LECTURE 12

Gradient Descent

An optimization method to numerically minimize loss functions.

Goals for this Lecture

Optimizing complex models – how do we select parameters when the loss function is “tricky”?

- Identifying cases where straight calculus or geometric arguments won’t work
- Introducing an alternative technique – gradient descent

Agenda

- Optimization: where are we?
- Minimizing an arbitrary 1D function
- Gradient descent on a 1D model
- Gradient descent on high-dimensional models
- Batch, mini-batch, and stochastic gradient descent

Optimization: Where Are We?

- **Optimization: where are we?**
- Minimizing an arbitrary 1D function
- Gradient descent on a 1D model
- Gradient descent on high-dimensional models
- Batch, mini-batch, and stochastic gradient descent

What we've done

Takeaways from the past few lectures:

- Choose a model
- Choose a loss function
- Optimize parameters – choose the values of θ that minimize the model's loss

How have we optimized?

1. Use calculus to solve for θ

   Take derivatives, set equal to 0, solve.

1. Use a geometric argument

   Using orthogonality, derive the OLS solution $\hat{\theta} = \left(\mathbb{X}^T\mathbb{X}\right)^{-1}\mathbb{X}^T\mathbb{Y}$

Where we’re going

We made some big assumptions with the calculus-based and geometric techniques.

- Calculus: assumed that the loss function was differentiable at all points and that the algebra was manageable
- Geometric: OLS *only* applies when using a linear model with MSE loss

To design more complex models with different loss functions, we need a new optimization technique: **gradient descent**.

Big Idea: use an algorithm instead of solving for an exact answer

Our roadmap

Big Idea: use an algorithm instead of solving for an exact answer

Structure for today's lecture:

1. Use a simple example (some arbitrary function) to build the intuition for our algorithm
2. Apply the algorithm to a *simple model* to see it in action
3. Formalize the algorithm to be applied to *any model*

Remember our goal: find the parameters that **minimize the model's loss**

Let's do it.

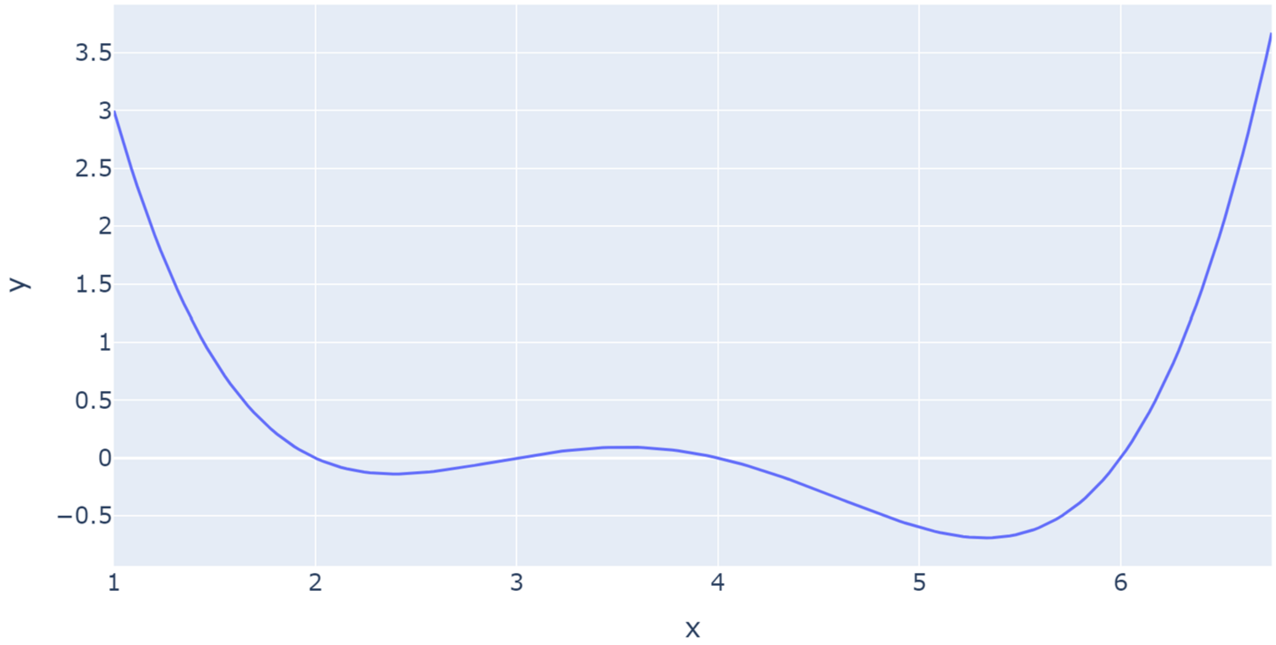
Minimizing an Arbitrary 1D Function

- Optimization: where are we?
- **Minimizing an arbitrary 1D function**
- Gradient descent on a 1D model
- Gradient descent on high-dimensional models
- Batch, mini-batch, and stochastic gradient descent

An arbitrary function

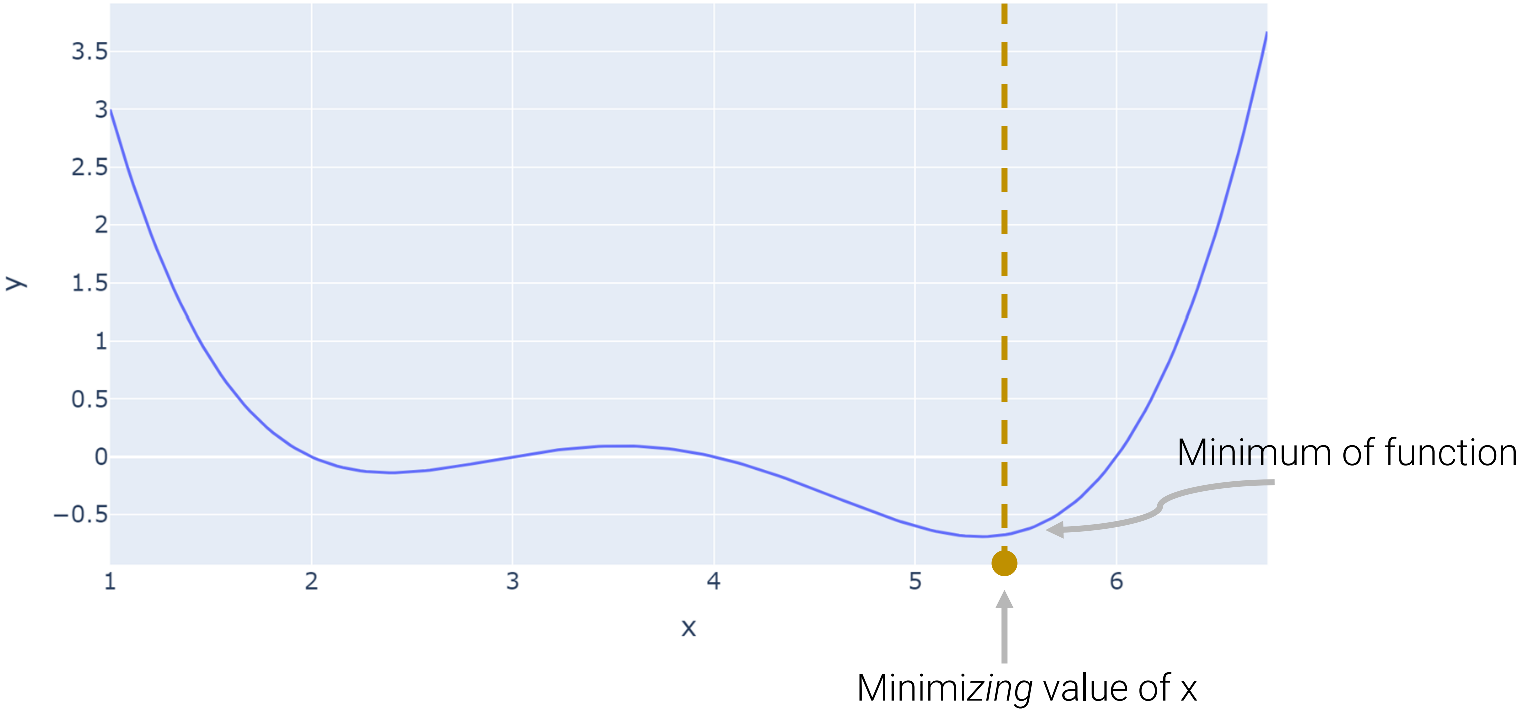
```
def arbitrary(x):
    return (x**4 - 15*x**3 + 80*x**2 - 180*x + 144)/10

x = np.linspace(1, 6.75, 200)
fig = px.line(y = arbitrary(x), x = x)
```



