Module 15: Gradient Descent and Optimization

Quick Reference Guide

Learning Outcomes:

1. Compute the approximate value of a function using its derivative.
2. Identify the basics of gradient descent and the impact of learning rate on convergence.
3. Compute x and/or y for each iteration of gradient descent.
4. Optimize a single-parameter linear regression model from scratch.
5. Recognize convex one-dimensional and two-dimensional functions.
6. Compute the gradient of a two-dimensional function.
7. Measure the impurity of a decision tree using entropy.
8. Use gradient descent to optimize a nonlinear two-dimensional regression model.
9. Use stochastic gradient descent to optimize a nonlinear two-dimensional regression model.
10. Compare the convergence behavior of gradient descent with stochastic gradient descent.
11. Articulate the impact of stochastic gradient on bias and variance.

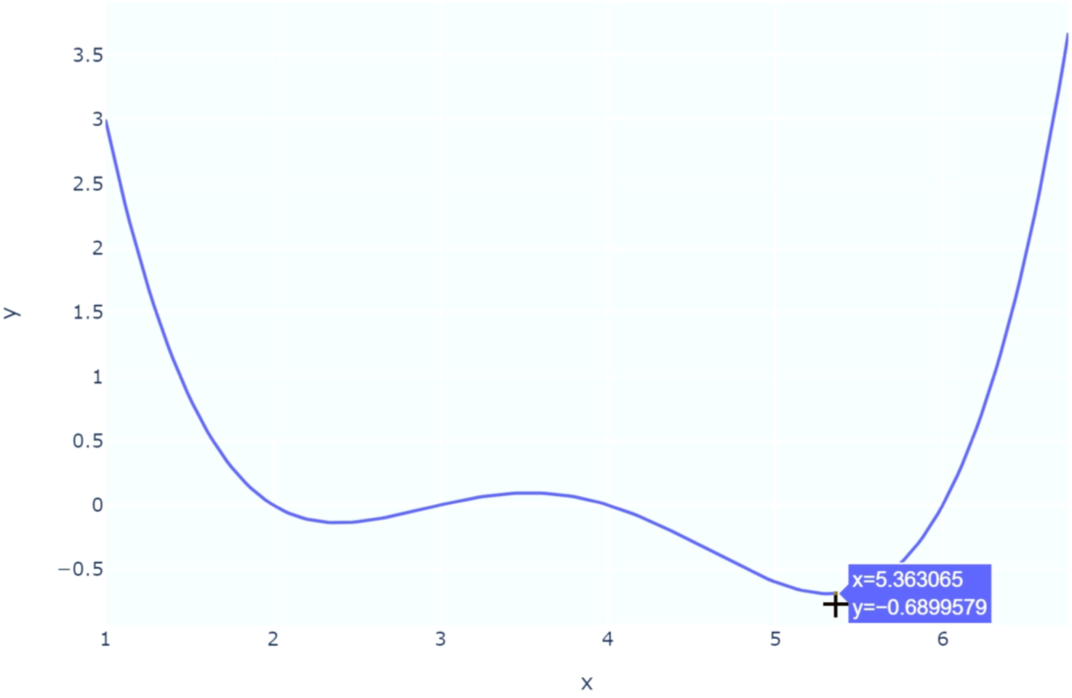
Minimizing an Arbitrary Function Using Guess and Check

Gradient descent is a technique that is used to **minimize functions**. You will start by looking at the contexts of functions in general.

After importing some libraries, you are going to define this arbitrary function of x.

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

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

When you plot this function, you see that it has a nice shape.



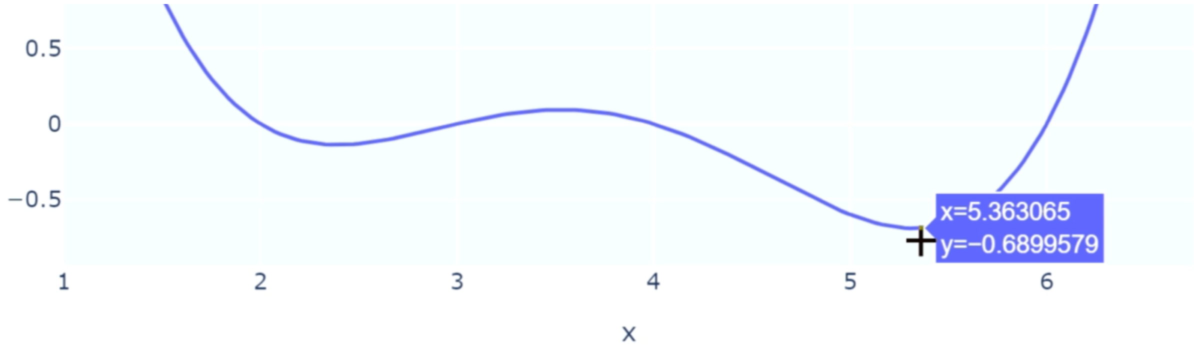
So, this function is a fourth-order function that has two local minima. What you are trying to do is find the x that minimizes this function, which you can see is at approximately 5.36. But there is also a local minimum. So how does this relate to machine learning? Recall that you are often trying to find parameters which find the minima of loss surfaces. This is a stand-in for the moment, for a loss surface you might have for linear or logistic regression.

In scikit-learn, the **scipy.optimize.minimize** function takes a function and a starting guess, and then tries to find the minimum. So, for example, if it minimizes an arbitrary function at a starting guess of 6, the minimum is 5.326.

from scipy.optimize import minimize

minimize(arbitrary, x0 = 6)

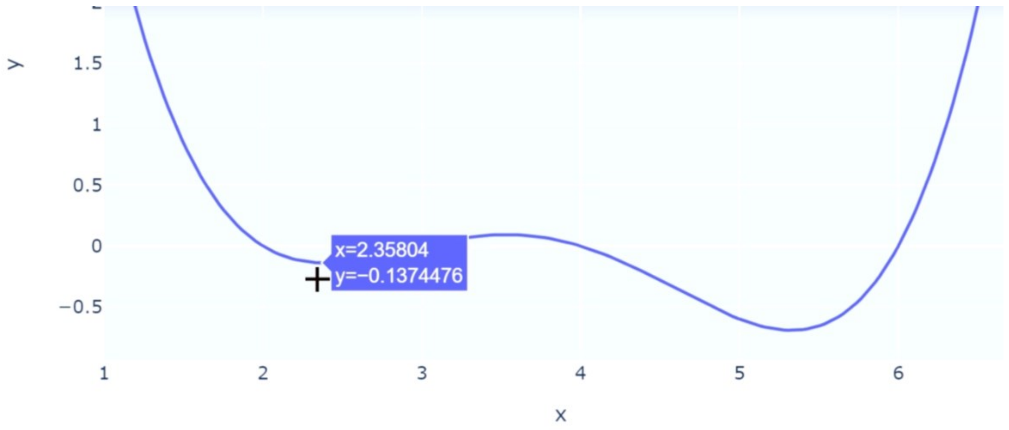
So, looking at the resulting graph, you find the bottom of the curve at 5.326.



Whatever starting point you give the **scipy.optimize.minimize** function matters. So looking at the graph, if you start at 1, it is actually going to get stuck at the first minimum.

minimize(arbitrary, x0 = 1)

If you run this, x is 2.392.



This minimum is a local minimum, but it is not the global minimum. Recall, **scipy.optimize** is using a version of gradient descent. It is going down the slope and trying to find the bottom point. Now you can try minimize it from scratch, using the **guess and check approach**. So, you could try 6 as the starting guess, the result is 0.0. What happens if you try 4.8? The result is around −0.5.

You can keep doing that over and over until you get the smallest number possible. Eventually, you will find the minimum somewhere around 5.3.

```
arbitrary(5.3)
```

```
-0.6906899999999995
```

So, that is one approach. That is not what gradient descent does, or what **scipy.optimize.minimize** does, but it is something you could do.

Another approach is creating a function called **simple_minimize**.

```
def simple_minimize(f, xs):
    y = [f(x) for x in xs]
    return xs[np.argmin(y)]
```

It computes a number of y-values and then gives you the argument that results in the smallest y-value.

