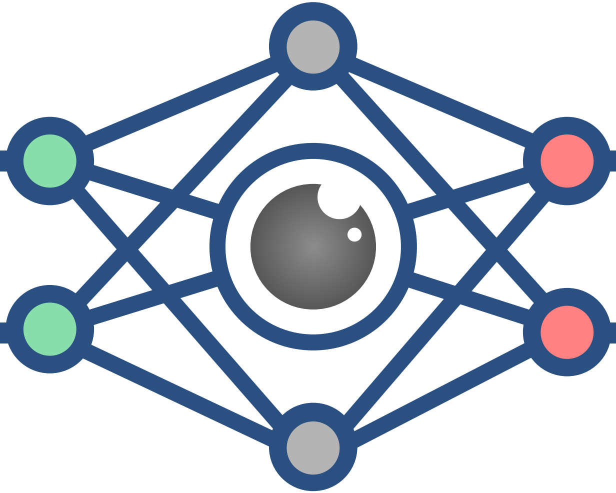
CS3485

Deep Learning for Computer Vision



Lec 4: Optimization in Deep Learning

Announcements

- Labs:
  - Lab 1 is due today at 11:59pm.
  - Lab 2 will be released this afternoon (report in latex, mandatory work in pairs).
- How to succeed in the **scientific lab report**s:
  - Write a concise report: Introduction, Methodology, Results, Discussion and Conclusion sections!
  - Explain what you want to study in the report and why this may be interesting to the reader (this can be done in the introduction).
  - In Methodology, you can: (1) Explain what experiments are taking place, (2) why you think they are relevant, (3) perhaps some background on the theory
  - In Results, you should (1) specify the parameters of your experiments (enough stuff so that another student could run them), (2) add plots and tables of your results.
  - In Discussion, you should interpret your results to the reader.
  - In Conclusion, you draw conclusions about what you discussed!
- **Quiz at the end of the lecture**

(Tentative) Lecture Roadmap

Basics of Deep Learning

1. Intro to Computer Vision
2. Linear Classifiers and Perceptron
3. Multilayer Perceptron
4. Optimization in Deep Learning
5. Pytorch I – MLPs
6. Pytorch II – Images and Regularization
7. Convolutional Neural Networks
8. Data Augmentation and Deep CNNs
9. Transfer Learning and Residual Nets
10. Inception Net and what CNNs learn
11. Adversarial Examples and Self-supervision

Deep Learning and Computer Vision in Practice

1. Intro to MLOps
2. Intro to Object Detection
3. Fast Object Detection
4. Intro to Image Segmentation
5. Applications of Detection and Segmentation
6. Autoencoders
7. Image Generation with GANs
8. Advanced GANs
9. The Attention Mechanism
10. Transformers and ChatGPT
11. Image Generation by Prompt
12. Misc. Topics

Finding the best weights

- Previously, we saw that we can learn the weights of a simple perceptron using the **Perceptron Algorithm**.
- We can extend that to multiclasses, by using multiple perceptron units and training each one separately.
- However, when we add the softmax layer, or add hidden layers, or changed the activation functions, the **perceptron algorithm is not helpful anymore**.
- Today we'll see how to find the weights of general neural networks using **optimization**!
- Before that, we'll see how to perform **Gradient Descent**, a core method in AI!

Loss minimization

- We saw that a Multilayer Perceptron classifies a data point x into a class y using:

$$\hat{y} = NN_\theta(x) = \mathrm{softmax}(W_L a(W_{L-1} \cdots a(W_0 x) \cdots))$$

where NN_θ is a shorthand notation for **the whole neural network** as **a function** and θ represents the weights W_0, W_1, ..., W_L in it.
- Since we want to do **supervised learning**, we have a set of n points $x^{(1)}$, ..., $x^{(n)}$ in D dimensions, each with a class $y^{(1)}$, ..., $y^{(n)}$, of K different classes.
- We can now assess NN_θ at classifying the points in our dataset via the average loss:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{n}\sum_{i=1}^{n} l(NN_\theta(x^{(i)}), y^{(i)})$$

- Naturally, we'd like to find best θ, i.e, those that **minimize** $L(\theta)$.
- Which means that learning in Deep Learning is "just" an **optimization problem.**

Minimization Techniques

- To minimize a **differentiable** function* $f(x)$ one can use **Gradient Descent (GD)**, which starting from some x_0, it finds x_1 such that $f(x_1)$ is lower than $f(x_2)$, and then repeats.
- It uses the derivative of f, defined as df/dx, to check its slope at each point to know where to go next.
- GD works just like a climber who wants to quickly go down a mountain:
  - He first steps around where he "feels" the **slope** of his location,
  - Then decides to take the direction where the slope is the **steepest**,
  - After that he walks a **step** on that direction.
  - He then **repeats** the process until he is at the bottom of the mountain.

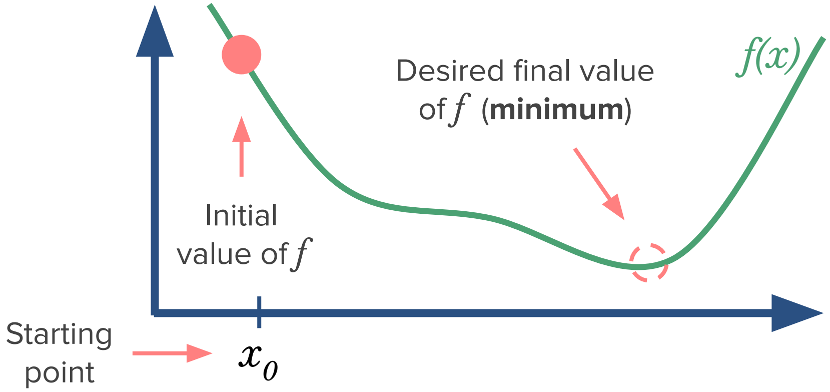
* We'll work on the general case for now and get back to Neural Networks/Deep Learning later.

Gradient Descent in 1D

- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:
  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.
- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".
** ϵ ("epsilon") is just a small number set by the user.

Gradient Descent in 1D

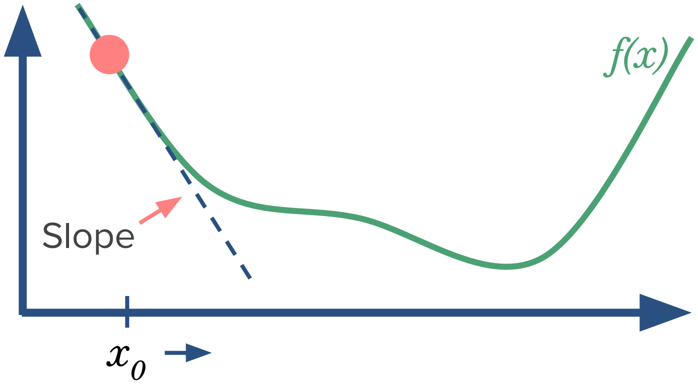
- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:

- If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
- Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".

** ϵ ("epsilon") is just a small number set by the user.

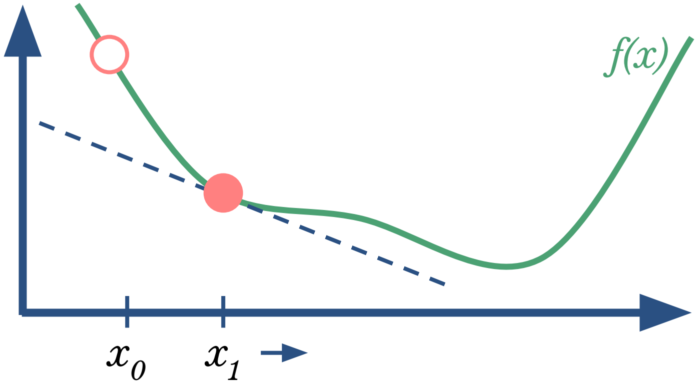
Gradient Descent in 1D

- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:
  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".
** ϵ ("epsilon") is just a small number set by the user.

Gradient Descent in 1D

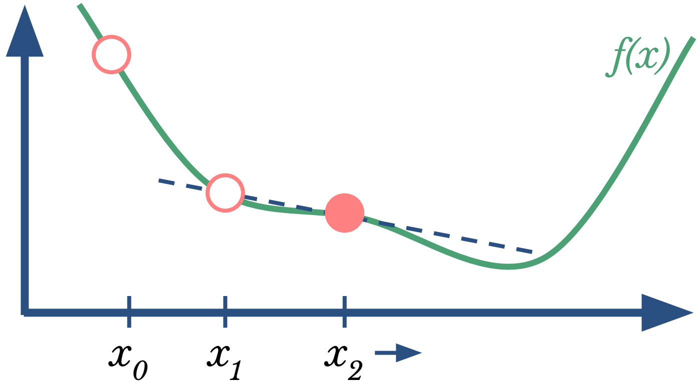
- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:

  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".

** ϵ ("epsilon") is just a small number set by the user.

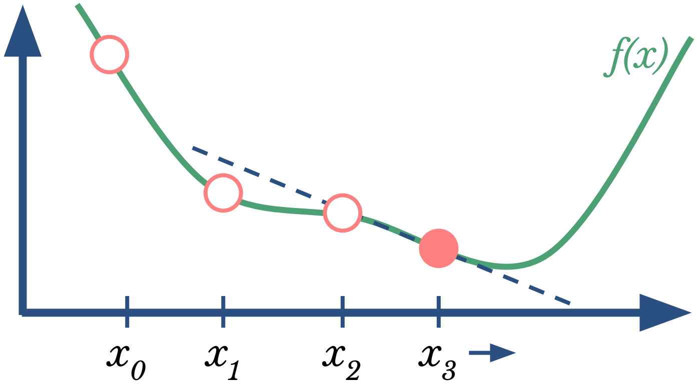
Gradient Descent in 1D

- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:
  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, ...$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like “eta”.
** ϵ (“epsilon”) is just a small number set by the user.