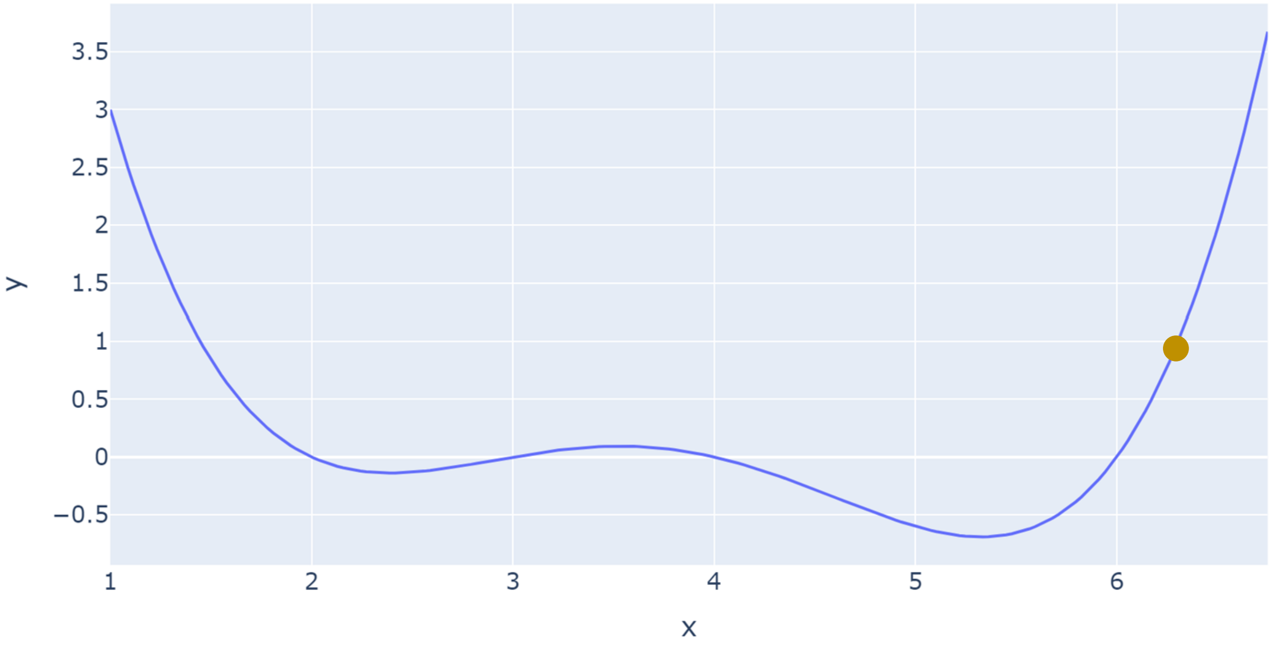
An arbitrary function

Our goal is to find the value of x that minimizes our function.



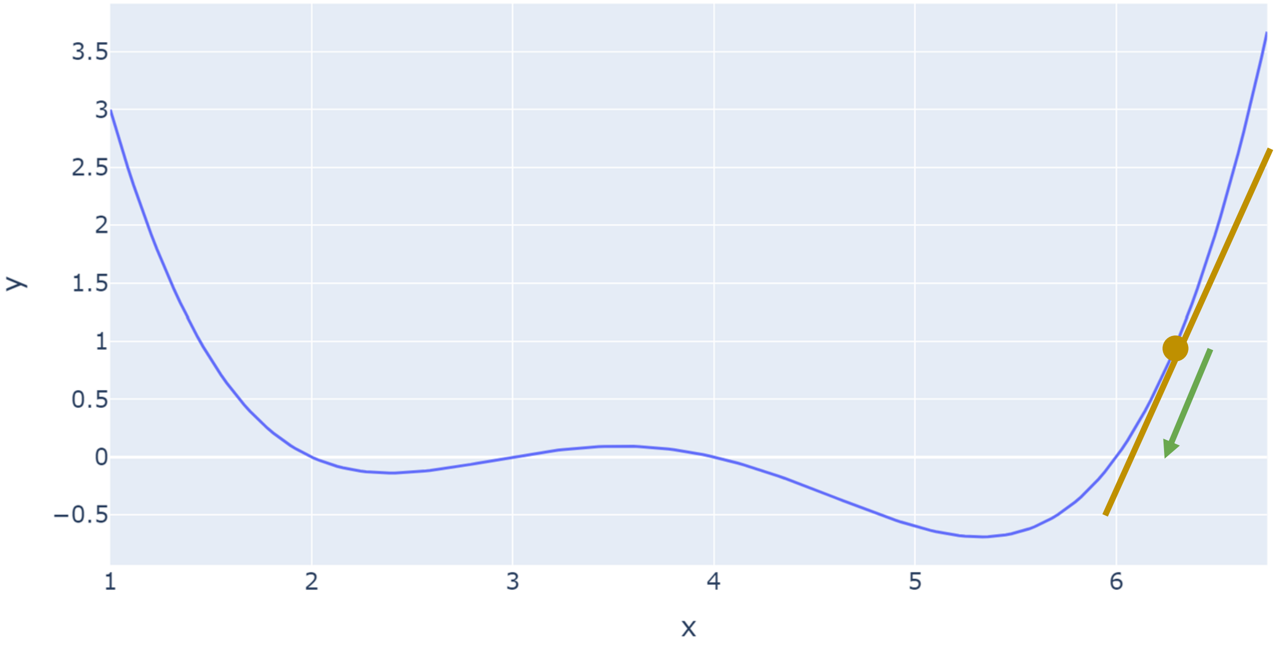
Finding the minimum

We could start with a random guess.



Finding the minimum

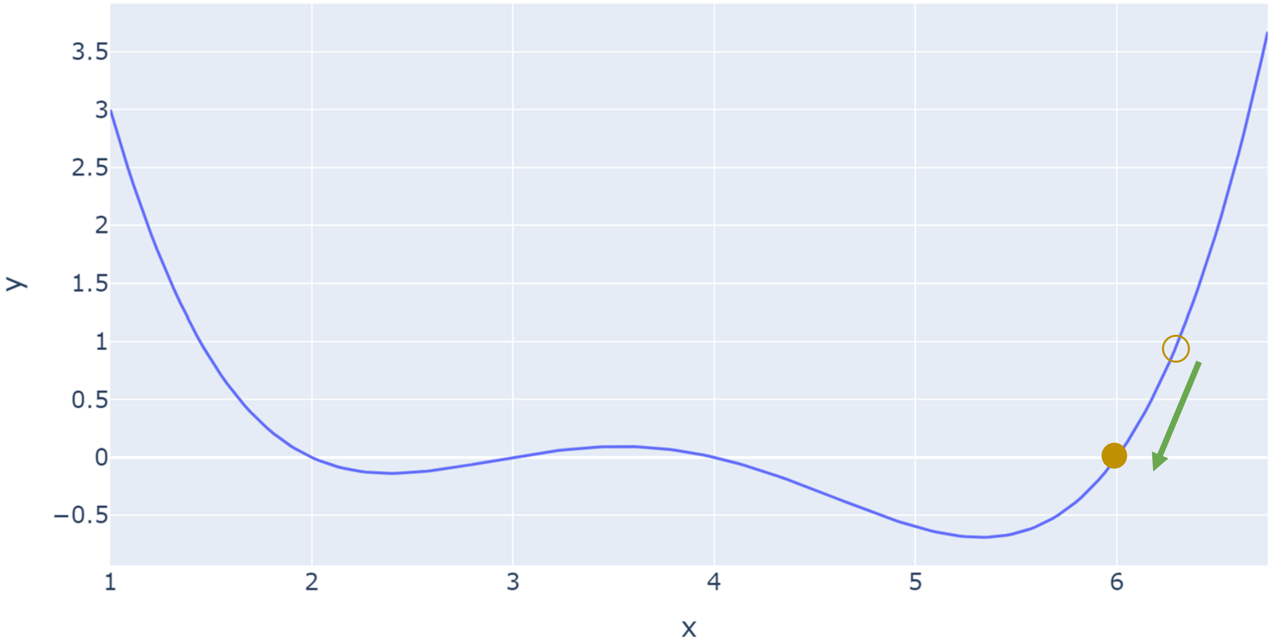
Where do we go next? We “step” downhill.



Follow the slope of the line down to the minimum.