So for this arbitrary function, you are plugging in the values between 1 and 7, and you are trying 20 different such values.

```
simple_minimize(arbitrary, np.linspace(1, 7, 20))
```

The result of this function is:

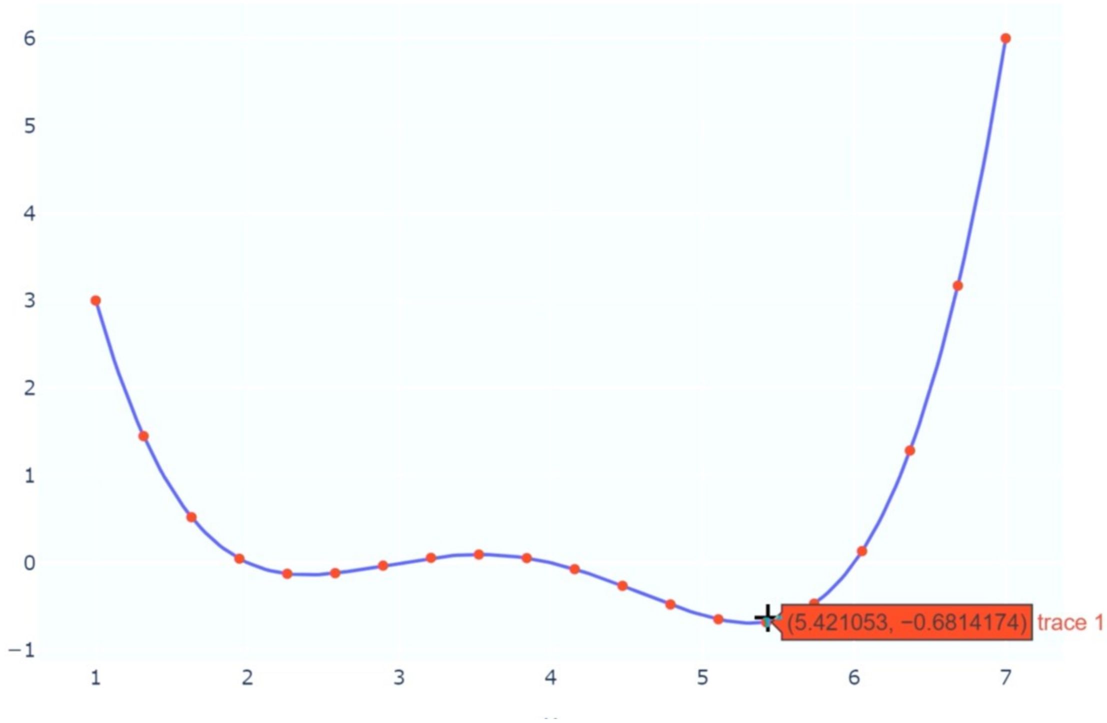
```
5.421052631578947
```

This is basically the manual guess-and-check approach you did earlier, except now you have automated the procedure of trying out a bunch of values. The result of this function, 5.42, is fairly close to the true minimum.

Now, as another way to think about what just happened, you can actually do a plot. So, this code plots the function.

```
xs = np.linspace(1, 7, 200)
sparse_xs = np.linspace(1, 7, 20)

ys = arbitrary(xs)
sparse_ys = arbitrary(sparse_xs)

fig = px.line(x = xs, y = arbitrary(xs))
fig.add_scatter(x = sparse_xs, y = arbitrary(sparse_xs), mode = "markers")
fig.update_layout(showlegend= False)
fig.show()
```

If you do that, you are plotting the function in blue and plotting all the different guesses in red. Basically, it is computing all the red dots on the plot and then telling you which one was the smallest.



This basic approach suffers from three major flaws:

- If the minimum is outside of your range of guesses, the answer will be completely wrong.
- Even if your range of guesses is correct, if the guesses are too coarse, your answer will be inaccurate.
- It is absurdly computationally inefficient, considering potentially vast numbers of guesses that are useless.

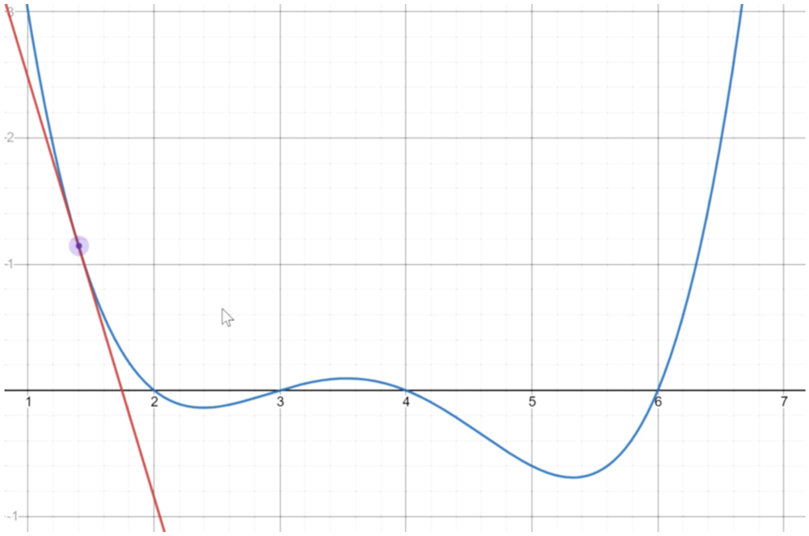
But it is a starting point that you can build on as you move forward.

Visualizing the Derivative

As an alternative to the guess-and-check approach, you can use the **derivative of the function** you are trying to optimize as a guide. You just start with derivatives of one-dimensional functions.

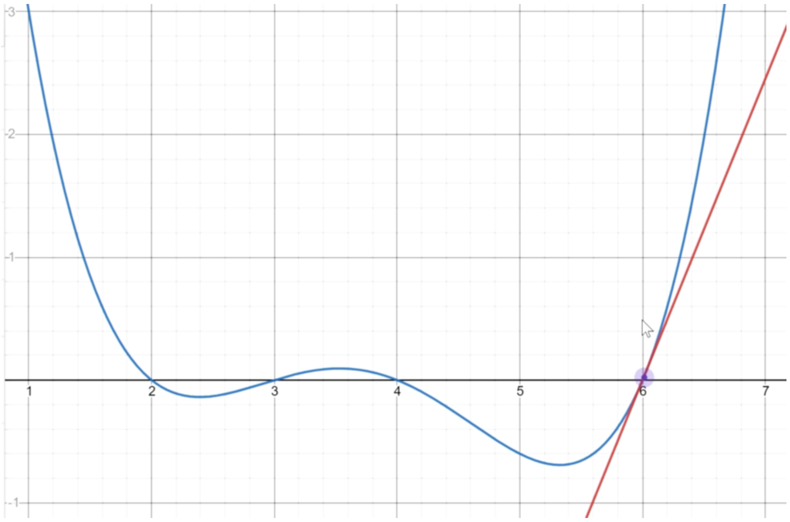
If the **derivative of the function is negative**, then that means the function is decreasing. So, you should go to the right and pick a bigger x value. And, if the **derivative of the function is positive**, that means the function is increasing. So, you should go to the left and pick a smaller x.

You can explore that visually to make it clearer.

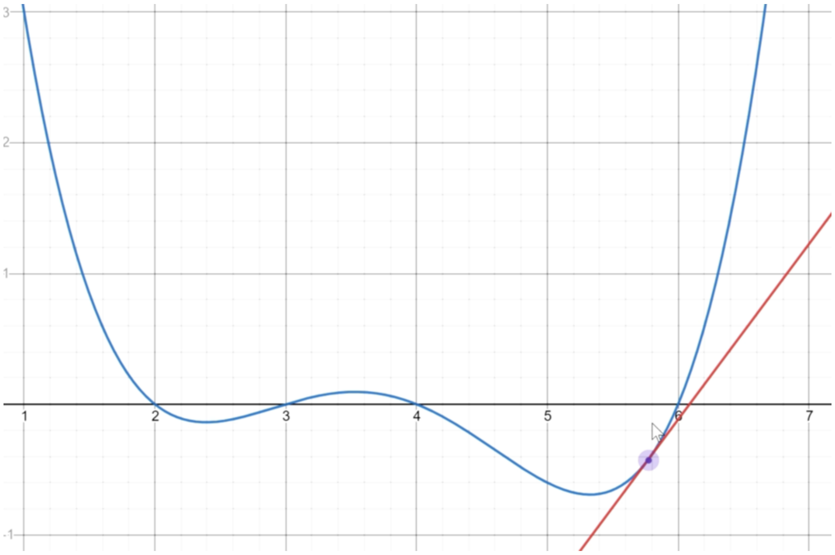


In this plot, the arbitrary function is in blue and the red line is the derivative, the instantaneous derivative, of the function. So, you are looking at the slope of the function. If you have a guess of 1.4, the derivative is negative, that is, the curve is decreasing. That means if you are trying to find the minimum, you should move to the right. You should go into the positive x-direction.

