At this point, we've grown quite familiar with the process of choosing a model and a corresponding loss function and optimizing parameters by choosing the values of θ that minimize the loss function. So far, we've optimized θ by

1. Using calculus to take the derivative of the loss function with respect to θ, setting it equal to 0, and solving for θ.
2. Using the geometric argument of orthogonality to derive the OLS solution $\hat{\theta} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{Y}$.

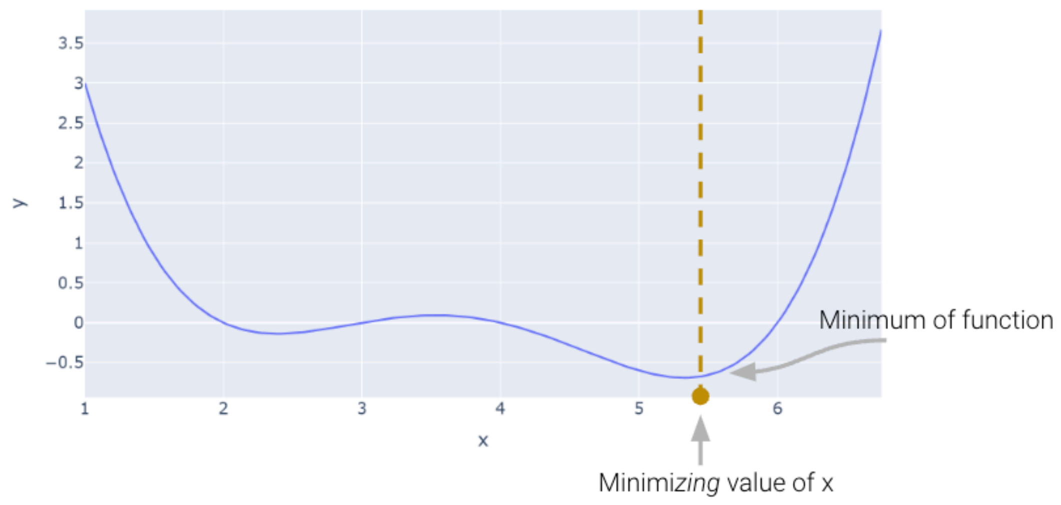
One thing to note, however, is that the techniques we used above can only be applied if we make some big assumptions. For the calculus approach, we assumed that the loss function was differentiable at all points and that we could algebraically solve for the zero points of the derivative; for the geometric approach, OLS *only* applies when using a linear model with MSE loss. What happens when we have more complex models with different, more complex loss functions? The techniques we've learned so far will not work, so we need a new optimization technique: **gradient descent**.

> **BIG IDEA**: use an iterative algorithm to numerically compute the minimum of the loss.

13.3.1 Minimizing an Arbitrary 1D Function

Let's consider an arbitrary function. Our goal is to find the value of x that minimizes this function.

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



13.3.1.1 The Naive Approach: Guess and Check

Above, we saw that the minimum is somewhere around 5.3. Let's see if we can figure out how to find the exact minimum algorithmically from scratch. One very slow (and terrible) way would be manual guess-and-check.

```python
arbitrary(6)
```

```
0.0
```

A somewhat better (but still slow) approach is to use brute force to try out a bunch of x values and return the one that yields the lowest loss.

