INFN Machine Learning course

Prof. Amir Farbin, Prof. Daniele Bonacorsi

20-22 May 2019

Camogli, Italy

Training models

Normal Equation vs Gradient Descent

Training a ML model → usually just few lines of code. Let's understand a bit "what's under the hood"

Take e.g. a simple, Linear Regression model. You can train in 2 ways:

- using the **Normal Equation** that directly computes the model parameters that minimize the cost function over the training set
- using an iterative optimization approach, called **Gradient Descent** (**GD**), that gradually tweaks the model parameters to minimise the cost function over the training set, eventually converging to the same set of parameters as the first method

A few variants of GD exist:

- **Batch GD**
- **Stochastic GD**
- **Mini-batch GD**

Towards an analytical solution

m examples $(\mathbf{x}^{(1)}, y^{(1)}), \dots (\mathbf{x}^{(m)}, y^{(m)})$ ← vectors

n features $x_1, x_2, \dots x_n$

$$x^{(i)} = \begin{pmatrix} x_0^{(i)} \\ x_1^{(i)} \\ \vdots \\ x_n^{i} \end{pmatrix} \in \mathbb{R}^{n+1} \qquad x_0^{(i)} = 1$$

$$y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}$$

m-dim vector

example: 1 feature only

$$x^{(i)} = \begin{pmatrix} x_0^{(i)} \\ x_1^{(i)} \end{pmatrix} = \begin{pmatrix} 1 \\ x_1^{(i)} \end{pmatrix}$$

$$X = \begin{pmatrix} 1 & x_1^{(1)} \\ 1 & x_1^{(2)} \\ & \vdots \\ 1 & x_1^{(m)} \end{pmatrix}$$

m x 2 matrix

$$X = \begin{pmatrix} \text{---} (x^{(1)})^T \text{---} \\ \text{---} (x^{(2)})^T \text{---} \\ \vdots \\ \text{---} (x^{(m)})^T \text{---} \end{pmatrix} = \begin{pmatrix} x_0^{(1)} & x_1^{(1)} & \dots & x_n^{(1)} \\ x_0^{(2)} & x_1^{(2)} & \dots & x_n^{(2)} \\ & & \vdots & \\ x_0^{(m)} & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix}$$

"DESIGN MATRIX"

m x (n+1) matrix

Towards an analytical solution

$$X = \begin{pmatrix} x_0^{(1)} & x_1^{(1)} & \cdots & x_n^{(1)} \\ x_0^{(2)} & x_1^{(2)} & \cdots & x_n^{(2)} \\ & & \vdots & \\ x_0^{(m)} & x_1^{(m)} & \cdots & x_n^{(m)} \end{pmatrix}$$

$$\theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix} \in \mathbb{R}^{n+1}$$

$$X\theta = \begin{pmatrix} x_0^{(1)} & x_1^{(1)} & \cdots & x_n^{(1)} \\ x_0^{(2)} & x_1^{(2)} & \cdots & x_n^{(2)} \\ & & \vdots & \\ x_0^{(m)} & x_1^{(m)} & \cdots & x_n^{(m)} \end{pmatrix} \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{pmatrix} = \begin{pmatrix} \theta_0 x_0^{(1)} + \theta_1 x_1^{(1)} + \ldots + \theta_n x_n^{(1)} \\ \theta_0 x_0^{(2)} + \theta_1 x_1^{(2)} + \ldots + \theta_n x_n^{(2)} \\ \vdots \\ \theta_0 x_0^{(m)} + \theta_1 x_1^{(m)} + \ldots + \theta_n x_n^{(m)} \end{pmatrix}$$

$m \times (n+1)$ — $(n+1)$-dim vector — m-dim vector

Towards an analytical solution

Start from:

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \dots + \theta_n x_n$$

$$J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

and rewrite:

$$h_\theta(x) = \theta^T x$$

$$J(\theta) = \frac{1}{2m} (X\theta - y)^T (X\theta - y)$$

(drop $\frac{1}{2m}$ as we are computing a derivative to θ anyway)

$$(X\theta - y)^T (X\theta - y) = ((X\theta)^T - y^T)(X\theta - y) =$$

$$= (X\theta)^T (X\theta) - (X\theta)^T y - y^T (X\theta) + y^T y$$

$$= \theta^T X^T X \theta - 2 (X\theta)^T y + y^T y$$

(note: $X\theta$ and y are both m-dim vectors)

Normal Equation

$$\hat{\boldsymbol{\theta}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1} \mathbf{X}^T \mathbf{y}$$

Let's discuss its computation complexity:

- it computes the inverse of a (n+1)×(n+1) matrix (n = # features), whose computational complexity is typically about **$O(n^{2.4})$** to **$O(n^3)$** (depending on the implementation)
  - if you double n, computation time grows a factor 5.3 to 8
- sklearn LinearRegression with Singular Value Decomposition (SVD) is about **$O(n^2)$**
- both are linear with the # instances → **O(m)**. So, they can handle large training sets efficiently (provided they can fit in memory)

Gradient Descent (GD)

Gradient Descent (**GD**) is an iterative algorithm capable of finding optimal solutions by measuring the local gradient of the error function with regards to the parameter vector θ, and it goes in the direction of descending gradient.

An important parameter in GD is the size of the steps, determined by the **learning rate** hyperparameter

Formulating the training problem differently

You have some function $J(\theta_0, \theta_1)$

you can generalize to $J(\theta_0, \theta_1, \dots, \theta_n)$

You want $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

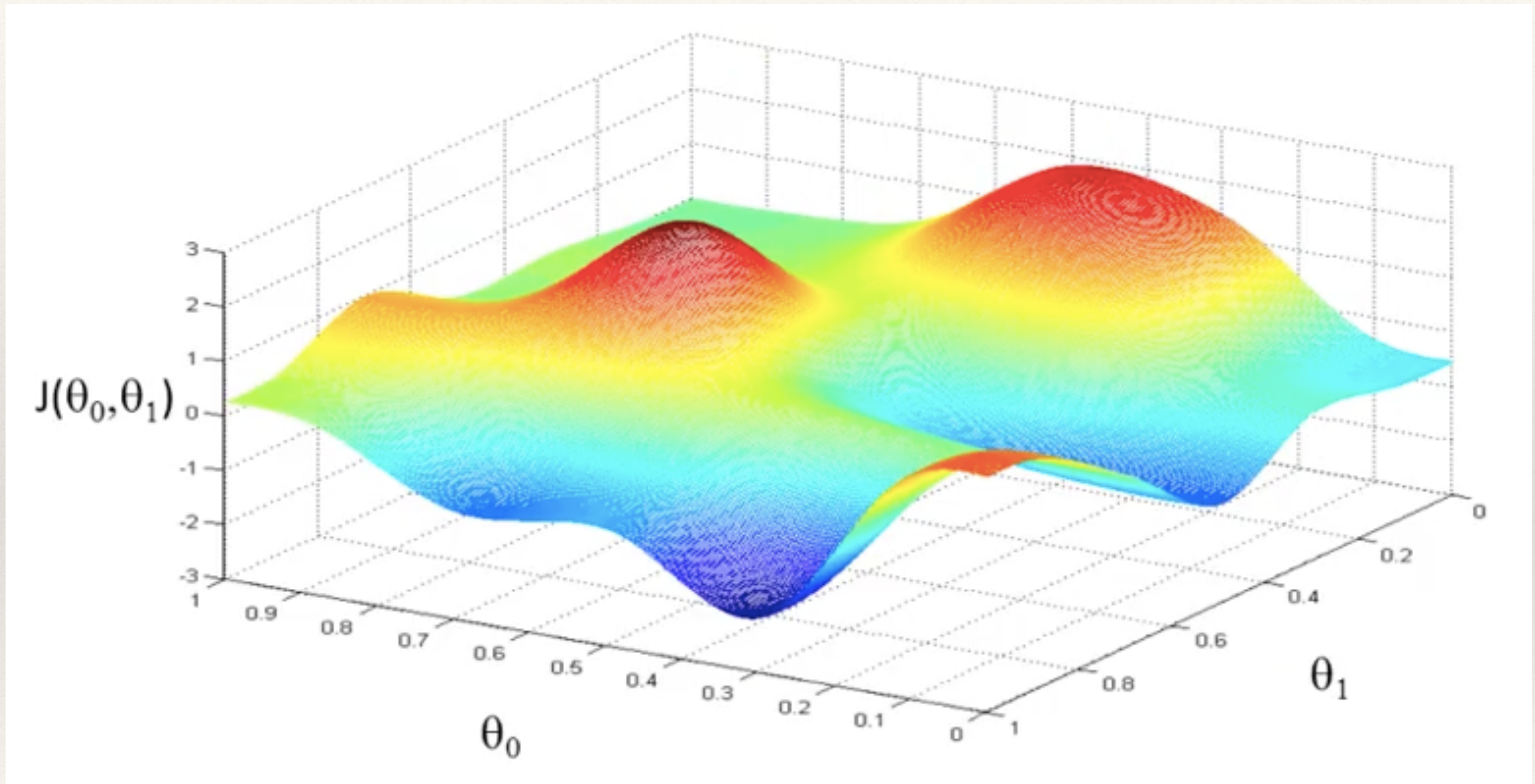
$\min_{\theta_0, \theta_1, \dots \theta_n} J(\theta_0, \theta_1, \dots, \theta_n)$

Algo:

- start with some θ_0, θ_1
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$
- stop when you (hopefully!) reach a min