Finding the minimum

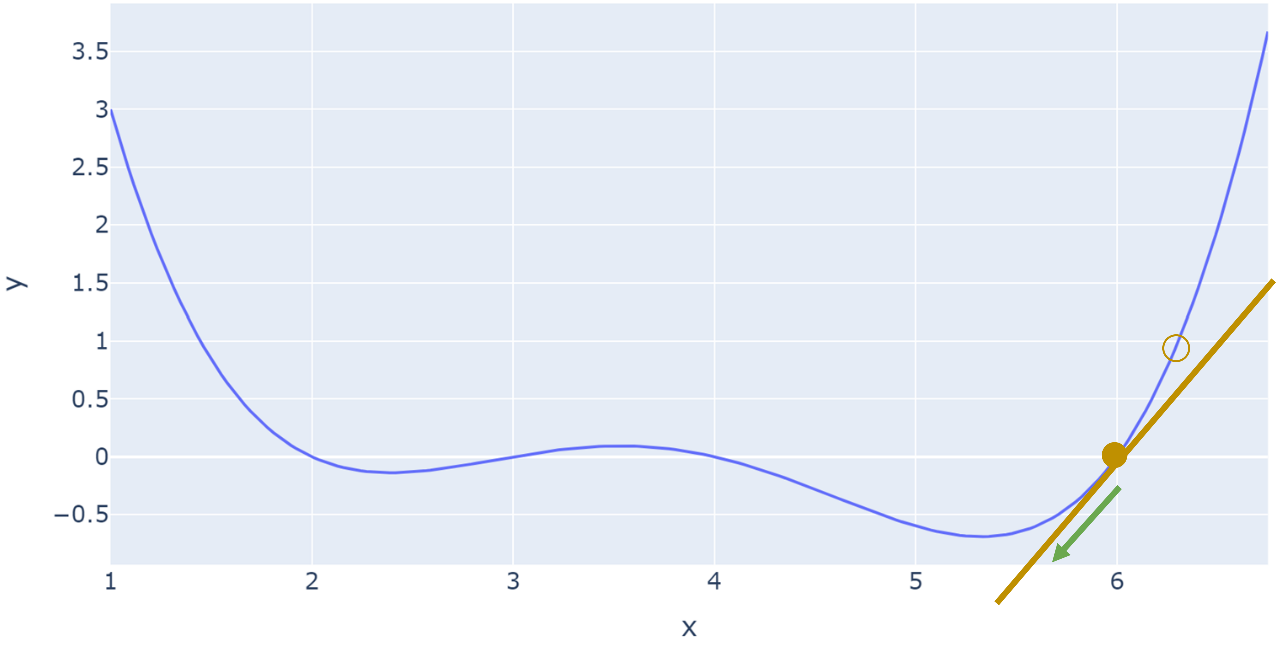
We arrive closer to the minimum.



Positive slope → step to the left

Finding the minimum

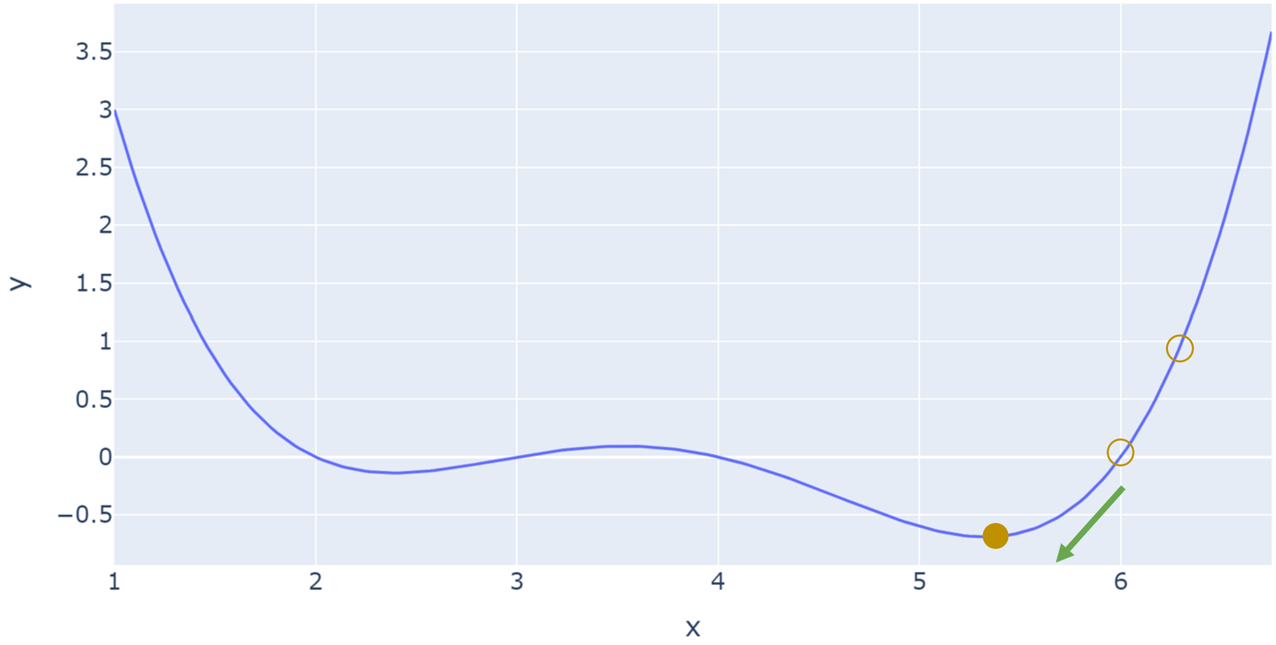
Do this again: follow the slope downwards towards the minimum



Positive slope → step to the left

Finding the minimum

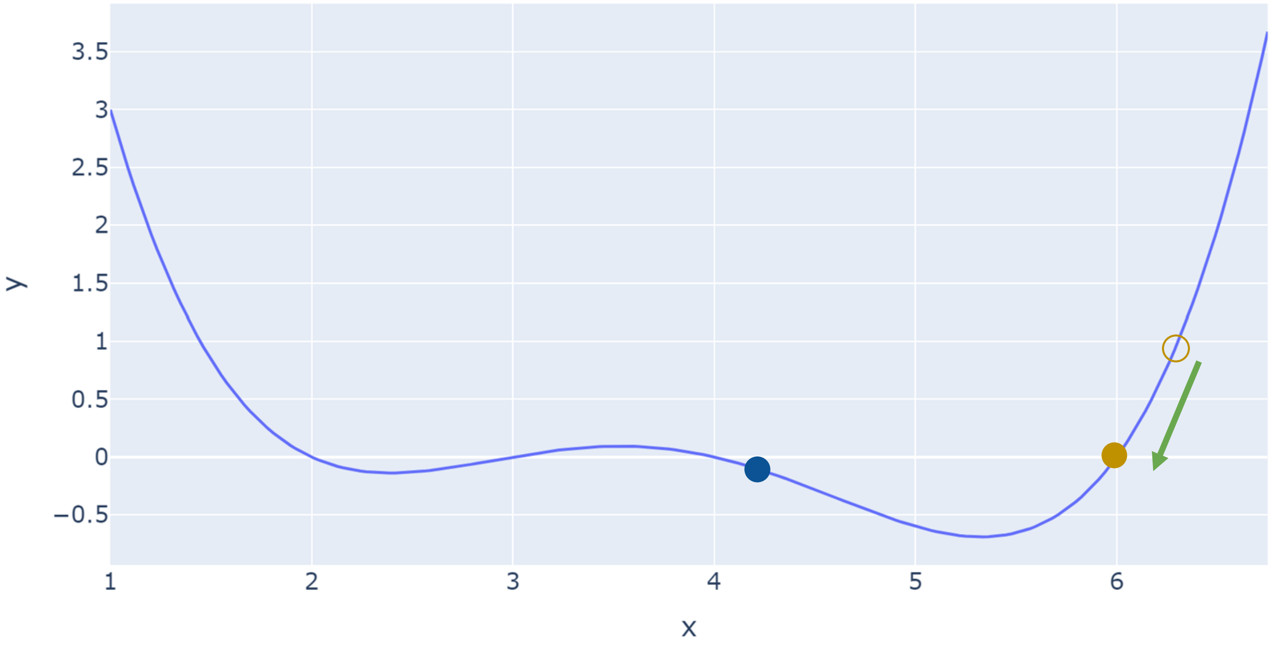
Do this again: follow the slope downwards towards the minimum