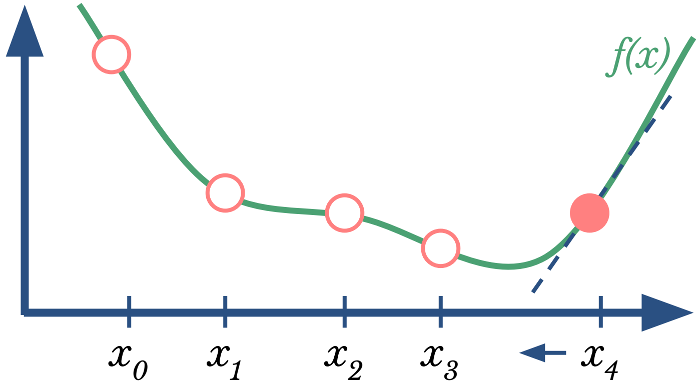
Gradient Descent in 1D

- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:
  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, \ldots$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$



* η reads like "eta".
** ϵ ("epsilon") is just a small number set by the user.

Gradient Descent in 1D

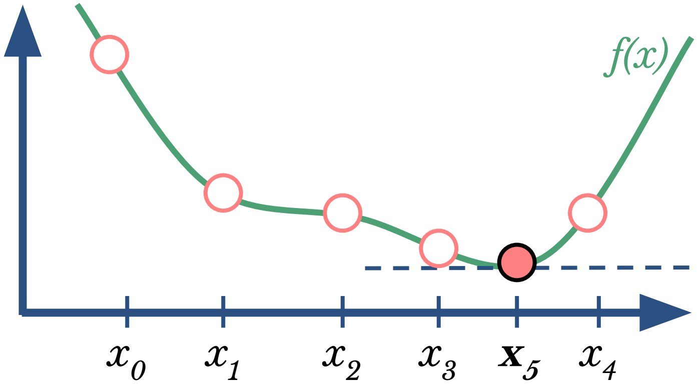
- We use this intuition to mathematically formulate our **minimizer** for functions in 1D:

$$x_{t+1} = x_t - \eta \frac{df}{dx}(x_t)$$

where η (called step size or **learning rate**) is a constant*. This equation simply says:

  - If you are at x_t, the next point you should go to is on the opposite direction of the slope of f at x_t.
  - Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:
  1. Pick a random starting point x_0,
  2. Repeat for $t = 0, 1, 2, ...$ until $|grad| < \epsilon$**
     a. Compute $grad = df(x_t)/dx$
     b. Update x as in $x_{t+1} = x_t - \eta \times grad$

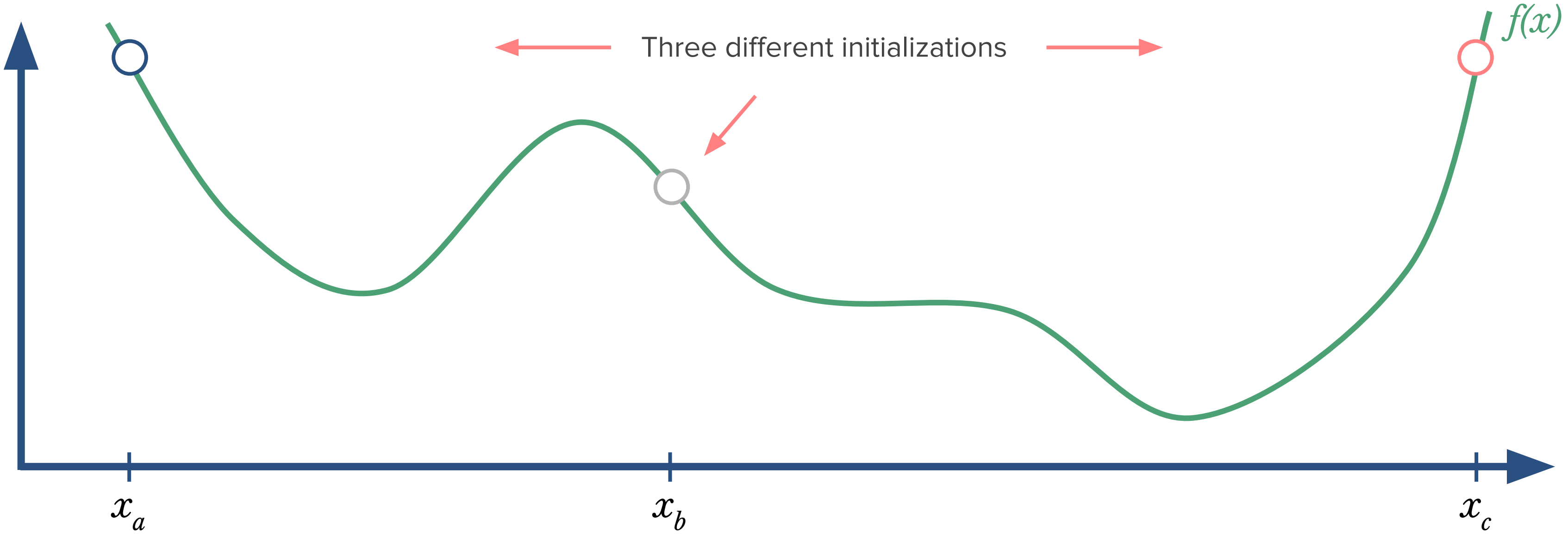


* η reads like "eta".

** ϵ ("epsilon") is just a small number set by the user.

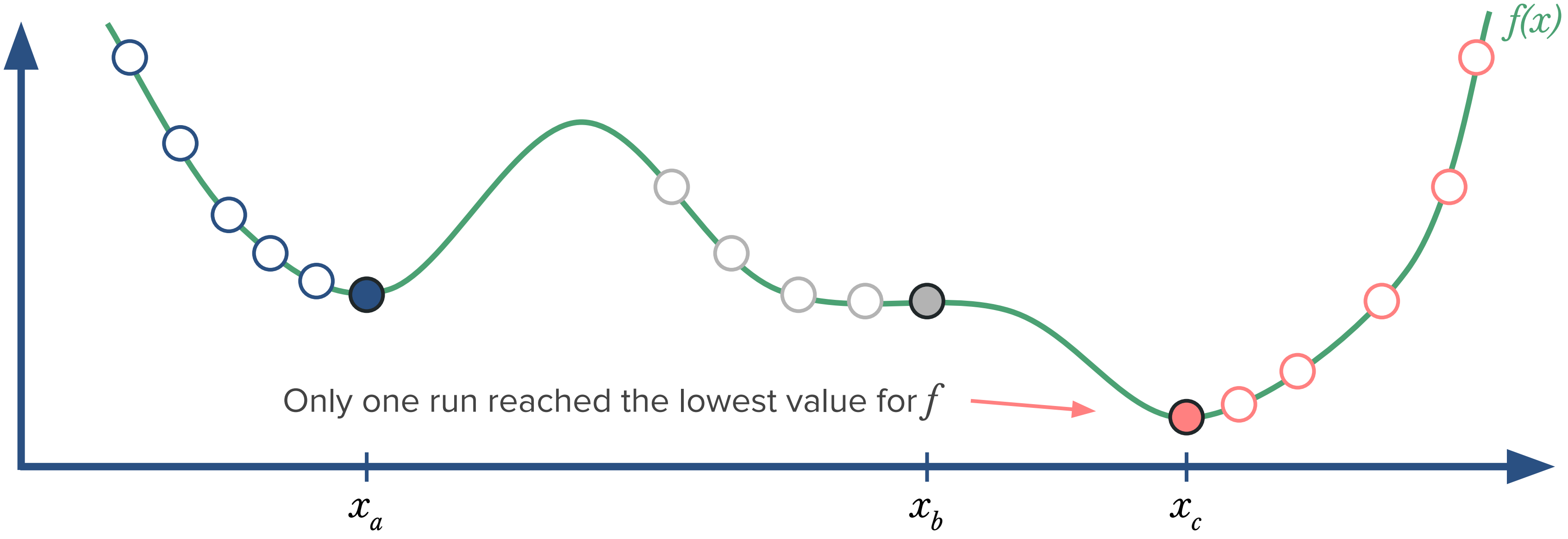
When Gradient descent is suboptimal

- Unfortunately, GD **doesn’t always** find the best x, the **global minimum**.
- Depending on where it is initialized, it output two possible **suboptimal solutions**: a **local minimum** or a **saddle point**.