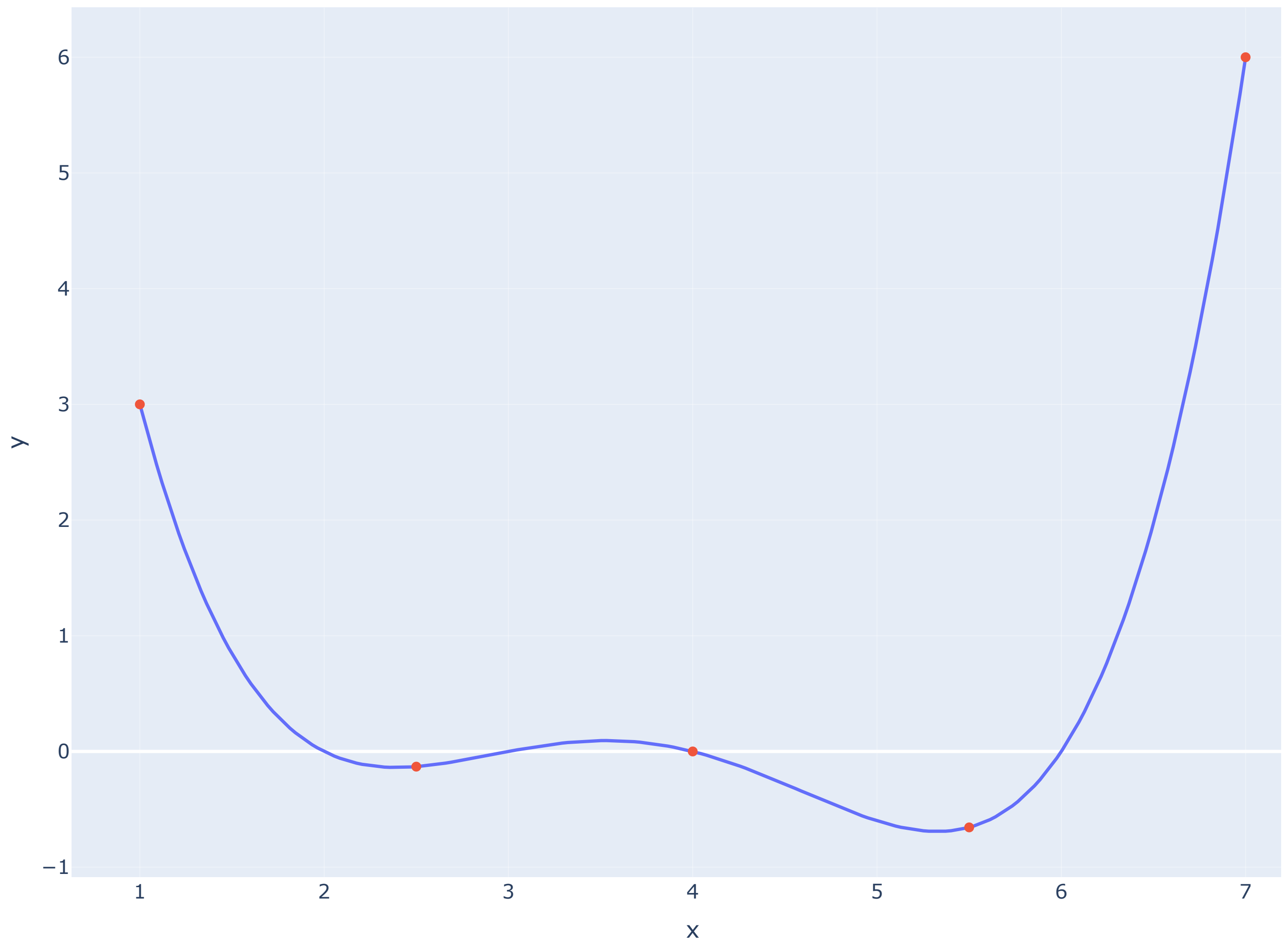
```python
def simple_minimize(f, xs):
    # Takes in a function f and a set of values xs.
    # Calculates the value of the function f at all values x in xs
    # Takes the minimum value of f(x) and returns the corresponding value x
    y = [f(x) for x in xs]
    return xs[np.argmin(y)]

guesses = [5.3, 5.31, 5.32, 5.33, 5.34, 5.35]
simple_minimize(arbitrary, guesses)
```

```
5.33
```

This process is essentially the same as before where we made a graphical plot, it's just that we're only looking at 20 selected points.

▶ Code



This basic approach suffers from three major flaws:

1. If the minimum is outside our range of guesses, the answer will be completely wrong.
2. Even if our range of guesses is correct, if the guesses are too coarse, our answer will be inaccurate.
3. It is *very* computationally inefficient, considering potentially vast numbers of guesses that are useless.

13.3.1.2 `Scipy.optimize.minimize`

One way to minimize this mathematical function is to use the `scipy.optimize.minimize` function. It takes a function and a starting guess and tries to find the minimum.

```
from scipy.optimize import minimize

# takes a function f and a starting point x0 and returns a readout
# with the optimal input value of x which minimizes f
minimize(arbitrary, x0 = 3.5)
```