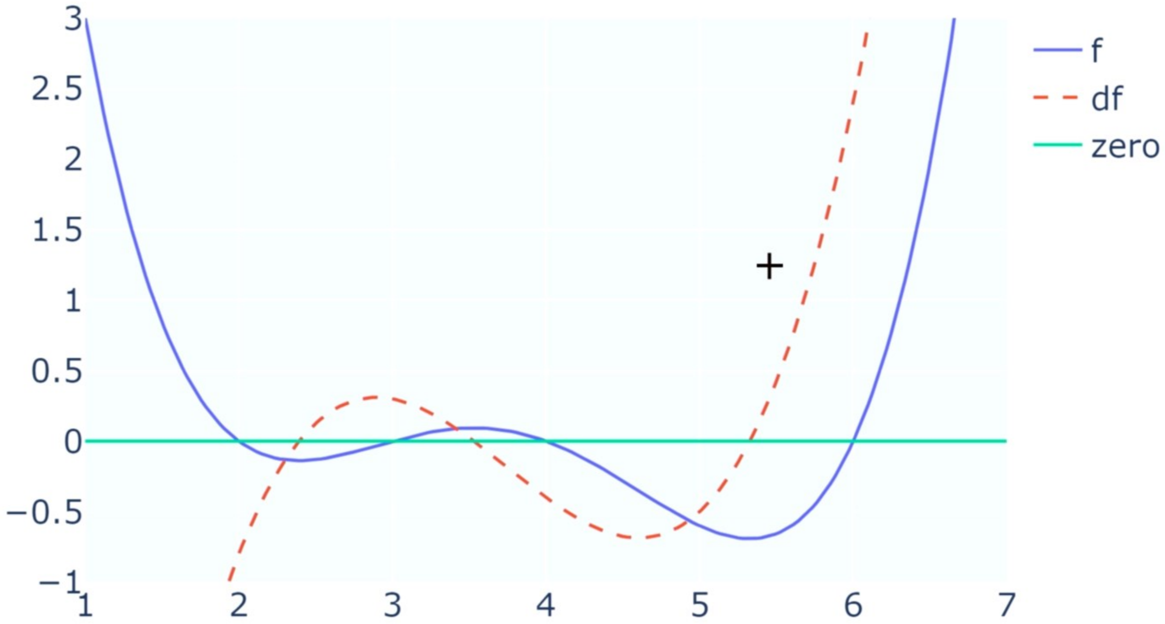
By contrast, if you have a guess of 6, the slope is positive, which means you should pick a smaller x if you want to work your way toward the minimum.



Using a guess of 5.77 is the insight that you are going to need in order to optimize your function.



Now, there is another way of thinking about it. This is going to be the entire derivative of your arbitrary function.



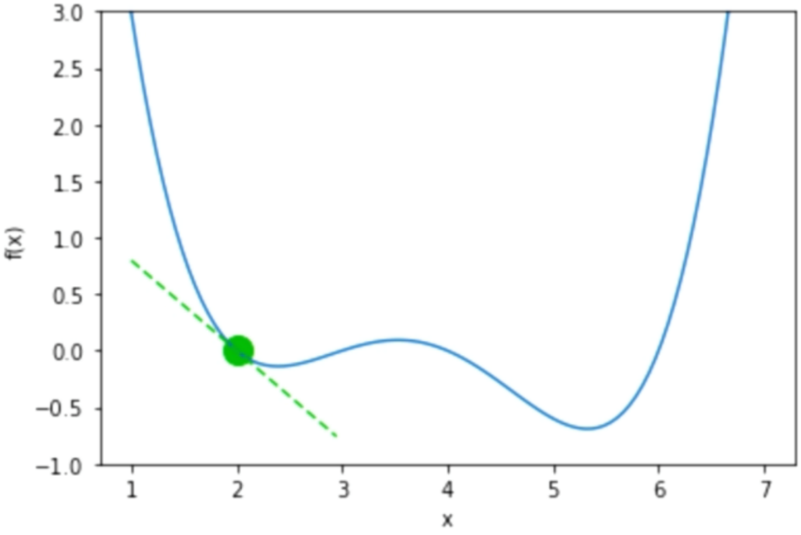
The function you are trying to optimize is in blue. The red dotted line is the entire derivative of the function. The green line highlights 0, so you can see where the zero crossings are. If you look at the derivative of the function, you can see that there are three places where the derivative crosses 0.

So, when the blue curve is flat, the red dotted curve is 0 because it is the derivative of the blue curve. So, it is at each of those points where you have a crossing of 0 that you can have minima or maxima of your functions. The bottom or the tops of your functions will be where the derivative is 0. So, you can try and use derivative descent to go down the curve until you reach a point where the derivative is 0. At that point you are done.

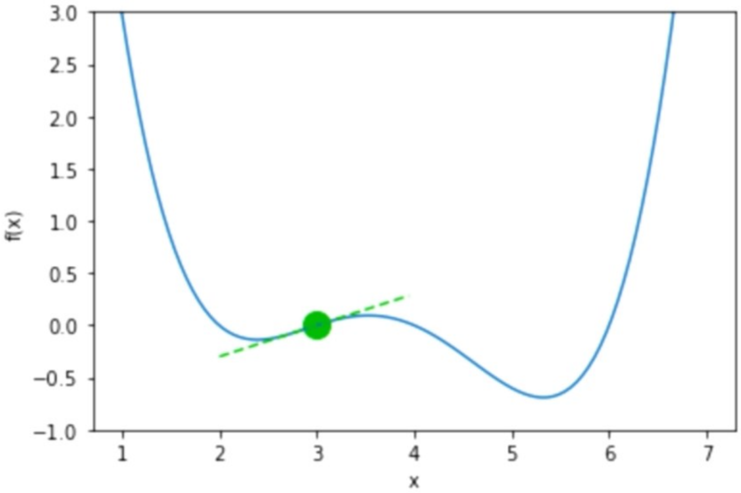
As another way of looking at it: You can use code to plot your function and its instantaneous derivatives in various locations.

plot_arbitrary()
plot_x_on_f(arbitrary, 2)

```
plot_tangent_on_f(arbitrary, 2)
plt.xlabel(‘x’)
plt.ylabel(‘f(x)’);
```



You can look at the function at a particular point. So, if you want to know what the function is at position 3 and what its tangent is, you amend that value in the code and the plot will show it to you.



You might find this to be a useful guide for visualizing what your procedure is going to be doing. Namely, if the derivative is positive, you are going to be picking a smaller x-value.

Manually Descending the Gradient

Once you know that the derivative will inform your next guess, you can try and do this in code. So, here is a print statement where you start to guess.

```
guess = 4
print(f"x: {guess}, f(x): {arbitrary(guess)}, derivative f'(x):
{derivative_arbitrary(guess)}")
```

So in this instance, you are going to print out your guess, the function at your guess, and the derivative of your function at your guess.

This is the derivative of the arbitrary function:

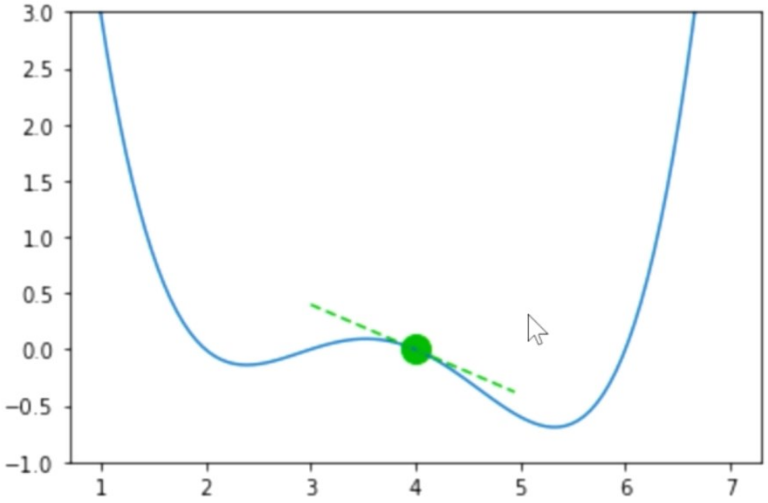
```
def derivative_arbitrary(x):

    return (4*x**3 – 45*x**2 + 160*x – 180)/10
```

When you run this, you see that at 4, the value is 0 and the derivative is negative 0.4.

```
x: 4, f(x): 0.0, derivative f'(x): -0.4
```

You can visualize that by changing the value in your code to 4.

```
plot_x_on_f(arbitrary, 4)
plot_tangent_on_f(arbitrary, 4)
```



You can see the slope is slightly negative. The derivative is giving me a sense of which way to go and a sense of how much to go. Because if the derivative is very steep, you should go further because you are very far away from a minimum. The next guess is 4 + 0.4.