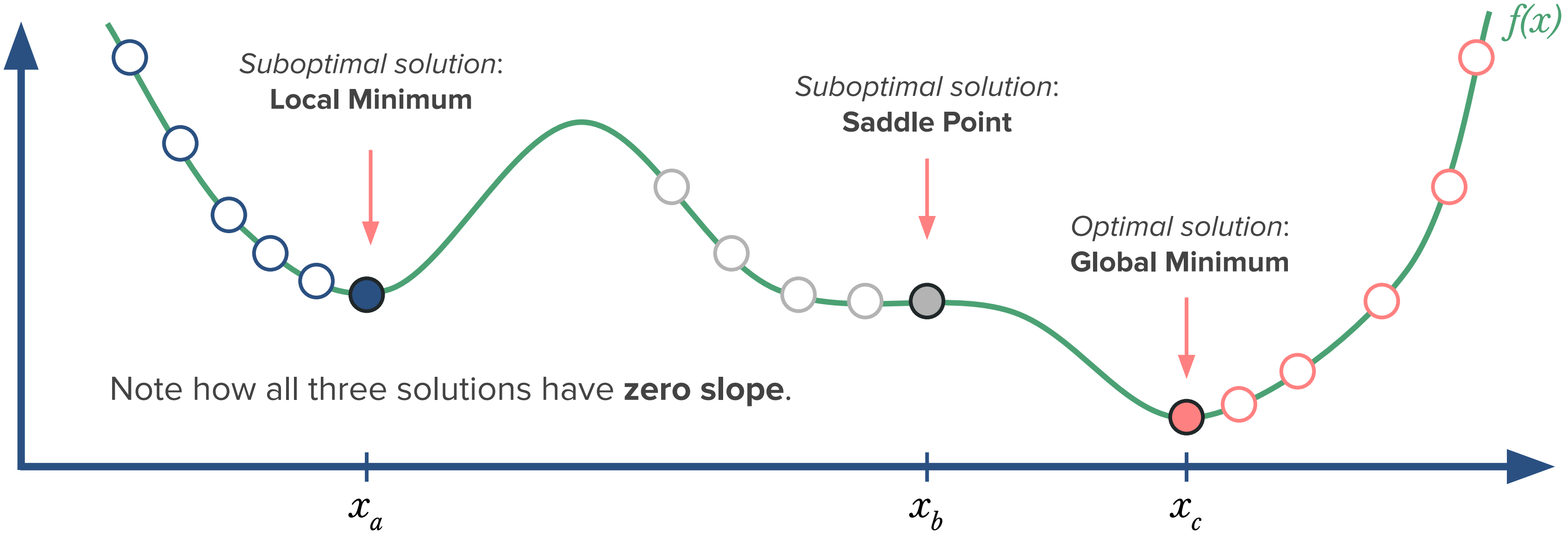
When Gradient descent is suboptimal

- Unfortunately, GD **doesn't always** find the best x, the **global minimum**.
- Depending on where it is initialized, it output two possible **suboptimal solutions**: a **local minimum** or a **saddle point**.



When Gradient descent is suboptimal

- Unfortunately, GD **doesn't always** find the best x, the **global minimum**.
- Depending on where it is initialized, it output two possible **suboptimal solutions**: a **local minimum** or a **saddle point**.



How about more dimensions?

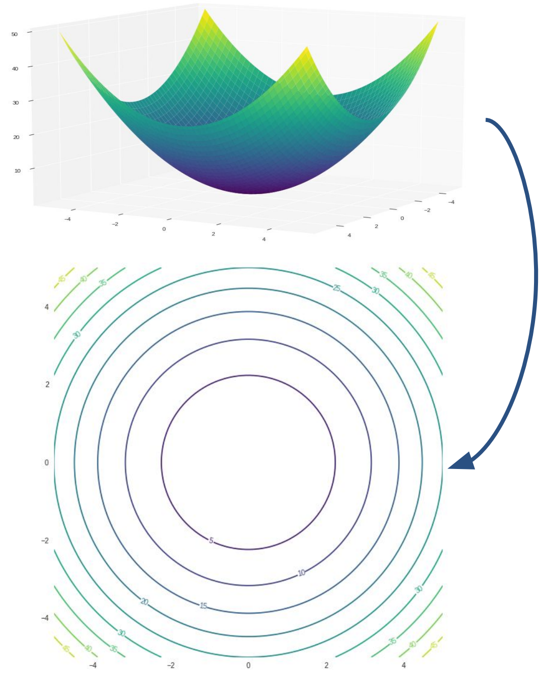
- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ in D dimensions.
- Call $x = [x_1, \dots, x_D]^T$ the vector of variables. Then, the Gradient Descent formula is now:

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

where $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ is the gradient of f.

- The principle is the same as in 1D, but now we have a direction in D dimensions to follow: **the negative of the function's gradient**, $-\nabla_x f(x)$.

$f(x_1, x_2) = x_1^2 + x_2^2$



Same function seen from above

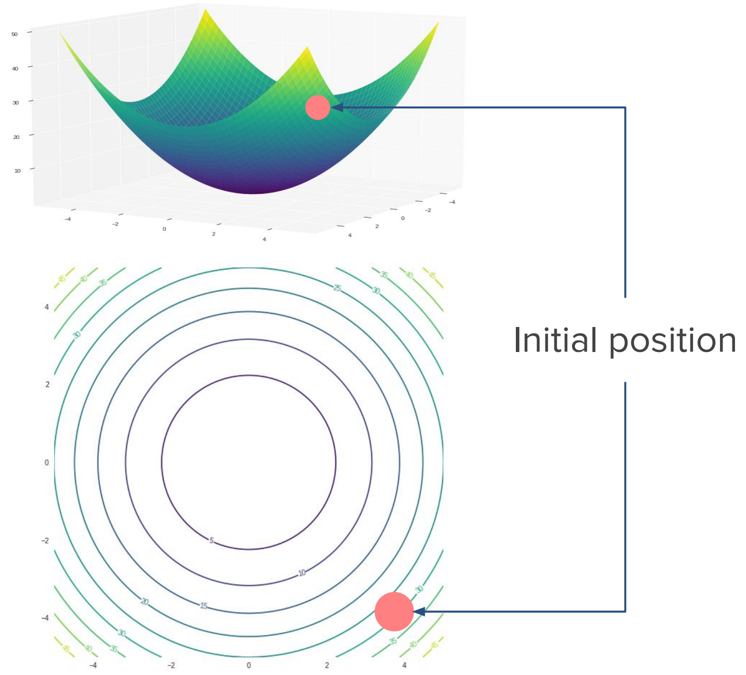
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ in D dimensions.
- Call $x = [x_1, \dots, x_D]^T$ the vector of variables. Then, the Gradient Descent formula is now:

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

where $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ is the gradient of f.
- The principle is the same as in 1D, but now we have a direction in D dimensions to follow: **the negative of the function's gradient,** $-\nabla_x f(x)$.

$f(x_1, x_2) = x_1^2 + x_2^2$



How about more dimensions?

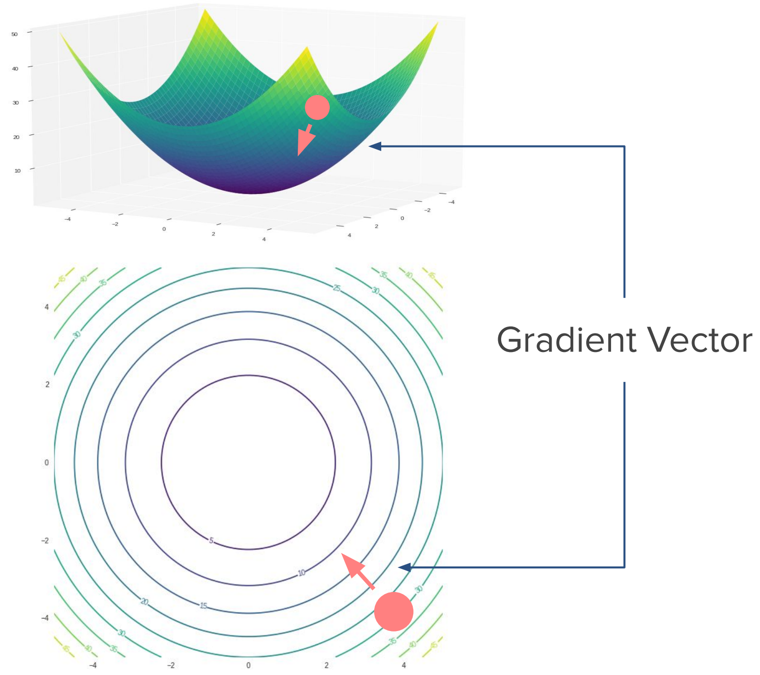
- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ in D dimensions.
- Call $x = [x_1, \dots, x_D]^T$ the vector of variables. Then, the Gradient Descent formula is now:

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

where $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ is the gradient of f.

- The principle is the same as in 1D, but now we have a direction in D dimensions to follow: **the negative of the function's gradient**, $-\nabla_x f(x)$.

$f(x_1, x_2) = x_1^2 + x_2^2$



How about more dimensions?

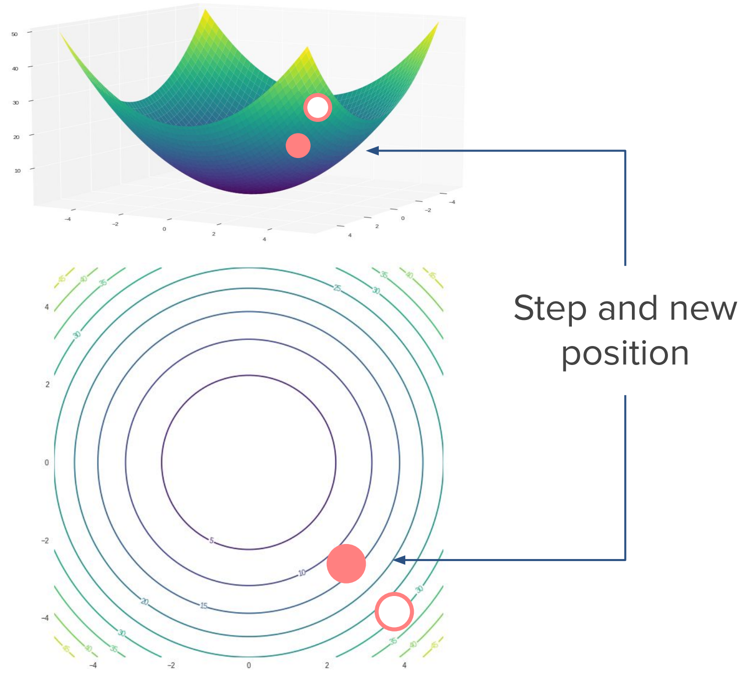
- What we did for 1D can be readily generalized to functions $f(x_1, \ldots, x_D)$ in D dimensions.
- Call $x = [x_1, \ldots, x_D]^\mathrm{T}$ the vector of variables. Then, the Gradient Descent formula is now:

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

where $\nabla_x f = [df/dx_1, \ldots, df/dx_D]^T$ is the gradient of f.

