.g., cyclic: $j = (t \bmod m) + 1$
5:         $\theta^{(j)} \leftarrow \theta^{(j)} - \eta \nabla_{\theta^{(j)}} L(\theta^{(1)}, \ldots, \theta^{(m)})$
6:         $t \leftarrow t + 1$
7:     end while
8:     return $\theta$
9: end procedure
```

2 Subgradient Methods

Definition 2.1 (Convex Function)

A function $f : U \subset \mathbb{R}^n \to \mathbb{R}$ defined on a convex set U is convex if and only if for any $\mathbf{x}, \mathbf{y} \in U$

$$f\big(\lambda \mathbf{x} + (1-\lambda)\mathbf{y}\big) \leq \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y}) \tag{19}$$

Now if f is differentiable, then convexity is equivalent to

$$f(x) \geq f(y) + \nabla f(y)^T \cdot (x-y) \tag{20}$$

for all $x, y \in U$. That is, its local linear approximation always underestimates f.

It is well known that the mean square error of a linear map is convex. However, when we impose the L1 penalty, the loss function is now not differentiable at $\mathbf{0}$. Therefore, we must introduce the notion of a subgradient.

Definition 2.2 (Subgradient)

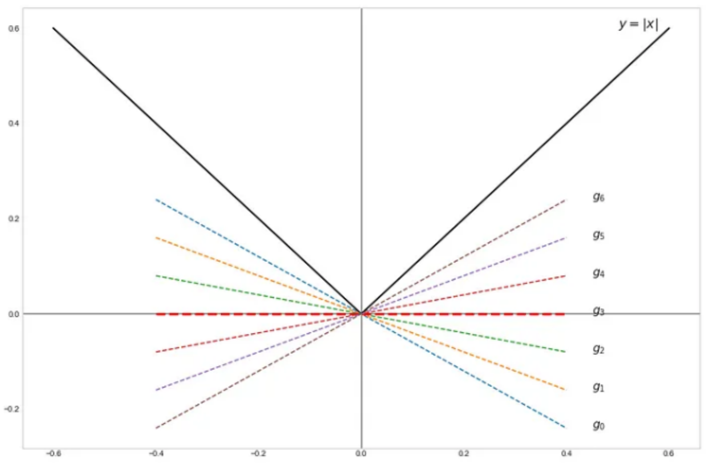
The subgradient of a convex function $f : U \subset \mathbb{R}^n \to \mathbb{R}$ is any linear map $\mathbf{A}(x) : \mathbb{R}^n \to \mathbb{R}$ such that

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \mathbf{A}(\mathbf{x})(\mathbf{y} - \mathbf{x}) \tag{21}$$

for any $\mathbf{y} \in U$. The set of all subgradients at $\mathbf{x}$ is called the **subdifferential** defined

$$\partial f(\mathbf{x}) = \{\mathbf{A} \in \mathbb{R}^n \mid \mathbf{A} \text{ is a subgradient of } f \text{ at } \mathbf{x}\} \tag{22}$$

The subgradient also acts as a linear approximation of f, but now at nondifferentiable points of convex functions, we have a set of linear approximations. It is clear that the subgradient at a differentiable point is uniquely the gradient ($\partial f(\mathbf{x}) = \{\nabla f(\mathbf{x})$), but for places like the absolute value, we can have infinite linear approximations.



Given the subdifferential, thus the optimality condition for any convex $\mathbf{f}$ (differentiable or not) is

$$f(\mathbf{x}^*) = \min_{\mathbf{x}} f(\mathbf{x}) \iff \mathbf{0} \in \partial f(\mathbf{x}^*) \tag{23}$$

known as the subgradient optimality condition, which clearly implies

$$f(\mathbf{y}) \geq f(\mathbf{x}^*) + \mathbf{0}^T(\mathbf{y} - \mathbf{x}^*) = f(\mathbf{x}^*) \tag{24}$$

Example 2.1 ()

The subdifferential of the absolute value function $f(x) = |x|$ at any given x is

$$\partial f(x) = \begin{cases} 1 & \text{if } x > 0 \\ [-1, 1] & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases} \tag{25}$$

2.1 Proximal Gradient Descent

Definition 2.3 (Proximal Operator)

Given a lower semicontinuous convex function f mapping from Hilbert space X to $[-\infty, +\infty]$, its **proximal operator** associated with a point u is defined

$$\text{prox}_{f,\tau}(u) = \underset{x}{\text{argmin}} \left(f(x) + \frac{1}{2\tau} ||x - u||^2 \right) \tag{26}$$

where $\tau > 0$ is a parameter that scales the quadratic term. This is basically the point that minimizes the sum of $f(x)$ and the square of the Euclidean distance between x and u, scaled by $1/2\tau$.