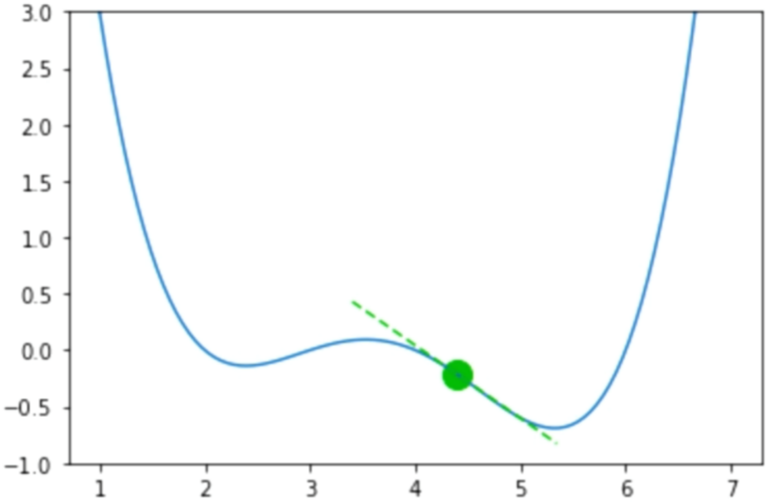
```
guess = 4 + 0.4
print(f"x: {guess}, f(x): {arbitrary(guess)}, derivative f'(x):
{derivative_arbitrary(guess)}")
```

```
x: 4.4, f(x): -0.21504000000003315, derivative f'(x): -
0.6464000000000055
```

At this point, f(x) is smaller, and the derivative is actually even slightly steeper. So, you can go and change the value in the code to 4.4.

```
plot_arbitrary()
plot_x_on_f(arbitrary, 4.4)
plot_tangent_on_f(arbitrary, 4.4)
```

As you can see, the slope is a little bit steeper.



There is one more function you can use to make this process a little more obvious to follow, **plot_one_step**.

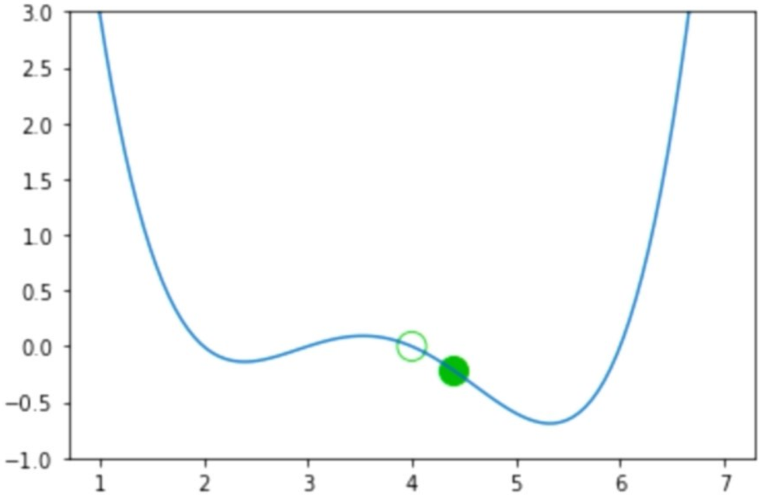
```
def plot_one_step(x):
    new_x = x – derivative_arbitrary(x)
    plot_arbitrary()
    plot_x_on_f(arbitrary, new_x)
    print(f’old x: {x}’)
    print(f’new x: {new_x}’)
```

So, the **plot_one_step** function is going to take a guess. It is going to compute the f(x) and the derivative of f'(x). Then using that, it is going to compute the next guess and then plot that next guess. If you do **plot_one_step(4)**, my old guess was 4 and my new guess was 4.4.

```
plot_one_step(4)
```

```
 old x: 4
new x: 4.4
```

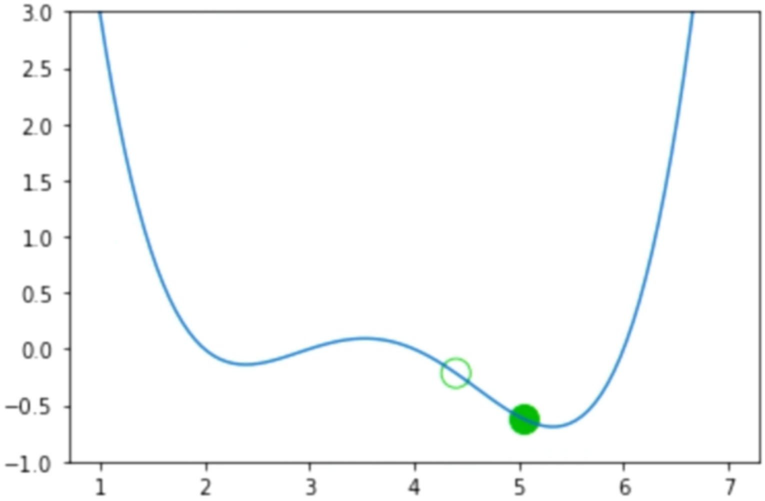
OK, so the first circle is where you were and the second circle is where you have moved to.



Now, try another guess. Do **plot_one_step(4.4)** and your next guess is 5.04.

plot_one_step(4.4)

old x: 4.4
new x: 5.0464000000000055



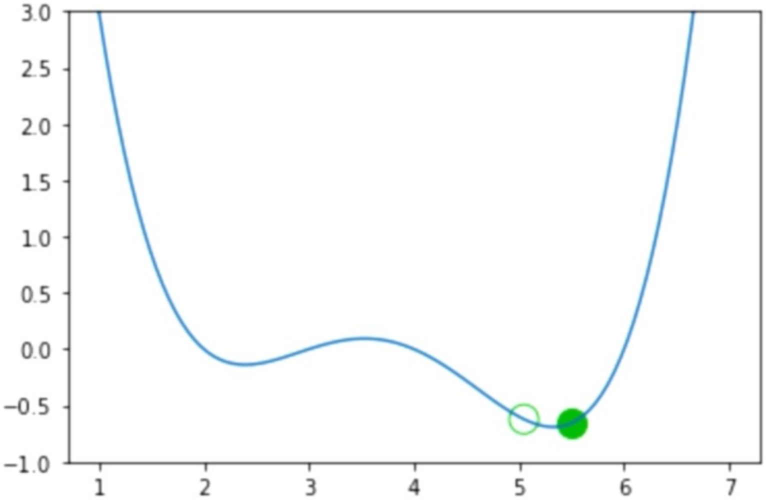
You went from 4 to 4.4 because the slope was 0.4. When you got to 4.4, the slope was bigger at 0.64, if you recall. So, the next jump was actually bigger.

Next, you can do another **plot_one_step** and do 5.0464 to manually descend the gradient.

```
plot_one_step(5.0464)
```

```
 old x: 5.0464
new x: 5.49673060106241
```

As a result, you get 5.4967.

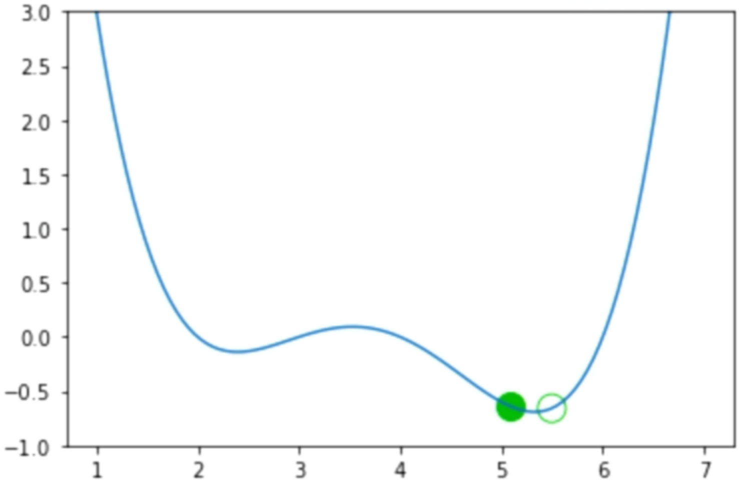


Now do one more. So, 5.4967.

```
plot_one_step(5.4967)
```

```
 old x: 5.4967
new x: 5.080917145374805
```

The next guess you get is 5.08.

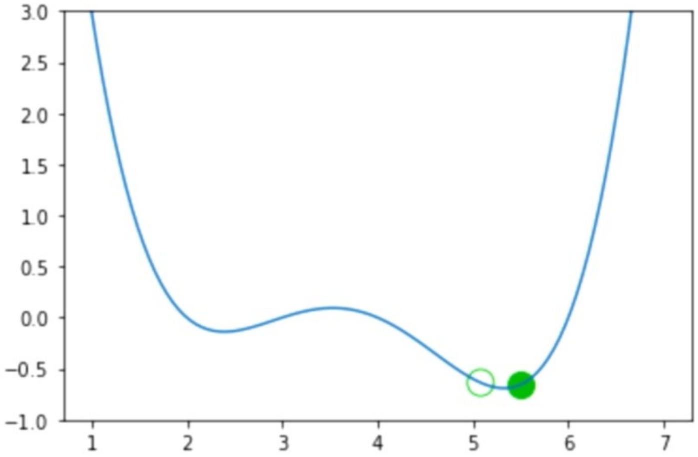


Now, notice that you went from 5.0464 to 5.967 to 5.08.