- The principle is the same as in 1D, but now we have a direction in D dimensions to follow: **the negative of the function's gradient,** $- \nabla_x f(x)$.

$f(x_1, x_2) = x_1^2 + x_2^2$



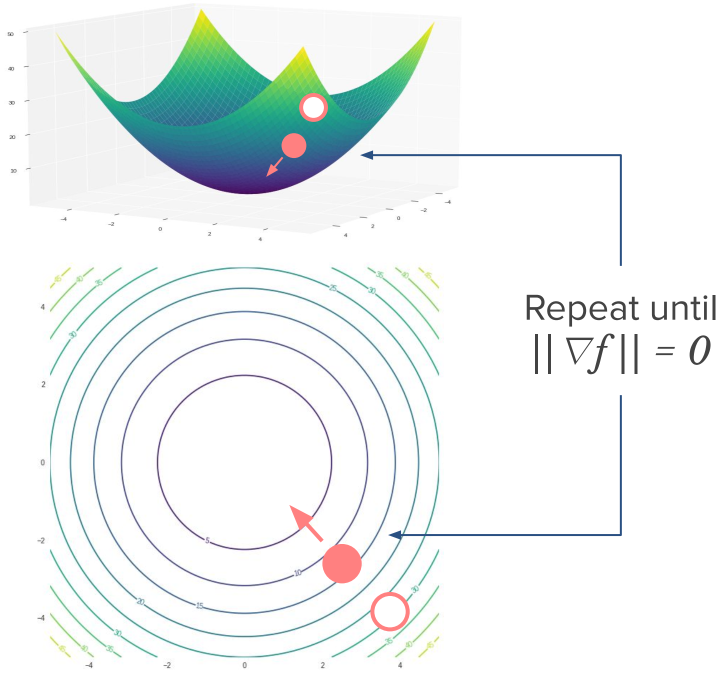
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \ldots, x_D)$ in D dimensions.
- Call $x = [x_1, \ldots, x_D]^T$ the vector of variables. Then, the Gradient Descent formula is now:

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

  where $\nabla_x f = [df/dx_1, \ldots, df/dx_D]^T$ is the gradient of f.
- The principle is the same as in 1D, but now we have a direction in D dimensions to follow: **the negative of the function's gradient**, $-\nabla_x f(x)$.

$f(x_1, x_2) = x_1^2 + x_2^2$



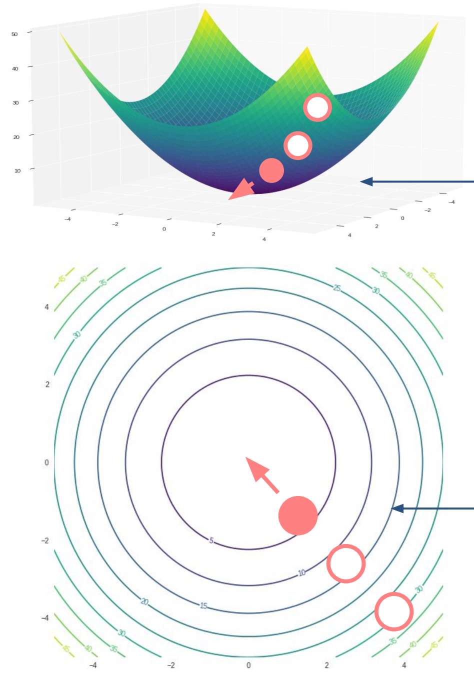
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \ldots, x_D)$ in D dimensions.
- Call $x = [x_1, \ldots, x_D]^T$ the vector of variables. Then, the Gradient Descent formula is now:

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

where $\nabla_x f = [df/dx_1, \ldots, df/dx_D]^T$ is the gradient of f.
- The principle is the same as in 1D, but now we have a direction in D dimensions to follow: **the negative of the function's gradient,** $- \nabla_x f(x)$.

$f(x_1, x_2) = x_1^2 + x_2^2$



Repeat until
$|| \nabla f || = 0$

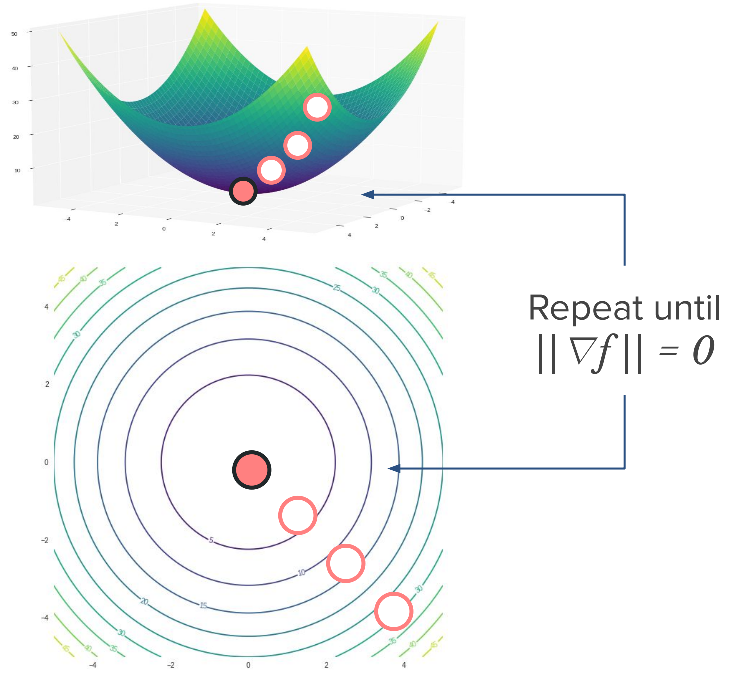
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \ldots, x_D)$ in D dimensions.
- Call $x = [x_1, \ldots, x_D]^{\mathrm{T}}$ the vector of variables. Then, the Gradient Descent formula is now:

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

where $\nabla_x f = [df/dx_1, \ldots, df/dx_D]^T$ is the gradient of f.
- The principle is the same as in 1D, but now we have a direction in D dimensions to follow: **the negative of the function's gradient**, $-\nabla_x f(x)$.

$f(x_1, x_2) = x_1^2 + x_2^2$

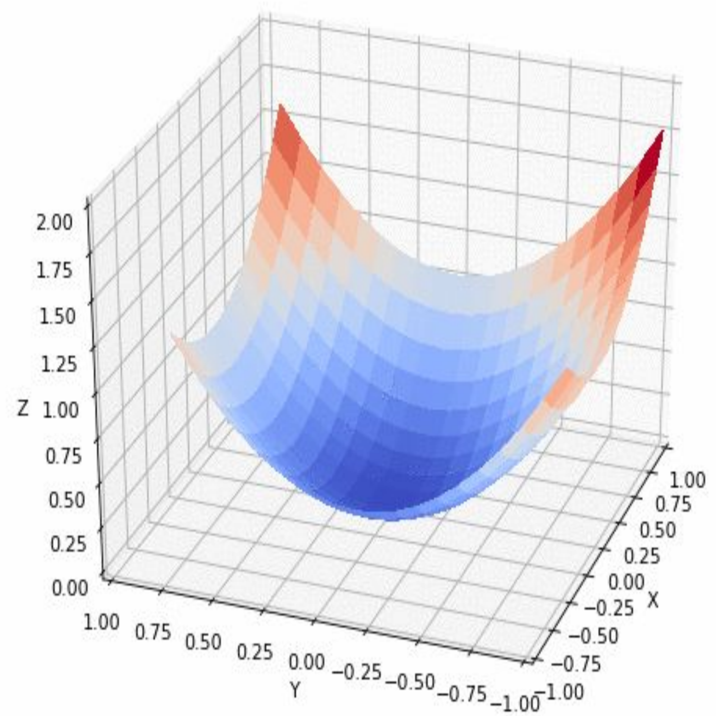


How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ in D dimensions.
- Call $x = [x_1, \dots, x_D]^\mathrm{T}$ the vector of variables. Then, the Gradient Descent formula is now:

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

where $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ is the gradient of f.
- The principle is the same as in 1D, but now we have a direction in D dimensions to follow: **the negative of the function's gradient**,- $\nabla_x f(x)$.

