Introduction to Numerical Optimization for Deep Learning

EECE 580G
Binghamton University

1 Reading assignment

This lecture covers a part of Chapter 8 from the textbook. Although the directions (and notations) we will take here are slightly different from the textbook. No reading assignment is required for this lecture.

2 Introduction

From the introduction to learning theory lecture, we now know that the gist of a learning algorithm is finding the minimum of the empirical risk, this is not a "pure" optimization problem because we are not really interested in the the empirical risk minimizer, but we hope (trough the choice of the learning algorithm, $\mathcal{H}$, loss function, etc.) that the empirical risk minimizer is close enough to the generalization risk minimizer. Let's recall the definition of the empirical risk and the learning algorithm for some loss function l :

$$\widetilde{L}(h) = \frac{1}{m}\sum_{i=1}^{m} l(h(x^{(i)}), y^{(i)})$$

$$\text{Empirical risk minimizer}: \mathcal{D} \to \mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$$
$$D \mapsto \widehat{f} = \text{argmin}_{h\in\mathcal{H}}\widetilde{L}(h).$$

In order to simplify the notations, we will write the empirical risk minimization problem using the fact that functions in $\mathcal{H}$ are parametrized by $\theta \in \mathbb{R}^c$ (c is the representation capacity seen previously):

$$\min_{\theta} \widetilde{L}(\theta) = \frac{1}{m}\sum_{i=1}^{m} l(\theta, (x^{(i)}, y^{(i)}))$$

This is an important re-writing because working in an abstract function space is not trivial, whereas using parametric functions to define the function spaces reduces the complexity, we are looking for parameters, not an abstract function.

In practice, even $\widehat{f}$ is difficult to evaluate. In fact $\widetilde{L}(\theta)$ might not be an easy function to minimize, we will use numerical methods which will output $\widehat{\widehat{f}}$, an estimate of the empirical risk minimizer.

In order to show how these numerical optimization methods work in the easiest conditions, we further impose that the risk function $\widetilde{L}(\boldsymbol{\theta})$ to be smooth[1] and convex. In practice these conditions (especially the convexity) are not verified, but numerical optimization is still able to yield good results. To make the document more readable, we will use the following change of notation $g(\theta) \leftarrow \widetilde{L}(\theta)$, which becomes an arbitrary function which we need to optimize.

Definition. Convex functions

[1]The smoothness definition used here is Lipschitz continuity, you can view it as an "almost" differentiable function

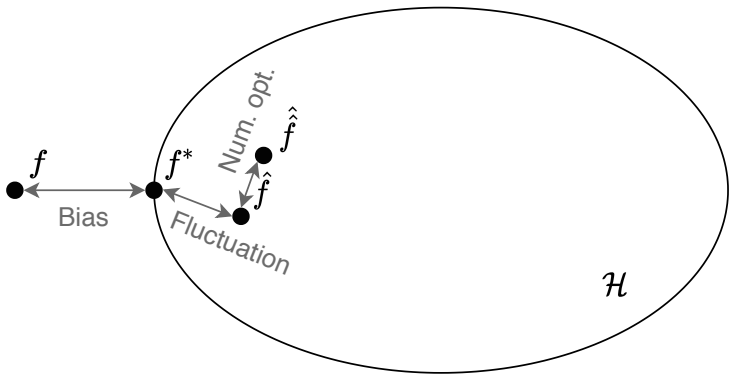

Figure 1: A diagram showing the added error of (approximate) numerical optimization.

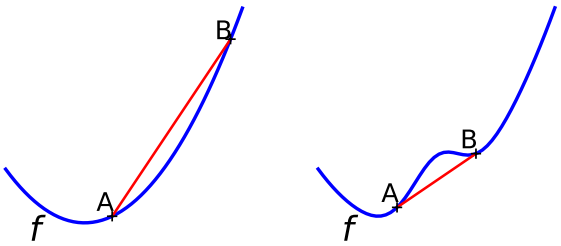

Figure 2: Example of a convex and non convex function

A function $g : \mathbb{R}^c \to \mathbb{R}$ is Convex if and only if for all $\theta_1, \theta_2 \in \mathbb{R}^c$ for all $0 \leq \lambda \leq 1$:

$$g\left(\lambda\theta^{(1)} + (1-\lambda)\theta^{(2)}\right) \leq \lambda g\left(\theta^{(1)}\right) + (1-\lambda)g\left(\theta^{(2)}\right)$$

Proposition. *Twice differentiable functions*

A twice differentiable function $g : \mathbb{R}^c \to \mathbb{R}$ is convex if and only if its Hessian matrix $H(\theta)$ is positive semi-definite[2] for all $\theta \in \mathbb{R}^c$.

Example. 1 dimensional functions

Let the function:

$$\begin{aligned} g\colon \mathbb{R} &\to \mathbb{R} \\ \theta &\mapsto \theta^2, \end{aligned}$$

g is twice differentiable and its Hessian (second derivative is) $g^{''}(\theta) = 2 > 0$. g is convex (left side of Figure 2)

Let the function:

$$\begin{aligned} g\colon \mathbb{R} &\to \mathbb{R} \\ \theta &\mapsto \theta^2 - e^{-5(\theta-0.5)^2}, \end{aligned}$$

g is twice differentiable and its Hessian (second derivative is) $g^{''}(\theta) = 2 - \left(100\theta^2 - 100\theta + 15\right) \mathrm{e}^{-5\left(\theta-\frac{1}{2}\right)^2}$. $g^{''}$ has positive and negative values (try, $\theta = 0$, and $\theta = 0.5$), therefore, g is non convex (right side of Figure 2).