So there is a pattern. You went down and missed the minimum. And now, in this next step, you have jumped past it. You can do one more and get a sense of what happens next.

```
plot_one_step(5.080917145374805)

  old x: 5. 080917145374805
new x: 5.489966698640582
```

And you end up at 5.4899—almost back where you started.



You keep overshooting the minimum. There is one last tweak that you can make to this procedure. Rather than adding the derivative, you can add the derivative times a small constant. So, in this case, you choose 0.3.

```
def plot_one_step_better(x):
    new_x = x – 0.3 * 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}’)
```

So, the only difference is that when picking your new guess, instead of subtracting the derivative, you are going to subtract the derivative times a small number. The intuition is, well, you are bouncing back and forth and keep missing the minimum. So, to avoid this oscillatory behavior, you are going to use this alternate approach. So in this one, when you say **plot_one_step_better(4)**, you are only moving one third as far.

```
plot_one_step_better(4)
```

```
  old x: 4
new x: 4.12
```

The derivative was 0.4, but you are only doing a third of 0.4. What happens if you go to your next guess, 4.12? OK, now you get 4.267.

```
plot_one_step_better(4.12)
```

```
  old x: 4.12
```

```
new x: 4.2672966399999997
```

What if you go again? With 4.267, you get 4.44.

```
plot_one_step_better(4.267)
```

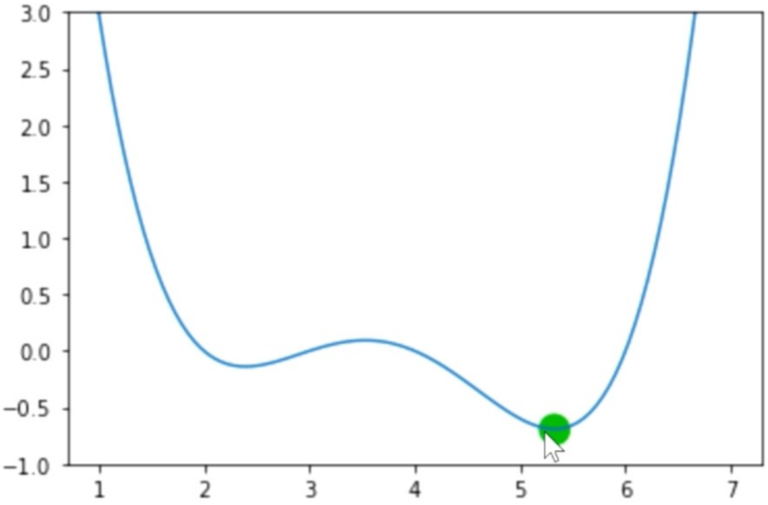
```
  old x: 4.267
new x: 4.44237989044
```

You are getting closer. You can repeat this process over and over and will eventually get to a nice minimum. So, for example, once you get to 5.23, it will be 5.325 and so forth.

```
plot_one_step_better(5.323)
```

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

And so, you converge down to something useful.



Gradient Descent in Code

You can write out the procedure you just carried out manually as a recurrence relation. Now, here is the equation for gradient descent.

$$x^{(t+1)} = x^{(t)} - \ 0.3\frac{d}{dx}f(x)$$

Given a current x, gradient descent creates its next guess for x based on the sign and magnitude of the derivative.

Now in this case, a value of 0.3 was picked totally arbitrarily. Naturally, you can generalize by replacing it with a parameter, typically represented by α, and often called the **learning rate**.

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

So, this learning rate, α, captures how quickly gradient descent learns the minimum. A large α moves quickly, but it can overshoot. With a small α, you move more slowly, but there is less chance of overshooting.

Next, consider gradient descent represented as a few lines of code.

This **gradient_descent** function takes the derivative of the function you want to optimize (df), the **initial_guess**, a learning rate (α), and a number of steps (n). It is going to return a NumPy array of all guesses over time.

```
def gradient_descent(df, initial_guess, alpha, n):
    guesses = [initial_guess]
    current_guess = initial_guess
    while len(guesses) < n:
        current_guess = current_guess - alpha * df(current_guess)
        guesses.append(current_guess)

    return np.array(guesses)
```

So, these lines of code implement gradient descent and give you the power of the SciPy optimization library. Now, there are some differences. Remember that **scipy.optimize** takes the original function, whereas you are asking for the derivative of the function. It also does not require you to specify the number of the steps to do. It just runs until it finds something it is happy with. But fundamentally, this is what **scipy.optimize** is all about.

Now take a look at this code in action.

```
trajectory = gradient_descent(derivative_arbitrary, 4, 0.3, 20)
trajectory
```

So, if you run this **gradient_descent** function, it should produce a trajectory that is basically what you got when you ran this example earlier by hand.

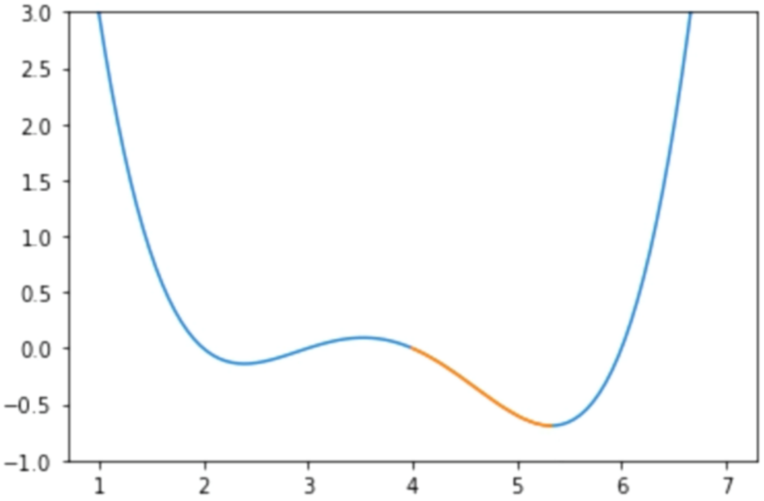
```
trajectory = gradient_descent(derivative_arbitrary, 4, 0.3, 20)
trajectory
```

As a result, you get 4, then 4.12, then 4.267 and so forth, up to 5.326. So, these are the same numbers you got previously. So that means that your procedure is working well.

Another thing you can do is visualize the trajectory taken by the algorithm. So, this is going to be a function that plots where you go over time.

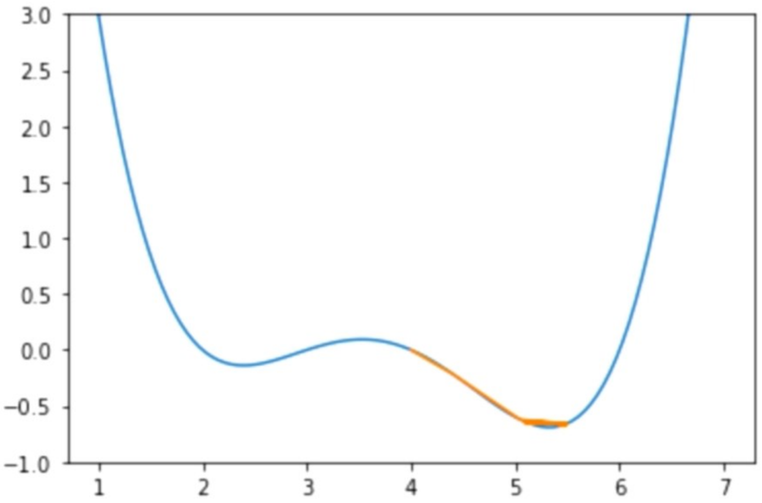
```
trajectory = gradient_descent(derivative_arbitrary, 4, 0.3, 20)
```

plot_arbitrary()
plt.plot(trajectory, arbitrary(trajectory));



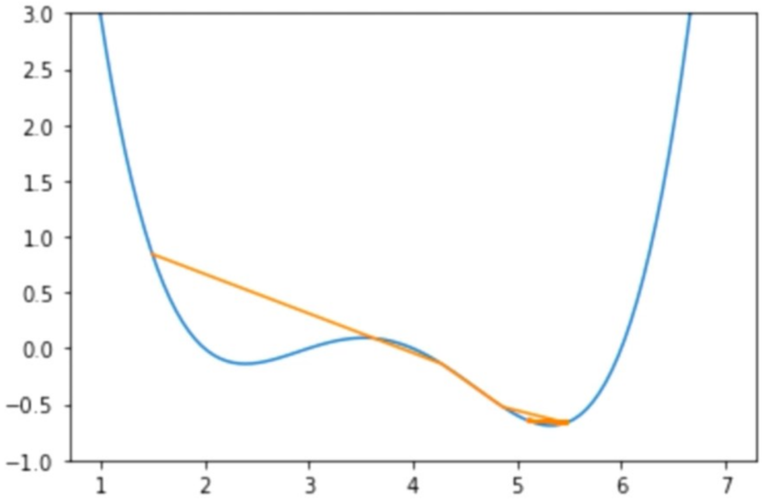
You can do 1 instead.

trajectory = gradient_descent(derivative_arbitrary, 4, 1, 20)
As a result, it goes down and then you bounce back and forth.