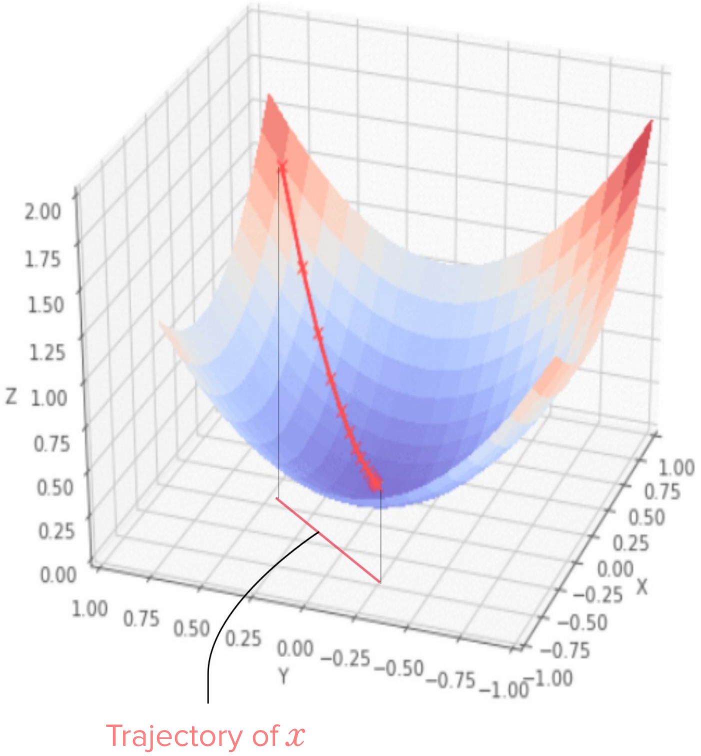


How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \ldots, x_D)$ in D dimensions.
- Call $x = [x_1, \ldots, x_D]^T$ the vector of variables. Then, the Gradient Descent formula is now:

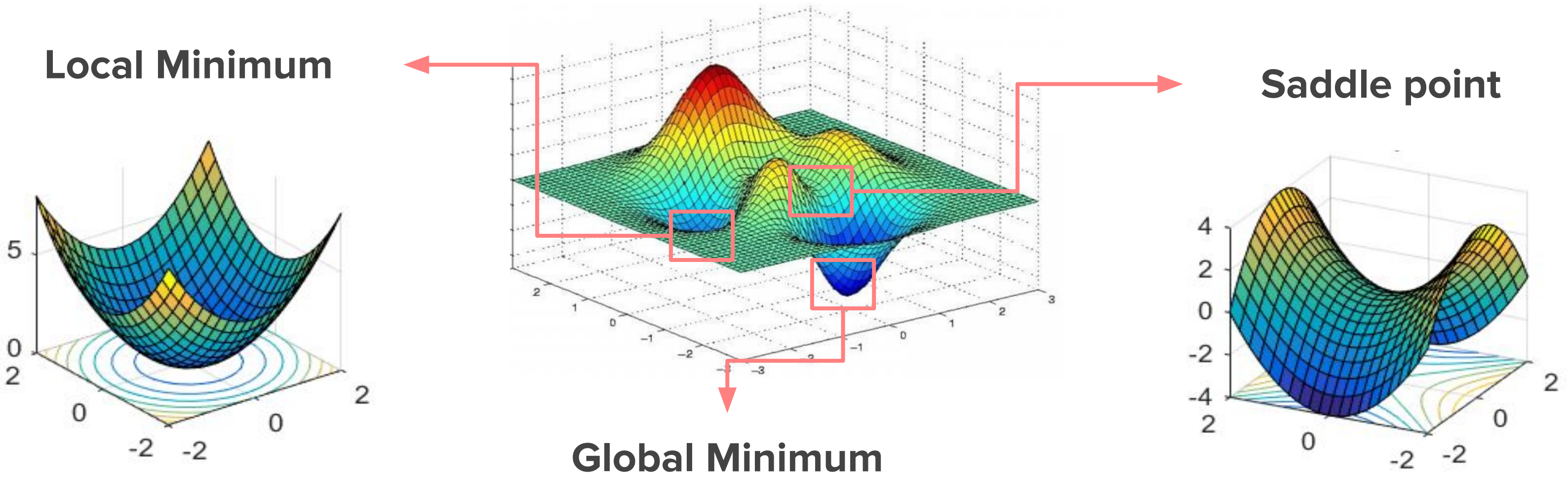
$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

where $\nabla_x f = [df/dx_1, \ldots, df/dx_D]^T$ is the gradient of f.
- The principle is the same as in 1D, but now we have a direction in D dimensions to follow: **the negative of the function's gradient**, $-\nabla_x f(x)$.



When Gradient Descent is suboptimal

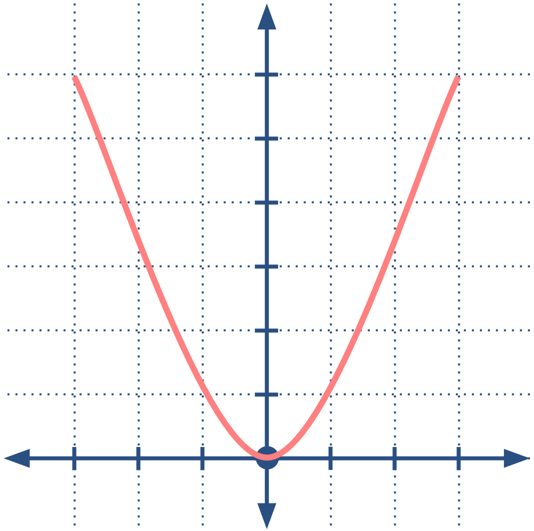
- Also, just like in 1D, gradient descent can get stuck in multidimensional suboptimal solutions and miss the global minimum:



- In fact, one can show that these points are prevalent in **Deep Learning loss surfaces**.
- How can we address this issue?

Exercise (*In pairs*)

- What would be the effect of very large or very small learning rates η in gradient descent for an easy optimization problem (like the parabola on the right)?
- Implement gradient descent for a function that you know the derivative of and its global minimum (like $f(x) = x^2$). Set η to 0.1 and make sure to print the value of the function as you do your GD interactions. *Hint*: create two Python functions `f(x)` and `df(x)`. Now try η equal to $0.01, 1, 100, 1000$. What do you observe? What does this tell you about weakness of gradient descent?



Going back to Neural Networks

- As a recap, in Neural Networks our goal is to find a set of weights θ^* defined by:

$$\theta^* = \arg\min_{\theta} L(\theta)$$

which reads as "θ^* *is the value for* θ *that minimizes* $L(\theta)$ *over all possible* θ" and where:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(NN_\theta(x^{(i)}), y^{(i)})$$

- In GD, we need to compute the gradient of $L(\theta)$, which is:

$$\nabla_\theta L(\theta) = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta l(NN_\theta(x^{(i)}), y^{(i)})$$

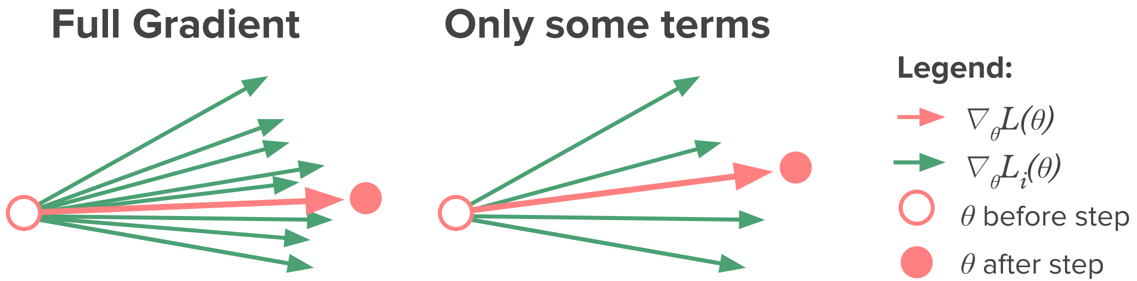
- That is, as we search for θ^*, we have to compute evaluate n gradients at each GD step.
- *One problem*: In modern datasets, $n > 100000$!

Going back to Neural Networks

- Ok, let's summarize our problems with GD so far:
  1. GD is prone to local minima and saddle points,
  2. It is very computationally expensive to compute a step of GD in modern Neural Networks.
- We'll try to solve both problems with the same solution: **randomness**!
- First, to make things easier, let's use the following shorthand notation:

$$L_i(\theta) = l(NN_\theta(x^{(i)}), y^{(i)})$$

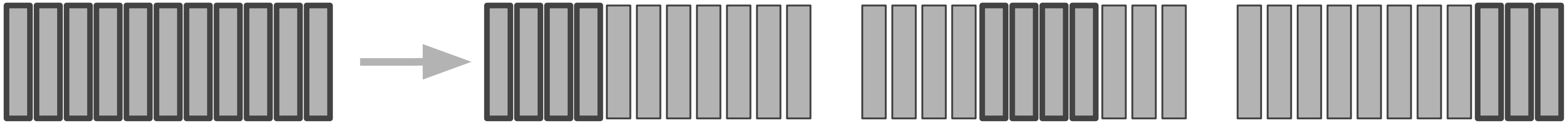
- That is, the gradient vector $\nabla_\theta L(\theta)$ is just an average sum of many vectors $\nabla_\theta L_i(\theta)$!
- *In other words*: we can compute the average of a few $\nabla_\theta L_i(\theta)$ and **the result won't be too far off** from the full gradient $\nabla_\theta L(\theta)$.



Stochastic Gradient Descent

- If we **randomly** choose the datapoints to compute these few $\nabla_\theta L_i(\theta)$ vectors, we are now dealing with **Stochastic* Gradient Descent (SGD)**.
- Since we won't be using the whole dataset to compute one step of SGD anymore, we need to introduce a bit more of deep learning linguo:
  - The set of chosen datapoints used to compute one step of SD is called **mini-batch** (or just batch**). The batches don't need to be exactly of the same size.

Full dataset **Three (deterministic) mini-batches**

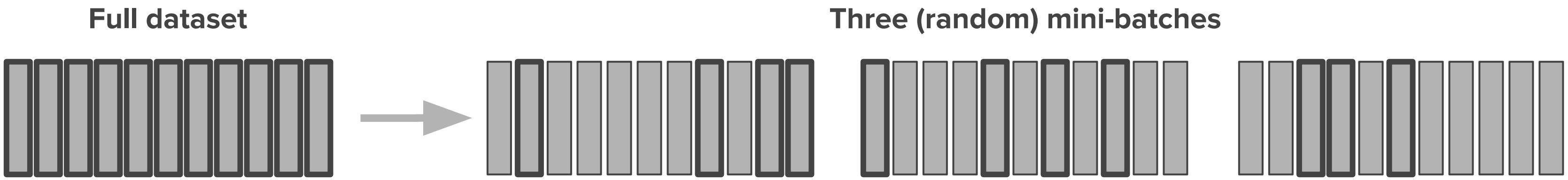


  - SGD will go over each batch and then restart. An **epoch** is over when it has finished going over all batches (and therefore all data points) once.