[2] H is positive semi-definite if and only if $\theta^T H\theta \geq 0$ for all $\theta \in \mathbb{R}^c$, another necessary and sufficient condition is for H's eigenvalues to be all non-negative.

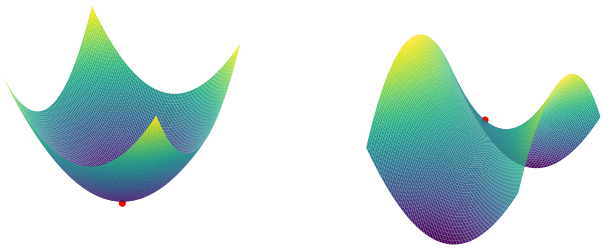

Figure 3: Example of a convex and non convex 2d functions

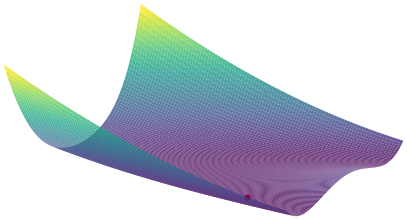

Figure 4: Rosenbrock function surface

Example. 2 dimensional functions

Let the function:

$$g\colon \mathbb{R}^2 \to \mathbb{R}$$
$$(\theta_1, \theta_2) \mapsto \theta_1^2 + \theta_2^2,$$

g is twice differentiable and its Hessian is $H = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$ which is positive semi-definite. g is convex.

Let the function:

$$g\colon \mathbb{R}^2 \to \mathbb{R}$$
$$(\theta_1, \theta_2) \mapsto \theta_1^2 - \theta_2^2,$$

g is twice differentiable and its Hessian is $H = \begin{bmatrix} 2 & 0 \\ 0 & -2 \end{bmatrix}$ which is not positive semi-definite. g is non convex.

Example. Rosenbrock function

We will also use the Rosenbrock function to try different optimization algorithm in challenging conditions (and you will work with it a little more in the assignment).

$$g\colon \mathbb{R}^2 \to \mathbb{R}$$
$$(\theta_1, \theta_2) \mapsto (1 - \theta_1)^2 + 100(\theta_2 - \theta_1^2)^2,$$

The Rosenbrock function is non convex and has a long, narrow, parabolic shaped flat "valley," which hides the minimum (red dot in figure 4).

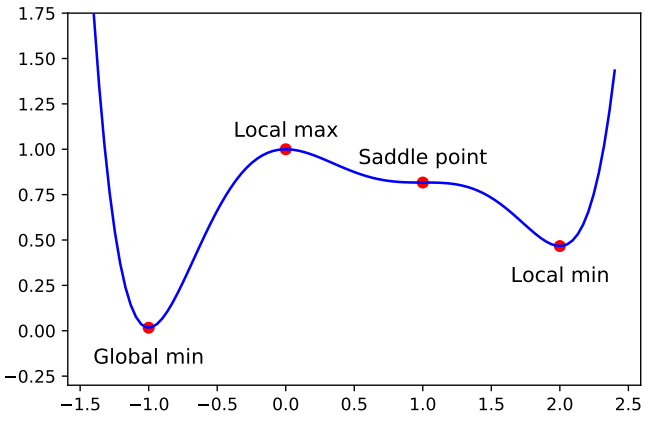

Figure 5: Critical points

3 Critical points

Critical points of a function g are points where the gradient $\nabla g(\theta)$ is zero or not defined. We will mostly be interested in zeros of the gradients, since the functions we will deal with will be differentiable (or differentiable almost everywhere).

$$\nabla g(\theta) = \begin{bmatrix} \frac{\partial g(\theta)}{\partial \theta_0} \\ \frac{\partial g(\theta)}{\partial \theta_1} \\ \dots \\ \dots \\ \frac{\partial g(\theta)}{\partial \theta_{c-1}} \end{bmatrix}$$

Critical points can be:

- Local or global minima
- Local or global maxima
- Saddle points

Proposition. *Critical points of convex functions*

Let $g : \mathbb{R}^c \to \mathbb{R}$ a convex function, any critical point of g is also a global minimum, g doesn't have any saddle points. Moreover, if g is strictly convex (i.e. the inequality of the convexity definition is strict), the global minimum is unique.

We will now show some numerical optimization methods and visualize their performance for convex and some non-convex functions. Please remember that most of these numerical methods have been developed for convex optimization, they are guaranteed to converge at a certain speed, but things get more crafty when we deal with non convex functions.

4 Gradient descent

One of the must famous optimization tool is Gradient Descent, it is an iterative algorithm, which starts from a hand picked initial point $\theta^{(0)}$ and updates it iteratively as:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla g(\theta^{(t)})$$

α is called the learning rate, or the step size, it controls how large the updates are made at each iterations. It is also allowed to change at every iteration if needed.