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: -0.13827491292966557
        x: [ 2.393e+00]
      nit: 3
      jac: [ 6.486e-06]
 hess_inv: [[ 7.385e-01]]
     nfev: 20
     njev: 10
```

`scipy.optimize.minimize` is great. It may also seem a bit magical. How could you write a function that can find the minimum of any mathematical function? There are a number of ways to do this, which we'll explore in today's lecture, eventually arriving at the important idea of **gradient descent**, which is the principle that `scipy.optimize.minimize` uses.

It turns out that under the hood, the `fit` method for `LinearRegression` models uses gradient descent. Gradient descent is also how much of machine learning works, including even advanced neural network models.

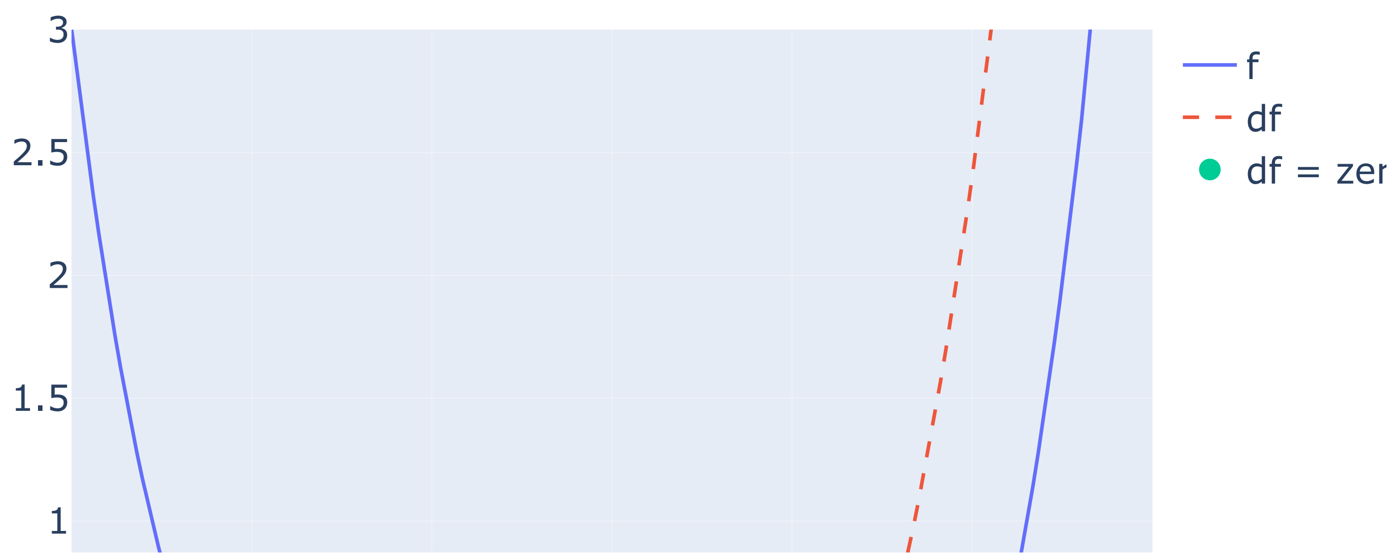
In Data 100, the gradient descent process will usually be invisible to us, hidden beneath an abstraction layer. However, to be good data scientists, it's important that we know the underlying principles that optimization functions harness to find optimal parameters.

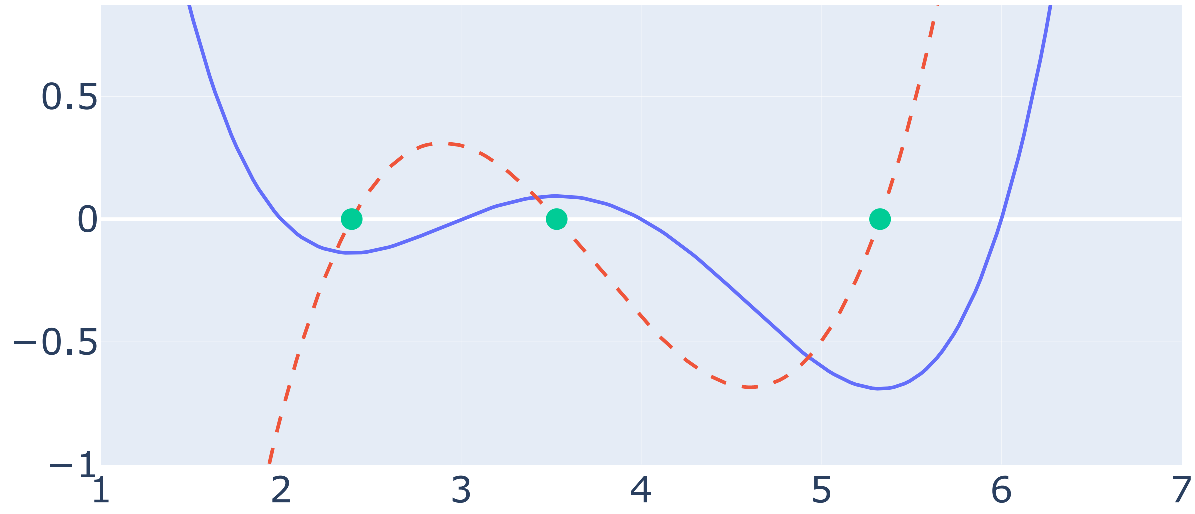
13.3.1.3 Digging into Gradient Descent

Looking at the function across this domain, it is clear that the function's minimum value occurs around $\theta = 5.3$. Let's pretend for a moment that we *couldn't* see the full view of the cost function. How would we guess the value of θ that minimizes the function?

It turns out that the first derivative of the function can give us a clue. In the graph below, the function and its derivative are plotted, with points where the derivative is equal to 0 plotted in light green.

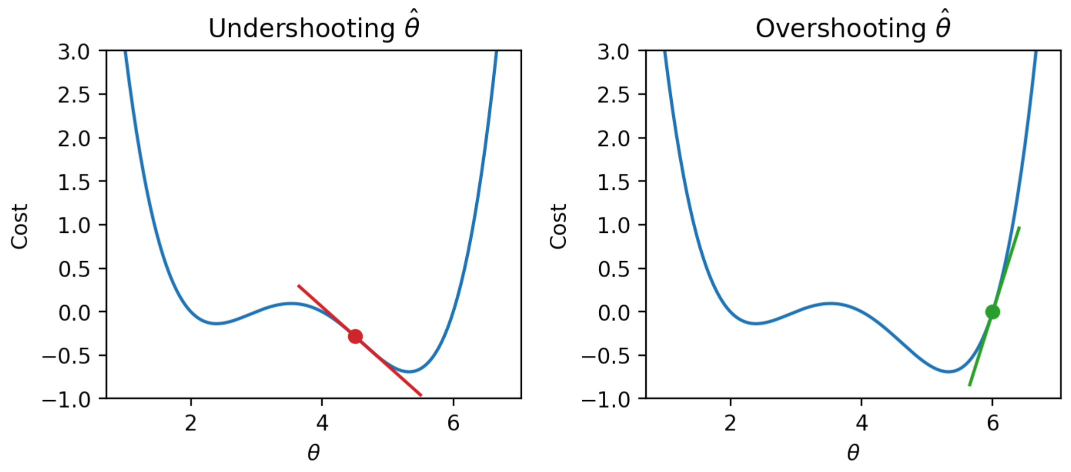
▶ Code



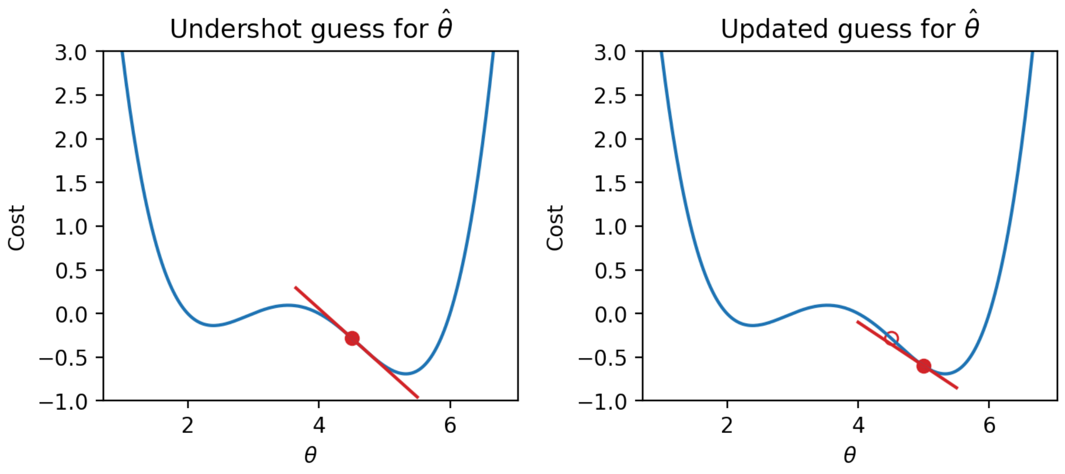