Gradient Descent (GD) algorithm

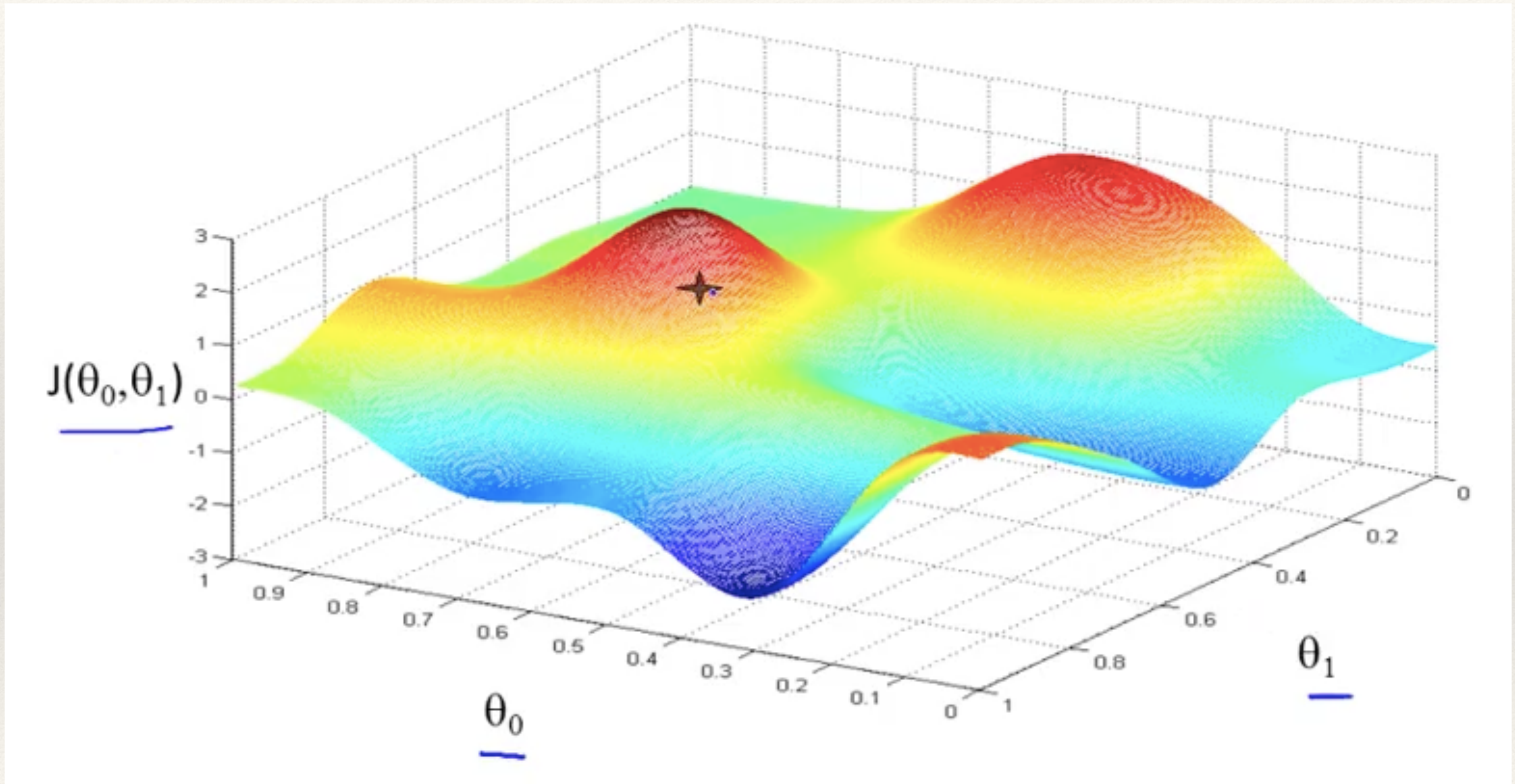
Note the axes.



(with 2 θs you can at least plot it 3D - but you can think and generalise to n θs..)

Gradient Descent (GD) algorithm

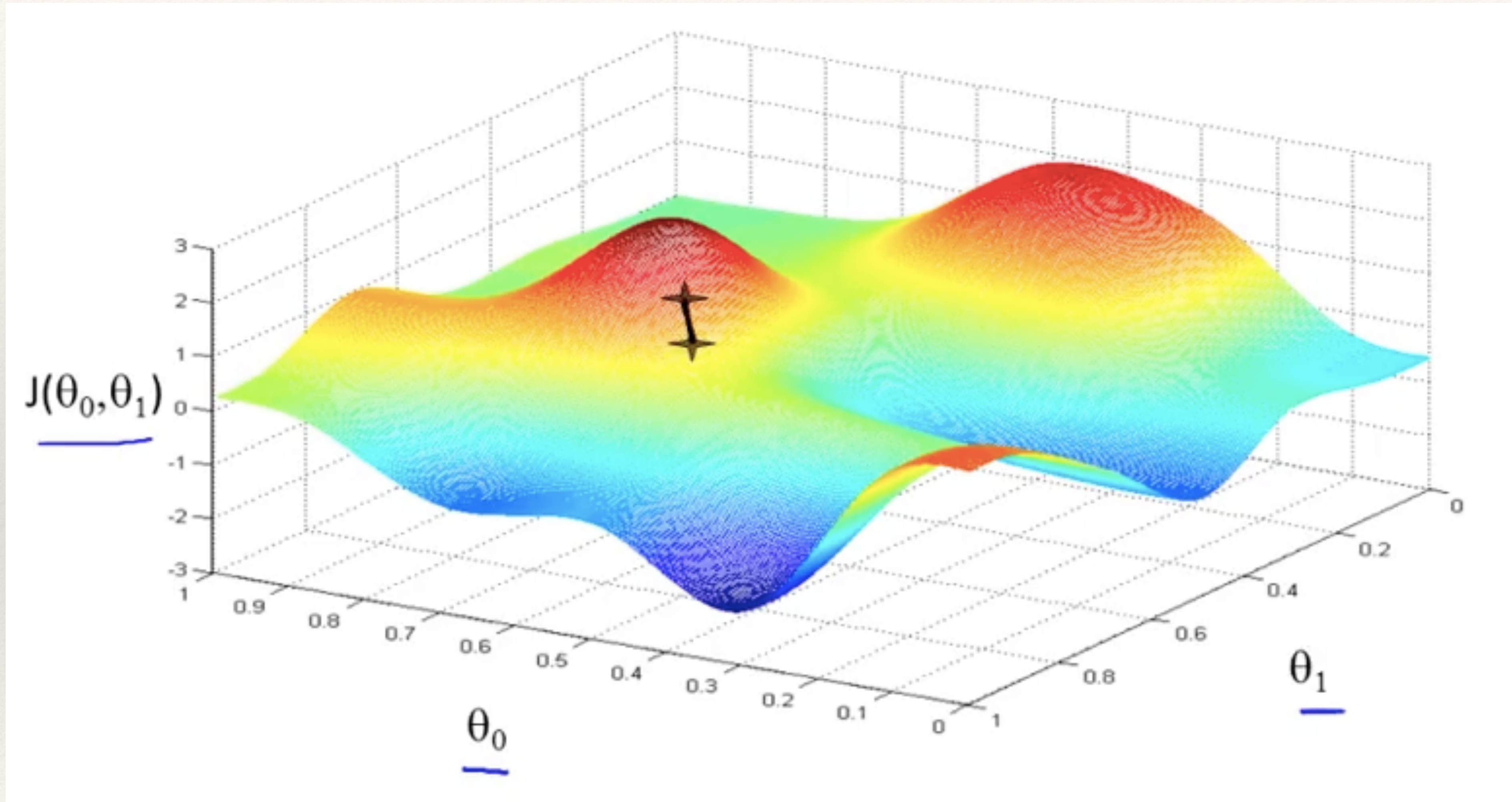
Pick a starting point, i.e. a given (θ_0, θ_1) pair



From the starting point, think physically as if these were hills: look around 360 degrees and make a step in the direction where I am going down quicker.

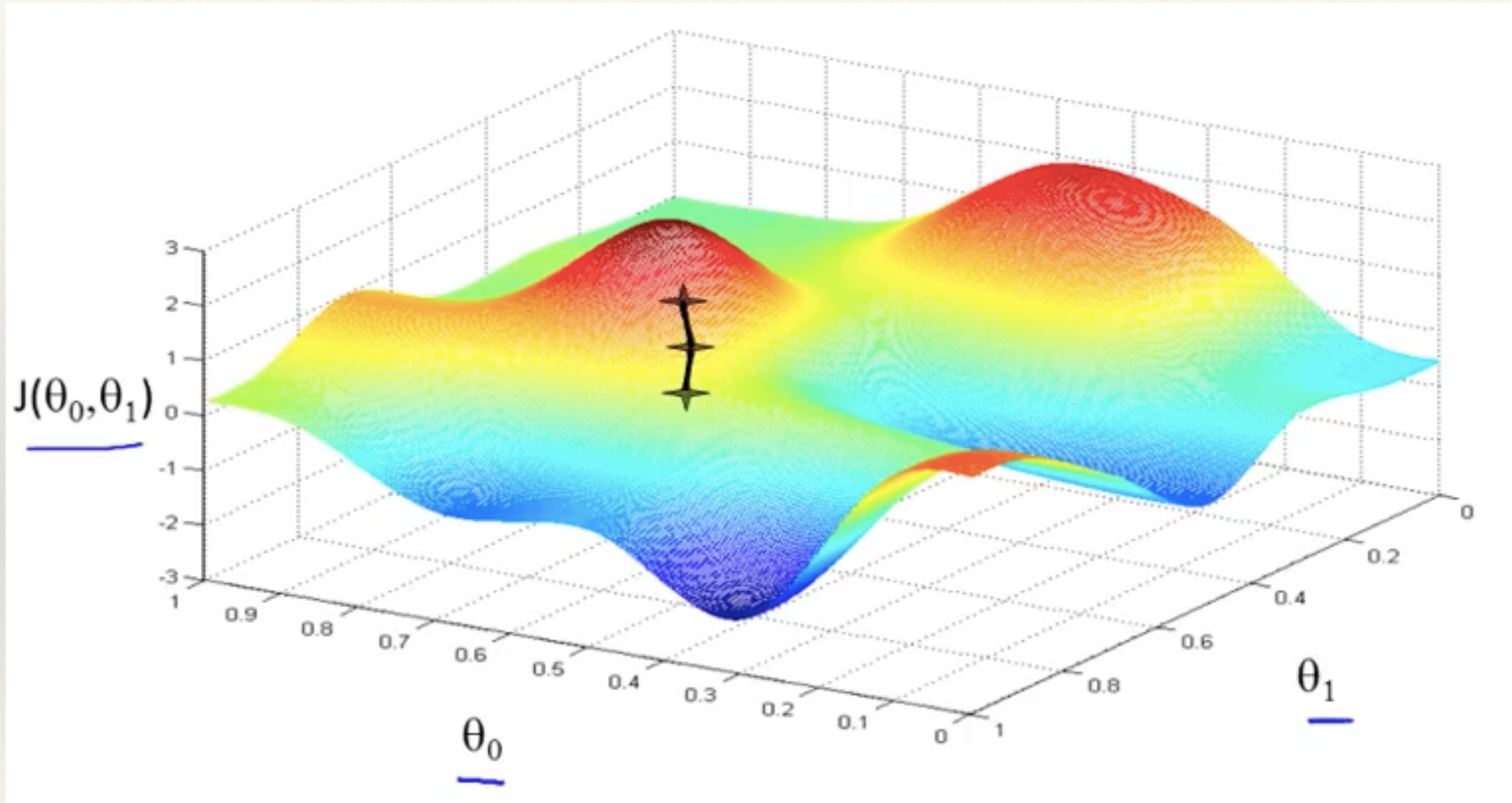
Gradient Descent (GD) algorithm

This is roughly the direction I should take.