Now given the loss function $L(\boldsymbol{\theta}) = L_{\text{obj}}(\boldsymbol{\theta}) + L_{\text{reg}}(\boldsymbol{\theta})$, we want to compute the proximal operator on the regularization loss and update that with the gradient of the smooth objective loss.

$$\boldsymbol{\theta}^{(k+1)} = \text{prox}_{L_{\text{reg}},\tau} \left[\boldsymbol{\theta}^{(k)} - \tau \nabla L_{\text{obj}}(\boldsymbol{\theta}^{(k)}) \right] \tag{27}$$

Let's compute the proximal operator of the L1 loss $h(\boldsymbol{\theta}) = \lambda ||\boldsymbol{\theta}||_1$. We can parameterize this loss by the λ, so we will use the notation $\text{prox}_{\lambda,\tau}$ rather than $\text{prox}_{h,\tau}$.

$$\begin{aligned} \text{prox}_{\lambda,\tau}(\mathbf{u}) &= \underset{\boldsymbol{\theta}}{\text{argmin}} \left(\lambda ||\boldsymbol{\theta}||_1 + \frac{1}{2\tau} ||\boldsymbol{\theta} - \mathbf{u}||_2^2 \right) \\ &= \underset{\boldsymbol{\theta}}{\text{argmin}} \left(\sum_{i=1}^n \lambda |\theta_i| + \frac{1}{2\tau} (\theta_i - u_i)^2 \right) \end{aligned}$$

These are separable functions that can be decoupled and optimized component-wise. So, we really just want to find

$$\theta_i^* = \underset{\theta_i}{\text{argmin}} \left(\lambda |\theta_i| + \frac{1}{2\tau} (\theta_i - u_i)^2 \right) \tag{28}$$

The sum of convex functions is convex, and so we should differentiate it and find where the gradient is 0 to optimize it.

1. When $\theta_i > 0$, then we minimize $\lambda \theta_i + \frac{1}{2\tau}(\theta_i - u_i)^2$, so taking the gradient and setting to 0 gives

$$\theta_i = u_i - \lambda \tau \tag{29}$$

   subject to the constraint that $\theta_i > 0$, or equivalently, that $u_i > \lambda \tau$.

2. When $\theta_i < 0$, then we minimize $-\lambda \theta_i + \frac{1}{2\tau}(\theta_i - u_i)^2$, so taking the gradient and setting to 0 gives

$$\theta_i = u_i + \lambda \tau \tag{30}$$

   subject to the constraint that $\theta_i < 0$, or equivalently, that $u_i < -\lambda \tau$.

3. When $\theta_i = 0$, then we minimize $\lambda|\theta_i| + \frac{1}{2\tau}(\theta_i - u_i)^2$, which doesn't have derivative at $\theta_i = 0$. So, we can compute the subdifferential of it to get

$$0 \in \partial\bigg(\lambda|\theta_i| + \frac{1}{2\tau}(\theta_i - u_i)^2\bigg) = \lambda\partial(|\theta_i|) + \frac{1}{\tau}(\theta_i - u_i)$$

Now at $\theta_i = 0$, the subdifferential can be any value in $[-1, 1]$, and the above reduces to

$$0 \in \lambda[-1, 1] - \frac{1}{\tau}u_i \tag{31}$$

this is equivalent to saying that u_i/τ is contained in the interval $[-\lambda, \lambda]$, meaning that $u_i \in [-\lambda\tau, \lambda\tau]$.