If you start at 1.5, you do a big jump and then bounce back and forth.

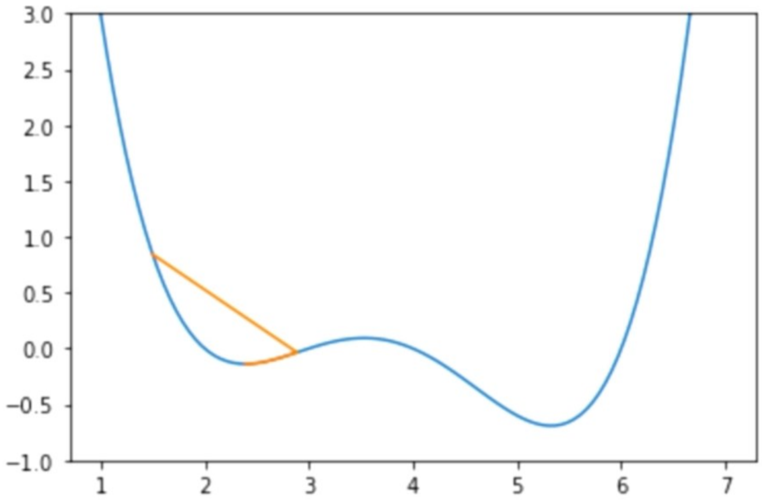
trajectory = gradient_descent(derivative_arbitrary, 1.5, 1, 20)
plot_arbitrary()
plt.plot(trajectory, arbitrary(trajectory));



You can try different values.

trajectory = gradient_descent(derivative_arbitrary, 1.5, 0.5, 20)

If you choose 0.5 as the learning rate, you end up with this minimum.



So, depending on where you start and your learning rate, you end up in different locations. But in principle, this algorithm can find minima. You run the algorithm a fixed number of times and the choices you make determine the location. It's worth noting that more sophisticated implementations, like the SciPy optimize library, will stop based on a variety of different stopping criteria, such as the error getting too small or the error getting too large.

Applying Techniques to Optimizing Linear Regression

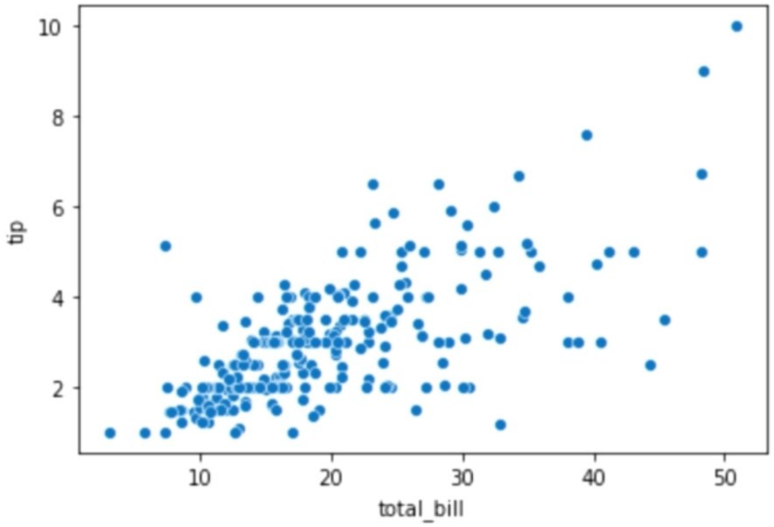
You can use the gradient descent algorithm on a machine learning problem. Specifically, you can do linear regression on the Tips dataset. So, you are going to first load the tips dataset.

tips = sns.load_dataset("tips")

And then you can plot the data using the Seaborn scatterplot library.

sns.scatterplot(x = tips["total_bill"], y = tips["tip"]);

In this case, you want to build a model to predict the tip based on the total bill.

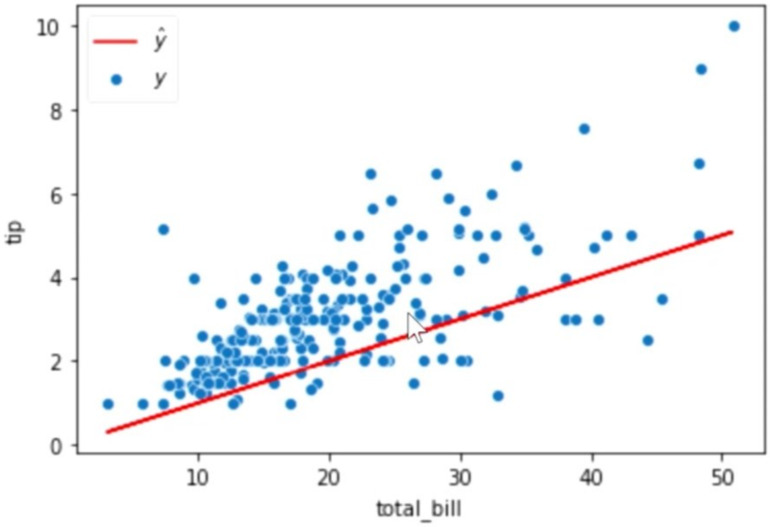


Now, you build a simple model. You have the parameter θ_1, which is going to be the slope. There is no offset. There will not be a y-intercept.

So, you want a one-parameter model, where your output is θ_1 times x, where x is the total bill. And, for example, if $\theta_1 = 0.1$ then $\hat{y} = \theta_1 x$.

If you do a scatterplot of the data, the resulting red line is your guess.

```
sns.scatterplot(x = tips["total_bill"], y = tips["tip"])
x = tips["total_bill"]
y_hat = 0.1 * x
plt.plot(x, y_hat, 'r')
plt.legend([‘$\hat{y}$’, '$y$']);
```



So, you want to find the best slope of this line. You can do that by creating a linear regression model in scikit-learn and then asking for its coefficients.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept = False)
model.fit(tips[["total_bill"]], tips["tip"])
model.coef_
```

So, if you create a LinearRegression model with no intercept, where you fit the total bill and tips, you get back 0.1437.

So, the best slope here is 14.37%. So, where does that come from? This is the tip percentage, which minimizes the mean squared error (MSE). So, this is the minimum of some arbitrary function. In this case, your arbitrary function is dependent on this data. Every single one of the data points will affect this function. So, if you use the L2 loss as your loss function, then you are going to be minimizing the MSE.

And so, you can write a function that captures that:

```
def mse_loss(theta1, x, y_obs):
    y_hat = theta1 * x
    return np.mean((y_hat - y_obs) ** 2)
```

It computes the MSE for a particular choice of θ_1 on your dataset.

OK, so once you have this function defined, you can plug in some values. So, if x is my total bill and y are my tips, then the loss of a 10% tip is 2.077.

```
x = tips[“total_bill”]
y_obs = tips["tip"]
mse_loss(0.1, x, y_obs)
```

```
2.0777683729508194
```

That is the average squared error on all of the tables. So, it is a function of all the blue dots on the graph that you used previously.

The best value would be 0.1437.

```
x = tips[“total_bill”]
y_obs = tips["tip”]
mse_loss(0.1437, x, y_obs)
```

So, how do you find the best value? So, recall, because this data has already been collected, it is not changing. You already have the whole tips dataset. The only real variable is θ_1, which makes it a little awkward to be working with this **mse_loss** function. So, you define a single-argument version of the MSE loss, where x and y are hard-coded values.

This is the **mse_single_arg** function that returns the MSE on the data for θ_1:

```
def mse_single_arg(theta1):
    x = tips["total_bill"]
    y_obs = tips["tip"]
    y_hat = theta1 * x
    return mse_loss(theta1, x, y_obs)
```

Why is this any different? Well, now if you want to run this sort of guess-and-check experiment, instead of having to enter in x and y_obs every time, you only need the θ_1 value. So, if you put in 0.1437, you get 1.178.

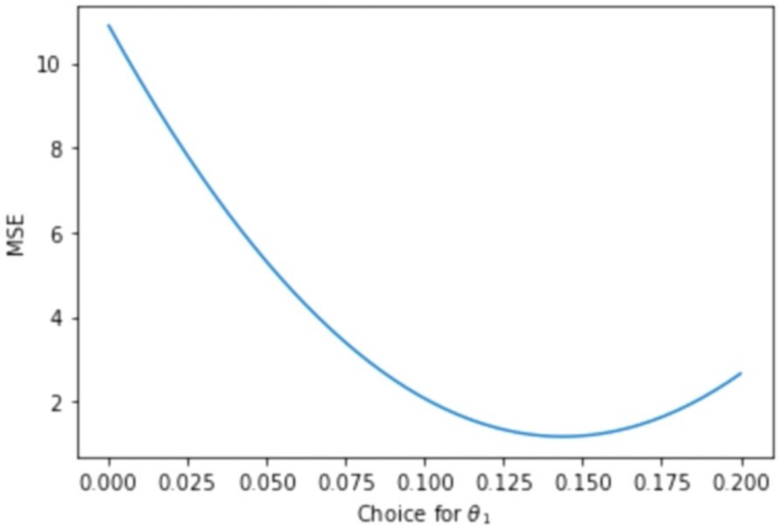
```
mse_single_arg(0.1437)
```

```
1.1781165940051925
```

So, this is a single-argument version of the MSE. So, why is that useful? Brute-forcing can be used to minimize the MSE.