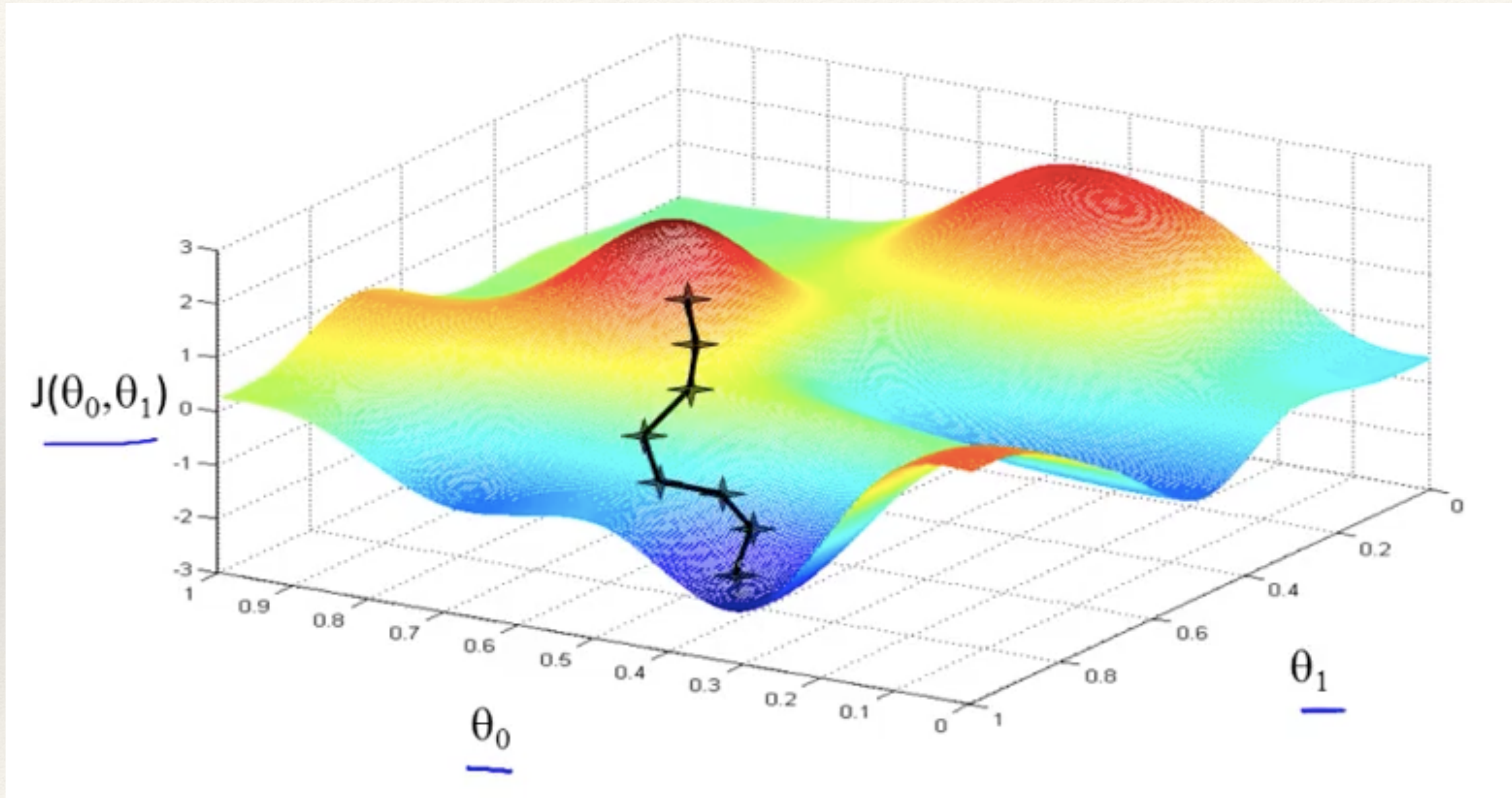


And now?

Gradient Descent (GD) algorithm



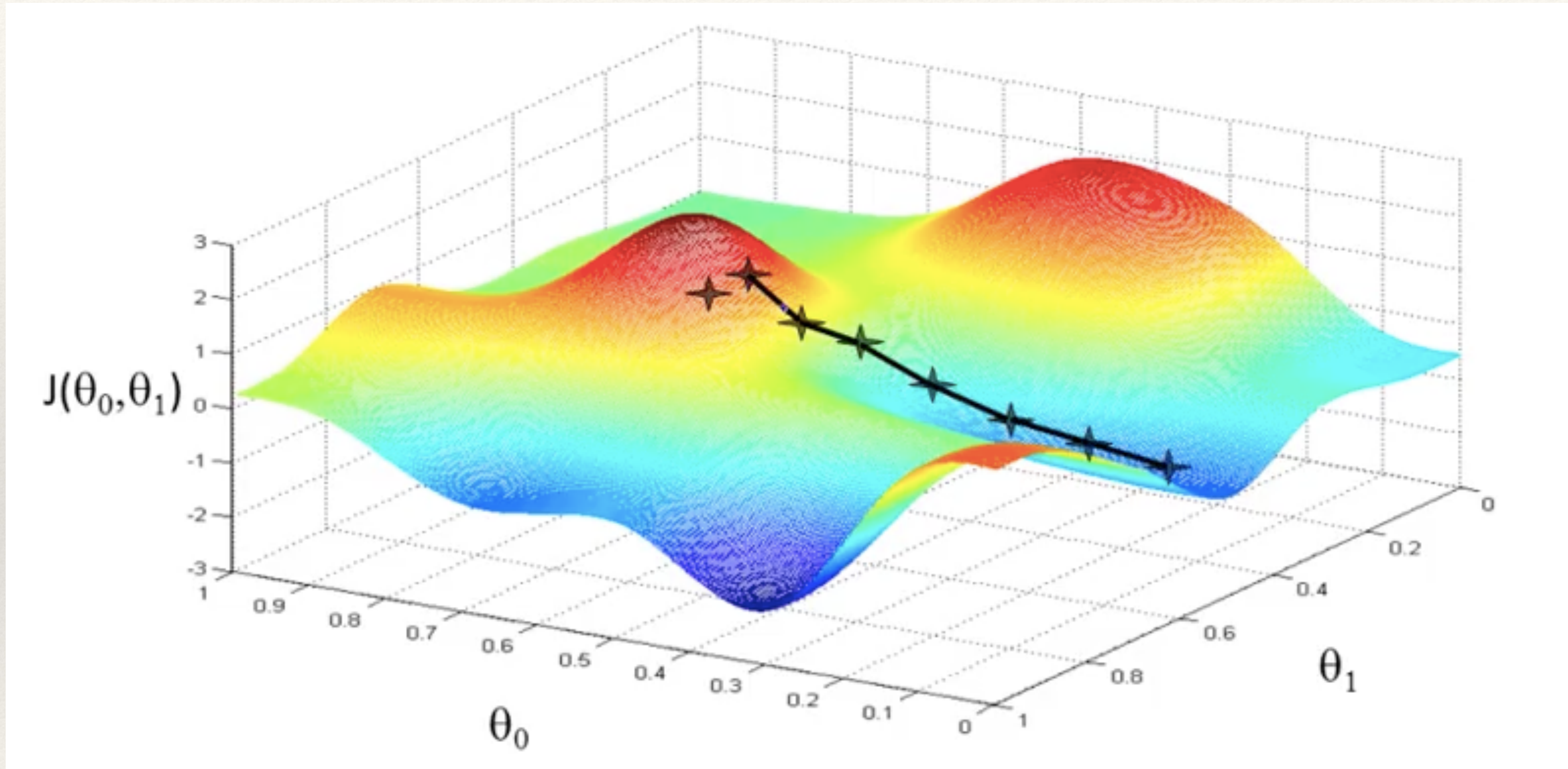
Gradient Descent (GD) algorithm



NOTE: this visual representation is easy to grasp but a bit misleading_ it is actually an hyperplane that intersect a hypersurface.. and projection on the thetas hyperplane gives you GD progression

Gradient Descent (GD) algorithm

If I had started just a couple of steps to the right…



… GD would have taken you to a **different local minimum**. This is a property of GD.

Definition of GD algo

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

}

simultaneous update
$j = 0$ and $j = 1$

feature index number

LEARNING RATE
it controls how aggressive the GD is

This is a good way to visualise the different between parameters and hyperparameters.

Implementation of GD algo

Let's look and digest all its parts.

- Firstly, let's look at the **θs**

Implementation of GD algo: θs

simultaneously = ?

$$
\begin{cases}
tmp\emptyset := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\
\theta_0 := tmp\emptyset \\
tmp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \\
\theta_1 := tmp1
\end{cases}
$$

QUIZ: is this correct?

Implementation of GD algo: θs

simultaneously = ?

$$\begin{cases} tmp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\ \theta_0 := tmp0 \\ tmp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \\ \theta_1 := tmp1 \end{cases}$$

incorrect ↯

Implementation of GD algo: θs

simultaneously = ?

$$\begin{cases} tmp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\ \theta_0 := tmp0 \\ tmp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \\ \theta_1 := tmp1 \end{cases}$$

incorrect

$$\begin{cases} tmp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\ tmp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \\ \theta_0 := tmp0 \\ \theta_1 := tmp1 \end{cases}$$

OK !

simultaneous update of θ_0, θ_1

NOTE: **simultaneously** update θ_0 and θ_1.

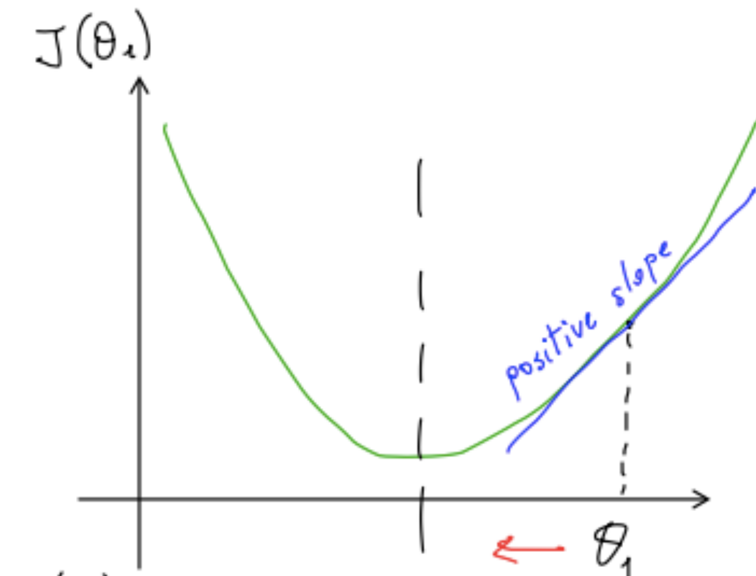
Be careful about a correct implementation of GD!