Positive slope → step to the left

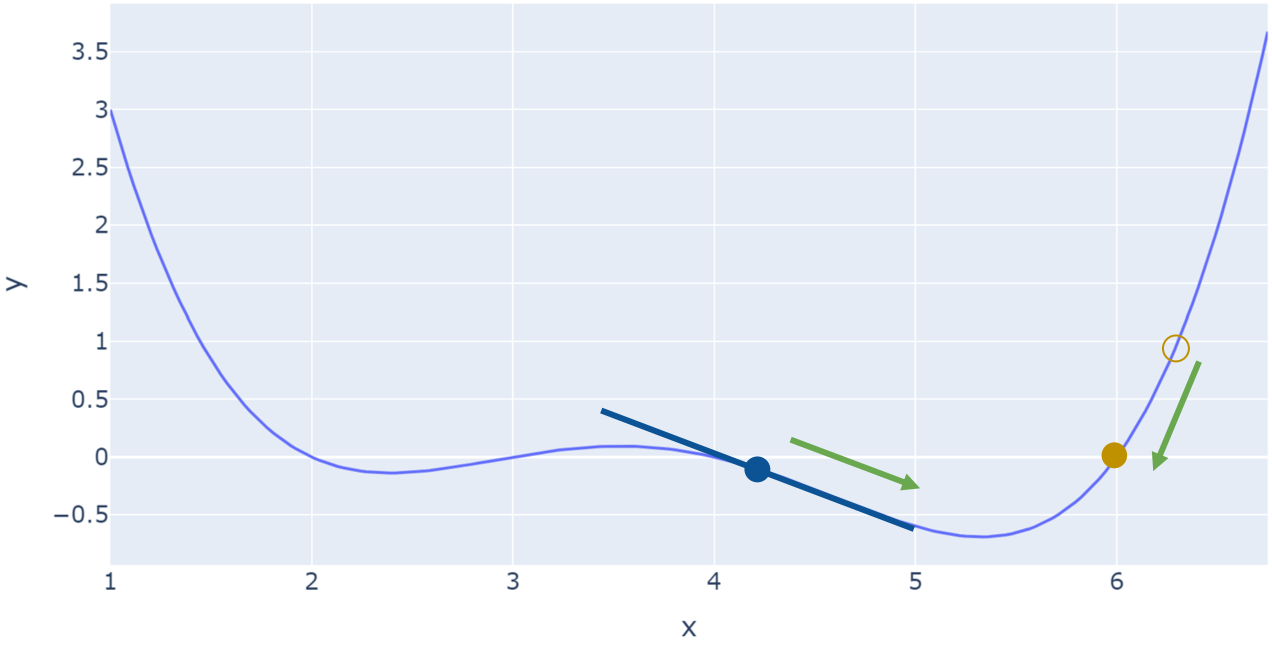
Finding the minimum

What if we had started elsewhere?



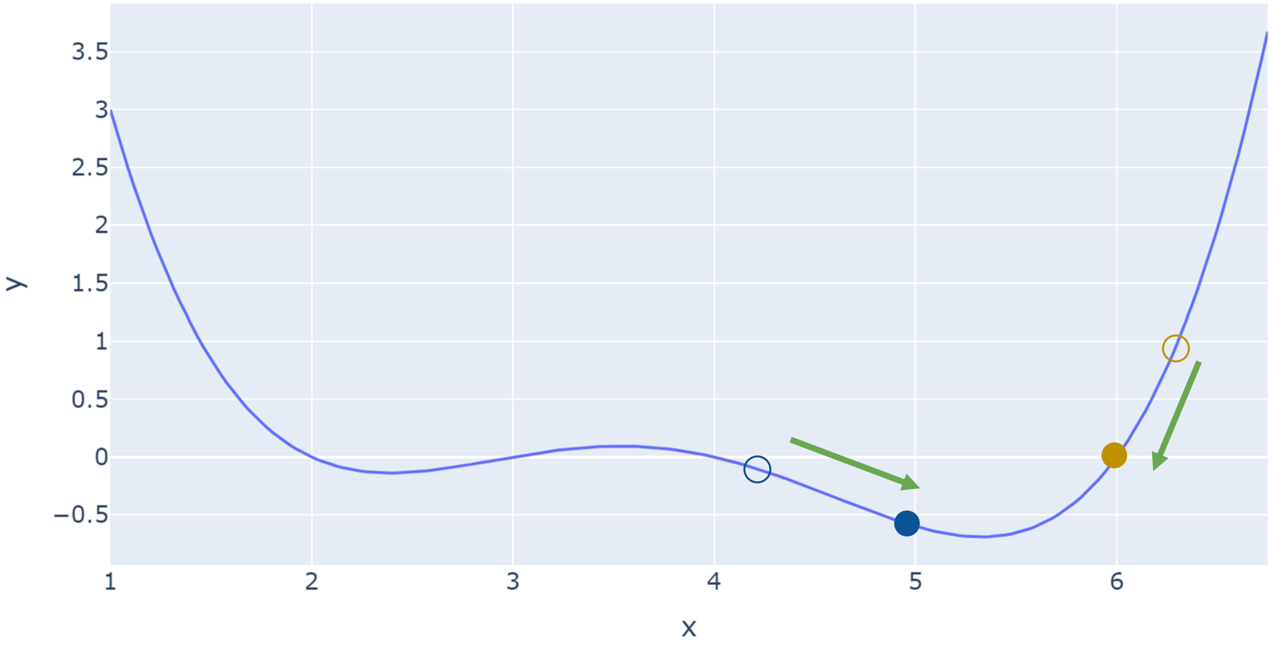
Finding the minimum

What if we had started elsewhere?



Finding the minimum

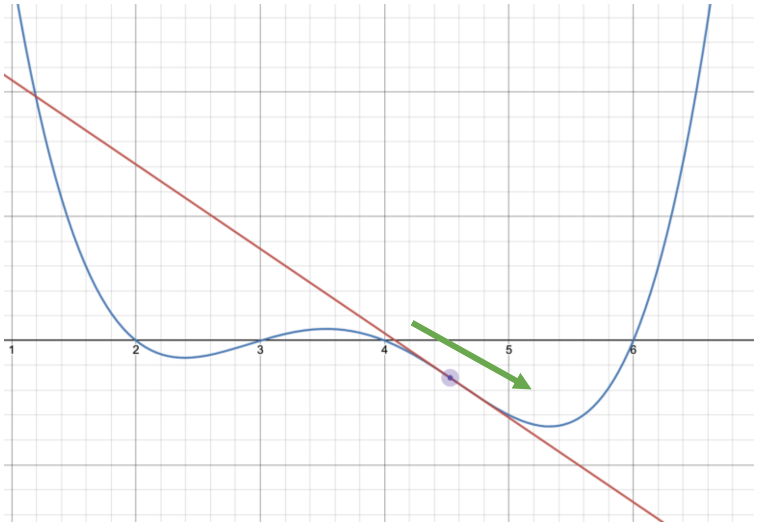
What if we had started elsewhere?



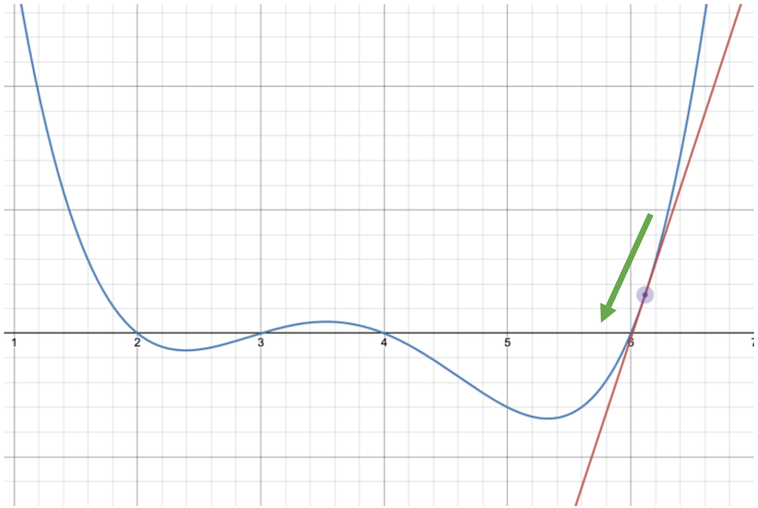
Negative slope → step to the right

Slopes tell us where to go

Negative slope → step to the right
Move in the *positive* direction



Positive slope → step to the left
Move in the *negative* direction



The derivative of the function at each point tells us the direction of our next guess.
Demo link: https://www.desmos.com/calculator/twpnylu4lr

Slopes tell us where to go

The derivative of the function at each point tells us the direction of our next guess.

Negative slope → step to the right
Move x in the *positive* direction

Positive slope → step to the left
Move x in the *negative* direction

How can we use this?

slido

What is a rule that could help us make our next guess (x at time t+1) from our previous guess (x at time t)?

Click **Present with Slido** or install our [Chrome extension] to activate this poll while presenting.

Slopes tell us where to go

The derivative of the function at each point tells us the direction of our next guess.