Ultimately we get that

$$\text{prox}_{\lambda,\tau}(u) = \begin{cases} u - \lambda\tau & \text{if } u > \lambda\tau \\ 0 & \text{if } |u| \leq \lambda\tau \\ u + \lambda\tau & \text{if } u < -\lambda\tau \end{cases} \tag{32}$$

which can be simplified to

$$\text{prox}_{\lambda,\tau}(u) = \text{sign}(u)\max\{|u| - \lambda\tau, 0) \tag{33}$$

3 Adaptive Gradient Methods

3.1 Adagrad

3.2 RMSProp and Adadelta

3.3 Adam

Adam and AdamW

4 Second-Order Optimizers

4.1 Newton's Method

Newton's method is an iterative algorithm for finding the roots of a differentiable function F. An immediate consequence is that given a convex C^2 function f, we can apply Newton's method to its derivative f' to get the critical points of f (minima, maxima, or saddle points), which is relevant in optimizing f. Given a C^1 function $f : D \subset \mathbb{R}^n \longrightarrow \mathbb{R}$ and a point $\mathbf{x}_k \in D$, we can compute its linear approximation as

$$f(\mathbf{x}_k + \mathbf{h}) \approx f(\mathbf{x}_k) + Df_{\mathbf{x}_k}\,\mathbf{h} = f(\mathbf{x}_k) + \nabla f(\mathbf{x}_k) \cdot \mathbf{h} \tag{34}$$

where $Df_{\mathbf{x}_k}$ is the total derivative of f at $\mathbf{x}_k$ and $\mathbf{h}$ is a small n-vector. Discretizing this gives us our gradient descent algorithm as

$$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \alpha\, f'(\mathbf{x}_k) \tag{35}$$

This linear function is unbounded, so we must tune the step size α accordingly. If α is too small, then convergence is slow, and if α is too big, we may overshoot the minimum. Netwon's method automatically tunes this α using the curvature information, i.e. the second derivative. If we take a second degree Taylor approximation

$$f(\mathbf{x}_k + \mathbf{h}) \approx f(\mathbf{x}_k) + Df_{\mathbf{x}_k}\,\mathbf{h} + \mathbf{h}^T\, Hf_{\mathbf{x}_k}\,\mathbf{h} \tag{36}$$

then we are guaranteed that this quadratic approximation of f has a minimum (existence and uniqueness can be proved), and we can calculate it to find our "approximate" minimum of f. We simply take the total derivative of this polynomial w.r.t. $\mathbf{h}$ and set it equal to the n-dimensional covector $\mathbf{0}$. This is equivalent to setting the gradient as $\mathbf{0}$, so

$$\begin{aligned}
\mathbf{0} &= \nabla_{\mathbf{h}}\big[f(\mathbf{x}_k) + Df_{\mathbf{x}_k}\,\mathbf{h} + \mathbf{h}^T\, Hf_{\mathbf{x}_k}\,\mathbf{h}\big](\mathbf{h}) \\
&= \nabla_{\mathbf{h}}[Df_{x_k}\mathbf{h}](\mathbf{h}) + \nabla_{\mathbf{h}}[\mathbf{h}^T\, Hf_{\mathbf{x}_k}\,\mathbf{h}](\mathbf{h}) \\
&= \nabla_{\mathbf{x}} f(\mathbf{x}_k) + Hf_{\mathbf{x}_k}\,\mathbf{h} \\
&\implies \mathbf{h} = -[Hf_{\mathbf{x}_k}]^{-1}\nabla_{\mathbf{x}} f(\mathbf{x}_k)
\end{aligned}$$

which results in the iterative update

$$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - [Hf_{\mathbf{x}_k}]^{-1}\nabla_{\mathbf{x}} f(\mathbf{x}_k) \tag{37}$$

Note that we require $\mathbf{f}$ to be convex, so that Hf is positive definite. Since f is C^2, this implies Hf is also symmetric, implying invertibility by the spectral theorem. Note that Newton's method is very expensive, since we require the computation of the gradient, the Hessian, *and* the inverse of the Hessian, making the computational complexity of this algorithm to be $O(n^3)$. We can also add a smaller stepsize h to control stability.