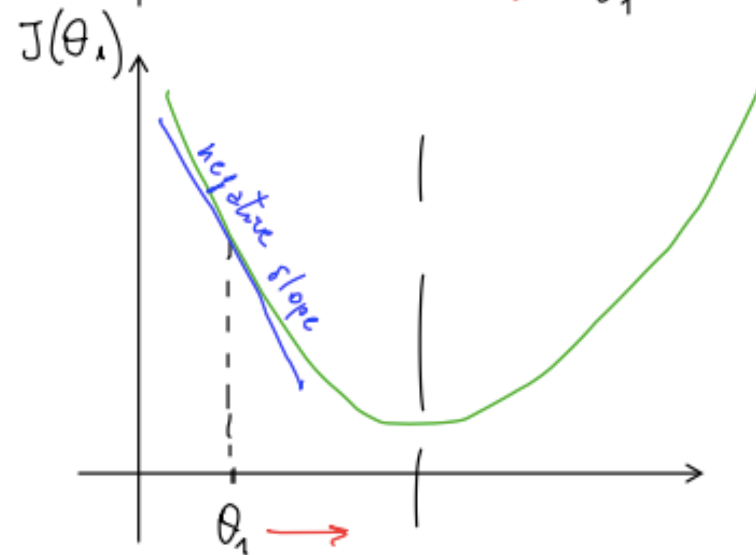
Implementation of GD algo

Let's look and digest all its parts.

- Firstly, let's look at the **θs**
- Secondly, let's look at the **derivative** term

Implementation of GD algo: **derivative**





$> 0 \Rightarrow \theta_1 := \theta_1 - \alpha \square$ ($\square > 0$): $\leftarrow \theta_1$

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

$< 0 \Rightarrow \theta_1 := \theta_1 - \alpha \square$ ($\square < 0$): $\theta_1 \rightarrow$

Of course: $= 0 \Rightarrow \theta_1$ already at min

Implementation of GD algo

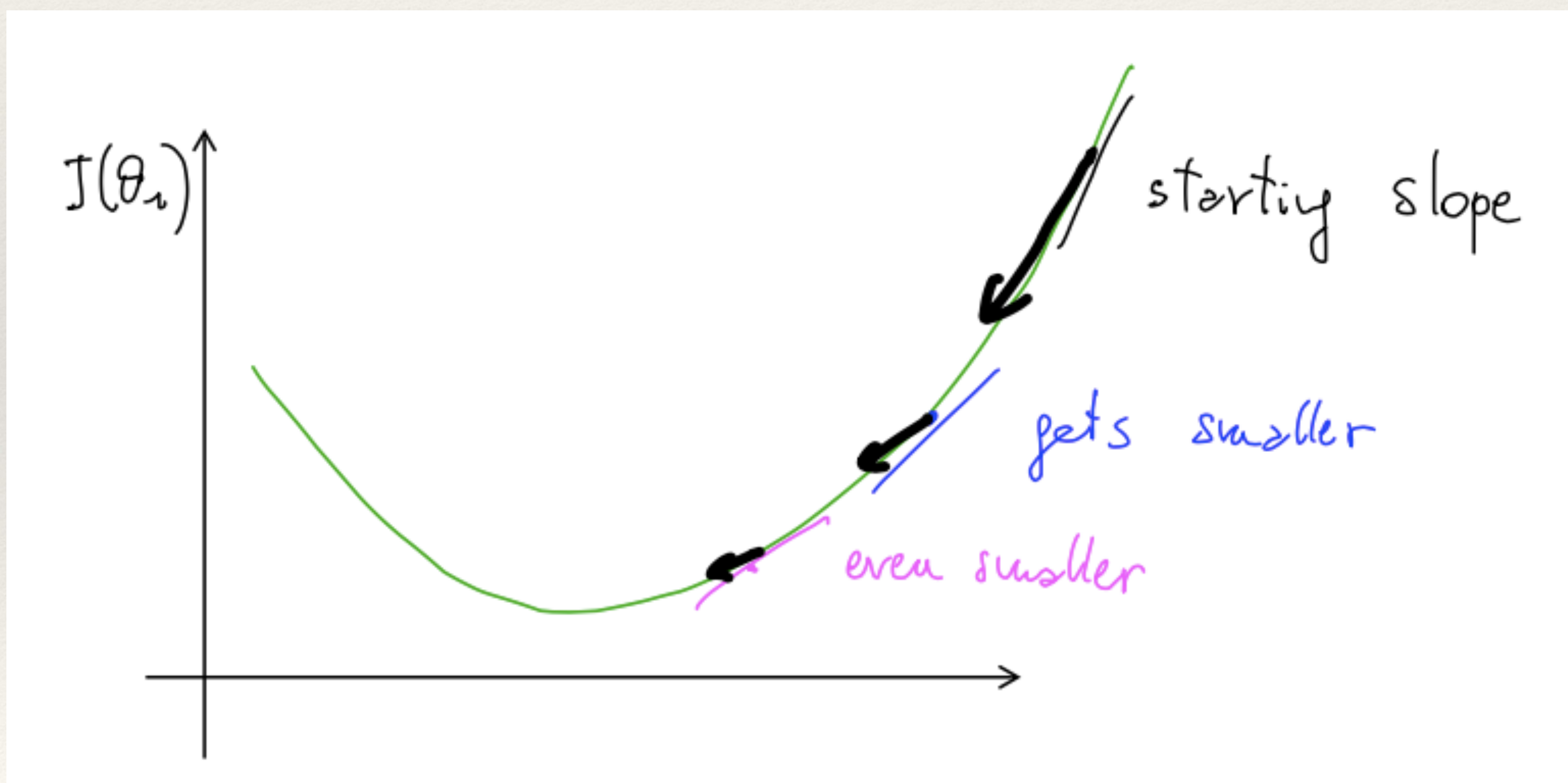
Let's look and digest all its parts.

- Firstly, let's look at the **θs**
- Secondly, let's look at the **derivative** term
- Thirdly, look at the **learning rate** term
    - constant/running
    - value

Why a _fixed_ learning rate?

The derivative also explains why GD can converge to a local minimum even with the learning rate _fixed_.

- As you approach a local minimum, GD will automatically take smaller steps. So, no need to decrease α over time.



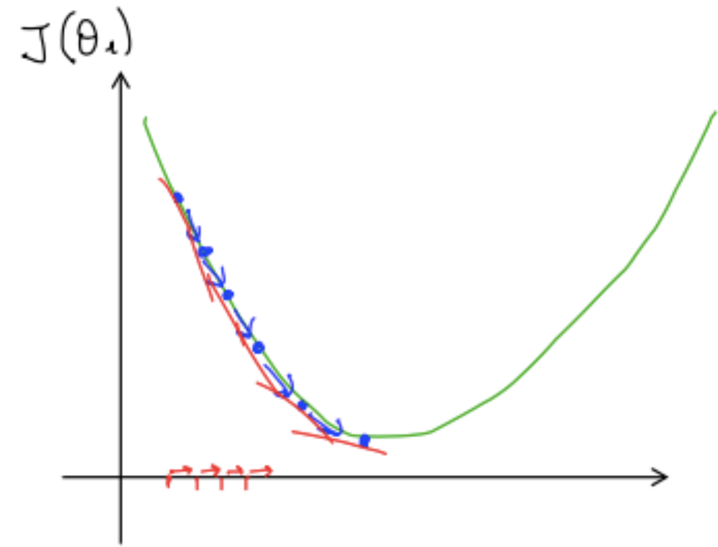
Implementation of GD algo: **learning rate**

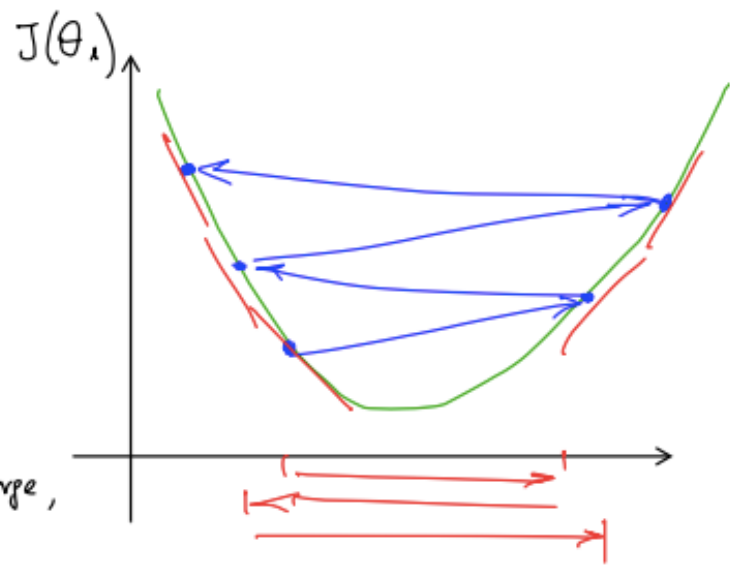
α too small $\Rightarrow$ GD can be slow