In the plots below, the line indicates the value of the derivative of each value of θ. The derivative is negative where it is red and positive where it is green.

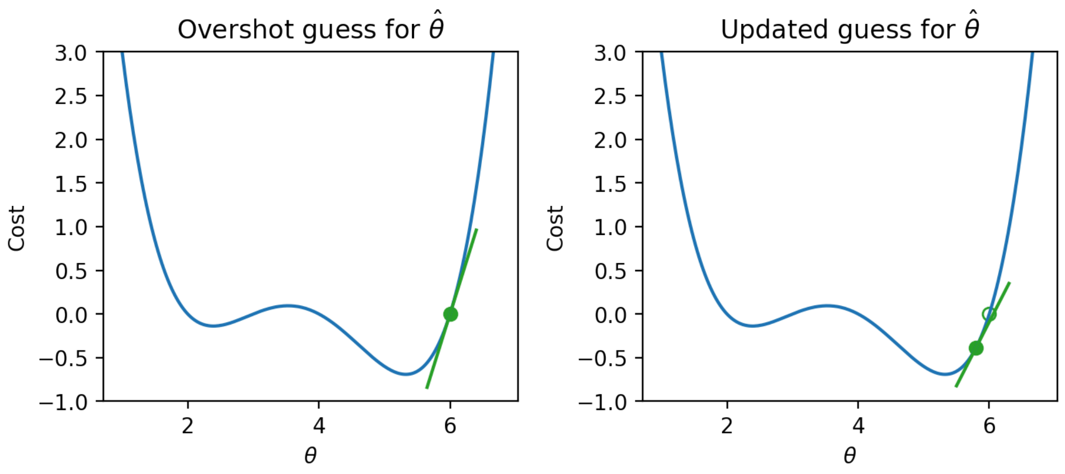
Say we make a guess for the minimizing value of θ. Remember that we read plots from left to right, and assume that our starting θ value is to the left of the optimal $\hat{\theta}$. If the guess "undershoots" the true minimizing value – our guess for θ is lower than the value of the $\hat{\theta}$ that minimizes the function – the derivative will be **negative**. This means that if we increase θ (move further to the right), then we **can decrease** our loss function further. If this guess "overshoots" the true minimizing value, the derivative will be positive, implying the converse.



We can use this pattern to help formulate our next guess for the optimal $\hat{\theta}$. Consider the case where we've undershot θ by guessing too low of a value. We'll want our next guess to be greater in value than our previous guess – that is, we want to shift our guess to the right. You can think of this as following the slope "downhill" to the function's minimum value.



If we've overshot $\hat{\theta}$ by guessing too high of a value, we'll want our next guess to be lower in value – we want to shift our guess for $\hat{\theta}$ to the left.



In other words, the derivative of the function at each point tells us the direction of our next guess.

- A negative slope means we want to step to the right, or move in the *positive* direction.
- A positive slope means we want to step to the left, or move in the *negative* direction.

13.3.1.4 Algorithm Attempt 1

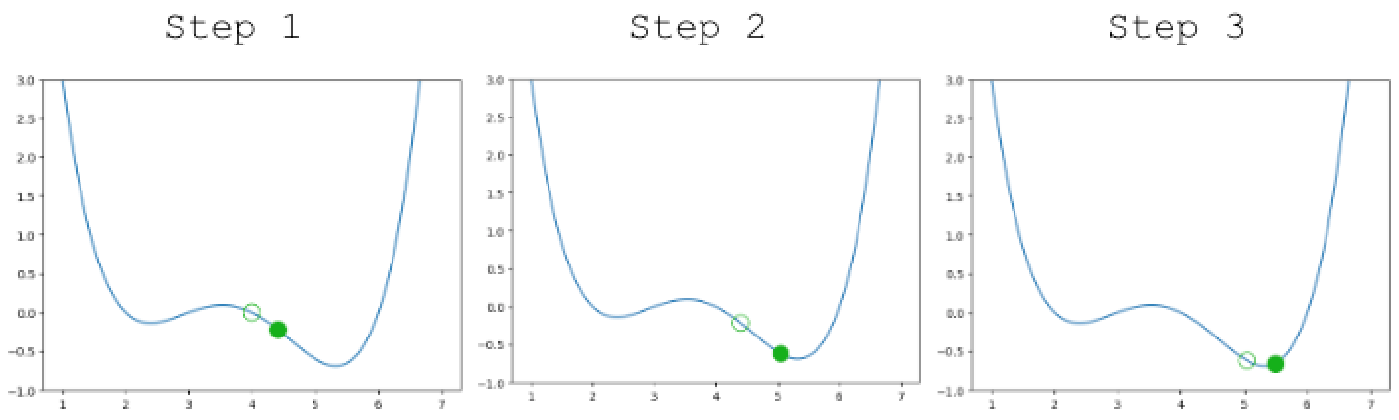