Algorithm 1 Newton's Method

```
Require: Initial $\mathbf{x}_0$, Stepsize $h$ (optional)
  for $t = 0$ to $T$ until convergence do
    $g(\mathbf{x}_t) \leftarrow \nabla f(\mathbf{x}_t)$
    $H(\mathbf{x}_t) \leftarrow Hf_{\mathbf{x}_t}$
    $H^{-1}(\mathbf{x}_t) \leftarrow [H(\mathbf{x}_t)]^{-1}$
    $\mathbf{x}_{t+1} \leftarrow \mathbf{x}_t - h\, H^{-1}(\mathbf{x}_t)\, g(\mathbf{x}_t)$
  end for
```

4.2 Gauss Newton Method

5 Quasi-Newton Methods

5.1 Secant Method

Algorithm 5.1 (Secant Method)

The Secant Method approximates Newton's method by estimating the derivative using two most recent iterates instead of requiring $f'(x)$.

```
Require: Function $f(x)$, initial guesses $x_0, x_1$, tolerance $\epsilon$, maximum iterations $N_{max}$
1: procedure SECANT($f, x_0, x_1, \epsilon, N_{max}$)
2:     $k \leftarrow 0$
3:     while $k < N_{max}$ do
4:         if $|f(x_1)| < \epsilon$ then
5:             return $x_1$
6:         end if
7:         $x_{new} \leftarrow x_1 - f(x_1) \dfrac{x_1 - x_0}{f(x_1) - f(x_0)}$
8:         $x_0 \leftarrow x_1$
9:         $x_1 \leftarrow x_{new}$
10:        $k \leftarrow k + 1$
11:    end while
12:    return Failure
13: end procedure
```

5.2 Davidon-Fletcher-Powell (DFP)

Algorithm 5.2 (DFP Quasi-Newton Method)

The DFP method is a quasi-Newton optimization algorithm that maintains an approximation H_k of the inverse Hessian to update parameters efficiently.

```
Require: Objective $f(\theta)$, gradient $\nabla f$, initial guess $\theta_0$, tolerance $\epsilon$
1: procedure DFP($f, \nabla f, \theta_0$)
2:     $H \leftarrow I$                                  ▷Initialize inverse Hessian approximation
3:     while not converged do
4:         $g \leftarrow \nabla f(\theta)$
5:         $d \leftarrow -Hg$
6:         $\alpha \leftarrow$ LineSearch($f, \theta, d$)
7:         $\theta_{new} \leftarrow \theta + \alpha d$
8:         $s \leftarrow \theta_{new} - \theta$
9:         $y \leftarrow \nabla f(\theta_{new}) - g$
10:        $H \leftarrow H + \frac{ss^T}{s^T y} - \frac{Hyy^T H}{y^T Hy}$
11:        $\theta \leftarrow \theta_{new}$
12:    end while
13:    return $\theta$
14: end procedure
```

5.3 Broyden's Method

Algorithm 5.3 (Broyden's Method)

Broyden's method is a quasi-Newton method that updates an approximation B_k to the Jacobian matrix without computing derivatives directly.

```
Require: Function $F(\theta)$, initial guess $\theta_0$, tolerance $\epsilon$
 1: procedure Broyden($F, \theta_0$)
 2:     $B \leftarrow I$                                  ▷Initial Jacobian approximation
 3:     while not converged do
 4:         $\Delta\theta \leftarrow -B^{-1}F(\theta)$
 5:         $\theta_{new} \leftarrow \theta + \Delta\theta$
 6:         $\Delta F \leftarrow F(\theta_{new}) - F(\theta)$
 7:         $B \leftarrow B + \frac{(\Delta F - B\Delta\theta)\Delta\theta^T}{\Delta\theta^T\Delta\theta}$
 8:         $\theta \leftarrow \theta_{new}$
 9:     end while
10:     return $\theta$
11: end procedure
```

5.4 Symmetric Rank-1 Update (SR1)

Algorithm 5.4 (Symmetric Rank-1 Update)

The SR1 update is another quasi-Newton method that maintains an approximation H_k of the inverse Hessian, using a symmetric rank-1 correction.