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

α too large $\Rightarrow$ GD may overshoot the minimum

→ it may fail to converge, or even diverge



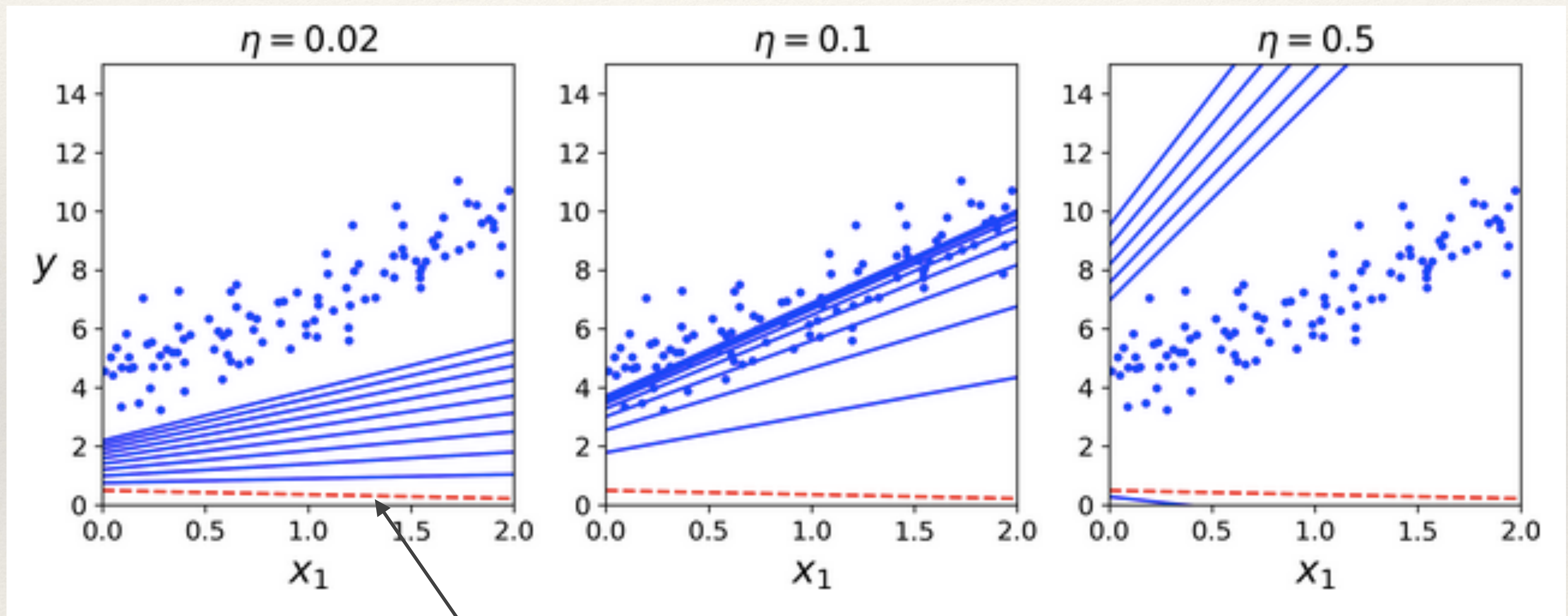


Batch GD and learning rate

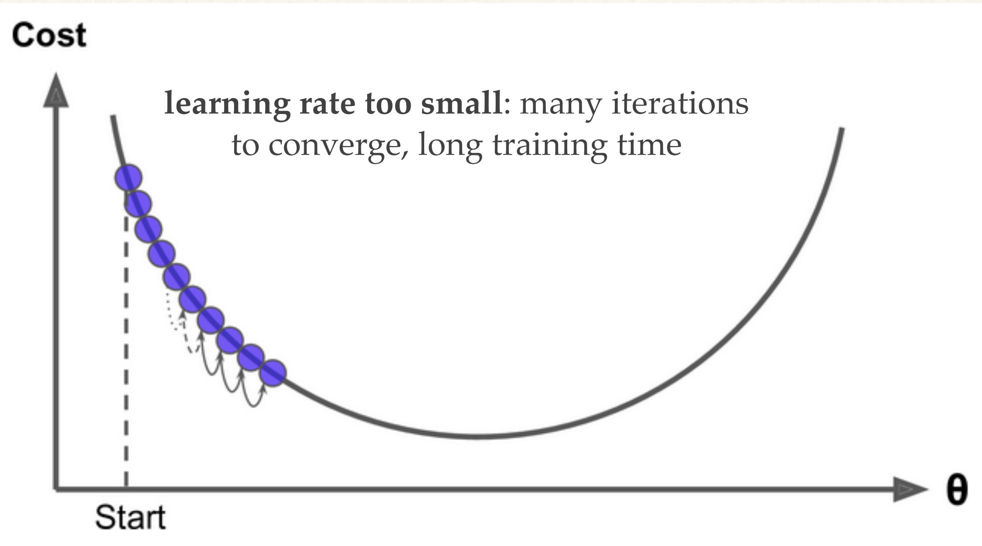
learning rate too low: will eventually converge, but slow..

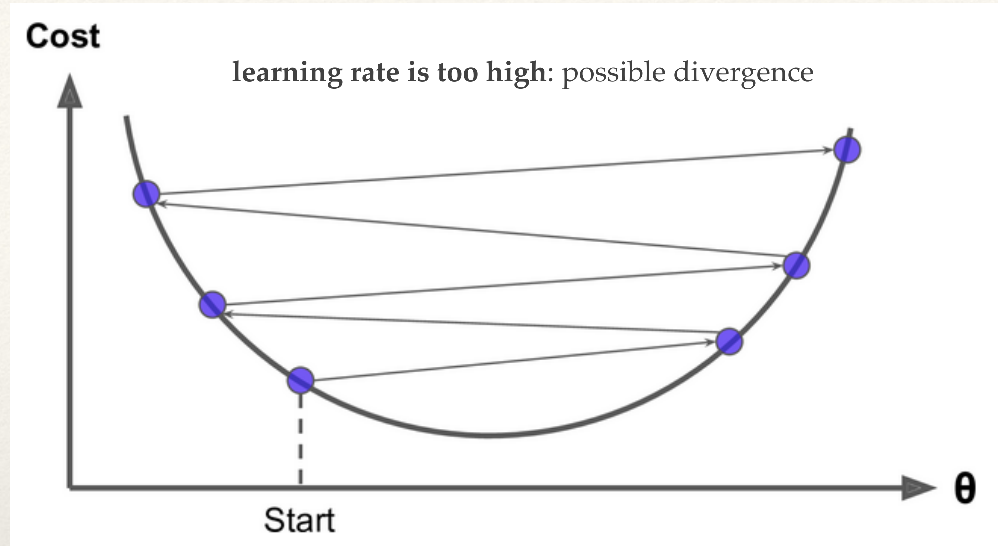
in just a few iterations, it has converged to the solution

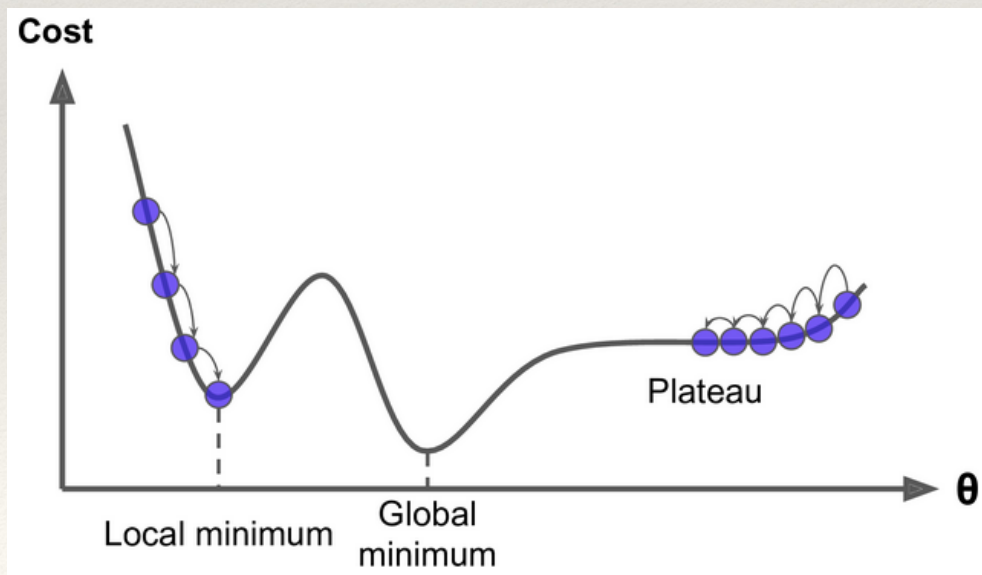
learning rate too high: may not converge at all



red line = start





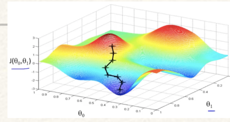


Linear Regression is a convex function, so no local minima, just a global minimum

Is GD guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high)?

GD vs normal equation

aka: an iterative process



need to choose alpha

- run it a few time and pick the best

needs many iterations

- depending on the details it would make it slower

needs feature scaling

works well even when n is large

- even millions of features
- cost ~scales as $O(kn^2)$

for some tasks (e.g. logistic regression algorithms) you need GD..

