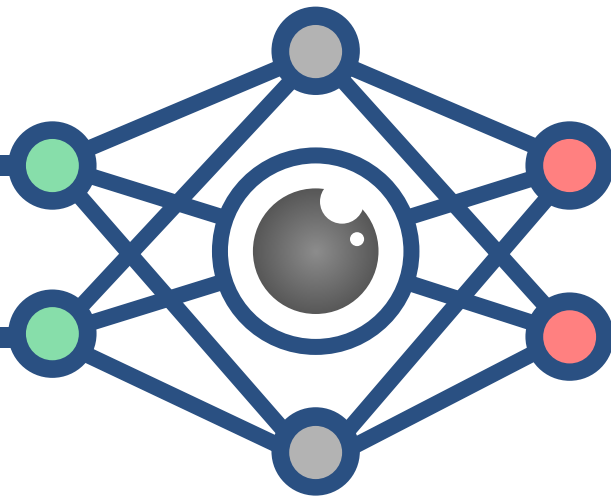


CS3485

Deep Learning for Computer Vision



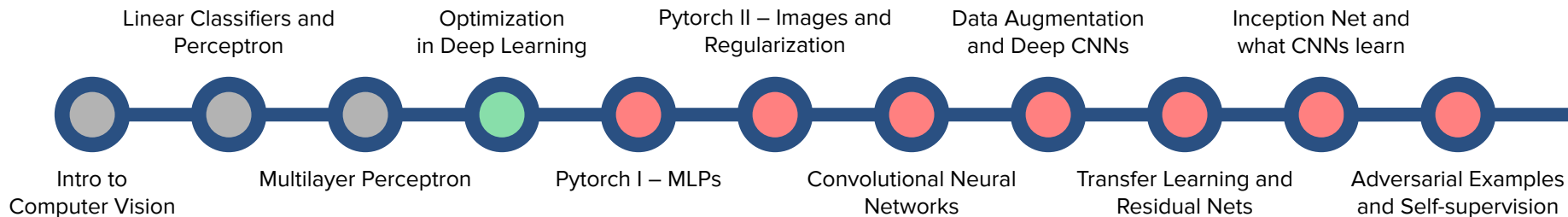
Lec 4: Optimization in Deep Learning

Announcements

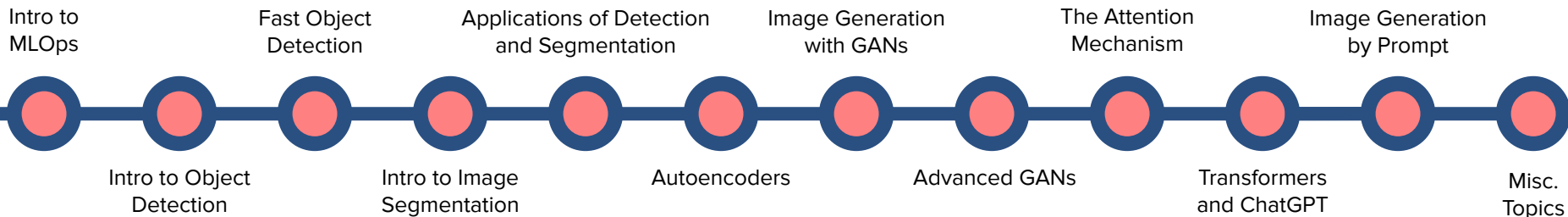
- Labs:
 - Lab 1 is due tomorrow at 11:59pm.
 - Lab 2 was released! Due next Friday. Report will be in latex this time.
- How to succeed in the **scientific lab reports**:
 - Write a concise report: Introduction, Methodology, Results, Discussion and Conclusion sections!
 - Explain what you want to study 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



Deep Learning and Computer Vision in Practice



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 multiclass, 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:

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

where NN_{θ} is a shorthand notation for **the whole neural network as a function** and θ represents the weights W_0, W_1, \dots, W_L in it.

- Since we want to do **supervised learning**, we have a set of n points $x^{(1)}, \dots, x^{(n)}$ in D dimensions, each with a class $y^{(1)}, \dots, y^{(n)}$, 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 is, in order to “feel” 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:

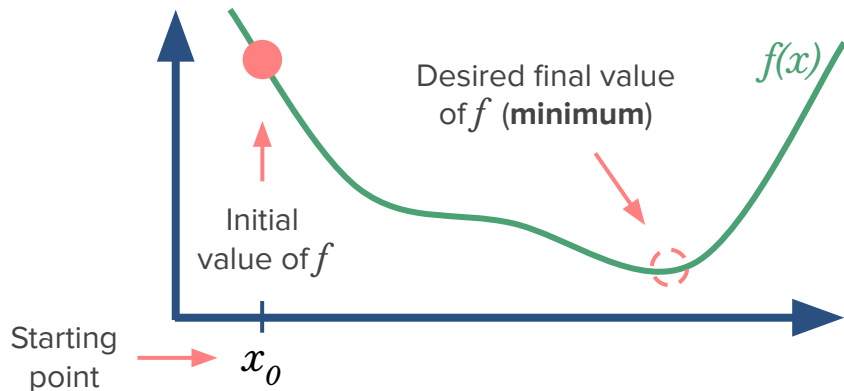
- If you are at x_t , the next point you should go to is on the opposite direction of the slope of f at x_t .
- Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:

1. Pick a random starting point x_0 ,
2. Repeat for $t = 0, 1, 2, \dots$ 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:

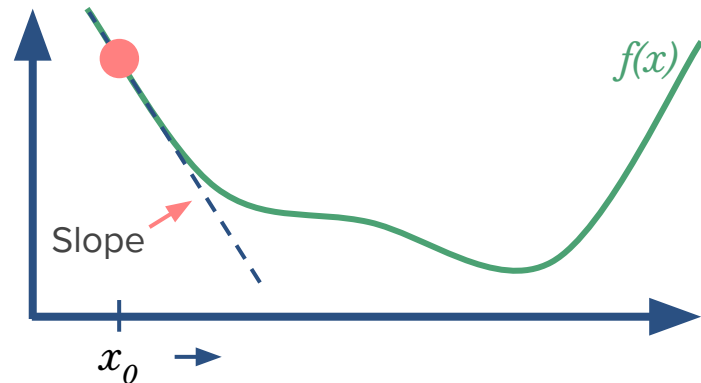
- If you are at x_t , the next point you should go to is on the opposite direction of the slope of f at x_t .
- Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:

1. Pick a random starting point x_0 ,
2. Repeat for $t = 0, 1, 2, \dots$ 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:

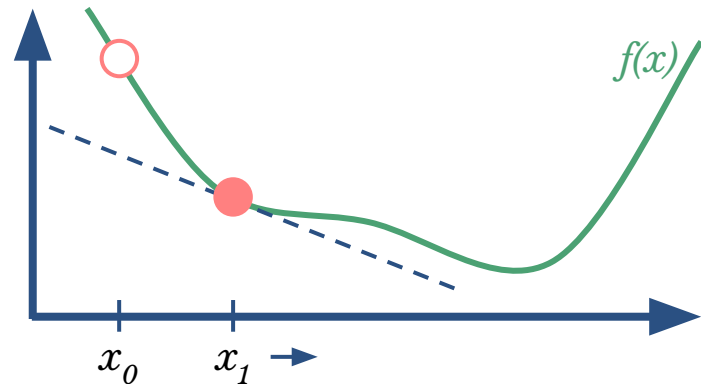
- If you are at x_t , the next point you should go to is on the opposite direction of the slope of f at x_t .
- Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:

1. Pick a random starting point x_0 ,
2. Repeat for $t = 0, 1, 2, \dots$ until $|\text{grad}| < \epsilon^{**}$
 - a. Compute $\text{grad} = df(x_t)/dx$
 - b. Update x as in $x_{t+1} = x_t - \eta \times \text{grad}$

* 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:

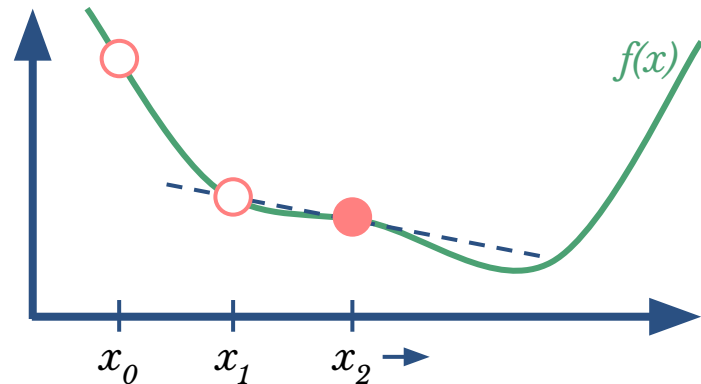
- If you are at x_t , the next point you should go to is on the opposite direction of the slope of f at x_t .
- Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:

1. Pick a random starting point x_0 ,
2. Repeat for $t = 0, 1, 2, \dots$ until $|\text{grad}| < \epsilon^{**}$
 - a. Compute $\text{grad} = df(x_t)/dx$
 - b. Update x as in $x_{t+1} = x_t - \eta \times \text{grad}$

* 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:

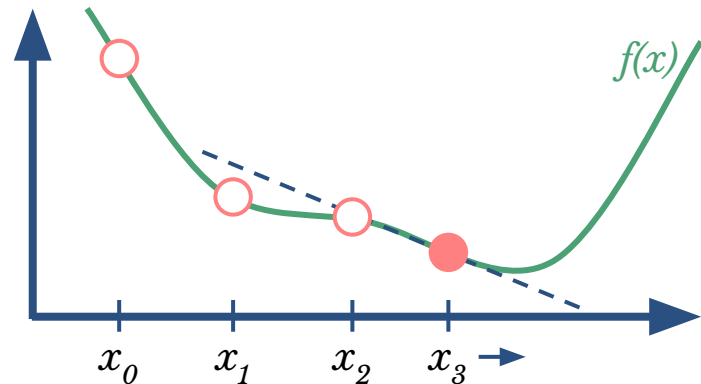
- If you are at x_t , the next point you should go to is on the opposite direction of the slope of f at x_t .
- Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:

1. Pick a random starting point x_0 ,
2. Repeat for $t = 0, 1, 2, \dots$ until $|\text{grad}| < \epsilon^{**}$
 - a. Compute $\text{grad} = df(x_t)/dx$
 - b. Update x as in $x_{t+1} = x_t - \eta \times \text{grad}$

* 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:

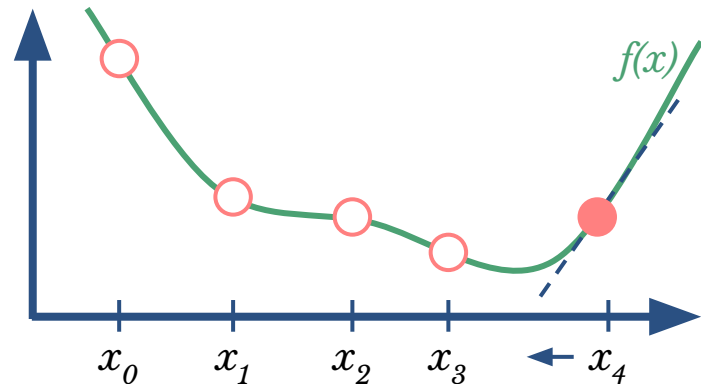
- If you are at x_t , the next point you should go to is on the opposite direction of the slope of f at x_t .
- Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:

1. Pick a random starting point x_0 ,
2. Repeat for $t = 0, 1, 2, \dots$ until $|\text{grad}| < \epsilon^{**}$
 - a. Compute $\text{grad} = df(x_t)/dx$
 - b. Update x as in $x_{t+1} = x_t - \eta \times \text{grad}$

* 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:

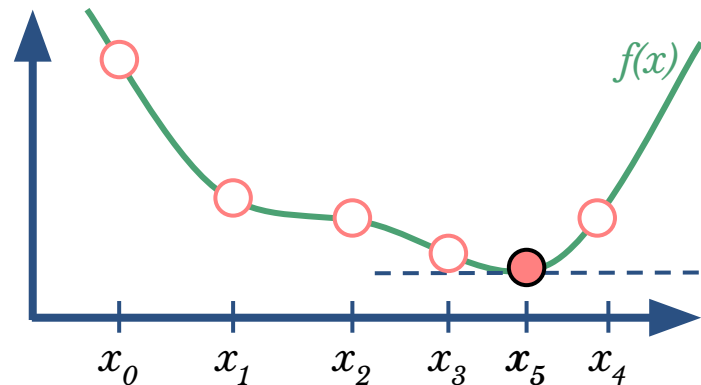
- If you are at x_t , the next point you should go to is on the opposite direction of the slope of f at x_t .
- Then walk a step of size proportional to how steep that slope is in the direction.

- With this definition, the **gradient descent algorithm** in 1D is very simple:

1. Pick a random starting point x_0 ,
2. Repeat for $t = 0, 1, 2, \dots$ until $|\text{grad}| < \epsilon^{**}$
 - a. Compute $\text{grad} = df(x_t)/dx$
 - b. Update x as in $x_{t+1} = x_t - \eta \times \text{grad}$

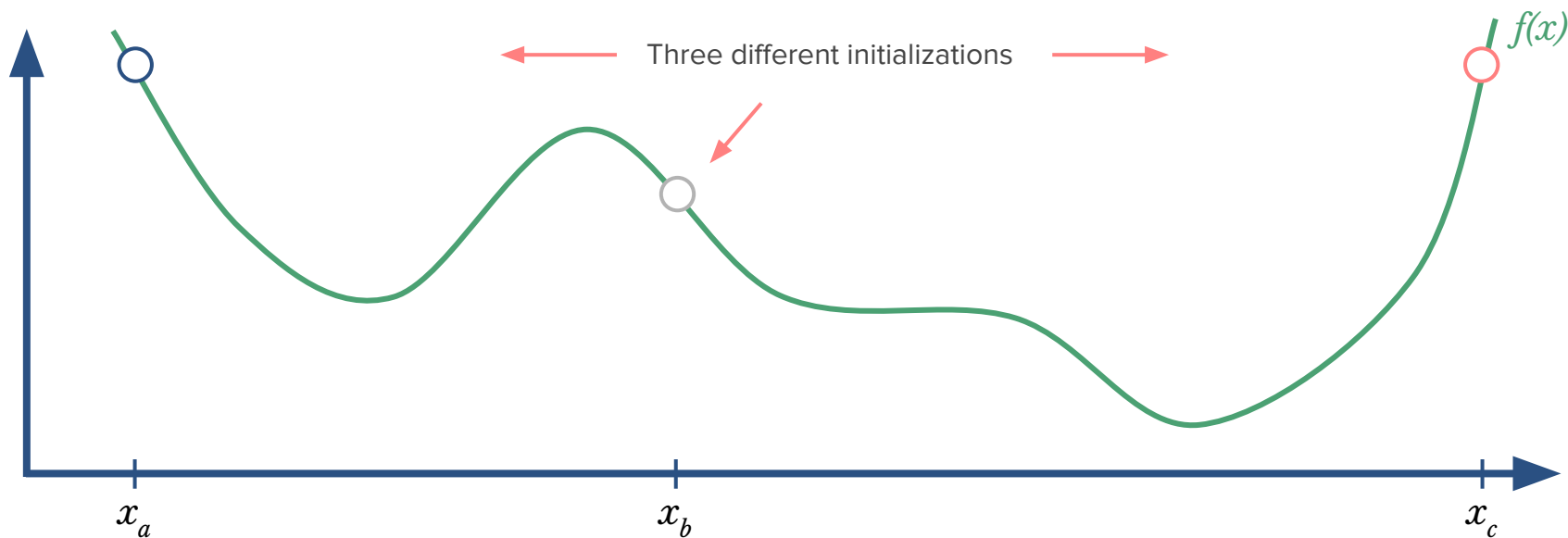
* η reads like “eta”.