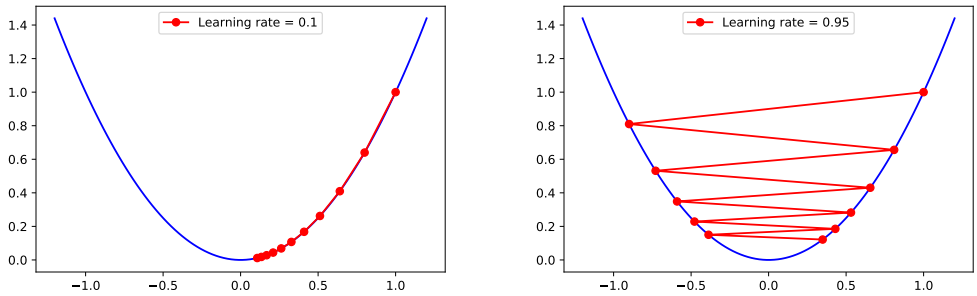

Figure 6: Comparing gradient (10) descent steps for 2 different learning rates, $\alpha = 0.1, 0.95$ for the function $g(\theta) = \theta^2$.

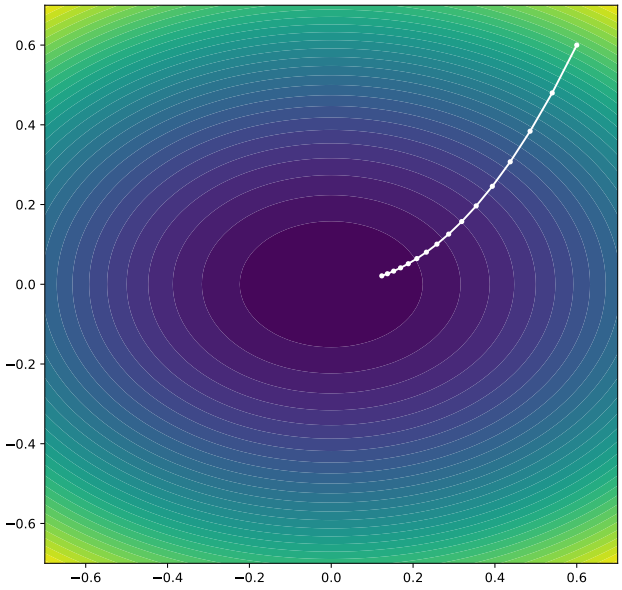

Figure 7: 10 gradient descent steps for the quadratic function $f(x, y) = x^2/2 + y^2$

$\nabla g(\theta^{(t)})$ is the gradient of the function g w.r.t. its parameters $\theta^{(t)}$, i.e. Gradient descent can be intuitively explained by the fact that in a neighborhood $\theta^{(t)}$, g decreases fastest if one goes from $\theta^{(t)}$ in the direction of the negative gradient of g, this is why gradient descent is also called steepest descent algorithm. Based on this observation and if at each iteration, the value of g is decreased: $g(\theta^{(0)}) \geq g(\theta^{(1)}) \geq ... \geq g(\theta^{(t)})$, then after some number of iterations, the algorithm converges to a local minimum of g. In case of convex functions, this local minimum is guaranteed to be the global minimum.

Remark. **Interpretation first order Taylor expansion**

Around a neighborhood of $\theta^{(0)}$ the function g can be approximated using first order Taylor expansion:

$$g(\theta) \simeq g(\theta^{(0)}) + (\theta - \theta^{(0)})^\top \nabla g(\theta^{(0)})$$

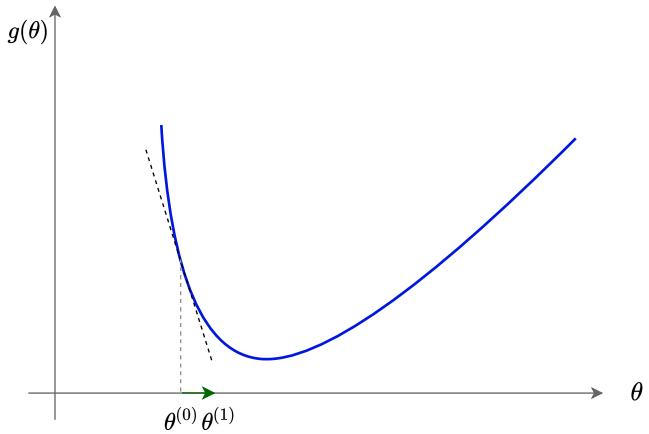

Figure 8: A 1d visualization of 1st order Taylor expansion and gradient descent. In 1d the direction is binary (to the right or to the left of $\theta^{(0)}$) and dictated by the sign of $\nabla g(\theta^{(0)})$ the size of the step is controlled by both α and $\left\|\nabla g(\theta^{(0)}\right\|)$.

$(\theta - \theta^{(0)})^\top \nabla g(\theta^{(0)})$ is a dot product, it is minimum when $(\theta - \theta^{(0)})$ and $\nabla g(\theta^{(0)})$ are aligned and in opposition, i.e. $\theta - \theta^{(0)} = -\alpha \nabla g(\theta^{(0)})$, which gives the gradient descent update presented above.