Negative slope → step to the right
Move x in the *positive* direction

Positive slope → step to the left
Move x in the *negative* direction

Our first attempt at making an algorithm: step in the opposite direction to the slope

$$x^{(t+1)} = x^{(t)} - \frac{d}{dx} f\left(x^{(t)}\right)$$

Our next guess for the minimizing x

…starts at the previous guess

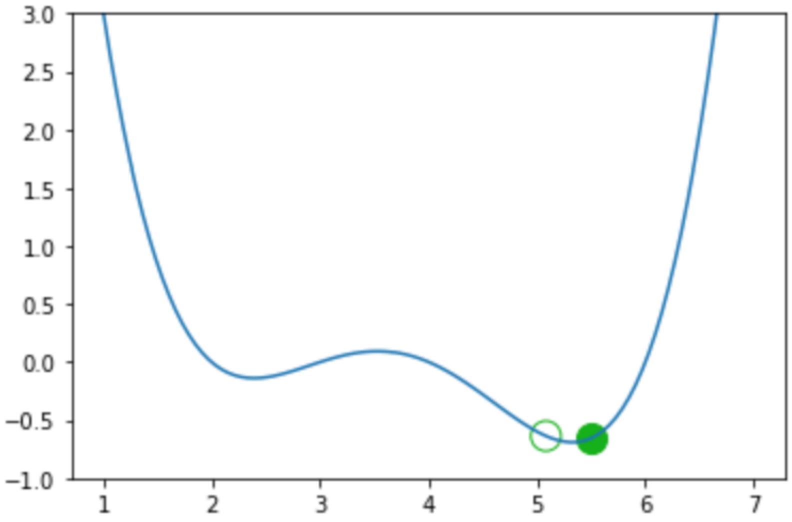
…and moves opposite to the slope

Algorithm attempt #1

```python
def plot_one_step(x):
    new_x = x - derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```python
plot_one_step(5.080917145374805)
```

```
old x: 5.080917145374805
new x: 5.489966698640582
```



We appear to be bouncing back and forth. Turns out we are stuck!

- Any suggestions for how we can avoid this issue?

Introducing a learning rate

Problem: each step is too big, so we overshoot the minimizing x

Solution: decrease the size of each step

Updated algorithm: α represents a **learning rate** that we choose. It controls the size of each step.

$$x^{(t+1)} = x^{(t)} - \alpha \frac{d}{dx} f\left(x^{(t)}\right)$$

Our next guess for the minimizing x

…starts at the previous guess

…and moves opposite to the slope

Let's try $\alpha = 0.3$

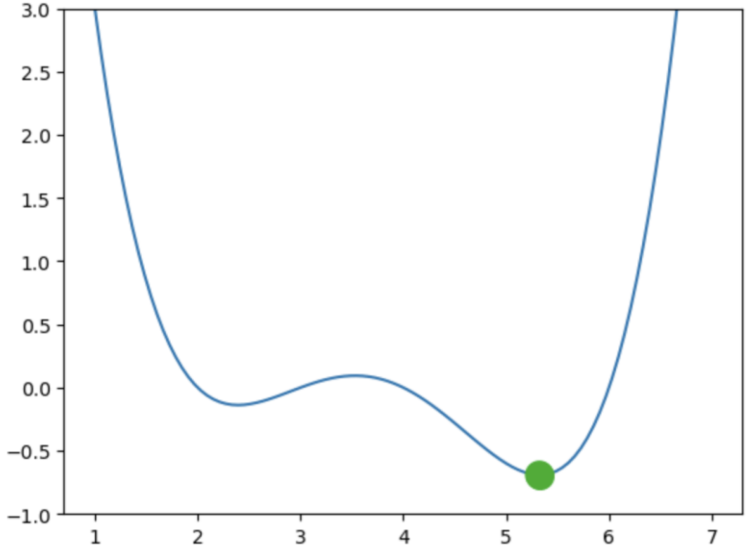
Algorithm attempt #2

```python
def plot_one_step_lr(x):
    # Implement our new algorithm with a learning rate
    new_x = x - 0.3 * derivative_arbitrary(x)

    # Plot the updated guesses
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    plot_x_on_f_empty(arbitrary, x)
    print(f'old x: {x}')
    print(f'new x: {new_x}')
```

```python
plot_one_step_lr(5.323)
```

```
old x: 5.323
new x: 5.325108157959999
```



When do we stop updating?

Some options:

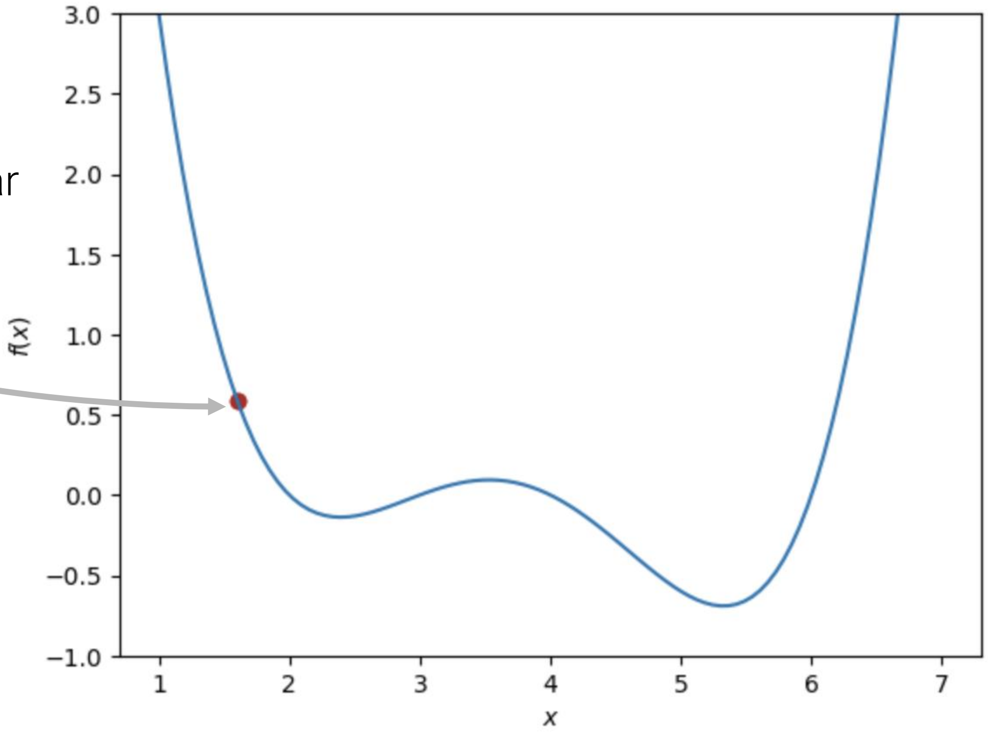
- After a fixed number of updates
- Subsequent update doesn't change "much"

Convergence: GD settles on a solution and stops updating significantly (or at all)

Convexity

What if our initial guess had been elsewhere?

Starting guess near
local minimum

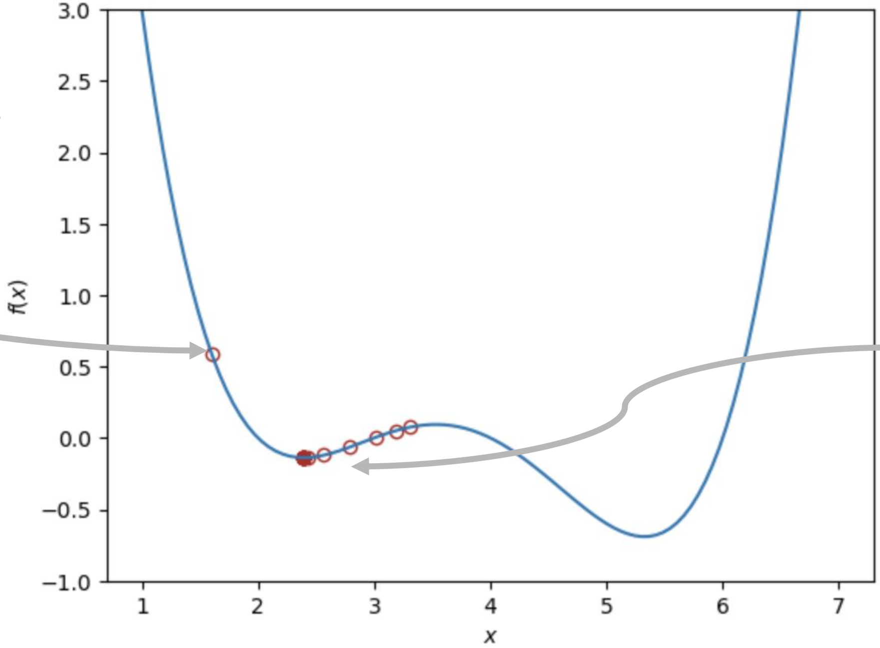


Convexity

What if our initial guess had been elsewhere?

The algorithm may have gotten “stuck” in a local minimum.