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



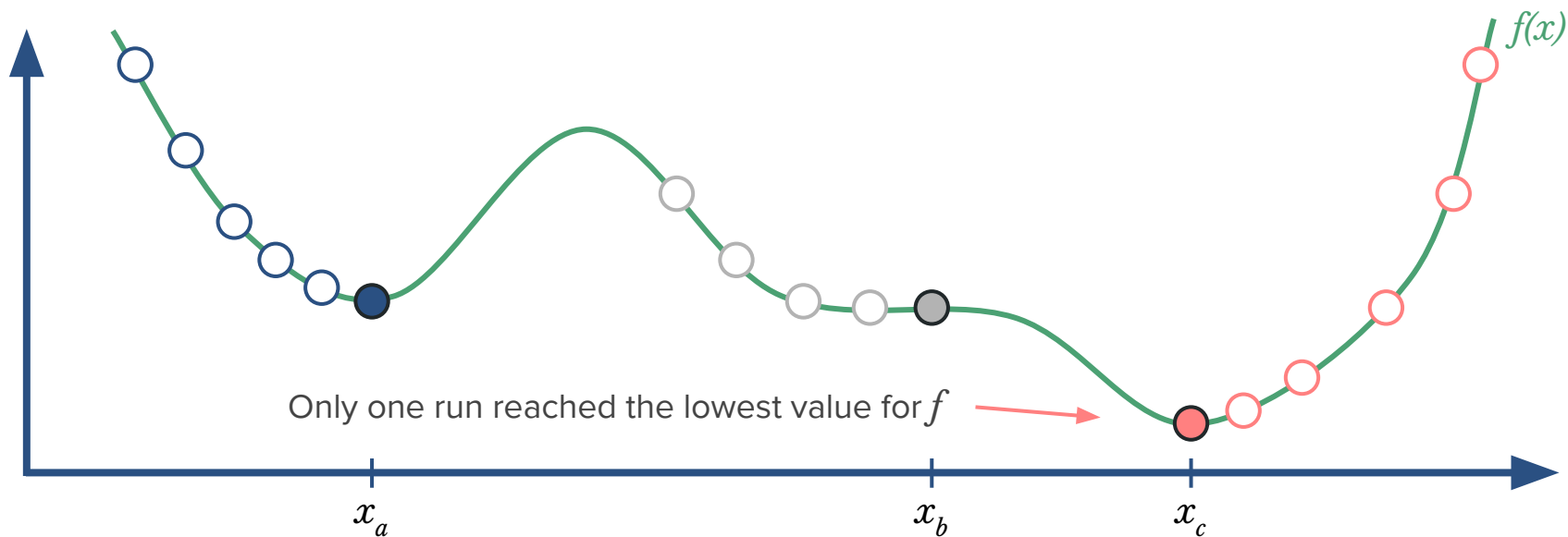
When Gradient descent is suboptimal

- 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**.



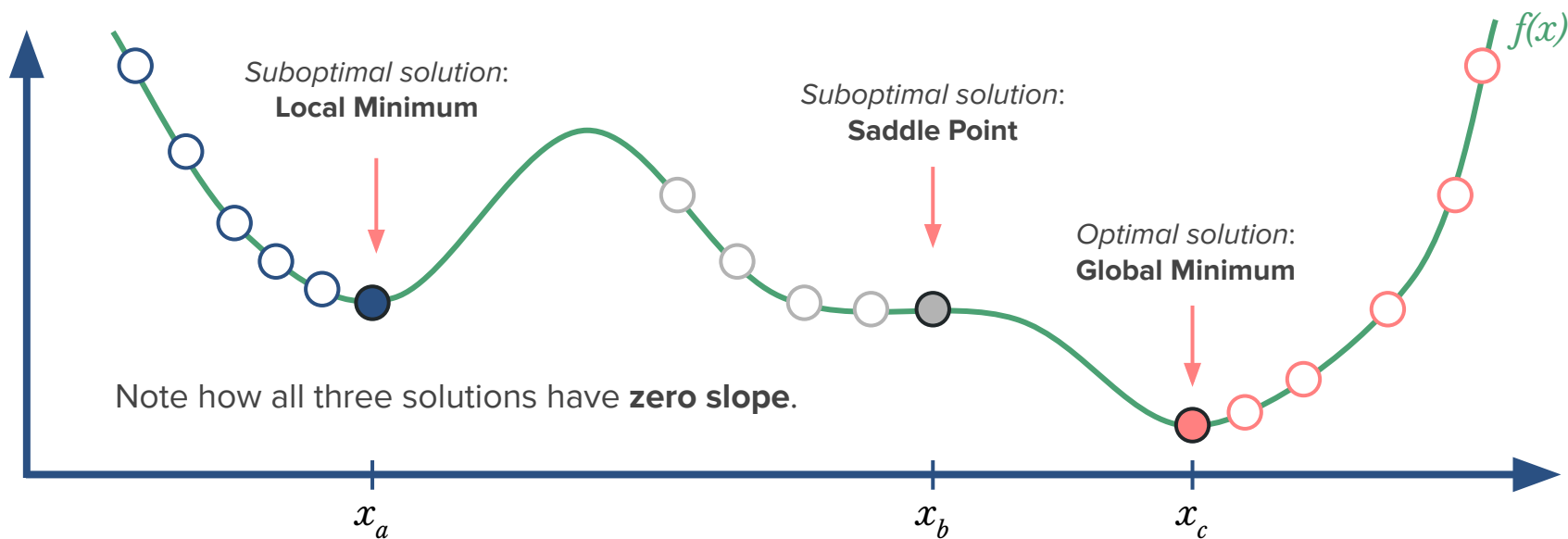
When Gradient descent is suboptimal

- 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**.



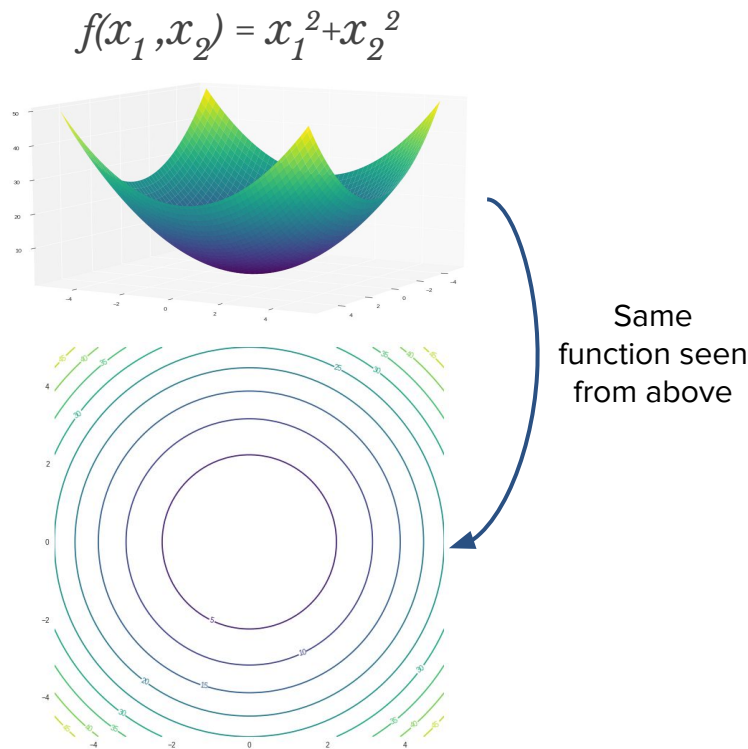
How about more dimensions?

- 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 $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ a **vector** corresponding to 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**, $-\nabla_x f(x)$.



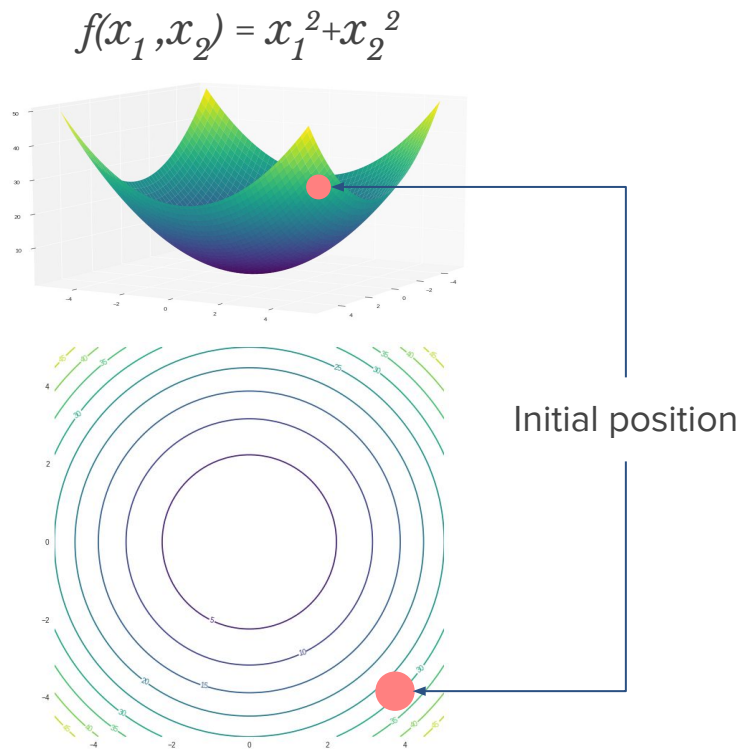
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ 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 $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ **a vector** corresponding to 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**, $-\nabla_x f(x)$.