Theorem. *Gradient descent for smooth convex functions*

Let $g : \mathbb{R}^c \to \mathbb{R}$ a convex and "smooth" function, for a sufficiently small learning rate α, we have

$$g(\theta^{(t)}) - g(\theta^*) \leq \frac{\left\|\theta^{(0)} - \theta^*\right\|_2^2}{2\alpha t}$$

Where $g(\theta^*)$ is the optimal value. Intuitively, this means that gradient descent is guaranteed to converge and that it converges with rate $O(1/t)$.

5 Gradient descent with momentum

As seen in figure 6, a large learning rate can considerably slow down convergence of gradient descent, sometimes even leading to divergence, especially in regions often called "valleys" where the function is rather constant alongside a narrow parameter subspace. We will see a version of a famous variant of gradient descent, called Momentum (Polyak's version), which tweaks gradient descent to "smooth" the gradient vector, but also leads to substantial acceleration of convergence.

$$\begin{aligned} \theta^{(t+1)} &= \theta^{(t)} + v^{(t)} \\ v^{(t)} &= \mu v^{(t-1)} - \alpha \nabla g(\theta^{(t)}) \end{aligned}$$

The velocity vector $\mathbf{v}_t$ acts as a memory that accumulates the directions of the updates in the previous t steps, while the influence of this memory is controlled by μ which is called the momentum coefficient. Notice that if $\mu = 0$ we recover gradient descent.

"Gradient descent is a man walking down a hill. He follows the steepest path downwards; his progress is slow, but steady. Momentum is a heavy ball rolling down the same hill. The added inertia acts both as a smoother and an accelerator, dampening oscillations and causing us to barrel through narrow valleys, small humps and local minima." Goh, Gabriel. Why Momentum Really Works. 2017.

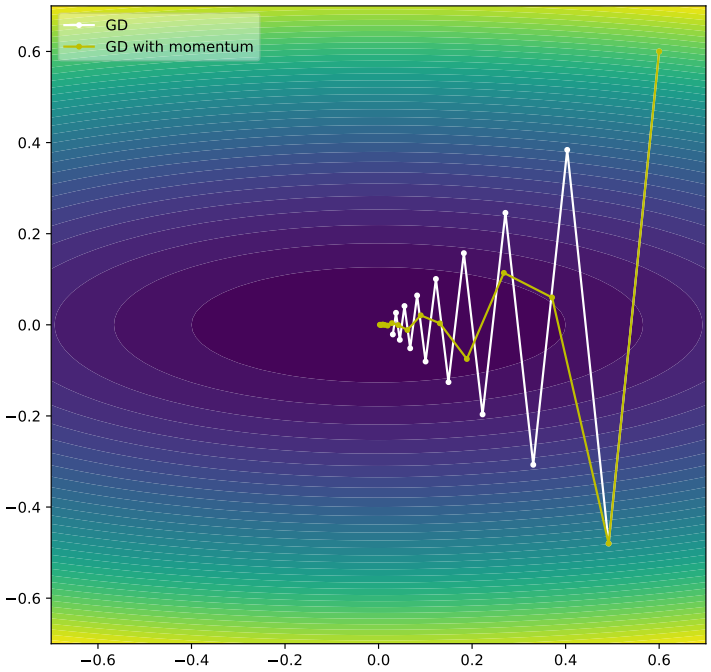

Figure 9: Trajectories of gradient descent and gradient descent with momentum. Both trajectories use the same learning rate and same number of iterations.

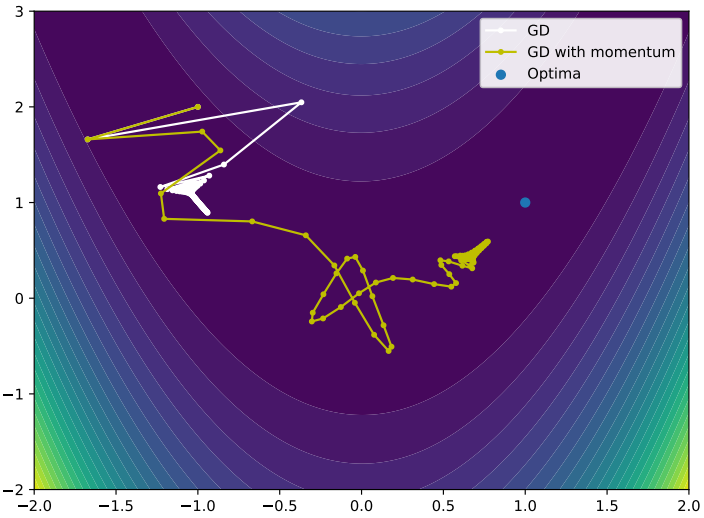

Figure 10: Trajectories of gradient descent and gradient descent with momentum in the Rosenbrock surface. Both trajectories use the same learning rate and same number of iterations. Notice how momentum leads to a faster and better solution, it also produces less oscillations. Momentum can also create its own oscillations (oscillation around (0, 0)).