* In most contexts, "stochastic" simply means "random". ** "*Batch Gradient Descent*" is sometimes used to refer to GD using all datapoints.

Stochastic Gradient Descent

- If we **randomly** choose the datapoints to compute these few $\nabla_\theta L_i(\theta)$ vectors, we are now dealing with **Stochastic* Gradient Descent (SGD)**.
- Since we won't be using the whole dataset to compute one step of SGD anymore, we need to introduce a bit more of deep learning linguo:
  - The set of chosen datapoints used to compute one step of SD is called **mini-batch** (or just batch**). The batches don't need to be exactly of the same size.

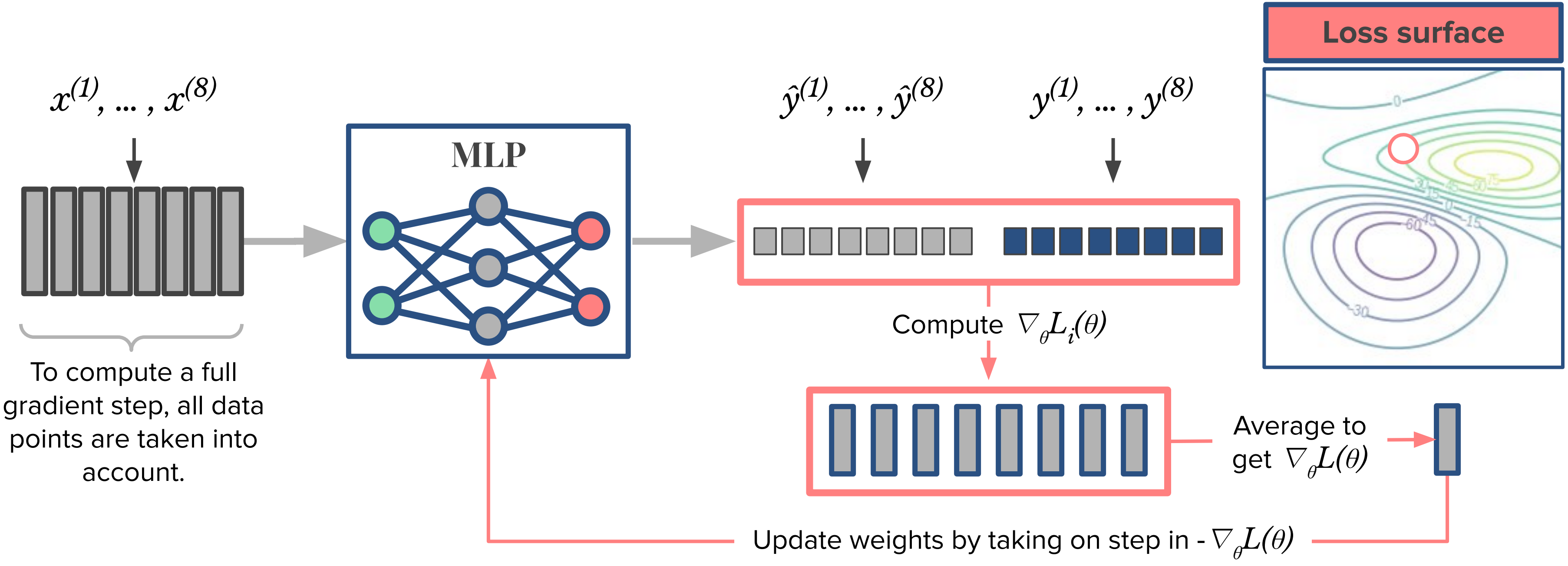


  - SGD will go over each batch and then restart. An **epoch** is over when it has finished going over all batches (and therefore all data points) once.

* In most contexts, "stochastic" simply means "random". ** "*Batch Gradient Descent*" is sometimes used to refer to GD using all datapoints.

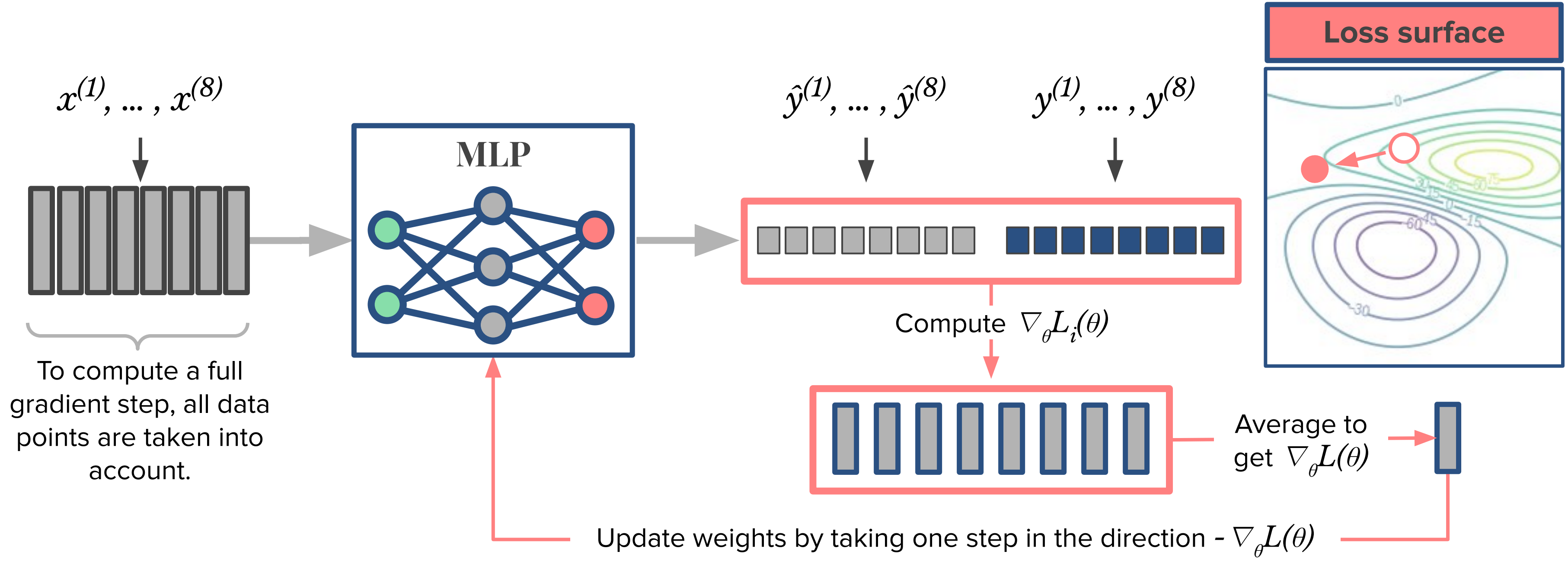
GD vs SGD

- In normal gradient descent, we need to compute all datapoints gradients (eight in the example below) to make one step.



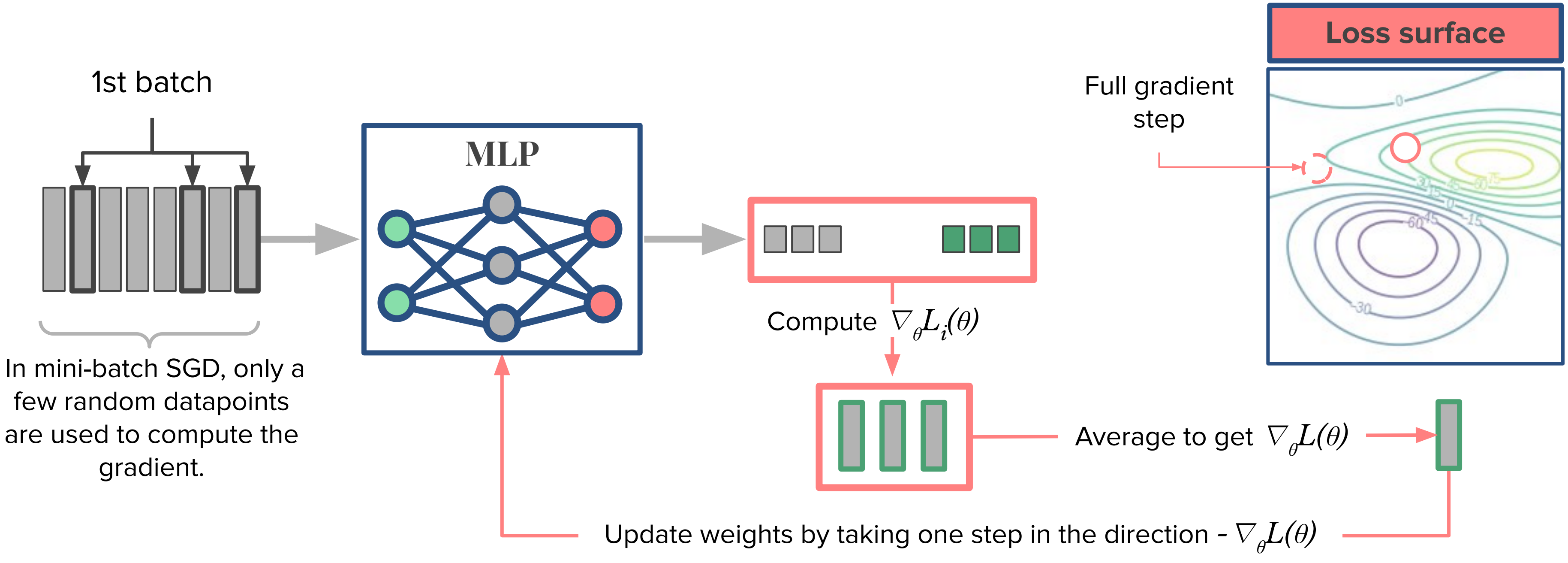
GD vs SGD

- In normal gradient descent, we need to compute all datapoints gradients (eight in the example below) to make one step.