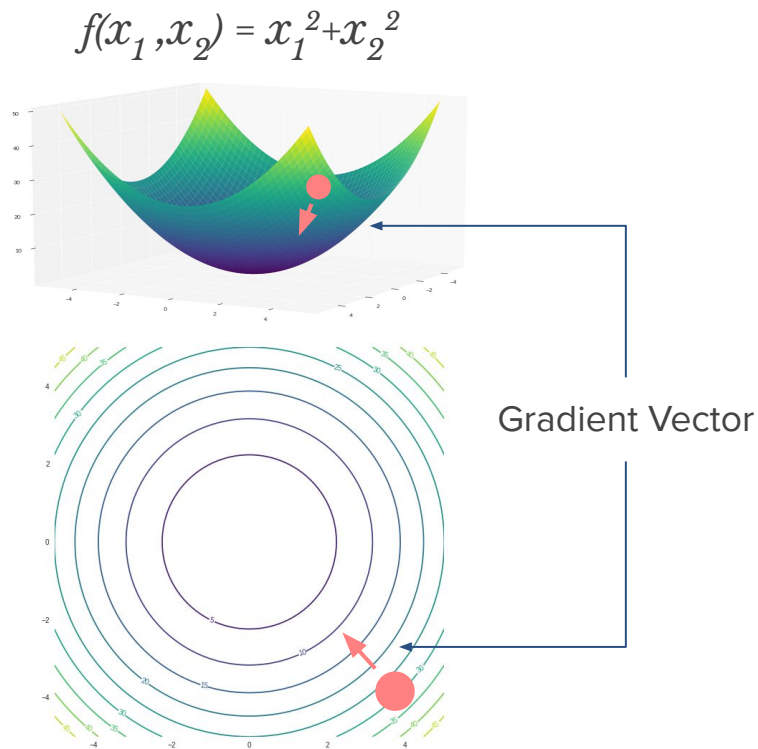
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ 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 $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ **a vector** corresponding to 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**, $-\nabla_x f(x)$.



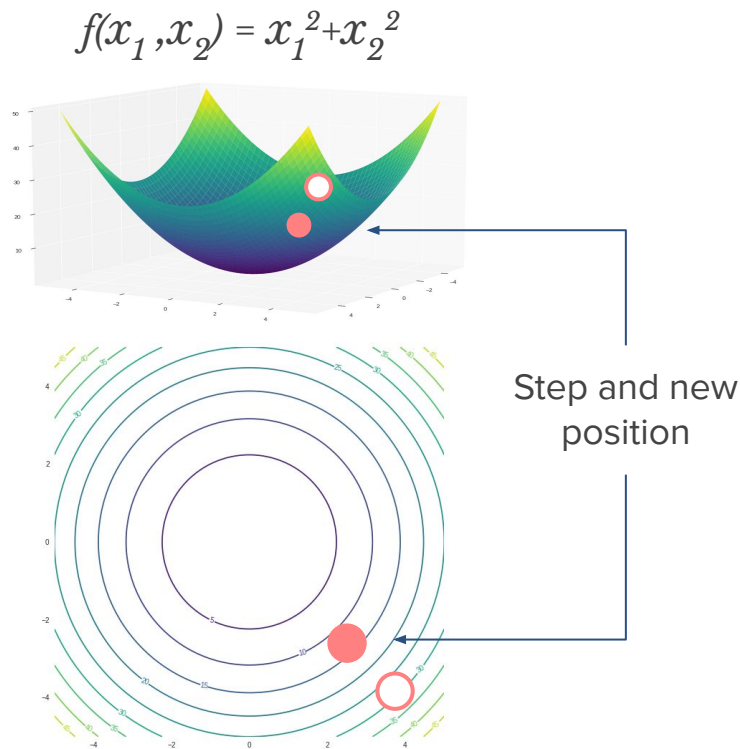
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ 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 $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ **a vector** corresponding to 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**, $-\nabla_x f(x)$.



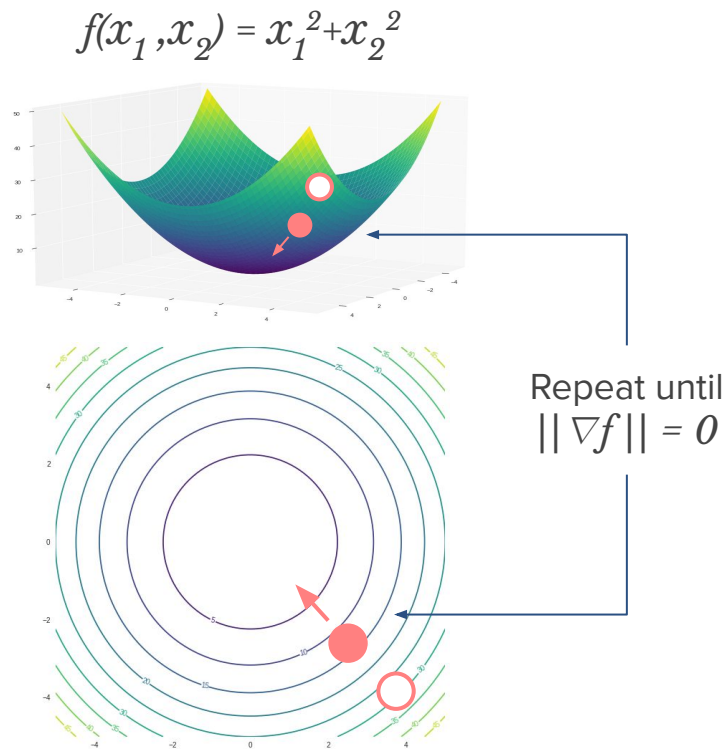
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ 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 $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ a **vector** corresponding to 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**, $-\nabla_x f(x)$.



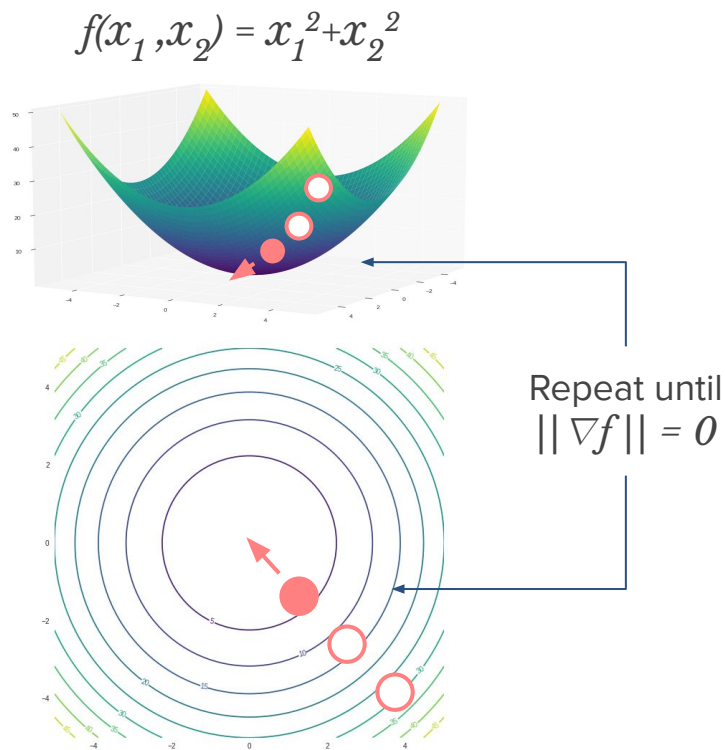
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ 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 $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ a **vector** corresponding to 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**, $-\nabla_x f(x)$.



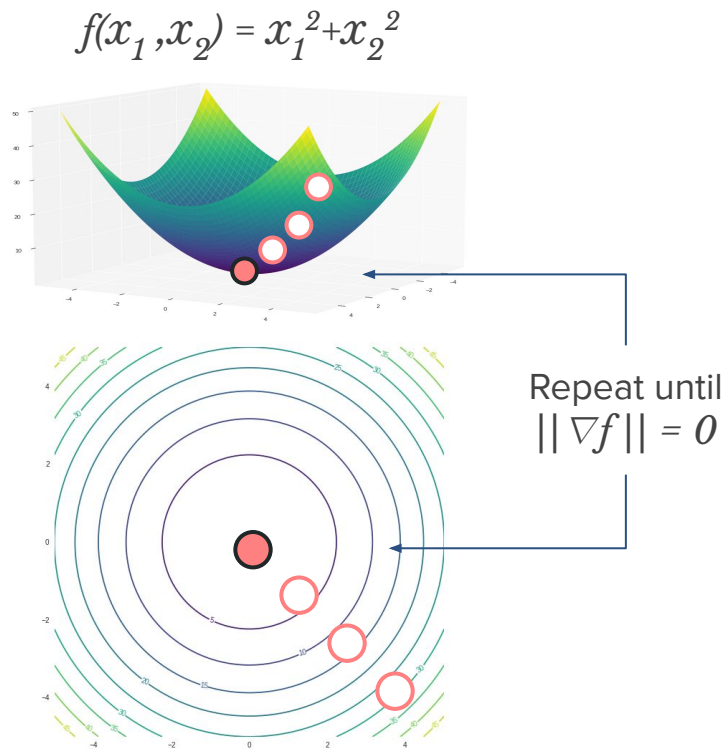
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ 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 $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ a **vector** corresponding to 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**, $-\nabla_x f(x)$.