aka: an analytic solution

$\theta = (X^T X)^{-1} X^T y$

no need to choose alpha

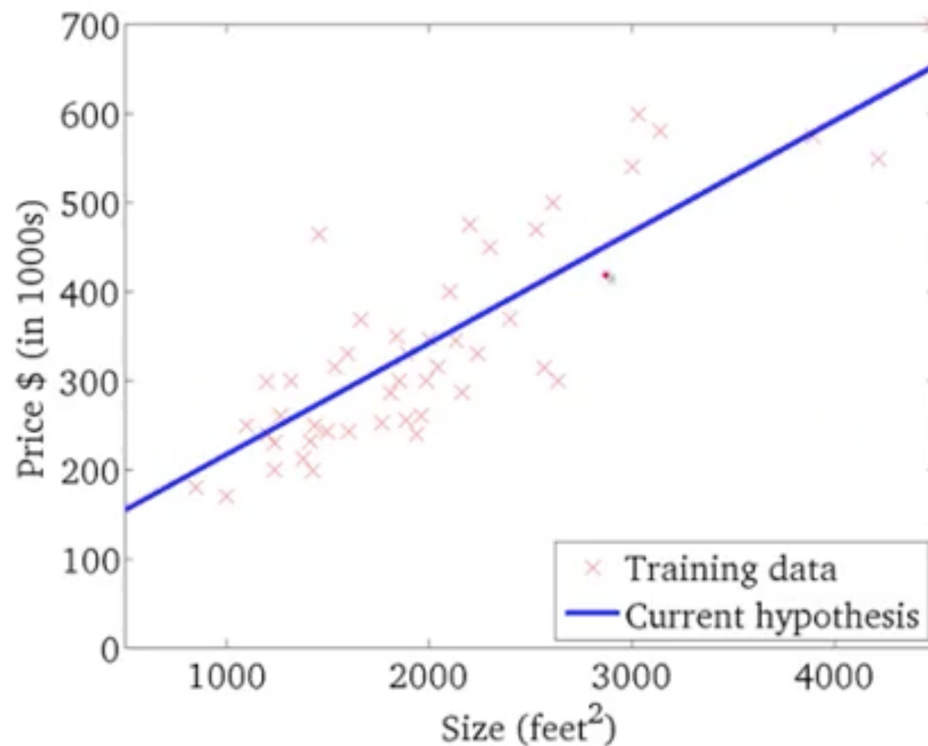
do not need any iterations

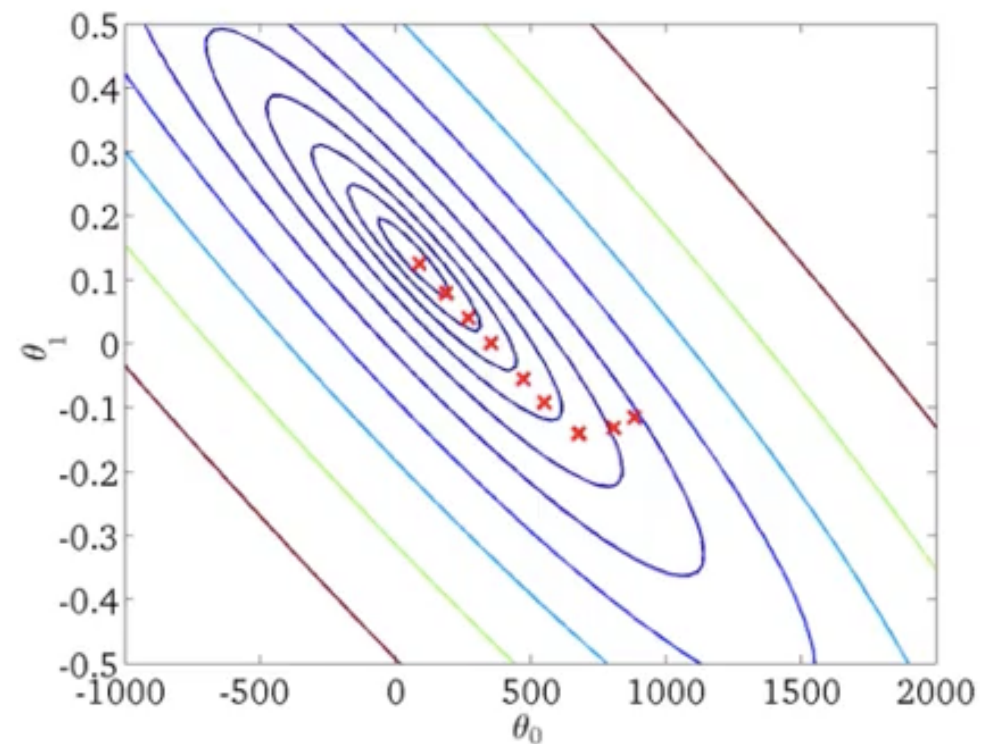
feature scaling is irrelevant

slow if n is large. Need to compute $(X^T X)^{-1}$

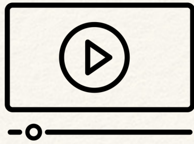
- nxn matrix, so very high for high n
- matrix inversion cost ~scales as $O(n^3)$

empirically, a limit at $n \sim 10^4$..





Eventually, it reaches the global minimum corresponds to having an hypothesis that gives me a good fit to the data



Bonus feature! GD in action

Set learning rate: 0.01

Execute single step: STEP 0

Reset the graph: RESET

Loss vs. Weight

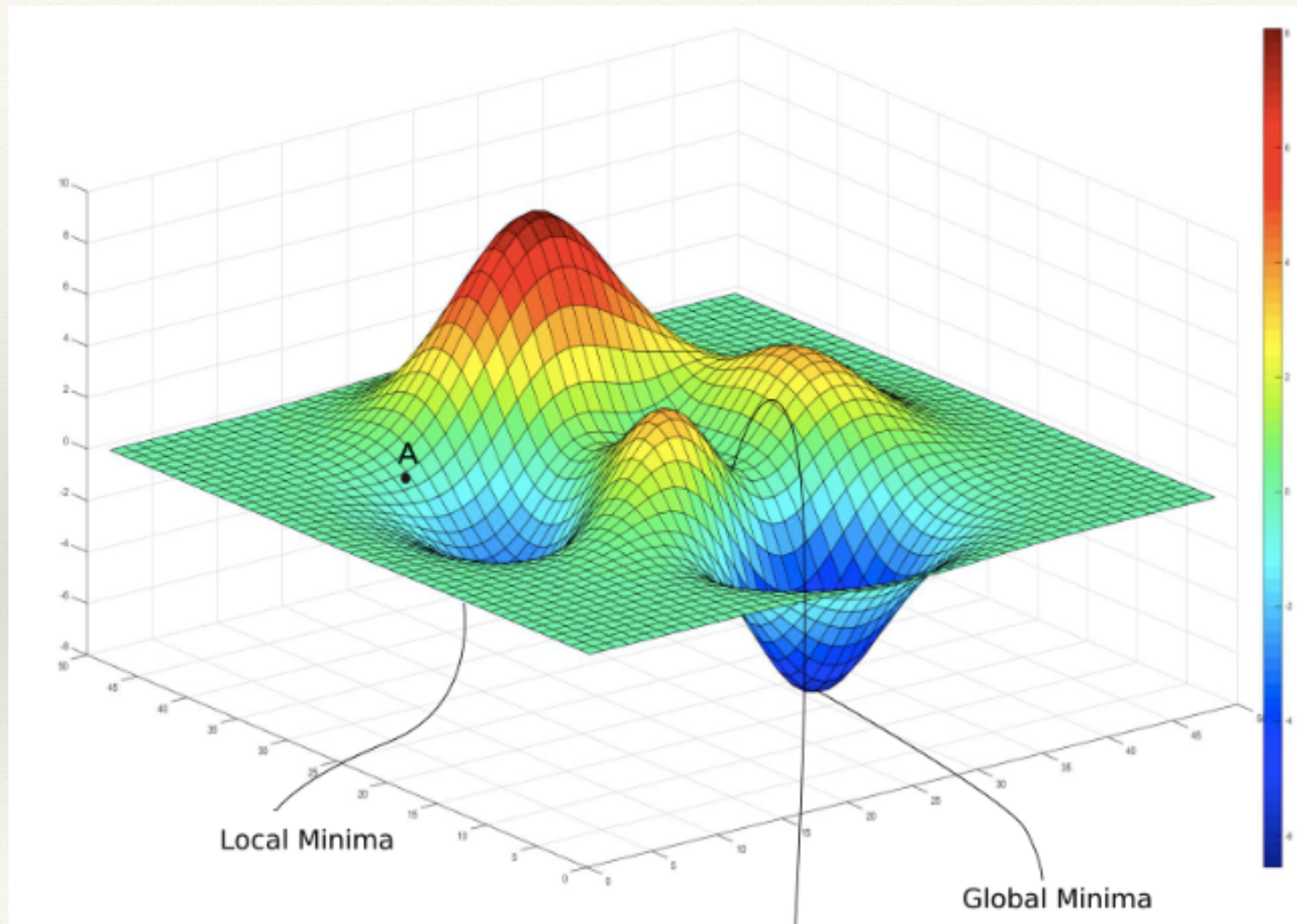
loss

10.0
7.5
5.0
2.5
0.0

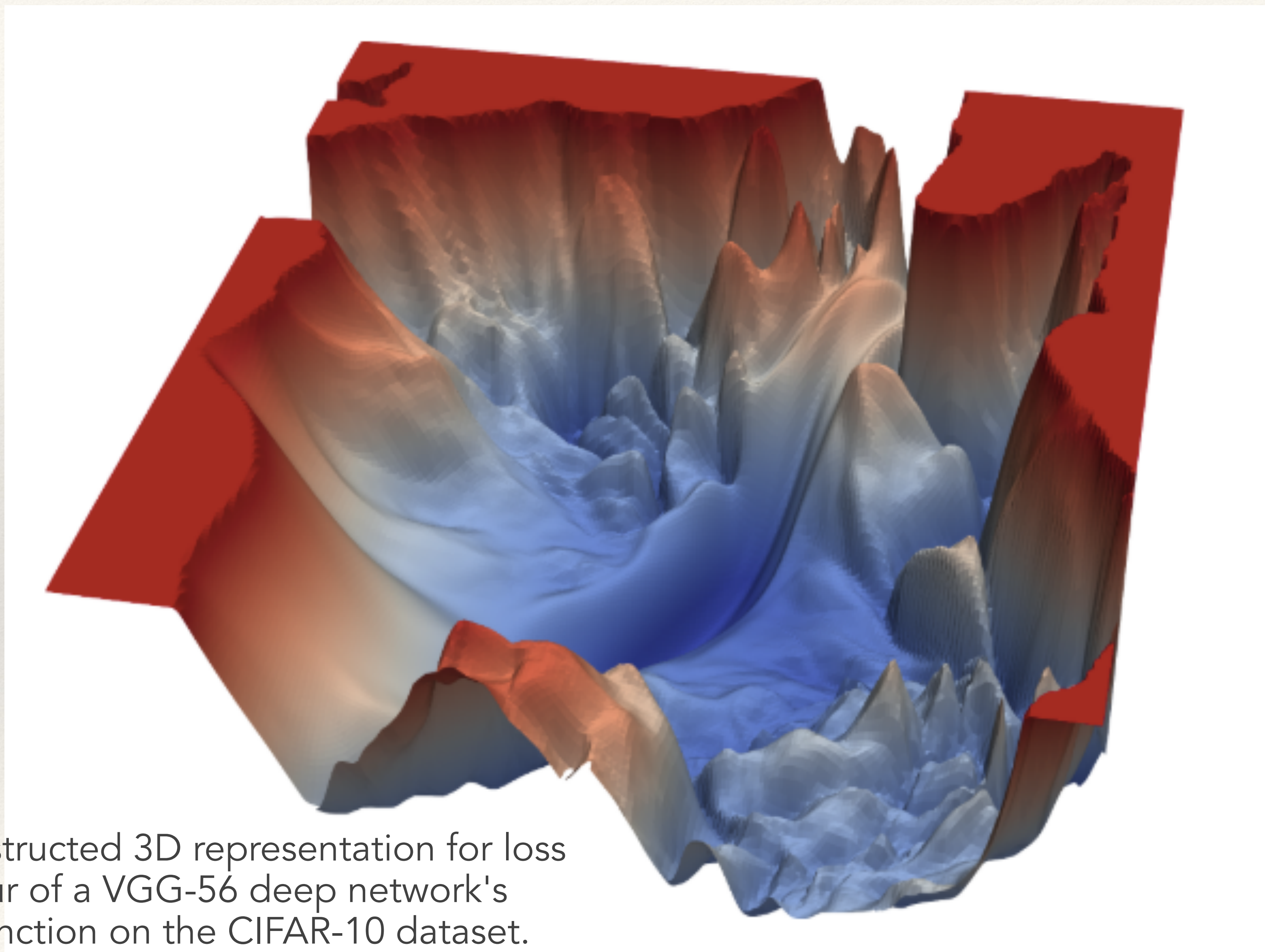
0 1 2 3 4

value of weight

Local minima?