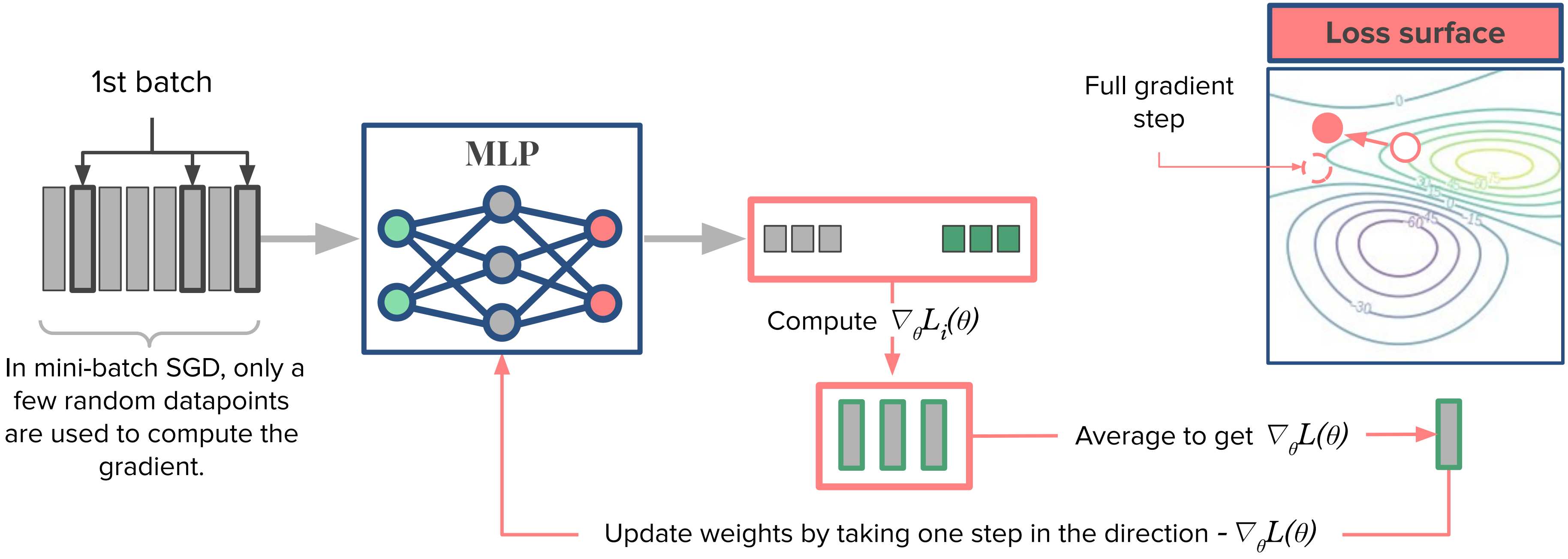
GD vs SGD

- In stochastic gradient descent, we only compute the gradients respective to the mini-batches' points to make a step.



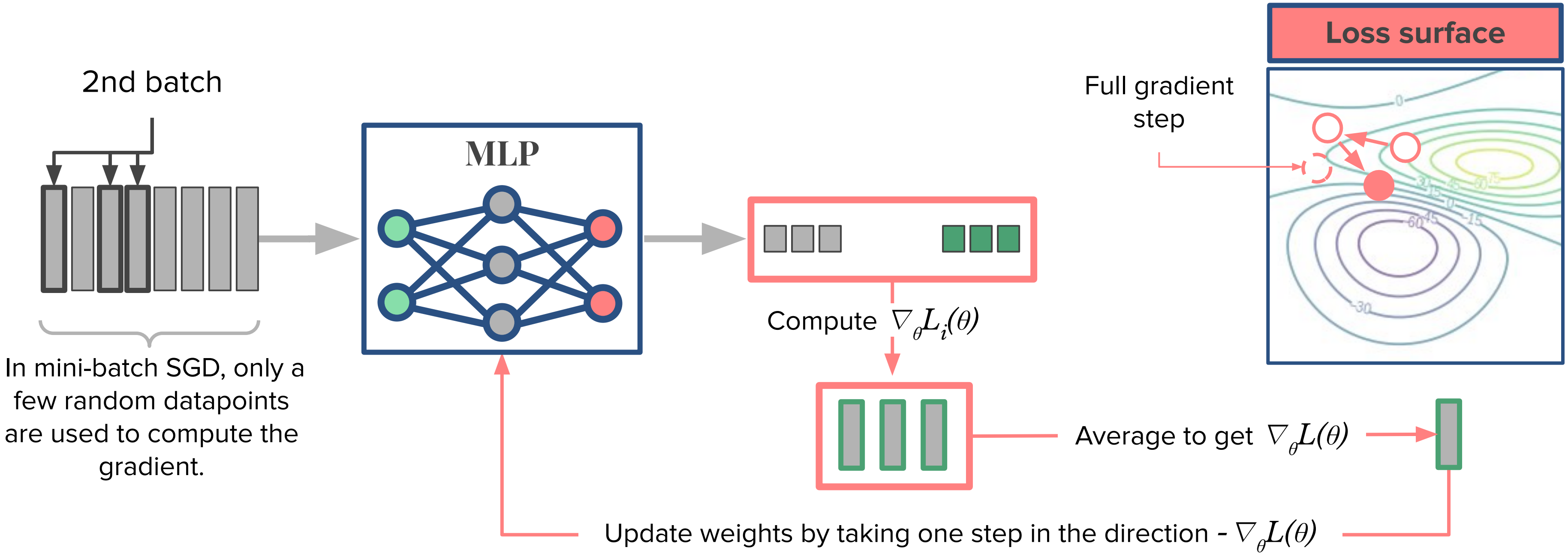
GD vs SGD

- In stochastic gradient descent, we only compute the gradients respective to the mini-batches' points to make a step.



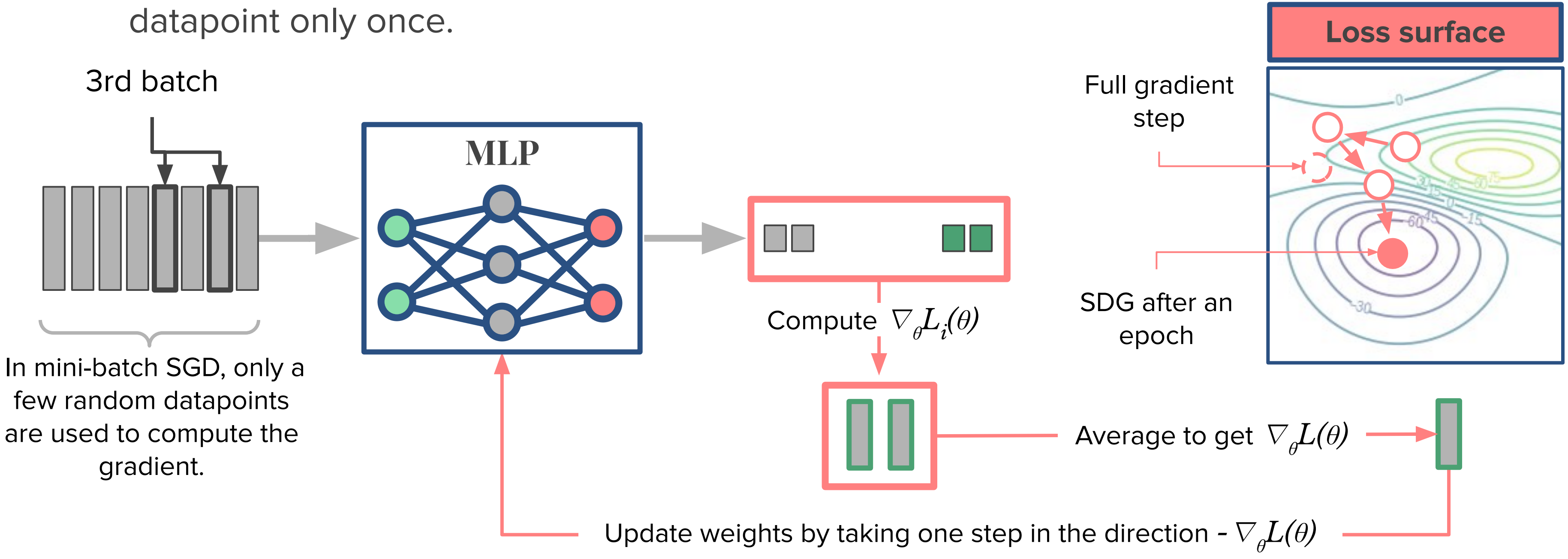
GD vs SGD

- The next mini-batch's step will start from the location found by the previous step.



GD vs SGD

- And we repeat that for the next batch and so on until we're done with one epoch. Note that **SGD made more progress than GD** using each datapoint only once.