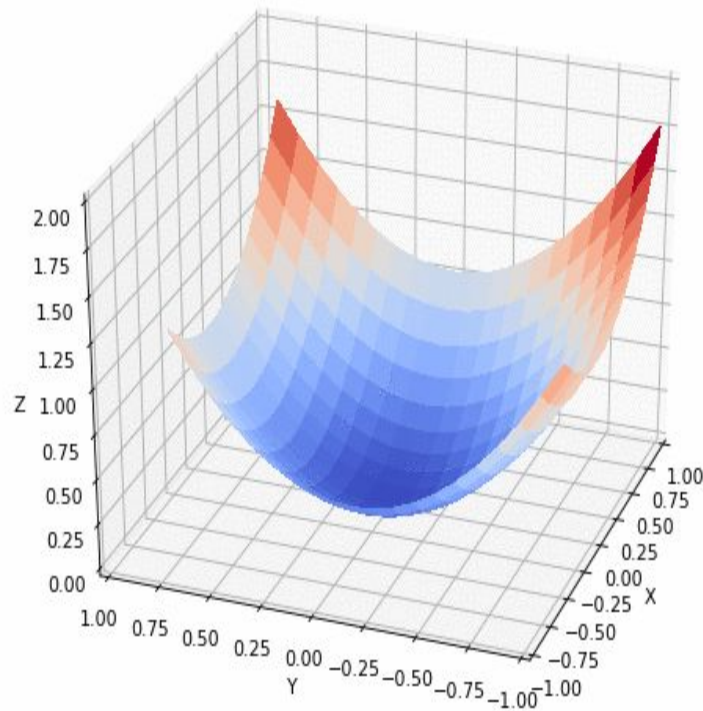
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ 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 $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ **a vector** corresponding to 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**, $-\nabla_x f(x)$.



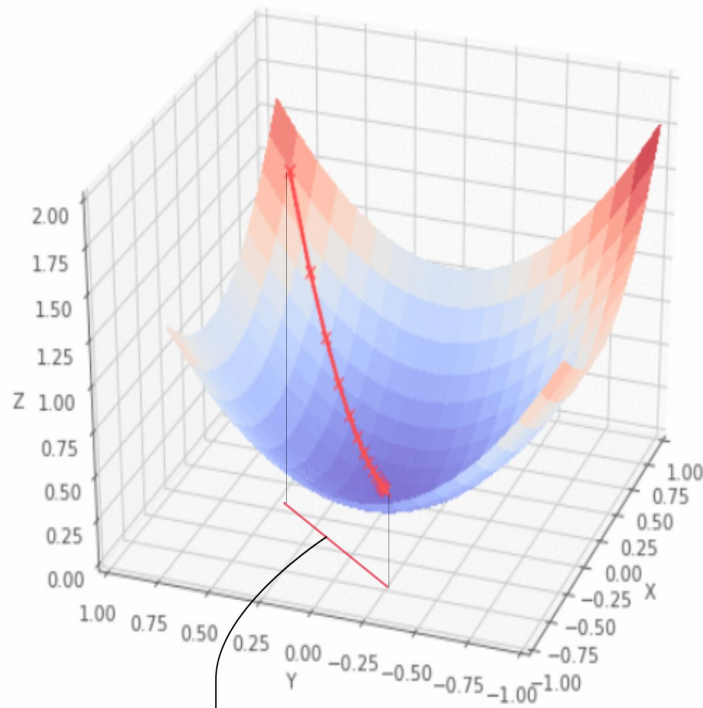
How about more dimensions?

- What we did for 1D can be readily generalized to functions $f(x_1, \dots, x_D)$ 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 $\nabla_x f = [df/dx_1, \dots, df/dx_D]^T$ **a vector** corresponding to the gradient of f .

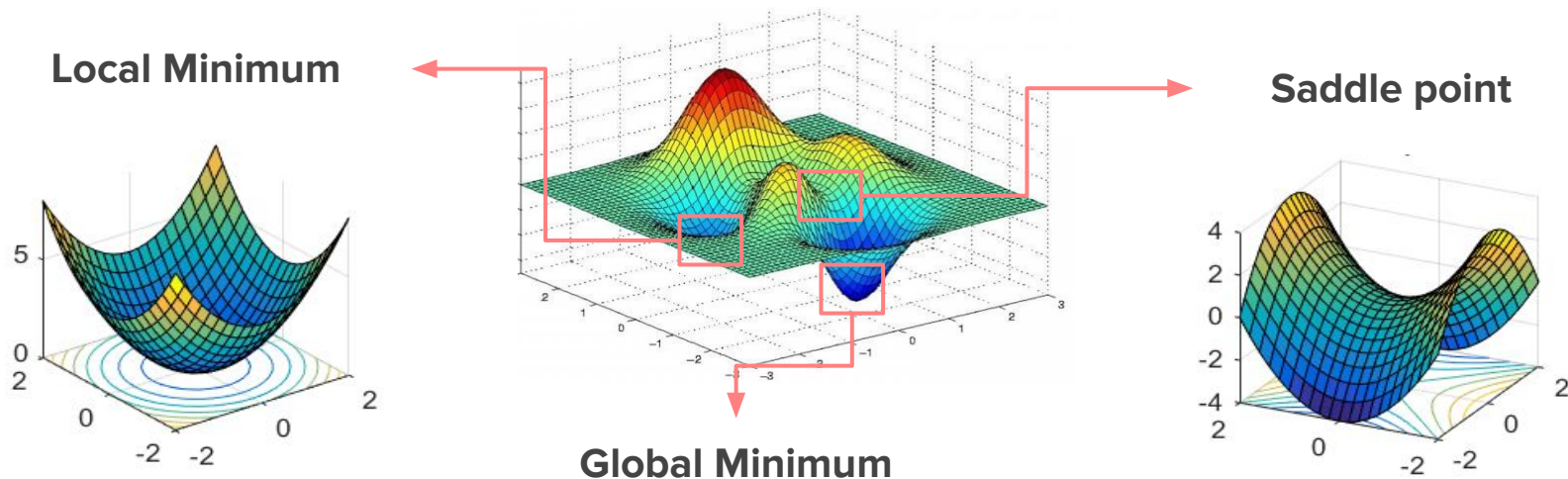
- The principle is the same as in 1D, but now we have a direction in D dimensions to follow: **the negative of the function's gradient**, $-\nabla_x f(x)$.



Trajectory of x

When Gradient Descent is suboptimal

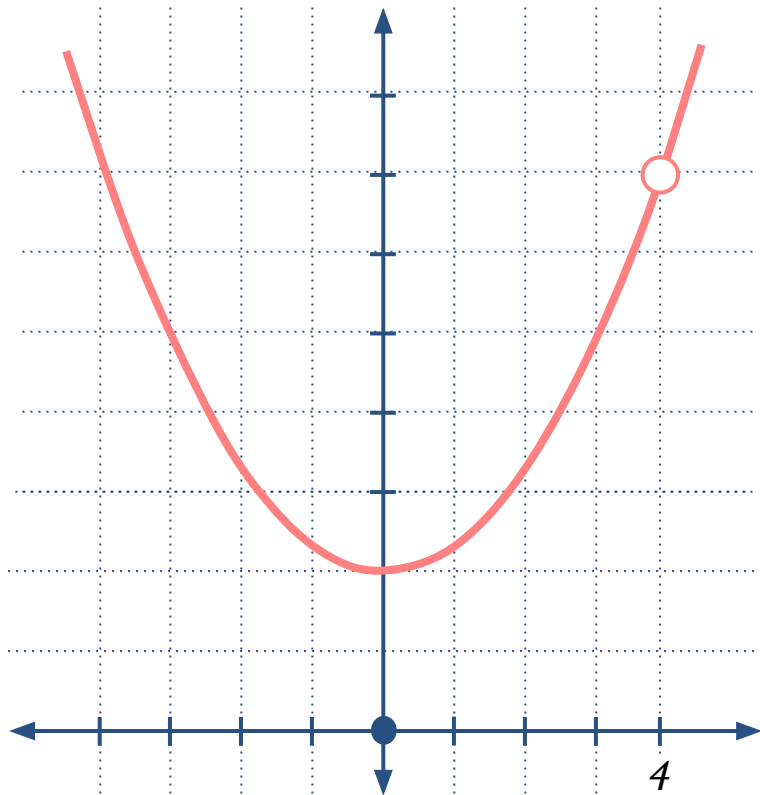
- Also, just like in 1D, gradient descent can get stuck in multidimensional suboptimal solutions and miss the global minimum:



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

Exercises (*In pairs*)

- Run 5 iterations (up to $t = 5$) of gradient descent on the function $f(x) = x^2 + 2$, whose derivative is $df(x)/dx = 2x$. Set the initial guess $x_0 = 4$ and use $\eta = 0.25$. Also compute the value of $f(x_t)$ as you go, marking it on the graph.
- 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)?