```
Require: Objective $f(\theta)$, gradient $\nabla f$, initial guess $\theta_0$, tolerance $\epsilon$
 1: procedure SR1($f, \nabla f, \theta_0$)
 2:     $H \leftarrow I$
 3:     while not converged do
 4:         $g \leftarrow \nabla f(\theta)$
 5:         $d \leftarrow -Hg$
 6:         $\alpha \leftarrow \text{LineSearch}(f, \theta, d)$
 7:         $\theta_{new} \leftarrow \theta + \alpha d$
 8:         $s \leftarrow \theta_{new} - \theta$
 9:         $y \leftarrow \nabla f(\theta_{new}) - g$
10:         if $(s - Hy)^T y \neq 0$ then
11:             $H \leftarrow H + \dfrac{(s - Hy)(s - Hy)^T}{(s - Hy)^T y}$
12:         end if
13:         $\theta \leftarrow \theta_{new}$
14:     end while
15:     return $\theta$
16: end procedure
```

5.5 BFGS

Netwon's method is extremely effective for finding the minimum of a convex function, but there are two disadvantages. First, it is sensitive to initial conditions, and second, it is extremely expensive, with a computational complexity of $O(n^3)$ from having to invert the Hessian. An alternative family of optimizers, called *quasi-Newton* methods, try to *approximate* the Hessian (or Jacobian) with $\hat{H}f$, reducing the computational cost without too much loss in convergence rates, and simply use this approximation in the Newton's update:

$$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - [\hat{H}f_{\mathbf{x}_k}]^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}_k)$$

The method of the Hessian approximation varies by algorithm, but the most popular is BFGS.

So how do we approximate the Hessian with only the gradient information? With secants. Starting off with $f : \mathbb{R} \longrightarrow \mathbb{R}$, let us assume that we have two points $(x_k, f(x_k))$ and $(x_{k+1}, f(x_{k+1}))$. We can approximate our derivative (gradient in dimension 1) at x_{k+1} using finite differences:

$$f'(x_{k+1})(x_{k+1} - x_k) \approx f(x_{k+1}) - f(x_k)$$

and doing the same for f' gives us the second derivative approximation:

$$f''(x_{k+1})(x_{k+1} - x_k) \approx f'(x_{k+1}) - f'(x_k)$$

which gives us the update:

$$x_{k+1} \leftarrow x_k - \frac{x_k - x_{k-1}}{f'(x_k) - f'(x_{k-1})} f'(x_k)$$

This method of approximating Netwon's method in one dimension by replacing the second derivative with its finite difference approximation is called the *secant method.* In multiple dimensions, given two points $\mathbf{x}_k, \mathbf{x}_{k+1}$ with their respective gradients $\nabla f(\mathbf{x}_k), \nabla f(\mathbf{x}_{k+1})$, we can approximate the Hessian $\hat{H}f_{\mathbf{x}_{k+1}} \approx D(\nabla f)_{\mathbf{x}_{k+1}}$ (which is the total derivative of the gradient) at $\mathbf{x}_{k+1}$ with the equation

$$\hat{H}f_{\mathbf{x}_{k+1}}(\mathbf{x}_{k+1} - \mathbf{x}_k) = \nabla_{\mathbf{x}} f(\mathbf{x}_{k+1}) - \nabla_{\mathbf{x}} f(\mathbf{x}_k)$$

This is solving the equation of form $A\mathbf{x} = \mathbf{y}$ for some linear map A. Since $\hat{H}f_{\mathbf{x}_{k+1}}$ is a symmetric $n \times n$ matrix with $n(n+1)/2$ components, there are $n(n+1)/2$ unknowns with only n equations, making this an underdetermined system. Quasi-Newton methods have to impose additional constraints, with the BFGS choosing the one where we want $\hat{H}f_{\mathbf{x}_{k+1}}$ to be as close as to $\hat{H}f_{\mathbf{x}_k}$ at each update $k+1$. Luckily, we can formalize this notion as minimizing the distance between $f_{\mathbf{x}_{k+1}}$ and $\hat{H}f_{\mathbf{x}_k}$. Therefore, we wish to find

$$\arg \min_{\hat{H}f_{\mathbf{x}_{k+1}}} ||\hat{H}f_{\mathbf{x}_{k+1}} - \hat{H}f_{\mathbf{x}_k}||_F,$$

where $|| \cdot ||_F$ is the Frobenius matrix norm, subject to the restrictions that $\hat{H}f_{\mathbf{x}_{k+1}}$ be positive definite and symmetric and that $\hat{H}f_{\mathbf{x}_{k+1}}(\mathbf{x}_{k+1} - \mathbf{x}_k) = \nabla_{\mathbf{x}} f(\mathbf{x}_{k+1}) - \nabla_{\mathbf{x}} f(\mathbf{x}_k)$ is satisfied. Since we have to invert it eventually, we can actually just create an optimization problem that directly computes the inverse. So, we wish to find

$$\arg \min_{(\hat{H}f_{\mathbf{x}_{k+1}})^{-1}} ||(\hat{H}f_{\mathbf{x}_{k+1}})^{-1} - (\hat{H}f_{\mathbf{x}_k})^{-1}||_F$$

subject to the restrictions that

1. $(\hat{H}f_{\mathbf{x}_{k+1}})^{-1}$ be positive definite and symmetric. It turns out that the positive definiteness restriction also restricts it to be symmetric.