Armed with this knowledge, let's try to see if we can use the derivative to optimize the function.

We start by making some guess for the minimizing value of x. Then, we look at the derivative of the function at this value of x, and step downhill in the *opposite* direction. We can express our new rule as a recurrence relation:

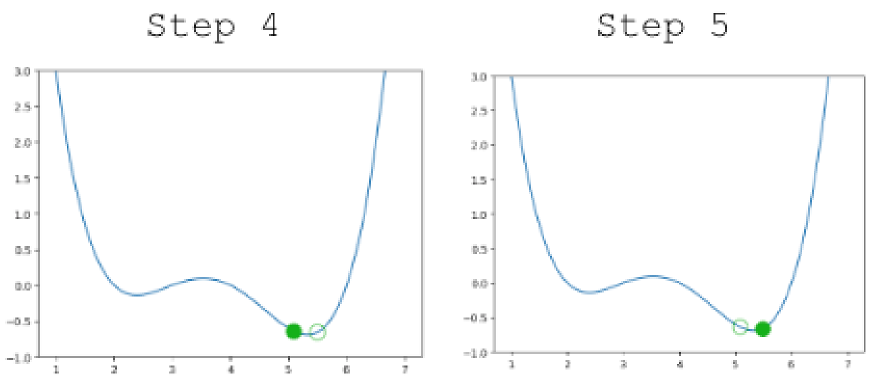
$$x^{(t+1)} = x^{(t)} - \frac{d}{dx} f(x^{(t)})$$

Translating this statement into English: we obtain **our next guess** for the minimizing value of x at timestep $t+1$ ($x^{(t+1)}$) by taking **our last guess** ($x^{(t)}$) and subtracting the **derivative of the function** at that point ($\frac{d}{dx} f(x^{(t)})$).

A few steps are shown below, where the old step is shown as a transparent point, and the next step taken is the green-filled dot.



Looking pretty good! We do have a problem though – once we arrive close to the minimum value of the function, our guesses "bounce" back and forth past the minimum without ever reaching it.



In other words, each step we take when updating our guess moves us too far. We can address this by decreasing the size of each step.

13.3.1.5 Algorithm Attempt 2

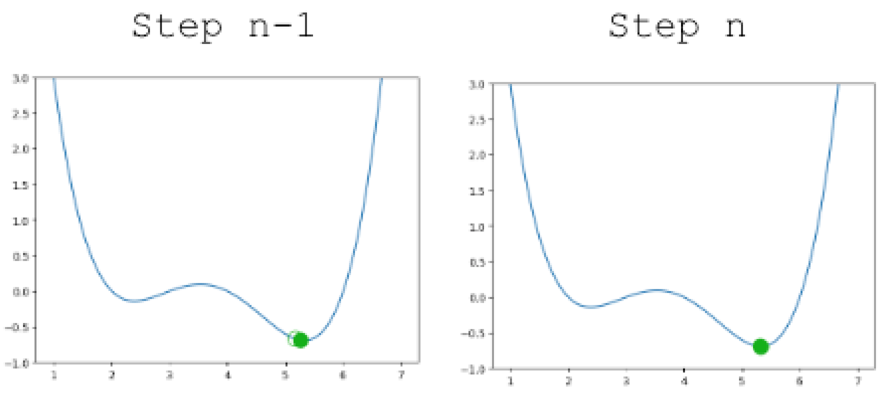
Let's update our algorithm to use a **learning rate** (also sometimes called the step size), which controls how far we move with each update. We represent the learning rate with α.

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

A small α means that we will take small steps; a large α means we will take large steps. When do we stop updating? We stop updating either after a fixed number of updates or after a subsequent update doesn't change much.

> **Note**
>
> In Data 100, the learning rate is constant. More generally, however, α could also decrease over time, or decay.