A complicate loss landscape..



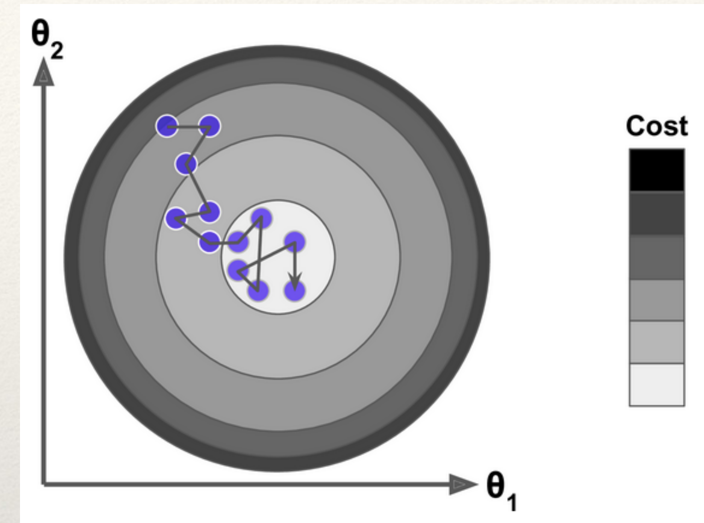
A constructed 3D representation for loss contour of a VGG-56 deep network's loss function on the CIFAR-10 dataset.

Stochastic GD

Stochastic GD

Batch GD uses the whole training set to compute the gradients at every step

- very slow when the training set is large



Stochastic GD is at the other extreme: it picks a single, random instance in the training set at every step and computes the gradients based only on that single instance

- pro: **much faster.** Also, possible to train on huge training sets (only one instance needs to be in memory at each iteration: SGD can be implemented as an out-of-core algorithm)
- con: **much less regular** than Batch GD, the cost function will bounce up and down, decreasing only on average, and will still bounce close to the minimum. so once the algorithm stops, the final parameter values are good, but not optimal
  - when the cost function is very irregular, this can actually help the algo jump out of local minima, so Stochastic GD has a better chance of finding the global minimum than Batch GD does

Stochastic GD

In Stochastic GD, randomness is good to escape from local optima, but bad because the algo can never settle at the minimum

You can **gradually reduce the learning rate**

- The steps start out large (which helps make quick progress and escape local minima)
- then get smaller and smaller, allowing the algo to settle at the global minimum.
    - if the learning rate is reduced too quickly, you may get stuck in a local minimum
    - if the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early

BATCH GD

$$J_{Train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

$$\frac{\partial}{\partial \theta_j} J_{Train}(\theta)$$

} (for every $j = 0, \dots, n$)

STOCHASTIC GD algo

1. randomly reshuffle the dataset → a standard pre-processing step

2. Repeat {

for $i = 1, \dots, m$ {

$$\theta_j := \theta_j - \alpha \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

} (for $j = 0, \dots, n$)

}

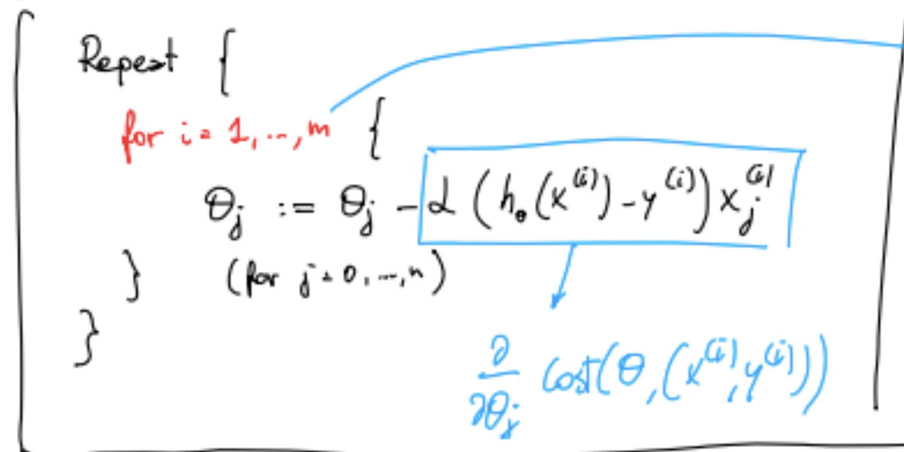
the same as $\frac{\partial}{\partial \theta_j} cost(\theta, (x^{(i)}, y^{(i)}))$

BATCH GD

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

} (for every $j = 0, \dots, n$)



a scan through each training examples

↓

look at $(x^{(i)}, y^{(i)})$

↓

does a little GD step on that example only w.r.t. the cost of just that training example

↓

look at that example and modify the params a little bit to fit just that training example a little bit better

←

then pass on to the next and modify the params θ_j to try to fit just that example better

done for all i → m, then "Repeat" causes multiple passes over the entire training set
(1x – 10x)

⇨ now we understand why reshuffling the entire set before starting!

Recap:

BATCH GD

Repeat {

$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$

} (for every $j = 0, \ldots, n$)

STOCHASTIC GD

Repeat {

for $i = 1, \ldots, m$ {

$\theta_j := \theta_j - \alpha \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$

} (for $j = 0, \ldots, n$)

}

$\frac{\partial}{\partial \theta_j} cost(\theta, (x^{(i)}, y^{(i)}))$

Mini-batch GD

Mini-batch GD

In Batch GD we will use **all m examples in each iteration**.

In Stochastic GD we will use **1 single example in each iteration**.

What **Mini-batch GD** does is somewhere in between. Specifically, with this algorithm we're going to use **b examples in each iteration**, where b is a parameter called the "mini batch size”.

- This is just like batch GD, except that I'm going to use a much smaller batch size. That’s why we call it “mini”.

Yes, **b is an additional hyperparameter.**.

- A typical range for b might be anywhere from2 up to b equals 100 (so, 10ish?)

mb - GD

Example : $b = 10$, $m = 1000$

Get bunches of 10 examples each : $(x^{(i)}, y^{(i)}), \ldots, (x^{(i+9)}, y^{(i+9)})$

Repeat {

for $i = 1, 11, 21, \ldots, 991$ {

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=1}^{i+9} \left(h_\theta(x^{(k)}) - y^{(k)} \right) \cdot x_j^{(k)}$$

} (for every $j = 0, \ldots, n$

}

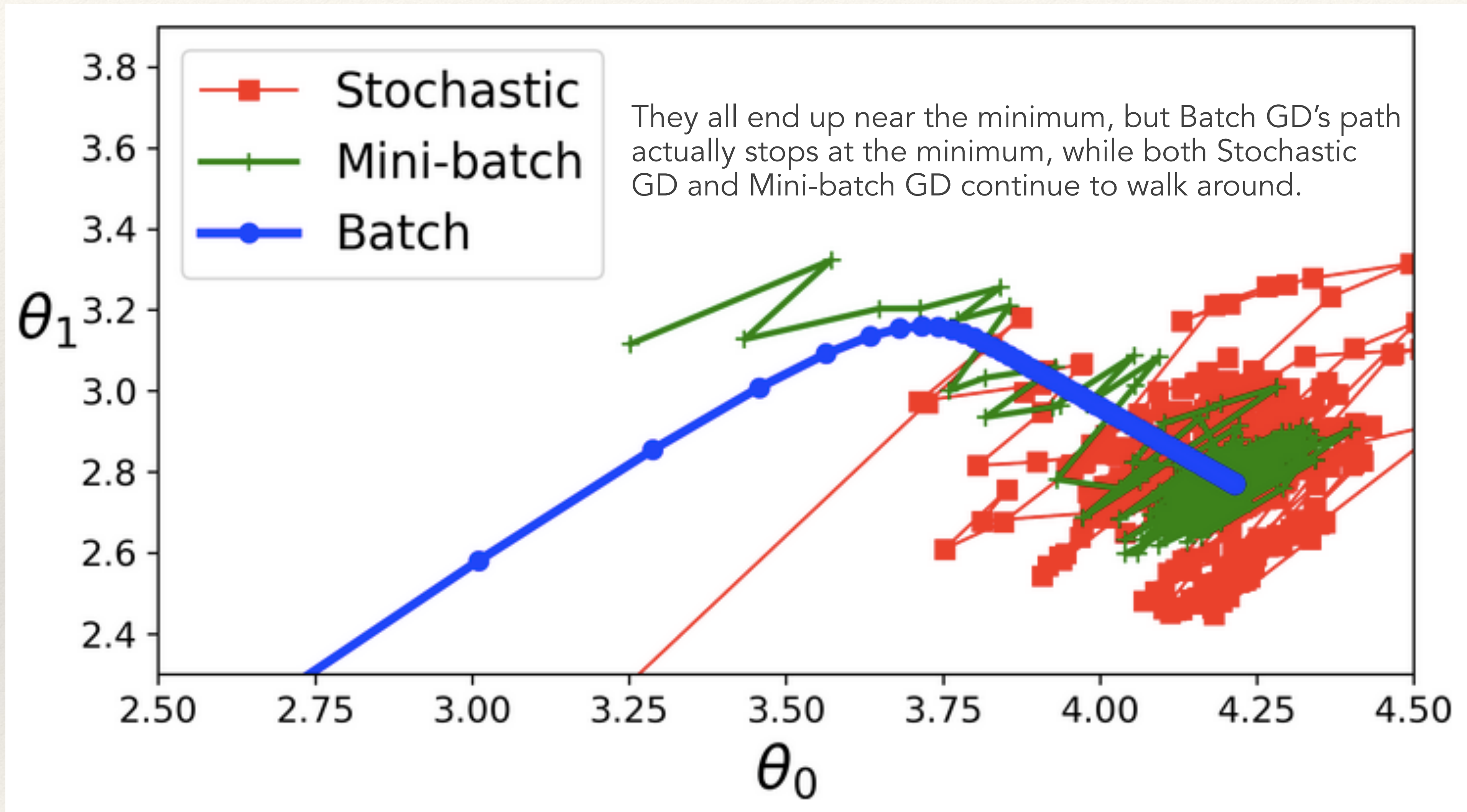
perform a GD update using b examples at a time

Mini-batch GD

In summary:

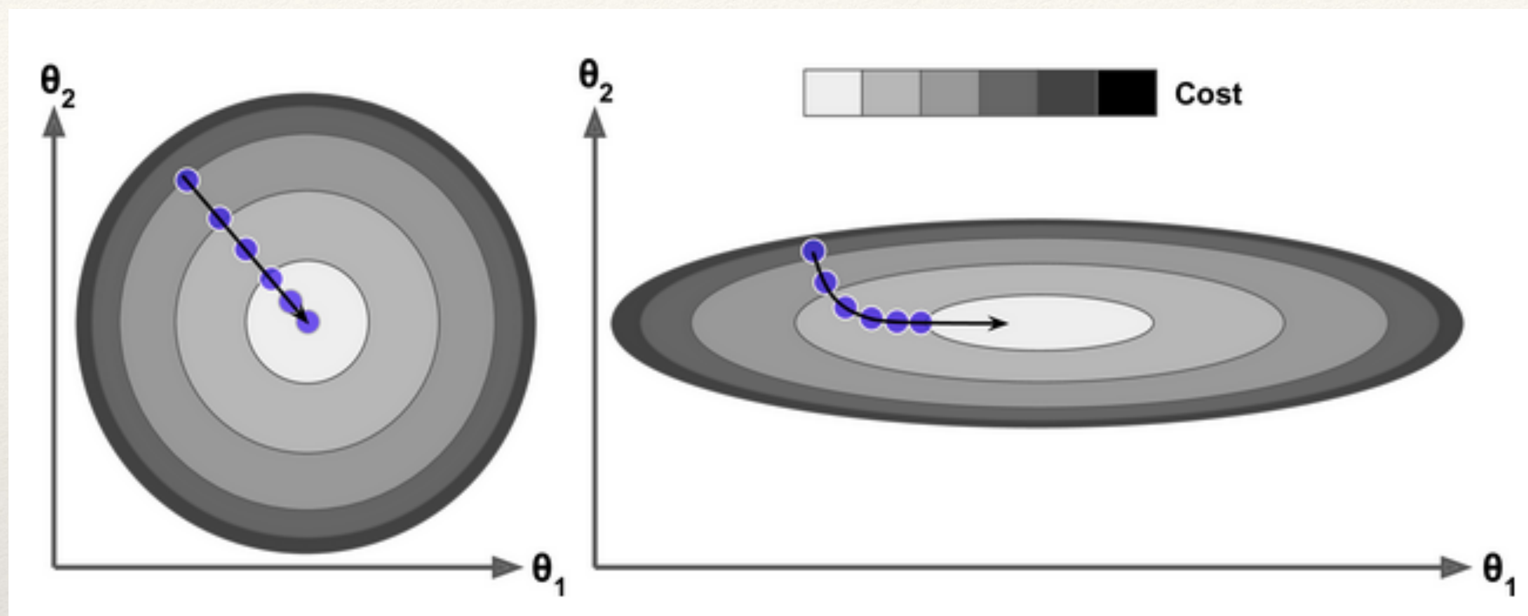
- pro: you can get a performance boost from hw optimization of matrix operations, especially when using GPUs
- pro: the algo’s progress in parameter space is less erratic than with Stochastic GD, especially with fairly large mini-batches.
- con: it may be harder than with SGC for it to escape from local minima (in the case of problems that suffer from local minima)

Batch vs Stochastic vs Minibatch GD



Note: there is almost no difference after training: all these algos end up with very similar models and make predictions in exactly the same way.

GD and feature scaling



When using GD, you should ensure that all features have a similar scale - or else it will take much longer to converge

- e.g. use sklearn's StandardScaler class

Polinomial Regression

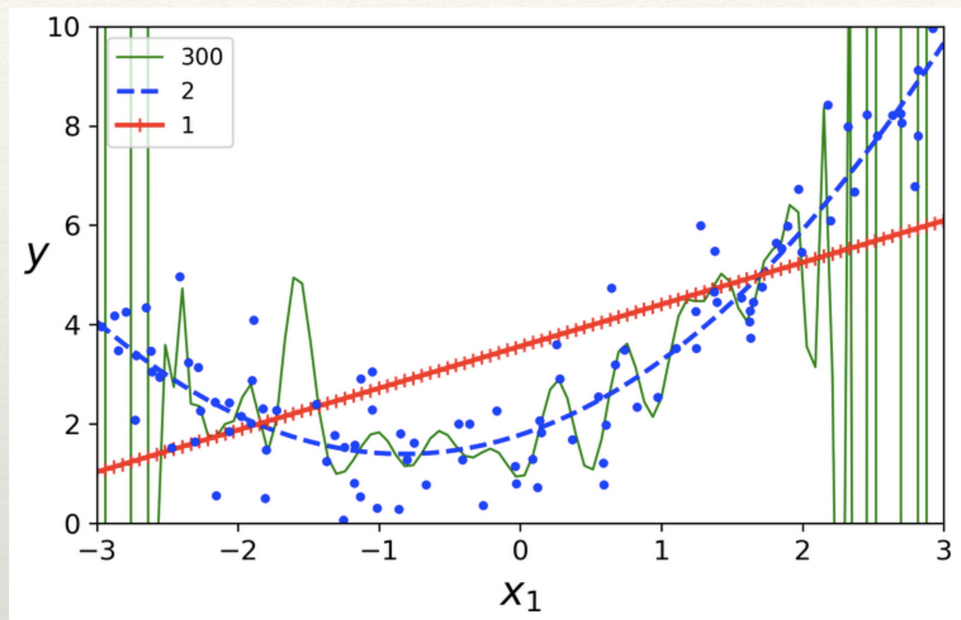
What if your data is actually more complex than a simple straight line?

You can actually use a linear model to fit nonlinear data:

- add powers of each feature as new features, then train a linear model on this "extended" set of features

Underfitting or Overfitting?

High-degree Polynomial Regression will likely fit the training data much better than plain Linear Regression



Linear model
→ **underfitting**

Polynomial regression
→ **overfitting**

How to diagnose this? We used CV to get an estimate of a model's generalisation performance:

- if a model performs **well on the training data** but **generalizes poorly** according to the CV metrics, then your model is **<u>overfitting</u>**
- if it performs **poorly on both**, then it is **<u>underfitting</u>**

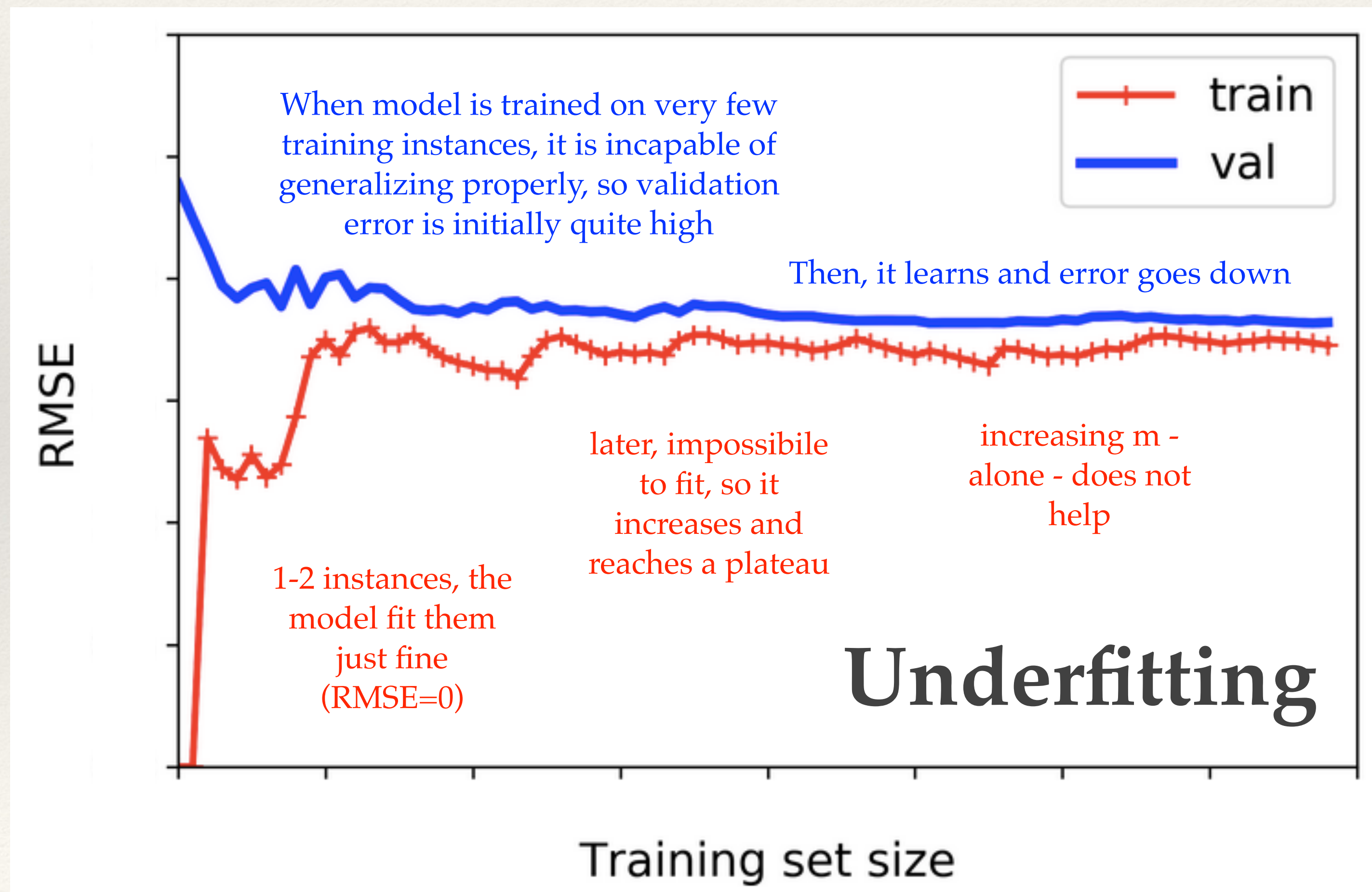
Concretely, this is one way to tell when a model is too simple or too complex.

Learning curves

Useful to look at the **learning curves**

- these are plots of the model's performance on the training set and the validation set as a function of the training set size (or the training iteration)

plain Linear Regression model: learning curves



Learning curves

A high-order polynomial model performs **significantly better on the training data than on the validation data**, which is the signature of an **overfitting** model.

- However, if you used a much larger training set, the two curves would continue to get closer.

10th-degree polynomial model: learning curves

One way to improve an overfitting model is to feed it more training data until the validation error reaches the training error.