```
theta1s = np.linspace(0, 0.2, 200)
x = tips["total_bill"]
y_obs = tips["tip"]

MSEs = [mse_single_arg(theta1) for theta1 in theta1s]

plt.plot(theta1s, MSEs)
plt.xlabel(r"Choice for $\theta_1$")
plt.ylabel(r"MSE");
```

So, if you plot that mean squared error as a function of θ_1, you see that it goes down and then back up, with a minimum of around 0.1437.



The curve is a function of the blue data points. So, if you move the data points, you will get a slightly different curve. But it will always be a bowl-shaped curve. Now, there are better ways of doing this. You could just optimize. Instead of minimizing the curve, you could use the **scipy.optimize** function and say, here is an MSE function that takes a single argument. If you start from 0, it finds a value of 0.1437.

```
import scipy.optimize
from scipy.optimize import minimize
minimize(mse_single_arg, x0 = 0)
```

So, scikit-learn sets up a single-argument MSE and then passes it to a minimization library to find the value. But you can still do it yourself using the gradient descent function that you created.

Here is your gradient descent algorithm.

```
def gradient_descent(df, initial_guess, alpha, n):
    guesses = [initial_guess]
    guess = initial_guess
    while len(guesses) < n:
        guess = guess - alpha * df(guess)
        guesses.append(guess)
    return np.array(guesses)
```

So, you can use the gradient descent function to find the optimal tip. Rather than relying on somebody else's code to come up with this value, you write it from scratch. Now to do this, you cannot just provide your MSE function to your gradient descent function. It requires the derivative. And so, you need to write the derivative of this **mse_loss** function. That is the thing you are trying to find the minimum of.

The derivative of this function is going to be as follows.

```
def mse_loss_derivative(theta_1, x, y_obs):
    y_hat = theta_1 * x
    return np.mean(2 * (y_hat - y_obs) * x)
```

Keep in mind that your derivative is not with respect to x but θ_1. So, once you have this expression, **mse_loss_derivative**, this is basically just like you had before, with your derivative of the arbitrary function.

Here, θ_1 is 0.1, x is the total bills, and then y_obs are the tips.

```
theta1 = 0.1
x = tips["total_bill”]
y_obs = tips["tip"]

mse_loss_derivative(theta1, x, y_obs)

-41.143986639344256
```

You can ask for the derivative with respect to θ_1 if θ_1 is 0.1. If you pick a different θ_1, you get different derivatives. So, when you are running the gradient descent procedure, when θ_1 is 0.1, the derivative is negative. So, that is something you will use in order to generate your next guess for θ_1.

For this MSE loss derivative function, rather than having to include the x and y values, you can create a single-argument version to make it more syntactically convenient.

So now, instead of having to plug in all the x and y-values, you plug in 0.1.