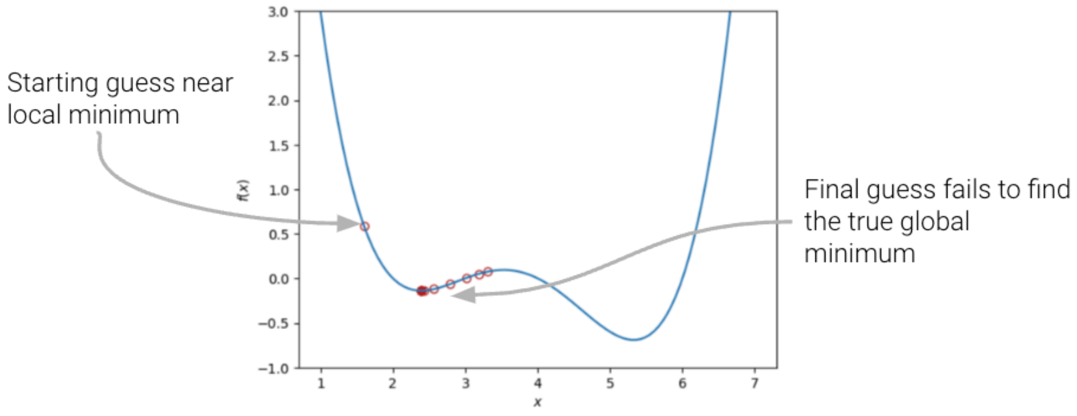
Updating our function to use $\alpha = 0.3$, our algorithm successfully **converges** (settles on a solution and stops updating significantly, or at all) on the minimum value.



13.3.2 Convexity

In our analysis above, we focused our attention on the global minimum of the loss function. You may be wondering: what about the local minimum that's just to the left?

If we had chosen a different starting guess for θ, or a different value for the learning rate α, our algorithm may have gotten "stuck" and converged on the local minimum, rather than on the true optimum value of loss.

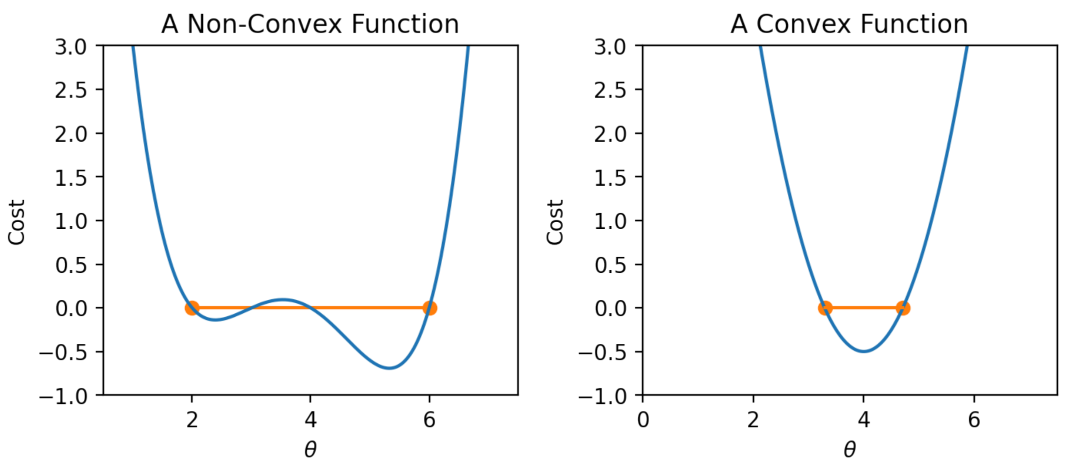


If the loss function is **convex**, gradient descent is guaranteed to converge and find the global minimum of the objective function. Formally, a function f is convex if:

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

for all a, b in the domain of f and $t \in [0, 1]$.

To put this into words: if you drew a line between any two points on the curve, all values on the curve must be *on or below* the line. Importantly, any local minimum of a convex function is also its global minimum so we avoid the situation where the algorithm converges on some critical point that is not the minimum of the function.



In summary, non-convex loss functions can cause problems with optimization. This means that our choice of loss function is a key factor in our modeling process. It turns out that MSE *is* convex, which is a major reason why it is such a popular choice of loss function. Gradient descent is only guaranteed to converge (given enough iterations and an appropriate step size) for convex functions.

13.3.3 Gradient Descent in 1 Dimension