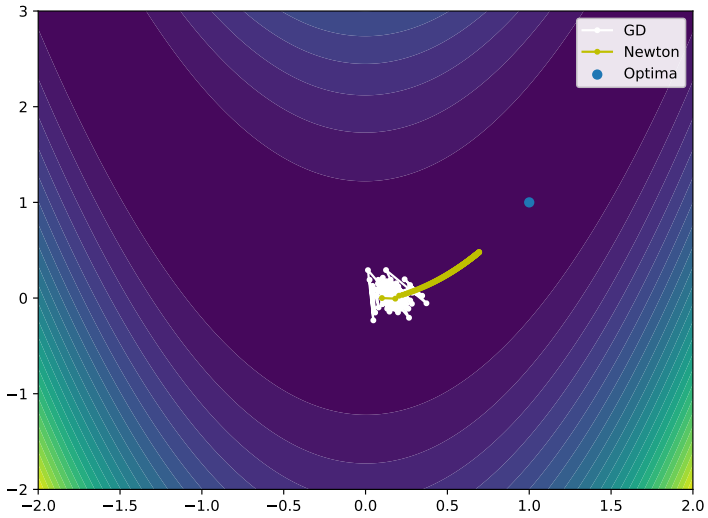

Figure 11: Trajectories of gradient descent and Newton's method in the Rosenbrock surface, inside a narrow valley.

6 Second order methods

6.1 Newton's method

We have seem numerical optimization algorithms based on iterative updates using the gradient, these are called "first order methods," this section describes Newton's method, which goes further by using the "second order" derivative to drive the updates:

$$\theta^{(t+1)} \quad = \quad \theta^{(t)} - [\nabla^2 g(\theta^{(t)})]^{-1} \nabla g(\theta^{(t)})$$

$\nabla^2 g(\theta^{(t)})$ is the second order derivative, also called the Hessian matrix $H(\theta^{(t)})$, it is a $c \times c$ matrix.

Second order methods are also superior when the loss surface has narrow "valleys," as figure 11 shows.

Remark. **Interpretation second order Taylor expansion**

Around a neighborhood of $\theta^{(0)}$ the function g can be approximated using first order Taylor expansion:

$$g(\theta) \simeq g(\theta^{(0)}) + (\theta - \theta^{(0)})^\top \nabla g(\theta^{(0)}) + \frac{1}{2}(\theta - \theta^{(0)})^\top H(\theta^{(0)})(\theta - \theta^{(0)})$$

Solving for critical point:

$$\begin{aligned}
\nabla_\theta g(\theta) = 0 \quad &\simeq \quad \nabla_\theta \left[g(\theta^{(0)}) + (\theta - \theta^{(0)})^\top \nabla g(\theta^{(0)}) + \frac{1}{2}(\theta - \theta^{(0)})^\top H(\theta^{(0)})(\theta - \theta^{(0)}) \right] \\
\Rightarrow 0 \quad &\simeq \quad \nabla g(\theta^{(0)}) + H(\theta^{(0)})(\theta - \theta^{(0)}) \\
\Rightarrow \theta \quad &\simeq \quad \theta^{(0)} - \left[H(\theta^{(0)}) \right]^{-1} \nabla g(\theta^{(0)})
\end{aligned}$$

In other terms, Newton method approximates the function using a second order Taylor expansion (parabola) and decides the next point as the minimum of that parabola.

Remark. **Curvature**

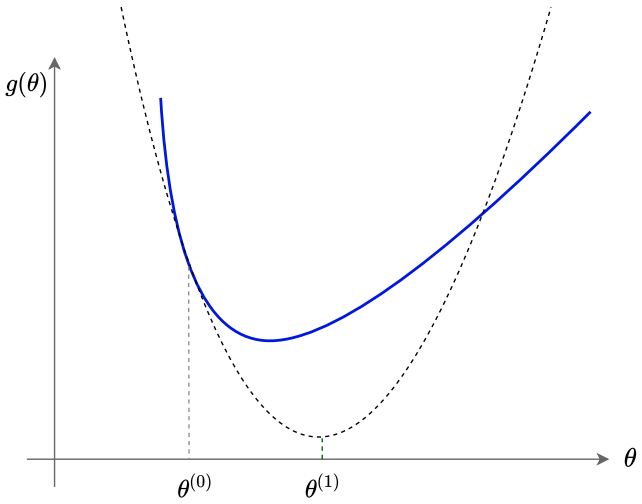

Figure 12: A 1d visualization of 2nd order Taylor expansion and Newton's method.

A useful interpretation of Newton's method is to see the Hessian matrix $H(\theta^{(t)})$ as an indication of the "curvature" of g. If the curvature is low in a particular descent direction, this means that the gradient of the objective changes slowly along that direction. Note that the link between the Hessian matrix and the "curvature" is not theoretically correct but rather qualitative. In Figure 12, points with low curvatures are points around the global minimum, while points further to the left/right have larger curvature.

Remember that the Hessian matrix is a $c \times c$ matrix, which also needs to be inverted, this becomes highly intractable when c grows large, for example, a super large Natural Language Processing model (GPT-3) has 175 Billion parameters... We will see later that this analogy will open doors to different optimization methods based on other approximate "curvature" matrices, which are more tractable to compute.