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, $n > 100000$!

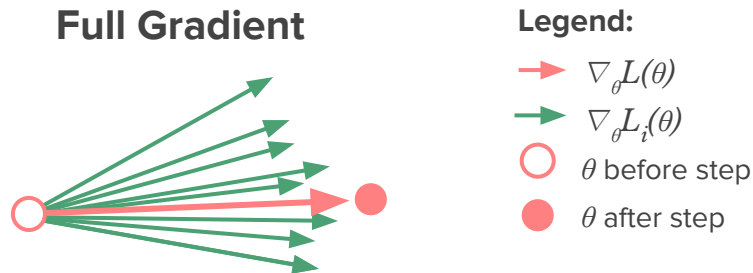
Going back to Neural Networks

- 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 for the loss on each data point:

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

- Note now that the gradient vector $\nabla_{\theta}L(\theta)$ is just an average of the gradient vectors on each data point, i.e. $\nabla_{\theta}L_i(\theta)$:

$$\nabla_{\theta}L(\theta) = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta}L_i(\theta)$$



Going back to Neural Networks

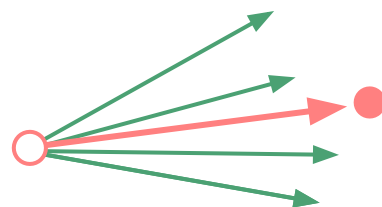
- Ok, let's summarize our problems with GD so far:
 1. GD is prone to local minima and saddle points,
 2. It is very computationally expensive to compute a step of GD in modern Neural Networks.
- We'll try to solve both problems with the same solution: **randomness!**
- First, to make things easier,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

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

- This also means that we can get the average of just a few $\nabla_{\theta} L_i(\theta)$ and the result won't be too far off from the full gradient $\nabla_{\theta} L(\theta)$.

$$\nabla_{\theta} L(\theta) = \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} L_i(\theta)$$

Approximate Gradient

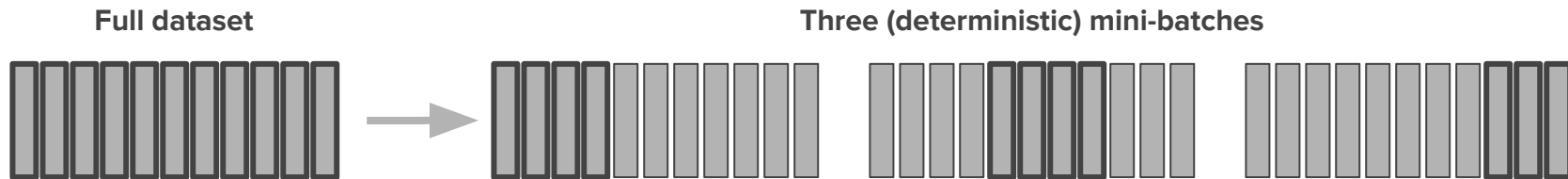


Legend:

- $\nabla_{\theta} L(\theta)$
- $\nabla_{\theta} L_i(\theta)$
- θ before step
- θ after step

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 lingo:
 - 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.



* 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 lingo:
 - 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.



* 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 lingo:
 - 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.

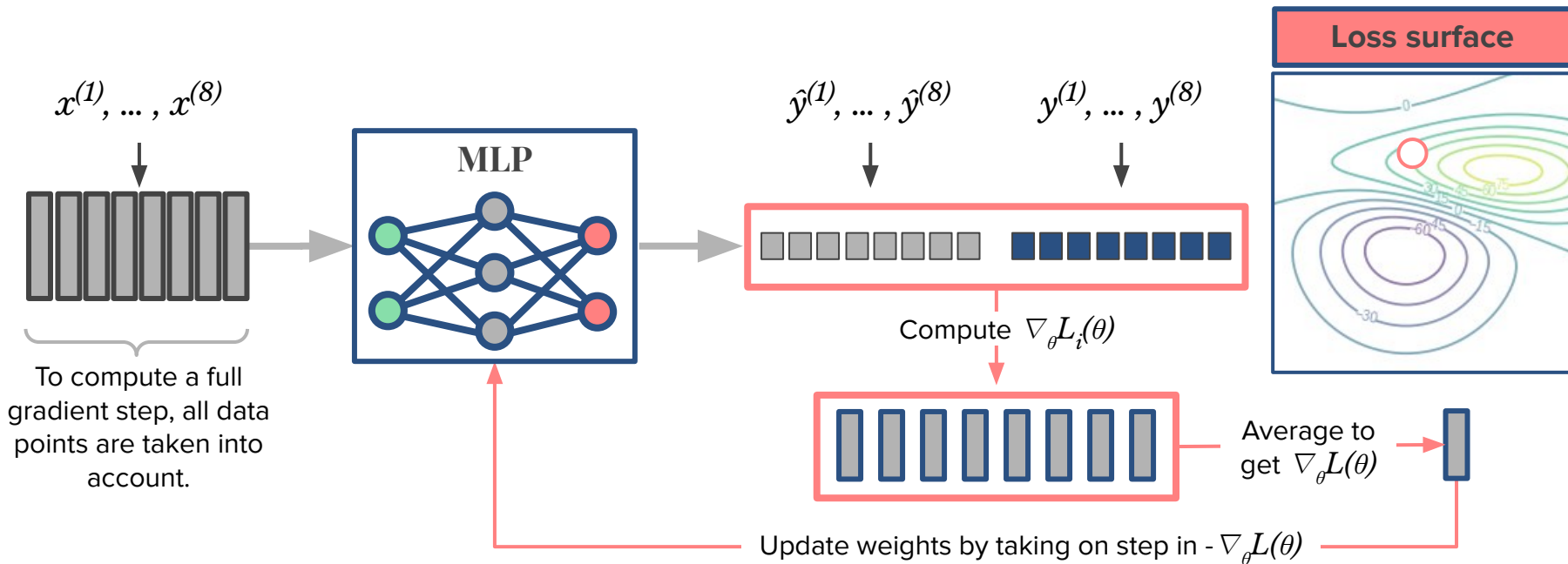


- SGD will go over each batch and then restart. An **epoch** is over when it has finished going over all batches (and therefore all data points) once.