> **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 (R), cost function, or objective function.
>
> $$L(\theta) = R(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(y, \hat{y})$$

In our discussion above, we worked with some arbitrary function f. As data scientists, we will almost always work with gradient descent in the context of optimizing *models* – specifically, we want to apply gradient descent to find the minimum of a *loss function*. In a modeling context, our goal is to minimize a loss function by choosing the minimizing model *parameters*.

Recall our modeling workflow from the past few lectures:

1. Define a model with some parameters θ_i
2. Choose a loss function
3. Select the values of θ_i that minimize the loss function on the data

Gradient descent is a powerful technique for completing this last task. By applying the gradient descent algorithm, we can select values for our parameters θ_i that will lead to the model having minimal loss on the training data.

When using gradient descent in a modeling context, we:

1. Make guesses for the minimizing θ_i
2. Compute the derivative of the loss function L

We can "translate" our gradient descent rule from before by replacing x with θ and f with L:

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

13.3.3.1 Gradient Descent on the `tips` Dataset

To see this in action, let's consider a case where we have a linear model with no offset. We want to predict the tip (y) given the price of a meal (x). To do this, we

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

Let's apply our `gradient_descent` function from before to optimize our model on the `tips` dataset. We will try to select the best parameter θ_i to predict the `tip` y from the `total_bill` x.