Example. Linear regression

Recall the linear regression problem again:

$$\begin{aligned}
\nabla_w \text{MSE} &= \nabla_w \frac{1}{m} \|\hat{y} - y\|_2^2 \\
&= \frac{2}{m}(\boldsymbol{X}^\top \boldsymbol{X} w - \boldsymbol{X}^\top y) \\
\nabla_w^2 \text{MSE} &= \frac{2}{m} \boldsymbol{X}^\top \boldsymbol{X}
\end{aligned}$$

Suppose the Newton algorithm starts from a randomly initialized w_0 :

$$\begin{aligned}
w_1 &= w_0 - [\frac{2}{m}\boldsymbol{X}^\top \boldsymbol{X}]^{-1}[\frac{2}{m}\boldsymbol{X}^\top \boldsymbol{X} w_0 - \frac{2}{m}\boldsymbol{X}^\top y] \\
w_1 &= \cancel{w_0 - w_0} + \left(\boldsymbol{X}^\top \boldsymbol{X}\right)^{-1} \boldsymbol{X}^\top y
\end{aligned}$$

We recover the normal equations! In other terms, for linear classification, Newton method converges converges in one iteration.[3]

Note that we can augment Newton's method by adding a learning rate parameter α similar to first order methods:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha[\nabla^2 g(\theta^{(t)})]^{-1} \nabla g(\theta^{(t)})$$

[3]More generally, any quadratic function can be optimized with Newton method in one iteration.

6.2 Another "curvature" matrix, the Adam optimizer

The Newton algorithm shows very promising results (performance in narrow valleys, ill conditioned problems, etc.), but as we noted it requires the computation and inversion of a $c \times c$ matrix, which we cannot afford with deep neural networks. In this section we will see another approximate "curvature" matrix, and how it is used as a building block of many popular optimization algorithms. The approximation consists of using the Fisher information matrix (empirical Fisher) instead of the Hessian matrix[4]:

$$H(\theta) \simeq F(\theta) \simeq \frac{1}{m} \sum_{i=1}^{m} \nabla g(\theta) \nabla g(\theta)^\top$$

Although the empirical Fisher is easier to computer, it is still a $c \times c$ matrix and needs to be inverted. The final approximation drops all non diagonal terms of the empirical Fisher:

$$F(\theta) \simeq \frac{1}{m} \sum_{i=1}^{m} \begin{bmatrix} \nabla g(\theta_0)^2 & & & \\ & \nabla g(\theta_1)^2 & & 0 \\ & & & \\ & 0 & & \nabla g(\theta_{c-1})^2 \end{bmatrix}$$

Which is now easy to compute, store, and invert. Although we lost a lot of information about the geometry, this still performs rather well as it takes care of scaling differences between parameters. In other terms, each parameter has a different learning rate, based on the importance of that parameter in the curvature.

This approximation is used in many state of the art optimization algorithms (RMSprop, adagrad, adam, etc.), but we will focus on the **Adam optimizer**

$$\begin{aligned} v^{(t)} &= \beta_1 v^{(t-1)} + (1-\beta_1) \nabla g(\theta^{(t)}) \\ w^{(t)} &= \beta_2 w^{(t-1)} + (1-\beta_2) \nabla g(\theta^{(t)})^\top \nabla g(\theta^{(t)}) \\ \hat{v}^{(t)} &= \frac{v^{(t)}}{1-\beta_1^t} \\ \hat{w}^{(t)} &= \frac{w^{(t)}}{1-\beta_2^t} \\ \theta^{(t+1)} &= \theta^{(t)} - \alpha \frac{\hat{v}^{(t)}}{\sqrt{\hat{w}^{(t)}} + \epsilon} \end{aligned}$$

First, it uses a "rolling average" of the gradient vector (just like momentum, but a slightly different version) and another rolling average of the diagonal approximation of the empirical fisher (the square of the gradients), then uses these two quantities in the update. Note that a scaling of the gradient and the fisher is done to correct the small bias in early iterations ($v^{(0)} = w^{(0)} = 0$), and a small ϵ is added to avoid numerical instabilities. Figure 13 shows how Adam compares to GD with and without momentum, as well as Newton's method, note that even if Adam was defined as an approximate Newton method, it outperforms it in this case!

7 Stochastic gradient descent

One disadvantage of all optimization algorithms we saw above is computational complexity. Indeed, the gradient of the empirical risk has to be evaluated on all data points (m of them), in many situations, we cannot afford this when the dataset is large. Stochastic gradient descent (SGD) uses an estimator for the gradient at each iteration instead of its exact evaluation over the entire dataset.

Remember that we are using $\widetilde{L}(\theta)$ as a proxy for $L(\theta) = E\left[l(h_{\boldsymbol{\theta}}(X), Y)\right]$, a natural estimator of $\nabla \widetilde{L}(\boldsymbol{\theta})$ is $\nabla l(\boldsymbol{\theta}, (x^{(i)}, y^{(i)}))$ where $(x^{(i)}, y^{(i)})$ is a random data sample from D, in fact this estimator is unbiased. This means that we only evaluate the gradient on a single datapoint at each iteration:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha_t \nabla l(\theta^{(t)}, (x^{(i)}, y^{(i)}))$$

[4]The theory behind the choice of this matrix is out of the scope of this class, but the reader is welcome to research "Natural gradient descent" if interested.