Starting guess near local minimum



Final guess fails to find the true global minimum

Convexity

For a **convex** function, any local minimum is a global minimum – we avoid the situation where the algorithm converges on some critical point that is not the minimum of the function.

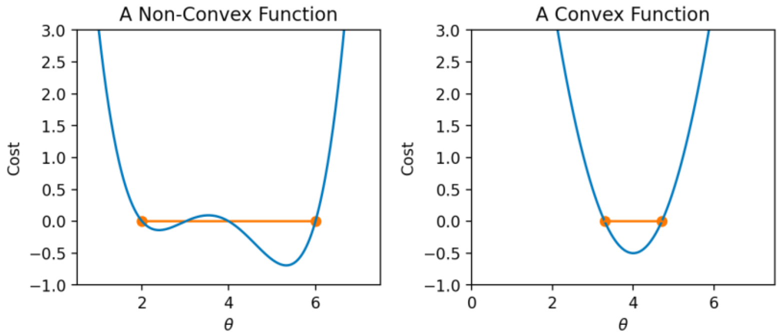
- Our arbitrary function is non-convex
- We will soon see – MSE *is* convex, which is why it is a popular choice of loss function

Algorithm is only guaranteed to converge (given enough iterations and an appropriate step size) for convex functions.

$$tf(a) + (1-t)f(b) \geq f(ta + (1-t)b)$$

$$\text{For all } a, b \text{ in domain of } f \text{ and } t \in [0, 1]$$

In plain English: if I draw a line between any two points on the curve, all values on the curve must be at or below the line.



Gradient Descent on a 1D Model

- Optimization: where are we?
- Minimizing an arbitrary 1D function
- **Gradient descent on a 1D model**
- Gradient descent on high-dimensional models
- Batch, mini-batch, and stochastic gradient descent

From arbitrary functions to loss functions

In a modeling context, we aim to minimize a *loss function* by choosing the minimizing model *parameters*.

Terminology clarification:

- In past lectures, we have used "loss" to refer to the error incurred on a *single* datapoint
- In applications, we usually care more about the average error across *all* datapoints

Going forward, we will take the "model's loss" to mean the model's average error across the dataset. This is sometimes also known as the empirical risk, cost function, or objective function.

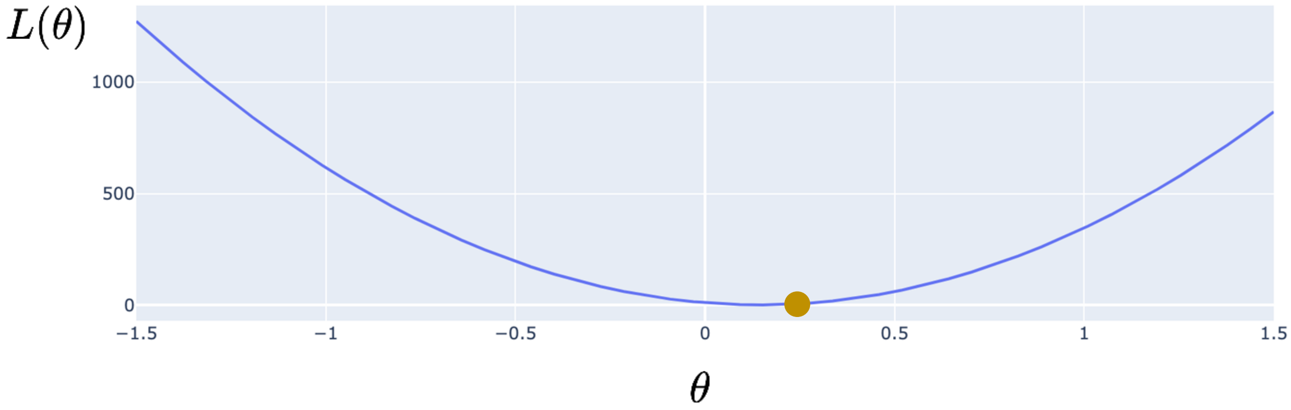
$$L(\theta) = R(\theta) = \frac{1}{n} \sum_{i=1}^{n} l(y, \hat{y})$$

From arbitrary functions to loss functions

In a modeling context, we aim to minimize a *loss function* by choosing the minimizing model *parameters*.

Goal: choose the value of θ that minimizes $L(\theta)$, the model's loss on the dataset

Our new framework:



Compute the model's loss for this choice of parameter

Test several values of the parameter θ

From arbitrary functions to loss functions

Goal: choose the value of θ that minimizes $L(\theta)$, the model's loss on the dataset

The **1D gradient descent** algorithm:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{d}{d\theta} L\left(\theta^{(t)}\right)$$

Take our algorithm from before, replace x with θ and f with L.

Demo: gradient descent on the `tips` dataset

We want to predict the tip (y) given the price of a meal (x). To do this:

- Choose a model: $\hat{y} = \theta_1 x$
- Choose a loss function: $L(\theta) = MSE(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta_1 x_i)^2$

Demo: gradient descent on the `tips` dataset

We want to predict the tip (y) given the price of a meal (x). To do this:

- Choose a model: $\hat{y} = \theta_1 x$
- Choose a loss function: $L(\theta) = MSE(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta_1 x_i)^2$
- Optimize the model parameter – apply gradient descent

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{d}{d\theta} L\left(\theta^{(t)}\right)$$

Demo: gradient descent on the `tips` dataset

- Optimize the model parameter – apply gradient descent

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{d}{d\theta} L\left(\theta^{(t)}\right)$$

Our loss function

$$L(\theta) = MSE(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta_1 x_i)^2$$

Demo: gradient descent on the `tips` dataset

- Optimize the model parameter – apply gradient descent

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{d}{d\theta} L\left(\theta^{(t)}\right)$$

Our loss function

$$L(\theta) = MSE(\theta) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \theta_1 x_i)^2$$

Take the derivative wrt θ_1

$$\frac{d}{d\theta_1} L\left(\theta_1^{(t)}\right) = \frac{-2}{n} \sum_{i=1}^{n} \left(y_i - \theta_1^{(t)} x_i\right) x_i$$

The gradient descent update rule

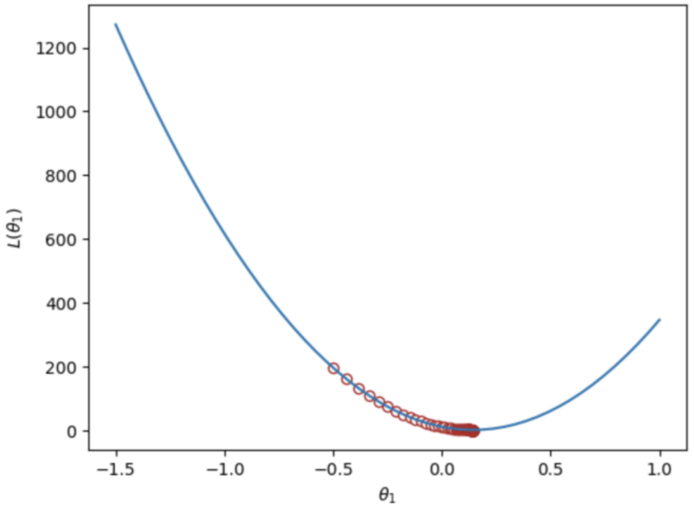
$$\theta_1^{(t+1)} = \theta_1^{(t)} - \alpha \frac{-2}{n} \sum_{i=1}^{n} \left(y_i - \theta_1^{(t)} x_i\right) x_i$$

Demo Slides

Gradient Descent on the `tips` dataset

Loss function: $MSE(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta_1 x_i)^2$

GD update rule: $\theta_1^{(t+1)} = \theta_1^{(t)} - \alpha\frac{-2}{n}\sum_{i=1}^{n}\left(y_i - \theta_1^{(t)}x_i\right)x_i$

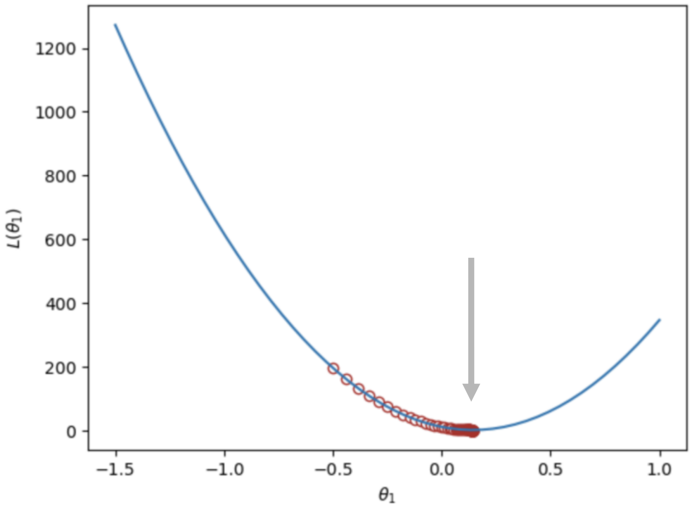


MSE is minimized when we set $\theta_1 = 0.1437$

MSE is convex!