2. $\mathbf{x}_{k+1} - \mathbf{x}_k = (\hat{H}f_{\mathbf{x}_{k+1}})^{-1}[\nabla_{\mathbf{x}} f(\mathbf{x}_{k+1}) - \nabla_{\mathbf{x}} f(\mathbf{x}_k)]$

After some complicated mathematical derivation, which we will not go over here, the problem ends up being equivalent to updating our approximate Hessian at each iteration by adding two symmetric, rank-one matrices U and V scaled by some constant, which can each be computed as an outer product of vectors with itself.

$$\hat{H}f_{\mathbf{x}_{k+1}} = \hat{H}f_{\mathbf{x}_k} + aU + bV = \hat{H}f_{\mathbf{x}_k} + a\mathbf{u}\mathbf{u}^T + b\mathbf{v}\mathbf{v}^T$$

where $\mathbf{u}$ and $\mathbf{v}$ are linearly independent. This addition of a rank-2 sum of matrices $aU + bV$, known as a rank-2 update, guarantees the "closeness" of $\hat{H}f_{\mathbf{x}_{k+1}}$ to $\hat{H}f_{\mathbf{x}_k}$ at each iteration. With this form, we now impose the quasi-Newton condition. Substituting $\Delta\mathbf{x}_k = \mathbf{x}_{k+1} - \mathbf{x}_k$ and $\mathbf{y}_k = \nabla_{\mathbf{x}} f(\mathbf{x}_{k+1}) - \nabla_{\mathbf{x}} f(\mathbf{x}_k)$, we have

$$\hat{H}f_{\mathbf{x}_{k+1}} \Delta\mathbf{x}_k = \hat{H}f_{\mathbf{x}_{k+1}} \Delta\mathbf{x}_k + a\mathbf{u}\mathbf{u}^T \Delta\mathbf{x}_k + b\mathbf{v}\mathbf{v}^T \Delta\mathbf{x}_k = \mathbf{y}_k$$

A natural choice of vectors turn out to be $\mathbf{u} = \mathbf{y}_k$ and $\mathbf{v} = \hat{H}f_{\mathbf{x}_k} \Delta\mathbf{x}_k$, and substituting this and solving gives us the optimal values

$$a = \frac{1}{\mathbf{y}_k^T \Delta\mathbf{x}_k}, \qquad b = -\frac{1}{\Delta\mathbf{x}_k^T \hat{H}f_{\mathbf{x}_k} \Delta\mathbf{x}_k}$$

and substituting these values back to the Hessian approximation update gives us the BFGS update:

$$\hat{H}f_{\mathbf{x}_{k+1}} = \hat{H}f_{\mathbf{x}_k} + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k^T \Delta\mathbf{x}_k} - \frac{\hat{H}f_{\mathbf{x}_k} \Delta\mathbf{x}_k \Delta\mathbf{x}_k^T \hat{H}f_{\mathbf{x}_k}}{\Delta\mathbf{x}_k^T \hat{H}f_{\mathbf{x}_k} \Delta\mathbf{x}_k}$$

We still need to invert this, and using the *Woodbury formula*

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}$$

which tells us how to invert the sum of an intertible matrix A and a rank-k correction, we can derive the iterative update of the inverse Hessian as

$$(\hat{H}f_{\mathbf{x}_{k+1}})^{-1} = \bigg(I - \frac{\Delta\mathbf{x}_k \mathbf{y}^T}{\mathbf{y}_k^T \Delta\mathbf{x}_k}\bigg)(\hat{H}f_{\mathbf{x}_k})^{-1}\bigg(I - \frac{\mathbf{y}_k \Delta\mathbf{x}_k^T}{\mathbf{y}_k^T \Delta\mathbf{x}_k}\bigg) + \frac{\Delta\mathbf{x}_k \Delta\mathbf{x}_k^T}{\mathbf{y}_k^T \Delta\mathbf{x}_k}$$

Remember that this is the iterative step that we want to actually compute, rather than the ones computing the regular Hessian. The whole point of using the Woodbury formula to derive an inverse update step was to do away with the tedious $O(n^3)$ computations of inverting an $n \times n$ matrix. This rank-2 update also preserves positive-definiteness.