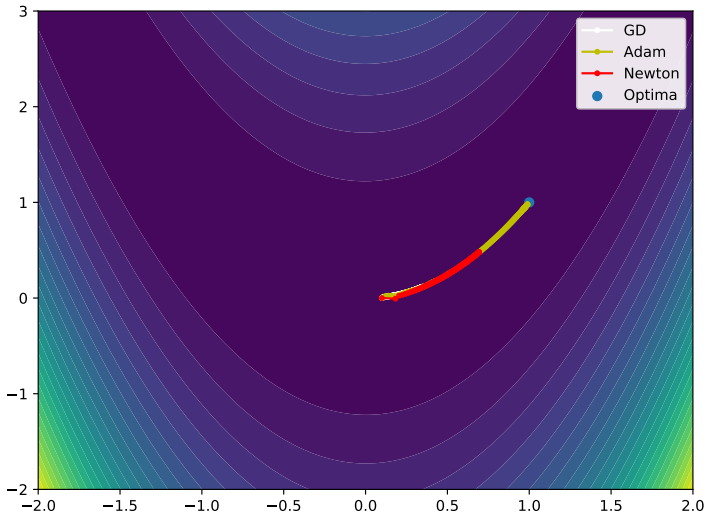

Figure 13: Trajectories of gradient descent with and without momentum and Newton's method in the Rosenbrock surface, inside a narrow valley. Note that when possible, learning rates were tuned to obtain the best performance for each algorithm.

Unlike gradient descent, the learning rate of stochastic gradient descent needs to be a function of the iteration, and satisfy:

$$\sum_{t=0}^{\infty} \alpha_t = \infty, \text{and } \sum_{t=0}^{\infty} \alpha_t^2 < \infty$$

The conditions on the learning rate ensure the convergence of SGD (The convergence for convex functions is proven by Robbins-Siegmund theorem and is not in the scope of this lecture). For SGD, the learning rate has to be a function of the iteration, more specifically, it has to decrease with the number of iterations. For example if you wish to set $\alpha_t = 1/t^p$ then $1/2 < p \leq 1$ in order to satisfy the Robbins conditions. Intuitively, the learning rate should decrease relatively fast to stabilize the search, but not too fast to ensure the optima is reached. $t \mapsto \alpha_t$ is usually called a learning rate schedule.

Example. Linear regression

Let's recall the linear regression problem where the target variable y is approached as $\hat{y} = w^T x$, we use the MSE error $\frac{1}{m} \|\hat{y} - y\|_2^2$ as loss function, we have shown that:

$$\begin{aligned}
\nabla_w \text{MSE} &= \nabla_w \frac{1}{m} \|\hat{y} - y\|_2^2 \\
&= \frac{2}{m}(\boldsymbol{X}^\top \boldsymbol{X} w - \boldsymbol{X}^\top y) \\
&= \frac{2}{m}(\boldsymbol{X}^\top \hat{y} - \boldsymbol{X}^\top y) \\
\text{For one sample: } \nabla_w \text{MSE}^{(i)} &= \frac{2}{m} x^{(i)}(\hat{y}^{(i)} - y^{(i)})
\end{aligned}$$

A stochastic gradient descent algorithm for linear regression would perform until convergence:

$$\begin{aligned}
\hat{y}_t^{(i)} &= w_t^T x^{(i)} \\
w_{t+1} &= w_t - \frac{2\alpha}{m} x^{(i)}(\hat{y}_t^{(i)} - y^{(i)})
\end{aligned}$$

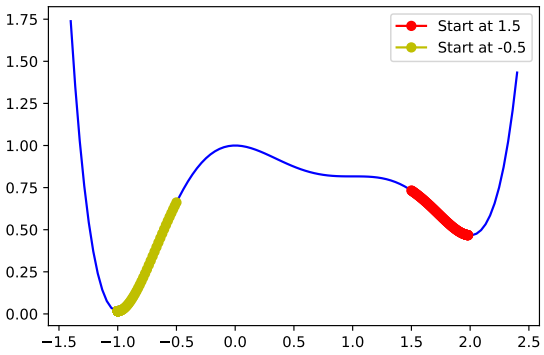

Figure 14: Trajectories of gradient descent with different initial points for a non convex function

Notice how the magnitude of the update is proportional to the error term $\hat{y}_t^{(i)} - y^{(i)}$.

Remark. **Mini-batch gradient descent**

We defined SGD as an iterative algorithm using the gradient evaluation at only one data point as an estimator of the true gradient function. Practitioners often use a mini-batch version of SGD, which consists of randomly drawing data points $\mathcal{B}$ at each iterations $|\mathcal{B}| \ll m$ and evaluating the gradients only on b points. The algorithm becomes:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha_t \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla l(\theta^{(t)}, (x^{(i)}, y^{(i)}))$$

$|\mathcal{B}|$ is called the batch size of the mini-batch SGD. mini-batch SGD is usually faster and more robust than single point SGD.

All algorithms we saw in this handout can be adapted to be mini-batch or stochastic, the only thing changing is the number of samples used to evaluate the gradient (or Hessian) at each iteration.