When we visualized the MSE loss on the `tips` data, there was a single global minimum

You will show in Homework #6 that L2 loss is convex – gradient descent will converge to the true minimum (assuming an appropriate choice of learning rate and enough time for convergence)



This is one reason why the MSE is a popular choice of loss function: it behaves "nicely" for optimization

Interlude


Machine learning
Gradient descent
Machine learning


You guys heard about the new perfume for ML engineers?
It's called Grädient Dè Scent


Your algorithm's pov right before it starts gradient descent

Gradient Descent on Multi-Dimensional Models

- Optimization: where are we?
- Minimizing an arbitrary 1D function
- Gradient descent on a 1D model
- **Gradient descent on multi-dimensional models**
- Batch, mini-batch, and stochastic gradient descent

Models in 2D or higher

Usually, models will have more than one parameter that needs to be optimized.

Simple linear regression: $\hat{y} = \theta_0 + \theta_1 x$

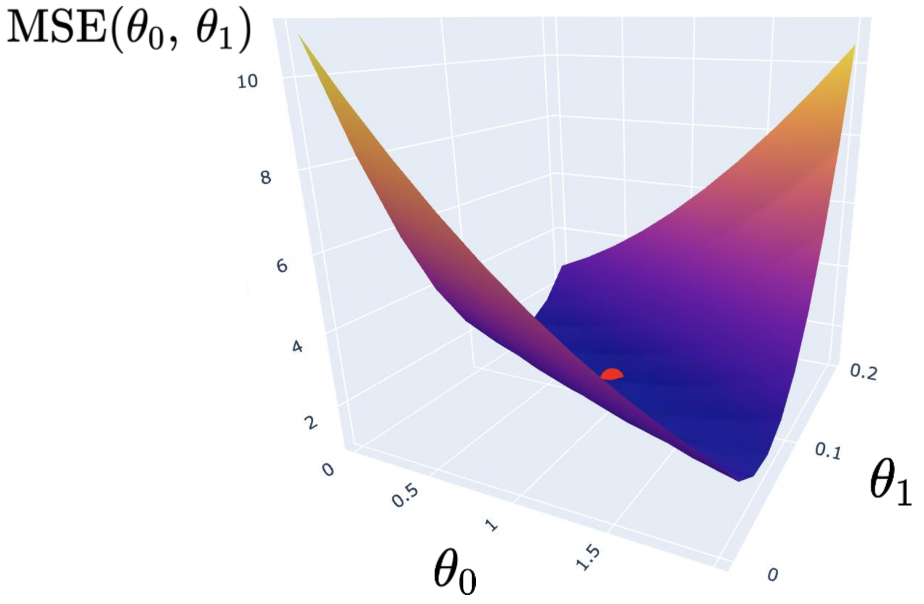
Multiple linear regression: $\hat{\mathbb{Y}} = \theta_0 + \theta_1 \mathbb{X}_{:,1} + \theta_2 \mathbb{X}_{:,\,2} \ldots + \theta_p \mathbb{X}_{:,\,p}$

Idea: expand gradient descent so we can update our guesses for *all* model parameters, all in one go

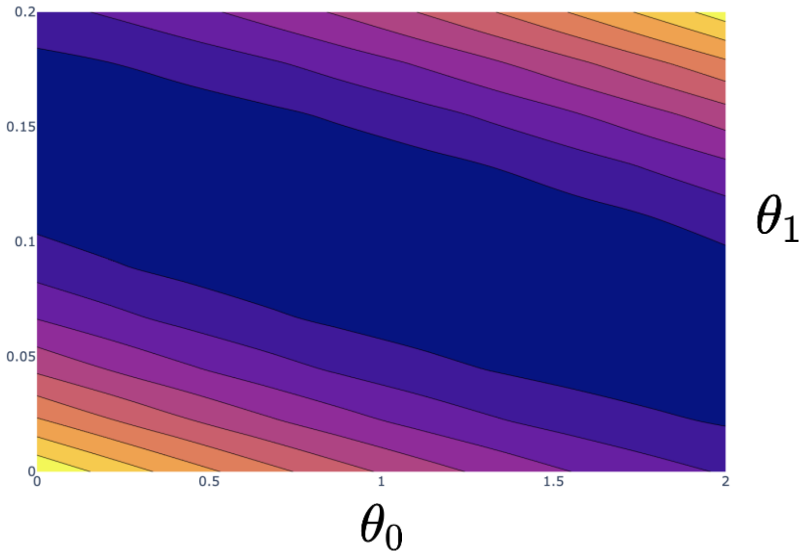
Models in 2D or higher

With multiple parameters to optimize, we consider a **loss surface**

- What is the model's loss for a particular *combination* of possible parameter values?



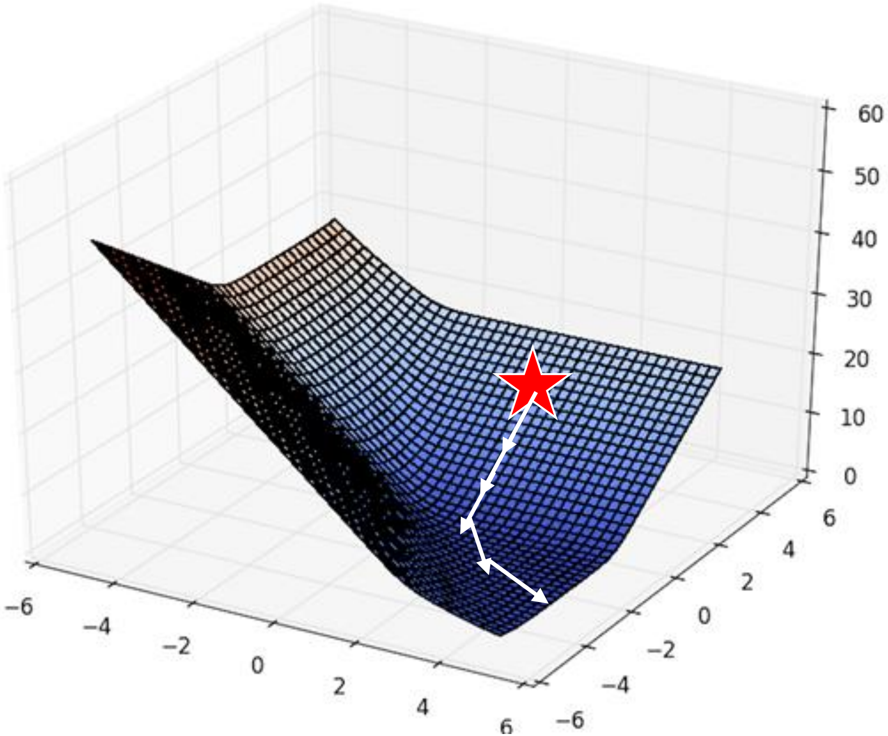
A bird's eye view from above:



The gradient vector

As before, the derivative of the loss function tells us the best way towards the minimum value

On a 2D (or higher) surface, the best way to go down (gradient) is described by a *vector*



For the *vector* of parameter values $\vec{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$

Take the *partial derivative* of loss with respect to each parameter θ_i

A math aside: partial derivatives

For an equation with multiple variables, we take a **partial derivative** by differentiating with respect to just one variable at a time.