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

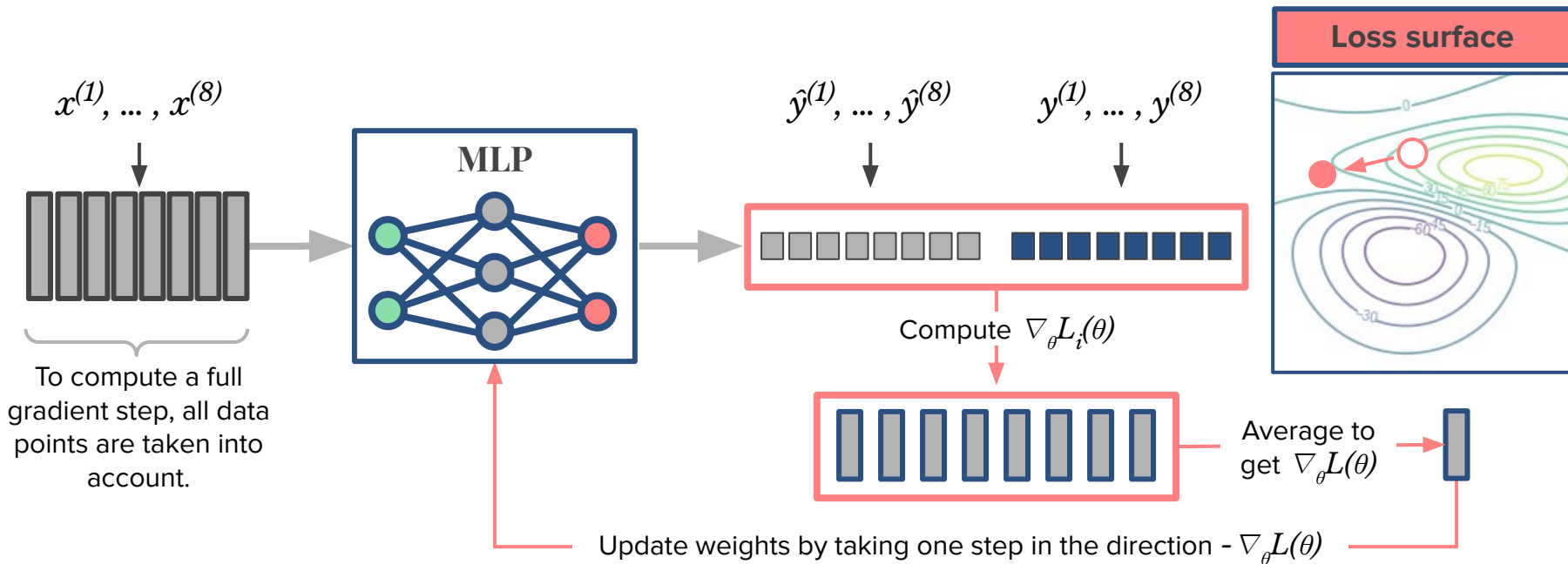
GD vs SGD

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



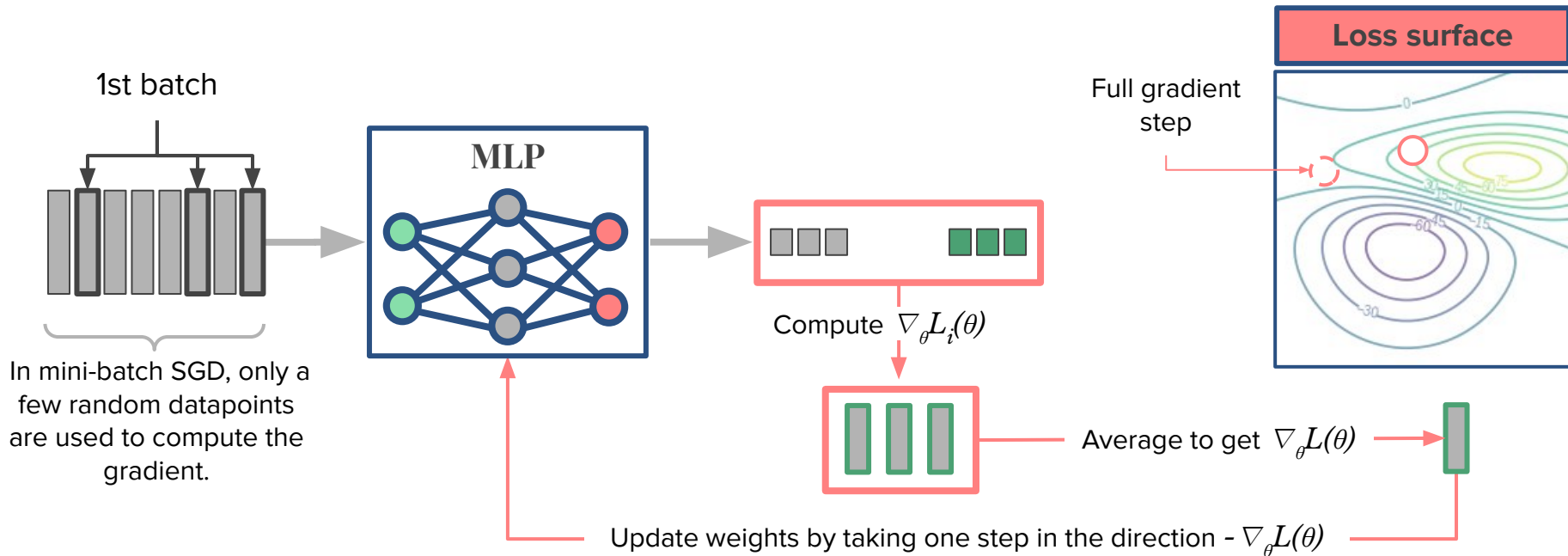
GD vs SGD

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



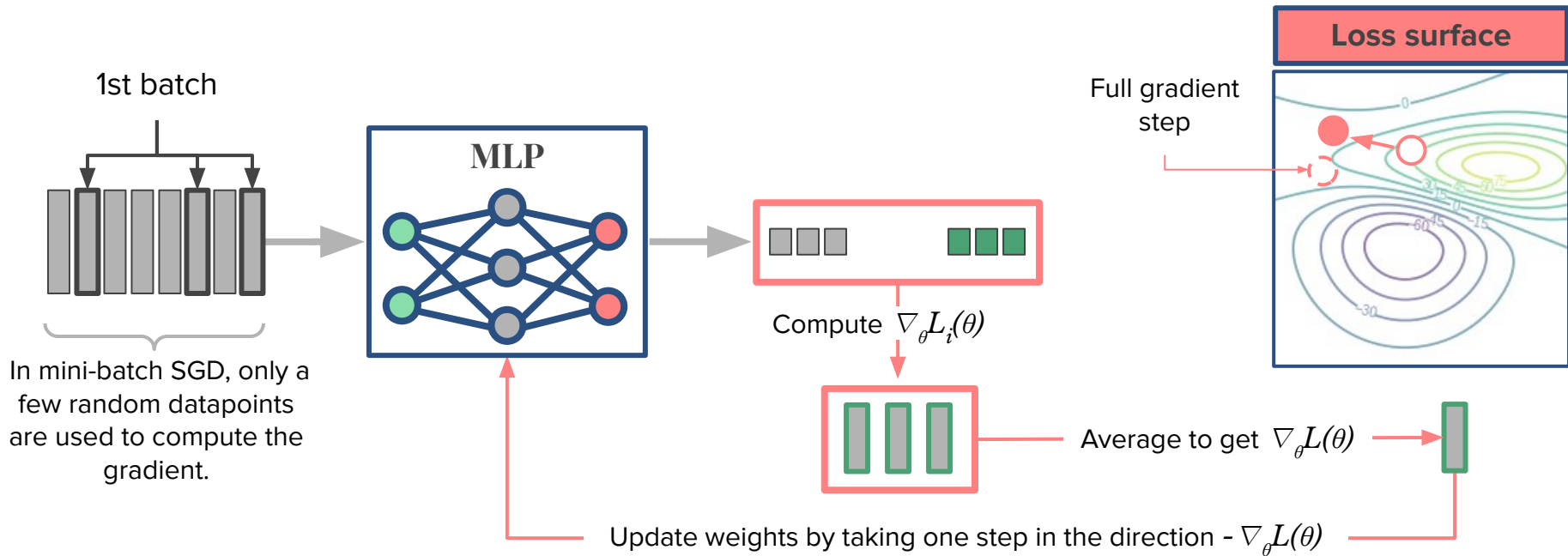
GD vs SGD

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



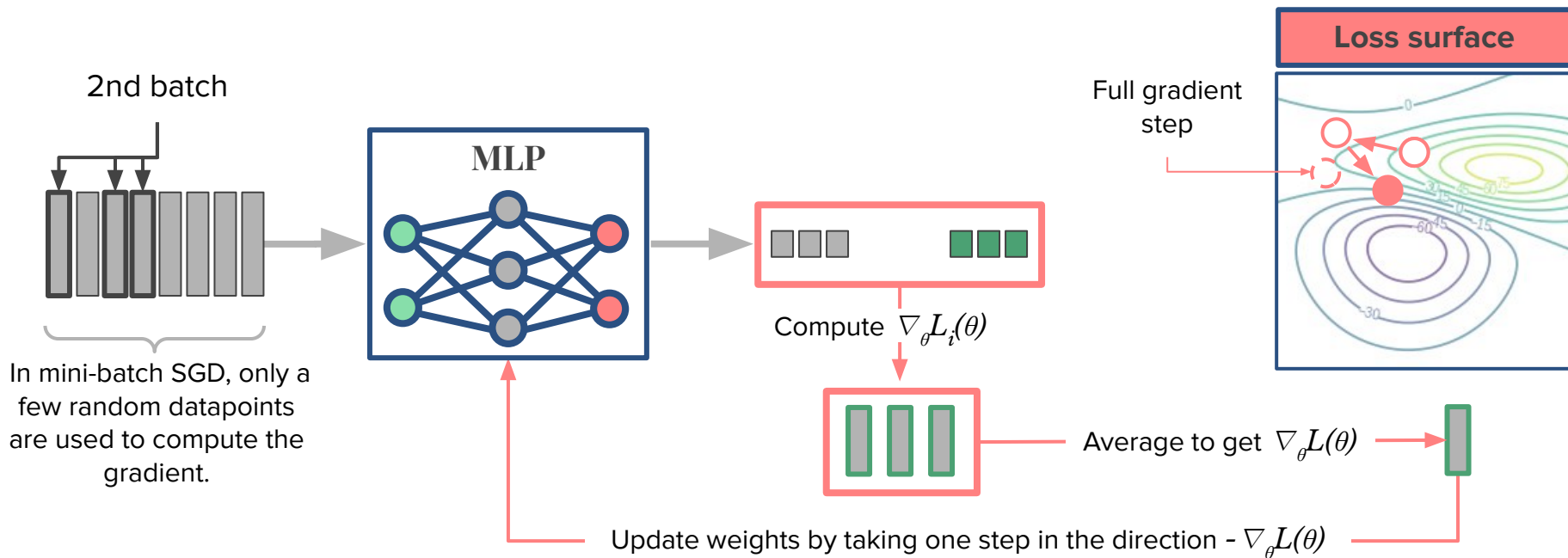
GD vs SGD

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



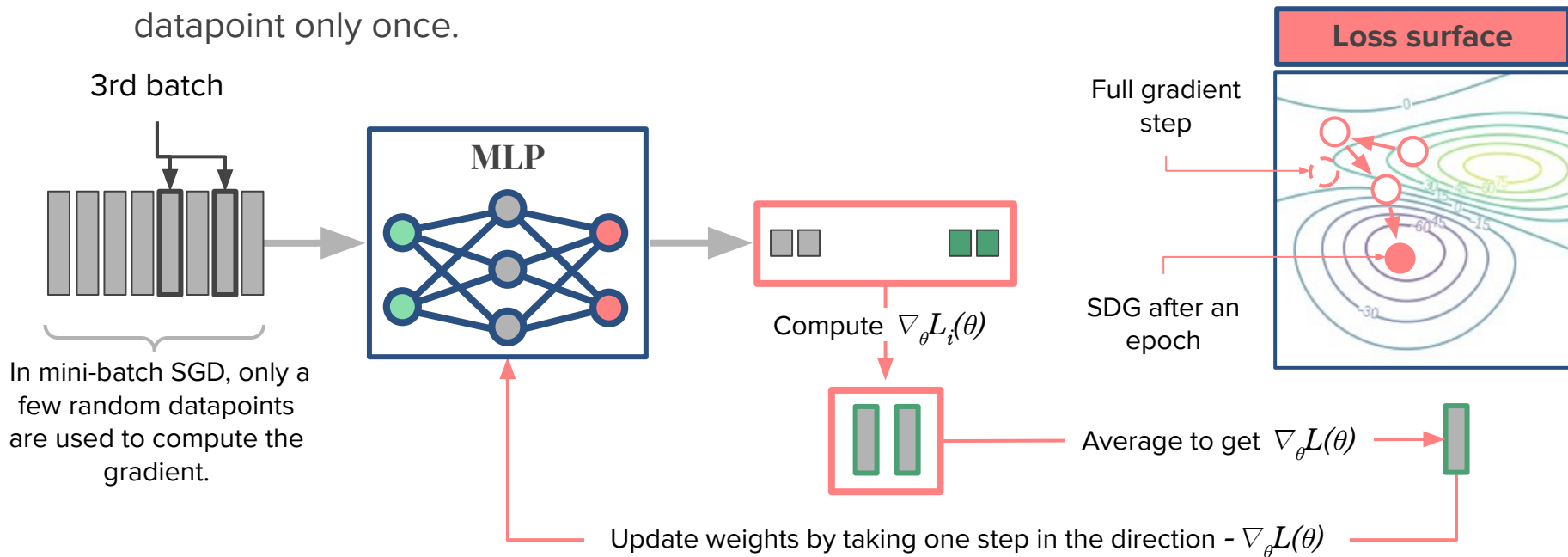
GD vs SGD

- 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 [paper](#) seems to conveniently answer this question:

[Submitted on 13 Feb 2019 (this version), latest version 4 Sep 2019 (v2)]

Stochastic Gradient Descent Escapes Saddle Points Efficiently

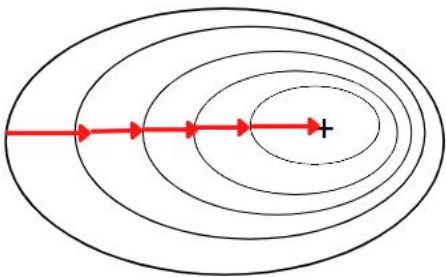
Chi Jin, Praneeth Netrapalli, Rong Ge, Sham M. Kakade, Michael I. Jordan

- 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.

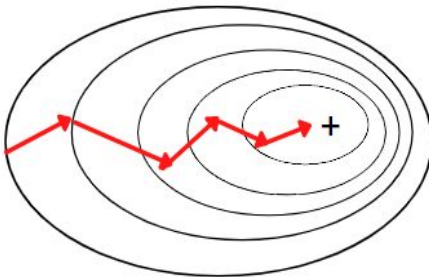
Analysing SGD

- 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.

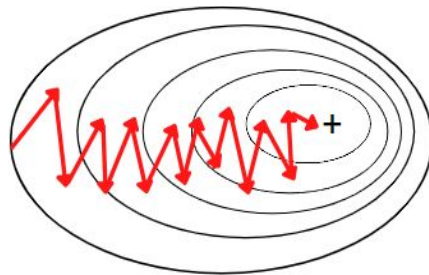
GD



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**.

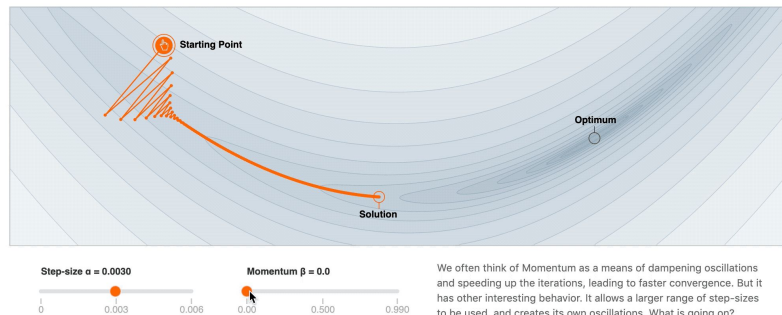
Adding Momentum

- 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).

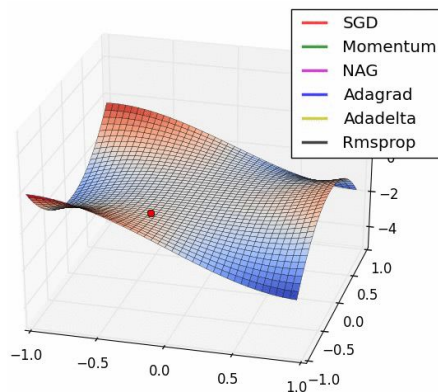
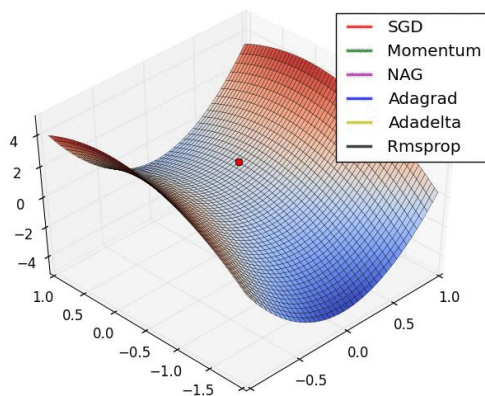