When using Stochastic algorithms or mini-batch algorithms, we will call an "epoch" the number of iterations needed to through all the training data m, i.e 1 epoch $= \frac{m}{|\mathcal{B}|}$ iterations.

8 In practice

8.1 Non convex optimization in deep learning

Most of the optimization theoretical guarantees are for convex functions, deep learning is almost never about convex functions... This is why this problem becomes much more difficult to optimize. For example the initial point is not an arbitrary choice anymore as shown in figure 14, the learning rate (and its schedule) also plays a major role.

Another interesting observation is that local optima are somewhat rare in deep learning, if c grows large (as it is the case with deep nets) the probability that all c parameters could be 0 and the Hessian's eigenvalues could be all positive becomes close to zero, it is more likely that deep learning loss surfaces have saddle points rather than local optima, and saddle points are much easier to escape. Another difficulty comes from the fact that we can't easily visualize a c dimensional loss surface and build intuition and heuristics.

8.2 Adam is good first choice

Adam with $\beta_1 = 0.9, \beta_2 = 0.999$ and a learning rate around 10^{-3} is almost always a good start. We will see later in the course and assignments how to tune some of these parameters efficiently.

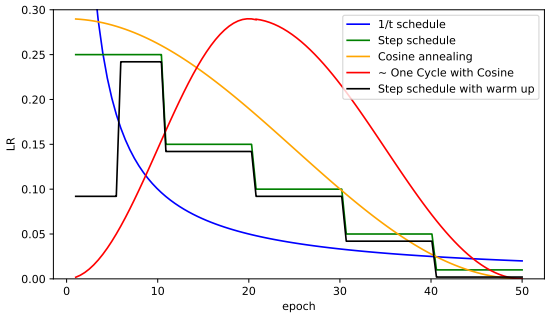

Figure 15: Some learning rate schedules used in practice.

8.3 Learning rate schedules

We introduced in section 7 the notion of learning rate schedule, i.e. decaying the schedule as a function of the iteration. In practice, you will have to experiment with several learning rate schedulers, as there is no theoretical guarantees for non-convex optimization problems such as those you will be dealing with in deep learning. We will see later in the course and assignments how these come into play, figure 15 shows examples of different learning rate schedules used in practice.

8.4 Computer scientist notation

When writing code or pseudo-code, notations such as $\nabla l(\theta, (x^{(i)}, y^{(i)}))$ become difficult to use. In most cases the loss function l is unique and it is obvious that the gradient is evaluated on $(x^{(i)}, y^{(i)})$ or on a batch $\mathcal{B}$ of them. When reading optimizers code, you will frequently see the notation $\nabla l(\theta, (x^{(i)}, y^{(i)})) \equiv d\theta$.

8.5 Other tricks

Early stopping

All figures in this handout are performed with a fixed number of steps. In general optimization algorithms have stopping conditions based on (i) the function value or (ii) the magnitude of the gradient, i.e. the algorithms stop if (i) the function value difference between successive iterations $\left|g(\theta^{(t)}) - g(\theta^{(t-1)})\right|$ is less than a small threshold, or if (ii) the magnitude of the gradient $\left\|\nabla g(\theta^{(t)})\right\|$ becomes smaller than a threshold.

However, as seen in **Introduction to learning theory**, the value of $g(\theta) = \frac{1}{m}\sum_{i=1}^{m} l(\theta, (x^{(i)}, y^{(i)}))$ is a biased estimation of the generalization risk when $(x^{(i)}, y^{(i)}) \in D^{\text{Train}}$. In practice we have access to $D^{\text{Validation}}$ to monitor the unbiased risk, which we can use to stop the training if its value difference between successive iterations is less than a threshold.

Cliffs and exploding gradients

The loss surface for highly nonlinear deep neural networks often contains sharp non-linearities, such as sharp cliffs. These cliffs produce very high derivatives which can lead to divergence or numerical instabilities. Figure 16 shows an example of such "exploding gradients." An easy way to avoid this issue is called "gradient clipping," which consists of "clipping" the gradient vector such as:

$$\text{if } \left\|\nabla g(\theta^{(t)})\right\| > v: \ \nabla g(\theta^{(t)}) \leftarrow \frac{v}{\left\|\nabla g(\theta^{(t)})\right\|} \nabla g(\theta^{(t)})$$

This has the advantage of making sure that the steps taken are not exploding due to some irregularity in the loss surface, the clipped gradient will still point to the direction of the original gradient, but with a smaller magnitude.

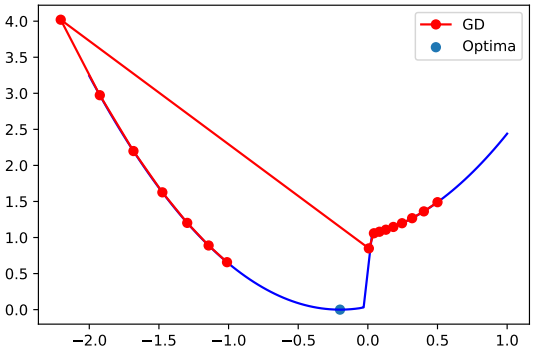

Figure 16: Trajectory of gradient descent in a highly non-linear function with a sharp cliff. The gradient explodes in the neighborhood of the cliff. Note that this function is non convex.