Finally, we can choose the initial inverse Hessian approximation $(\hat{H}f_{\mathbf{x}_{k+1}})^{-1}$ to be the identity I or the true inverse Hessian $(Hf_{\mathbf{x}_{k+1}})^{-1}$ (computed just once), which would lead to more efficient convergence. The pseudocode for BFGS is a bit too long and confusing to include here, but most of the time, we won't be implementing BFGS by hand; efficient and established BFGS optimizers are already in numerous packages. Like most optimizers, BFGS is not guaranteed to converge to the true global minimum.

Algorithm 5.5 (BFGS)

The BFGS algorithm maintains an approximation B_k to the inverse Hessian, updating it using gradient differences and parameter steps. This avoids explicitly computing or inverting the Hessian at each iteration.

```
Require: Objective $f(\theta)$, gradient $\nabla f$, initial guess $\theta_0$, tolerance $\epsilon$, maximum iterations $N_{max}$
 1: procedure BFGS($f, \nabla f, \theta_0$)
 2:     $B \leftarrow I$                                  ▷Initialize inverse Hessian approximation
 3:     $\theta \leftarrow \theta_0$
 4:     $k \leftarrow 0$
 5:     while $k < N_{max}$ and $||\nabla f(\theta)|| > \epsilon$ do
 6:         $g \leftarrow \nabla f(\theta)$
 7:         $d \leftarrow -Bg$
 8:         $\alpha \leftarrow \text{LineSearch}(f, \theta, d)$          ▷Enforce Wolfe conditions
 9:         $\theta_{new} \leftarrow \theta + \alpha d$
10:         $s \leftarrow \theta_{new} - \theta$
11:         $y \leftarrow \nabla f(\theta_{new}) - g$
12:         if $y^T s \leq 0$ then
13:             break                                  ▷Curvature condition violated
14:         end if
15:         $B \leftarrow \left(I - \frac{sy^T}{y^T s}\right) B \left(I - \frac{ys^T}{y^T s}\right) + \frac{ss^T}{y^T s}$
16:         $\theta \leftarrow \theta_{new}$
17:         $k \leftarrow k + 1$
18:     end while
19:     return $\theta$
20: end procedure
```

Algorithm 5.6 (Limited-memory BFGS)

L-BFGS avoids storing the full inverse Hessian by keeping only the last m update pairs (s_i, y_i). At each step, the search direction is computed using a two-loop recursion. This reduces storage from $O(n^2)$ to $O(mn)$ and is widely used in large-scale optimization.