- In practice, adding some momentum makes SGD's path more stable/smooth*, leading to quicker convergences.
- However, adding too much momentum can also hurt convergence*.



* Check out this [website](#) and try adding momentum to GD yourself.

Modern Optimizers

- The literature offers many possible **optimizers** to find the best neural network weights.
- All of them employ either **Stochasticity** or **Momentum** or **both** (beside a few more tricks to set the best 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, Adadelata 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.

Exercise (*Homework*)

- Say you computed at some point you computed $\nabla f(x) = [1, 2]$ and your learning rate η is equal to 0.5 , where does the update direction? If you were at point $x = [4, 3]$, where will you end up after that step of gradient descent?
- Suppose our dataset with 5 points we have the following gradients:

$$\nabla_{\theta} L_1(\theta) = [7, 2], \quad \nabla_{\theta} L_2(\theta) = [3, 1], \quad \nabla_{\theta} L_3(\theta) = [2, 3], \quad \nabla_{\theta} L_4(\theta) = [4, 2], \quad \nabla_{\theta} L_5(\theta) = [3, 3],$$

Compute the full gradient and the mini-batch gradient when your batch consists of datapoint points indexed at 2, 4, 5.

- Your dataset has $n = 1200$ points and the batch size is $B = 50$. How many SGD updates per epoch? If batch size changes to $B = 200$, how many updates now? What if $B = 32$?