```
df = sns.load_dataset("tips")
df.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

We can visualize the value of the MSE on our dataset for different possible choices of θ_1. To optimize our model, we want to select the value of θ_1 that leads to the lowest MSE.

To apply gradient descent, we need to compute the derivative of the loss function with respect to our parameter θ_1.

- Given our loss function,

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

- We take the derivative with respect to θ_1

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

- Which results in the gradient descent update rule

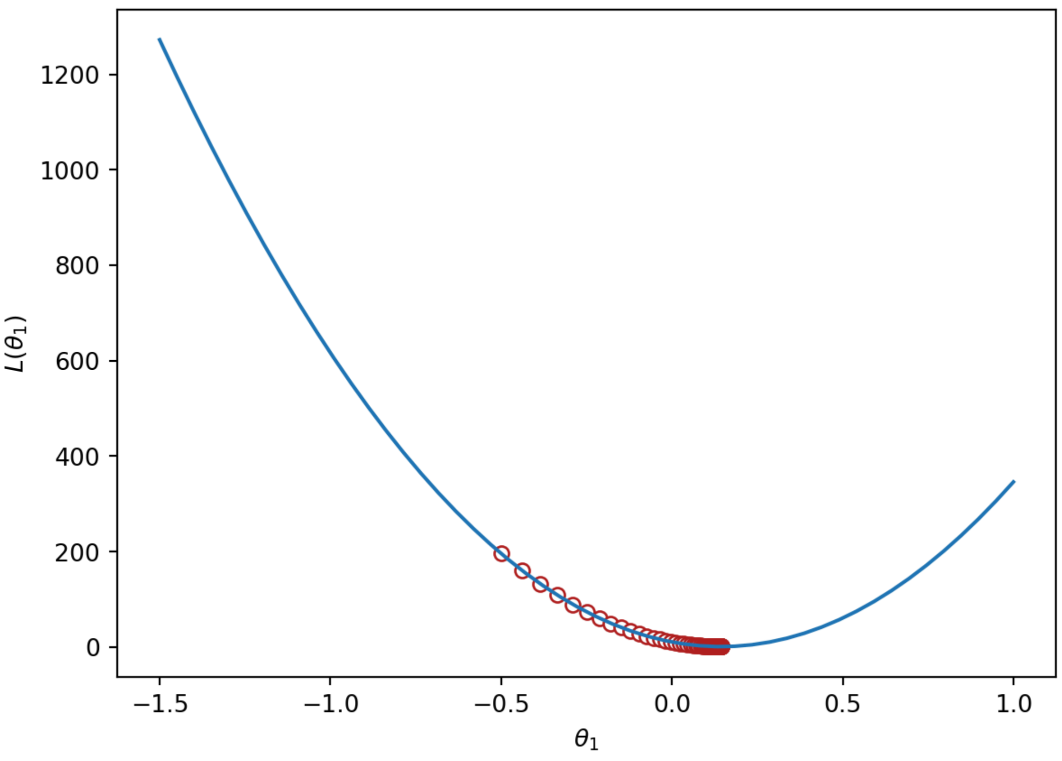
$$\theta_1^{(t+1)} = \theta_1^{(t)} - \alpha \frac{d}{d\theta} L(\theta_1^{(t)})$$

for some learning rate α.

Implementing this in code, we can visualize the MSE loss on the `tips` data. **MSE is convex**, so there is one global minimum.

▶ Code

```
Final guess for theta_1: 0.14369554654231262
```


$L(\theta_1)$
θ_1
1200
1000
800
600
400
200
0
−1.5
−1.0
−0.5
0.0
0.5
1.0