```
Require: Objective $f(\theta)$, gradient $\nabla f$, initial $\theta_0$, history size $m$, tolerance $\epsilon$, maximum iterations $N_{max}$
procedure L-BFGS($f, \nabla f, \theta_0, m$)
    Initialize $\theta \leftarrow \theta_0$
    Initialize empty history lists $S, Y$
    $k \leftarrow 0$
    while $k < N_{max}$ and $||\nabla f(\theta)|| > \epsilon$ do
        $g \leftarrow \nabla f(\theta)$
        $q \leftarrow g$
        Initialize empty list $\alpha$                                  ▷First loop: backward pass
        for $i = |S|$ down to 1 do
            $\rho_i \leftarrow 1/(y_i^T s_i)$
            $\alpha_i \leftarrow \rho_i s_i^T q$
            $q \leftarrow q - \alpha_i y_i$
        end for
        Choose scalar $H_0$ (e.g., $H_0 = \frac{s_{k-1}^T y_{k-1}}{y_{k-1}^T y_{k-1}} I$ if available, else $I$)
        $r \leftarrow H_0 q$                                            ▷Second loop: forward pass
        for $i = 1$ to $|S|$ do
            $\beta \leftarrow \rho_i y_i^T r$
            $r \leftarrow r + s_i(\alpha_i - \beta)$
        end for
        $d \leftarrow -r$                                               ▷Search direction
        $\alpha \leftarrow$ LineSearch($f, \theta, d$)
        $\theta_{new} \leftarrow \theta + \alpha d$
        $s \leftarrow \theta_{new} - \theta$
        $y \leftarrow \nabla f(\theta_{new}) - g$
        if $y^T s > 0$ then                                             ▷Curvature condition
            Append $s, y$ to $S, Y$
            if $|S| > m$ then
                Remove oldest pair
            end if
        end if
        $\theta \leftarrow \theta_{new}$
        $k \leftarrow k + 1$
    end while
    return $\theta$
end procedure
```

6 Gradient Free Methods

6.1 Simulated Annealing

Unlike the previous optimizers, *simulated annealing* is useful in finding *global* optima in the presence of multimodal functions within a usually very large discrete space $\mathcal{S}$. Given some function f defined on $\mathcal{S}$, we would like to find its global maximum. Rather than picking the "best move" using gradient information (like SGD), we propose a random move. Let us start at a state θ_k and propose a random P_{k+1}. We denote $\Delta f = f(P_{k+1}) - f(\theta_k)$.

1. If the selected move improves the solution (i.e. $\Delta f \geq 0$, then it is always accepted and we set $\theta_{k+1} \leftarrow P_{k+1}$.
2. Otherwise, when $\Delta f < 0$ it makes the move with the following acceptance probability

$$p(\theta_{k+1} \leftarrow P_{k+1} \mid \Delta f < 0) = e^{\Delta f / T(t)}$$

We can see that if Δf is very negative (the move is very bad), then this probability of acceptance decreases as well. Furthermore, $T(t)$ represents some sort of "temperature" that we anneal as a function of time, called the *annealing schedule*. T starts off at a high value, increasing the rate at which bad moves are accepted, which promotes exploration of $\mathcal{S}$ and allows the algorithm to travel to suboptimal areas. As T decreases, the vast majority of steps move uphill, promoting exploitation, which means that once the algorithm is in the right search space, there is no need to search other sections of the search space.

Algorithm 2 Simulated Annealing

```
Require: Initial $\theta_0$, Transition kernel $\pi(\theta_{k+1} \mid \theta_k)$, Annealing schedule $T(t)$
for $t = 0$ to $T$ until convergence do
    $P_{t+1} \sim \pi(\cdot \mid \theta_t)$
    if $f(P_{t+1}) - f(\theta_t) \geq 0$ then
        $\theta_{t+1} \leftarrow P_{t+1}$
    else
        $\delta \sim \text{Uniform}[0, 1]$
        if $\delta < \exp[(f(P_{t+1}) - f(\theta_t))/T(t)]$ then
            $\theta_{t+1} \leftarrow P_{t+1}$
        else
            $\theta_{t+1} \leftarrow \theta_t$
        end if
    end if
end for
```

This algorithm is very easy to implement and provides optimal solutions to a wide range of problems (e.g. TSP and nonlinear optimization), but it can take a long time to run if the annealing schedule is very long. We can stop either if T reaches a certain threshold or if we have determined convergence.

6.2 Nelder-Mead

Uses simplex.

7 Lagrangian Optimizers for Constraints

7.1 Lagrange Multipliers

For equality.

7.2 KKT Conditions

For inequality.

7.3 ADMM

8 Non-Lagrangian Optimizers for Constraints

8.1 Penalty

8.2 Projection

8.3 Saddle Point Problem in Nonconvex Optimization

[PDGB14]

9 Sparsity-Inducing Optimizers

9.1 Clipping

We can do SGD with clipping.

References

[PDGB14] Razvan Pascanu, Yann N. Dauphin, Surya Ganguli, and Yoshua Bengio. On the saddle point problem for non-convex optimization, 2014.