Analysing SGD

- SGD definitely makes the gradient computation quicker, but how about the local minima and saddle points?
- Well, the following recent paper seems convenient to answer this question:

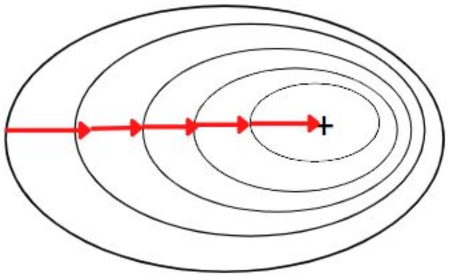
> [Submitted on 13 Feb 2019 (this version), **latest version 4 Sep 2019** (v2)]
>
> **Stochastic Gradient Descent Escapes Saddle Points Efficiently**
>
> Chi Jin, Praneeth Netrapalli, Rong Ge, Sham M. Kakade, Michael I. Jordan

- Why does this happen? In simple english, it happened because SGD can be seen as **adding noise** to every step a full gradient would take.
- That means that it tries out directions that GD would not take, allowing it to **explore the loss surface better** and to hopefully “fall into” the global minimum region.
- This also means that SGD also makes more steps per datapoint than GD, due to this exploration feature.

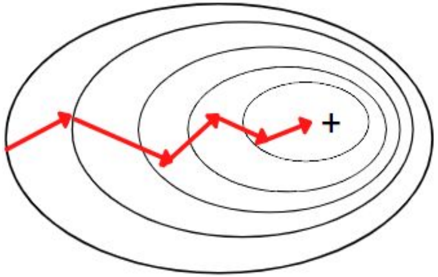
Analysing SGD

- This behaviour is even more explicit when we change the batch size:
  - With a large batch size, SGD makes fewer steps per epoch and each step is more expensive. On the other hand, the full path is more stable.
  - With a smaller batch size, SGD explore the loss surface better and each step becomes cheaper. On the other hand, the path may be too erratic and SGD may take long to converge.

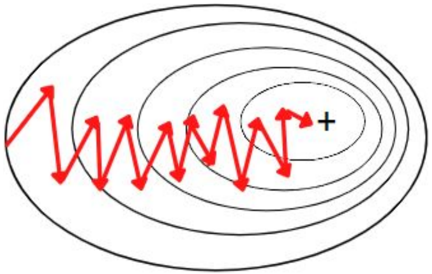
GD



SGD with large batch size



SGD with small batch size



- One solution to this issue is to use **smaller step sizes** (which may make the convergence even slower), other is to add **momentum**.

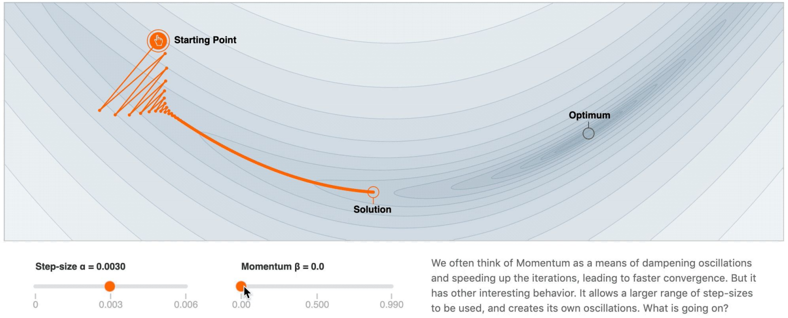
Adding Momentum

- Adding momentum means **using previous steps (gradient directions)** to compute the current direction to go.
- Intuitively, it **hinders the walk from making very sharp turns** from one step to the next.
- Mathematically, we compute a step of SGD with momentum as follows:

$$g_t = \beta g_{t-1} + \nabla_x f(x_t) \qquad x_{t+1} = x_t - \eta g_t$$

where β is called the **momentum parameter** (or simply momentum).

- In practice, adding some momentum makes SGD's path more stable/smooth*, leading to quicker convergences.
- However, adding too much momentum can also hurt convergence*.



* Check out this website and try adding momentum to GD yourself.

Adding adaptive learning rates

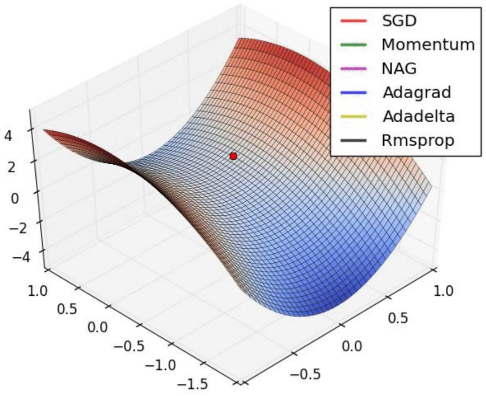
- The final trick to improve SGD is to use **adaptive learning rates (ALR)**, i.e. change the learning rates according to the “intensity” of previous steps.
- Mathematically, the new gradient descent formula would look like the following:

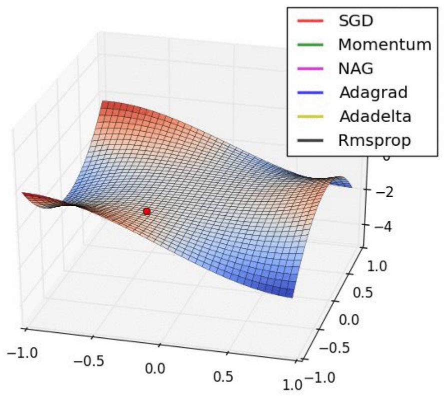
$$x_{t+1} = x_t - \frac{\eta}{\sqrt{\epsilon + \sum_{j=1}^{t} \|\nabla_x f(x_j)\|^2}} \nabla_x f(x_t)$$

  where ϵ is just a small number added to the numerator to avoid division by zero.
- Since the notation $|| \nabla f(x)||^2$ in the numerator represents a gradient magnitude, the intuition behind the whole formula above is: **the larger the previous gradients/steps were, the smaller the next steps will be**.
- Most practical modern implementations of SGD for deep learning nowadays use ALR with slight changes, but keeping the same intuition.

Modern Optimizers

- The literature offers many possible **optimizers** to find best the neural network weights.
- All of them employ one or more of the three main techniques: **Stochasticity**, **Momentum** and **Adaptive Learning Rates**.
- Below, we see how some of these optimizers* are able to escape saddle points.





- In practice, Deep Learning practitioners tend to use an optimizer called **ADAM (Adaptive Moment Estimation)**, since it uses the three techniques above in its algorithm**.

* NAG (Nestorov Accelerated Grad.) is a variation of momentum and Adagrad, Adadelta and RMSprop are different implementations of ALR.

** Here's two websites where you can compare ADAM's performance to SGD's, Momentum's and RMSprop's.

Chain rule and Backpropagation

- After seeing all this theory of optimization, we only miss one thing: **how can we apply it to the neural networks we saw before???**
- Well, the first step is to write out the function we need to minimize.
- If we are using cross-entropy loss, this is the average loss function for our network:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(NN_\theta(x^{(i)}), y^{(i)}) = -\frac{1}{n}\sum_{i=1}^{n}[y^{(i)}]^\top \log(\text{softmax}(W_L a(W_{L-1}\cdots a(W_0 x^{(i)})\ldots)))$$

- Now we "just" need to compute the gradient of $L(\theta)$ with respect to θ! Although not straightforward, one just has to use the **Chain Rule** from calculus.
- Say that you have two **differentiable** functions f and g. Let $y = f(g(x))$ and $u = g(x)$ for a value x. Then we have that:

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

Chain rule and Backpropagation

- *Example*: if $f(x) = x^2$ and $g(x) = 3x^3 + 2$, then the derivative of $y = f(g(x))$ (call $u = g(x)$):

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx} = (2u)(9x^2) = (2(3x^3+2))(9x^2) = 54x^5+36x^2$$

- Using a similar approach one can consider $y = f_1(f_2(f_3...f_n(x)...))$. Let $u_1 = f_2(f_3...f_n(x)...)$, $u_2 = (f_3...f_n(x)...)$ and so on. Then we have that:

$$\frac{dy}{dx} = \frac{dy}{du_1}\frac{du_1}{du_2}\frac{du_2}{du_3}\cdots\frac{du_n}{dx}$$

- For (simple) neural networks, one only has to apply the chain rule to get the weight updates*.
- In that case the first step it to "see" our loss definition as a series of composed function such as $y = f_1(f_2(f_3...f_n(x)...))$.

* Mathematically speaking, the ReLU activation is not differentiable, which should complicate things. In practice, however, the deep learning community simply simply **disregards** this issue. This paper goes in detail about this issue.

Chain rule and Backpropagation

- To make things simple, let's consider a network of just one hidden layer, the loss on only one datapoint (called x with true label y) and that we just want to optimize W_0. Then:

$$u_0(W_0) = l(NN_\theta(x), y) = -y^\top \log(\text{softmax}(W_1 a(W_0 x)))$$

- Let $u_1(z) = -y^\top z$, $u_2(z) = log(z)$, $u_3(z) = \text{softmax}(z)$, $u_4(z) = W_1 z$, $u_5(z) = a(z)$, $u_6(z) = z^\top x$, where z is a **vector** or a **matrix.** Then have that $u_0 = u_1(u_2(u_3(u_4(u_5(u_6(W_0))))))$ and that

$$\frac{du_0}{dW_0} = \frac{du_0}{du_1}\frac{du_1}{du_2}\frac{du_2}{du_3}\frac{du_3}{du_4}\frac{du_4}{du_5}\frac{du_5}{du_6}\frac{du_6}{dW_0}$$

- Now things are much easier: for example, using matrix calculus*, we have $du_1/dz = -y$, $du_4(z)/dz = W_1$ and so on. Remember that one has to compute Jacobians sometimes here.
- Note that you'd need to do something similar to W_1 to optimize it too.

* Here, you can find more refreshing information on derivatives with respect to vectors and matrices

Video: Go AlphaGo!