Intuitively: how does the function change if we vary one variable, while holding the others constant?

$$f(x, y) = 3x^2 + y$$

Take the partial derivative wrt x: treat y as a constant

$$\frac{\partial f}{\partial x} = 6x$$

Take the partial derivative wrt y: treat x as a constant

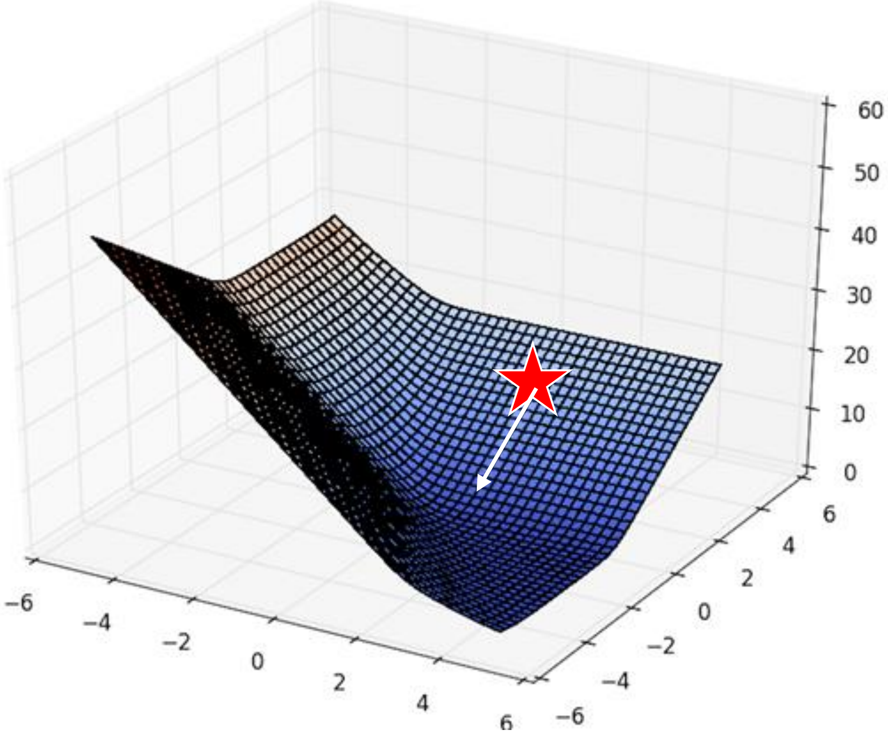
$$\frac{\partial f}{\partial y} = 1$$

This symbol means "partial derivative"

The gradient vector

For the *vector* of parameter values $\vec{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$

Take the *partial derivative* of loss with respect to each parameter: $\dfrac{\partial L}{\partial \theta_0}, \dfrac{\partial L}{\partial \theta_1}$



The **gradient vector** is

$$\nabla_{\vec{\theta}} L = \begin{bmatrix} \frac{\partial L}{\partial \theta_0} \\ \frac{\partial L}{\partial \theta_1} \end{bmatrix}$$

$-\nabla_{\vec{\theta}} L$ always points in the downhill direction of the surface.

Gradient descent in multiple dimensions

Recall our 1D update rule:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{d}{d\theta} L\left(\theta^{(t)}\right)$$

Now, for models with multiple parameters, we work in terms of vectors:

$$\begin{bmatrix} \theta_0^{(t+1)} \\ \theta_1^{(t+1)} \\ \vdots \end{bmatrix} = \begin{bmatrix} \theta_0^{(t)} \\ \theta_1^{(t)} \\ \vdots \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial L}{\partial \theta_0} \\ \frac{\partial L}{\partial \theta_1} \\ \vdots \end{bmatrix}$$

Written in a more compact form:

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \nabla_{\vec{\theta}} L\left(\vec{\theta}^{(t)}\right)$$

Gradient descent update rule

Gradient descent algorithm: nudge θ in negative gradient direction until θ converges.

For a model with multiple parameters:

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \alpha \overbrace{\nabla_{\vec{\theta}} L\left(\vec{\theta}^{(t)}\right)}^{\text{gradient of the loss function evaluated at current } \theta}$$

Next value for **θ**

θ: Model weights

L: loss function

α: Learning rate (ours is constant; other techniques have α decrease over time)

Batch, Mini-Batch, and Stochastic Gradient Descent

- Optimization: where are we?
- Minimizing an arbitrary 1D function
- Gradient descent on a 1D model
- Gradient descent on multi-dimensional models
- **Batch, mini-batch, and stochastic gradient descent**

Batch gradient descent

We have just derived **batch gradient descent.**

- We used our *entire* dataset (as one big batch) to compute gradients
- Recall the derivative of MSE for our 1D model – involves working with *all* n datapoints

$$\frac{d}{d\theta_1} L\left(\theta_1^{(t)}\right) = \frac{-2}{n} \sum_{i=1}^{n} \left(y_i - \theta_1^{(t)} x_i\right) x_i$$

Using all datapoints is often impractical when our dataset is large.

Computing each gradient will take a long time; gradient descent will converge slowly because each individual update is slow.

Mini-batch gradient descent

An alternative: use only a *subset* of the full dataset at each update.

Estimate the true gradient of the loss surface using just this subset of the data.

Batch size: the number of datapoints to use in each subset

In mini-batch GD:

- Compute the gradient on the first x% of the data. Update the parameter guesses.
- Compute the gradient on the next x% of the data. Update the parameter guesses.
- …
- Compute the gradient on the last x% of the data. Update the parameter guesses.

Training Epoch

Mini-batch gradient descent

In mini-batch GD:

- Compute the gradient on the first x% of the data. Update the parameter guesses.
- Compute the gradient on the next x% of the data. Update the parameter guesses.
- …
- Compute the gradient on the last x% of the data. Update the parameter guesses.

Training Epoch

In a single training epoch, we use every datapoint in the data once.

We then perform several training epochs until we are satisfied.

Stochastic gradient descent

In the most extreme case, we may perform gradient descent with a batch size of just *one* datapoint – this is called **stochastic gradient descent**.

- Works surprisingly well in practice! Averaging across several epochs gives a similar result as directly computing the true gradient on all the data.

In stochastic GD:

- Compute the gradient on the first datapoint. Update the parameter guesses.
- Compute the gradient on the next datapoint. Update the parameter guesses.
- …
- Compute the gradient on the last datapoint. Update the parameter guesses.

Training Epoch

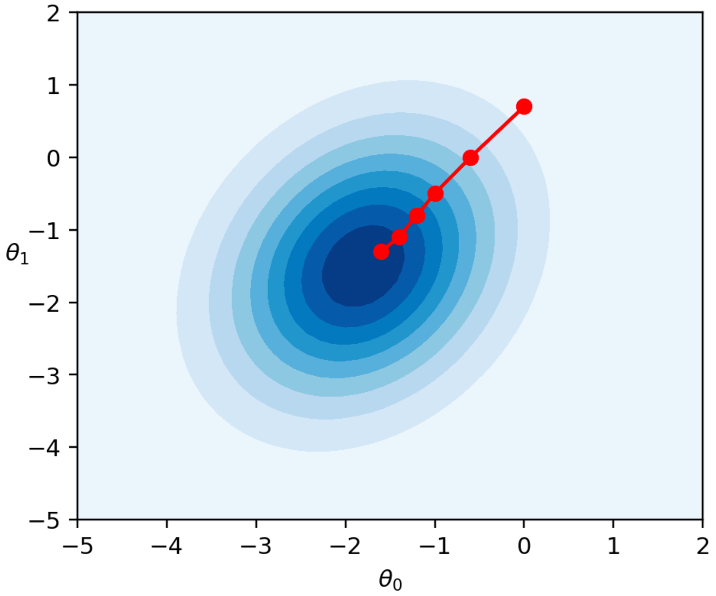
Comparing GD techniques

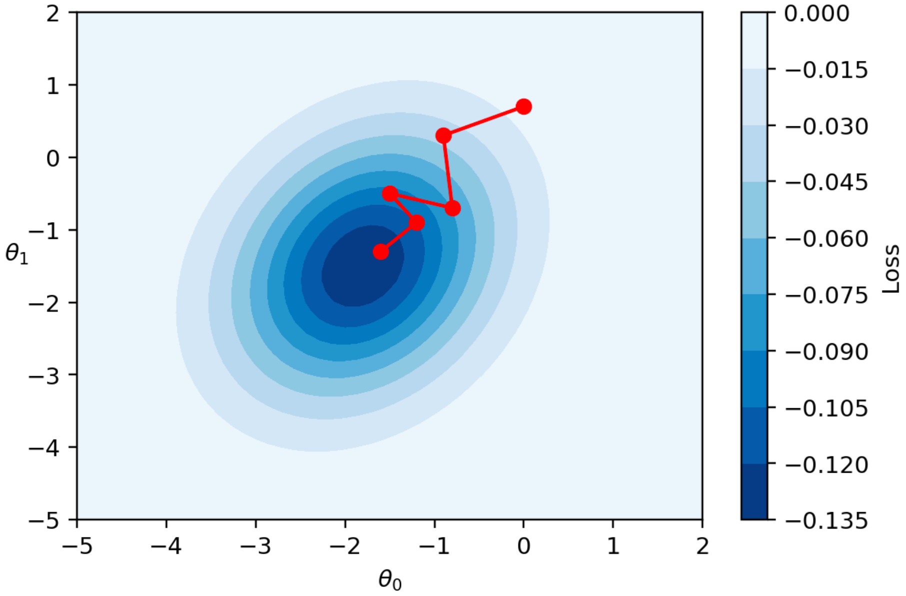
Batch gradient descent:

- Computes the true gradient
- Always descends towards the true minimum of loss

Mini-batch/stochastic gradient descent:

- *Approximates* the true gradient
- May not descend towards the true minimum with each update





LECTURE 12

Gradient Descent