```
def mse_loss_derivative_single_arg(theta1):
    x = tips["total_bill"]
    y_obs = tips["tip"]
    return mse_loss_derivative(theta1, x, y_obs)
```

Now you can use gradient descent with this function you wrote yourself. You call **gradient_descent** on **mse_loss_derivative_single_arg**, which you wrote. Then you pick a learning rate, starting value, and number of points.

So, you are going to start with giving them a 5% tip, have a relatively small learning rate, and you will do 100 guesses.

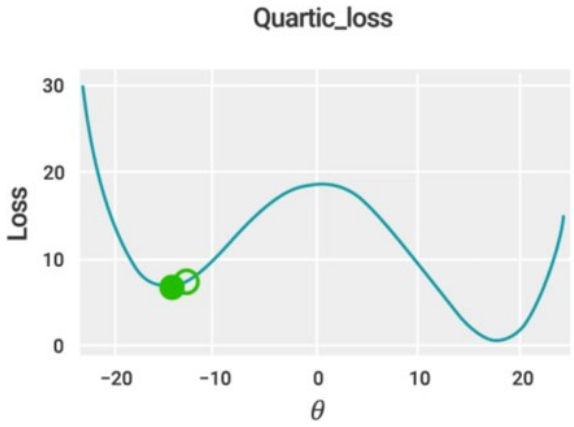
gradient descent(mse_loss_derivative_single_arg, 0.05, 0.0001, 100)

As a result, the original tip that the algorithm decides is 5%, because that is what you told it. And it says that is too low.

The derivative says you need a bigger θ. So the algorithm keeps trying different tips, over and over, until eventually it converges to a value.

Convexity

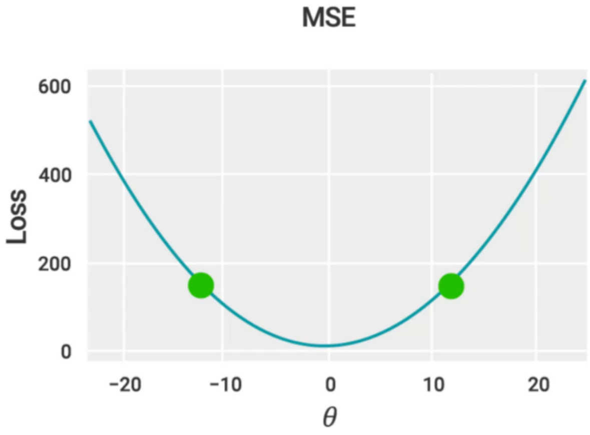
Gradient descent is able to give you the same optimum parameter as scikit-learn. It can find the global minimum of the mean squared error (MSE) function. However, does gradient descent always give the optimal linear regression parameter? No. Gradient descent can get stuck in a local minimum.



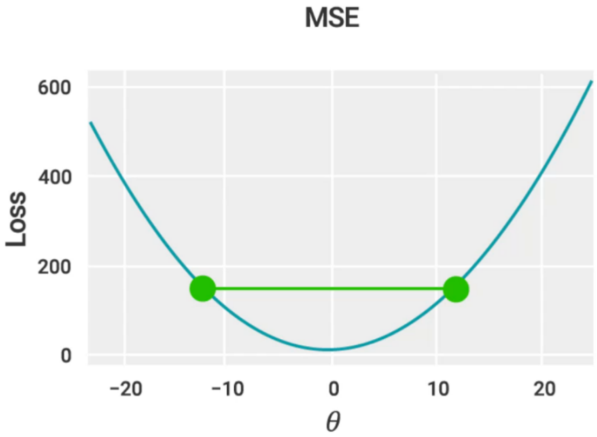
But if a function has **convexity**, then gradient descent is guaranteed to find the global minimum. Normally, you will say that a function, f, is convex if it obeys this inequality:

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

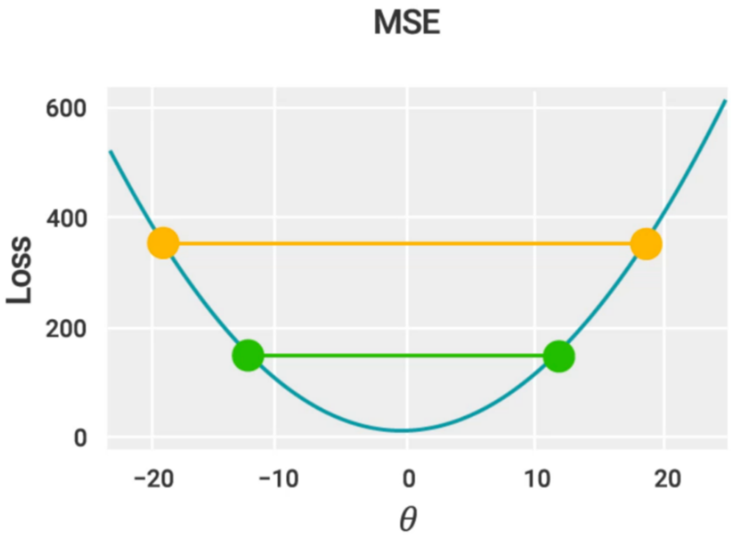
For all a and b in the domain of f and $t \in [0,1]$. This graph illustrates the formula.



If you draw a line between two points on the curve, all values on the curve must be on or below the line. The MESE loss function in one dimension is convex.



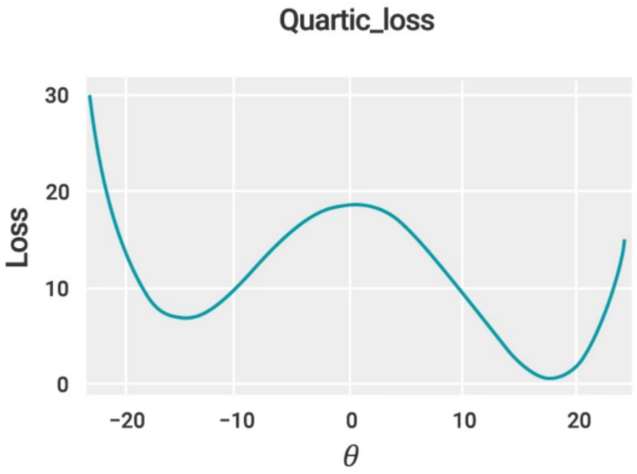
So, if you draw a line segment between points on the MSE curve, all the values on the curve are below the line segment.



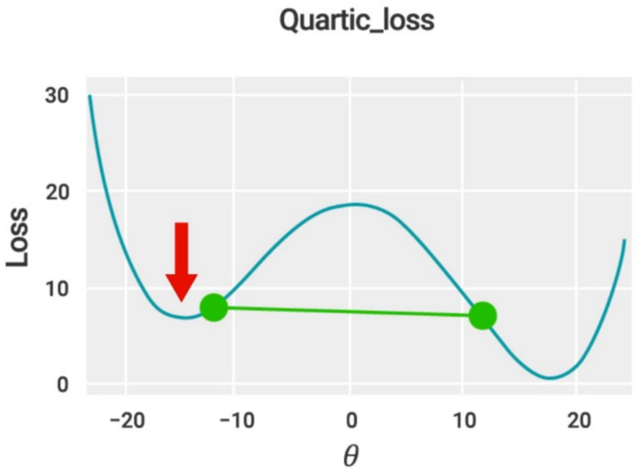
If you draw any line segment between two points on the MSE function, all values on the curve between those two points will be below that line segment. So, for any dataset, the MSE is always convex.

For any convex function, f, any local minimum is a global minimum. If the loss function is convex, gradient descent always finds the globally optimum parameter in one dimension, or a set of parameters in higher dimensions.

Now consider an arbitrary function with two local minima.



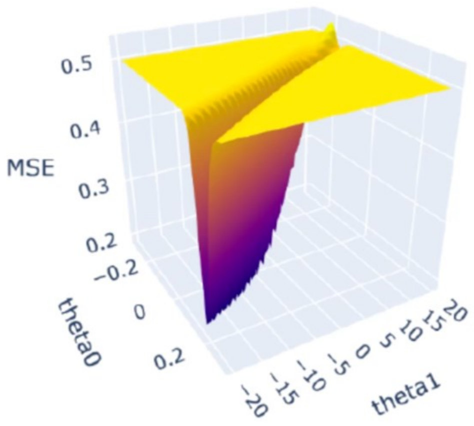
Here, you have drawn a line segment between two points on that curve such that not every point on the curve is below the line segment.

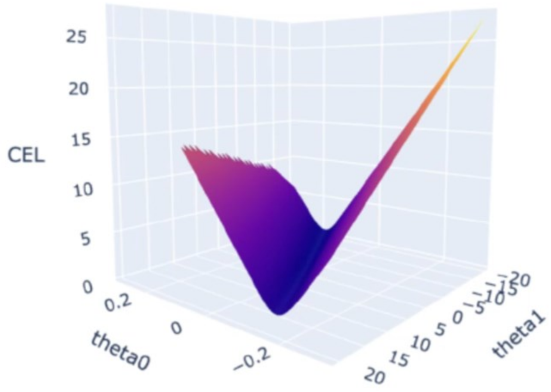


By the definition of convexity, this arbitrary function is therefore **not convex**. Thus, gradient descent may get stuck in that local minimum to the left.

Another reason to care about convexity is that optimization is generally much easier for convex functions.

So, as an example, consider the two, two-dimensional functions shown:





The left function is non-convex and the right one is convex. For the left function, if you are far from the minimum in that large, relatively-flat region, the slope is too shallow to efficiently follow. Only very near the minimum, is the slope steep enough to follow toward the minimum. By contrast, the convex function, it has enough steepness anywhere so that you can work your way down towards the bottom no matter where you start.

Optimizing 2D Linear Regression Using Non-Gradient Descent Techniques

A regression model can be generalized to have not just one parameter, but two. So, specifically, for each prediction, you assume that the tip is going to be the original bill times the tip percentage. But now you add this θ_0 offset.

$$\text{tip} = \theta_0 + \theta_1 \text{ bill}$$

So, you reload the tips dataset from scratch.

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

And then you can fit the model. In this instance, you include an intercept, θ_0.

```
model = LinearRegression(fit_intercept = True)
X = tips[["total_bill"]]
y = tips["tip"]
model.fit(X, y)
```

And so, when you fit this model and you ask for the coefficient and the intercept, you get back a tip percentage of 10.5% and an intercept of $0.92.

You can compute these two values from scratch, without help from any library, if you reframe the question slightly.

First, you create a new copy of the tips dataset.

```
tips_with_bias = tips.copy()
tips_with_bias[“bias”] = 1
X = tips_with_bias[["bias", "total_bill"]]
X.head(5)
```

Next, you create a bias column of all 1s.

	bias	total_bill
0	1	16.99
1	1	10.34
2	1	21.01
3	1	23.68
4	1	24.59

Next, you tell the linear regression model that you do not want to include an intercept. Instead, this bias column is effectively giving you an intercept.

```
LinearRegression(fit_intercept=False)
```

So, in other words, if you fit the model and then ask for the coefficients, the intercept is θ_0 and the slope is θ_1.

$$\hat{y} = \theta_0 + \theta_1 \times bill$$

So, for example, if θ_0 is 1.5 and θ_1 is 0.05, then my predictions are just θ_0 times X, the first column of X, plus 0.05 times the next column of X.

```
1.5 * X.iloc[:, 0] + 0.05 * X.iloc[:, 1]
```

You get a bunch of predictions as a result.

0	2.3495
1	2.0170
2	2.5505
3	2.6840
4	2.7295
…	…
239	2.9515
240	2.8590
241	2.6335

Note: You do not actually need **X.iloc[:, 0]** because the column is all ones and if you delete it, you get the same predictions. So, this is just an alternate framing to get the coefficient and intercept in the same array.

So you can create a linear regression model where you have a bias column and each prediction is simply θ_0 times that bias column + θ_1 times the actual data—the X values that you care about.

Next, you create a mean squared error (MSE) function to optimize against.

```
def mse_loss(theta, X, y_obs):
    y_hat = theta[0] * X.iloc[:, 0] + theta[1] * X.iloc[:, 1]
```

You compute your predictions, and then you return the difference between your predictions and the truth, all squared.

```
return np.mean((y_hat - y_obs) **2)
```

Then you compute the mean. As before, you could use scikit-learn metrics to compute the MSE. And so, once you have written this function, you can then plug in values for θ_0 and θ_1 and see what happens. So, if you pick a fixed tip offset of a $1.50 and a 5% tip, this is your MSE loss.

```
mse_loss(np.array([1.5, 0.05]), X, y_obs)
```

```
1.5340521752049179
```

If you make this $2.50, you get a slightly different loss and so forth. If you change this to a 0% tip, you get a larger loss.

```
mse_loss(np.array([1.5, 0]), X, y_obs)
```

```
4.1514475409836065
```

You can use this to find the optimal θ_0 and optimal θ_1. Just plug in a bunch of values and see what happens. Now, repeat the same idea as before, but on a two-dimensional function. So, the first approach is **brute forcing.** In this approach, you try a bunch of different θ values with the same exact **mse_loss** function used previously.

But it is convenient to create a version of this function that only has one argument. So, set aside X and y in a function that has a single argument.

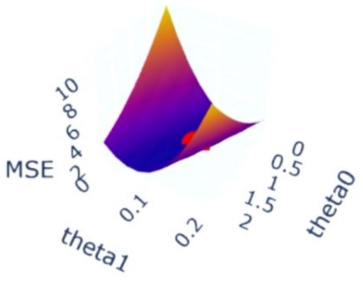
```
def mse_loss_single_arg(theta):
    X = tips_with_bias[["bias", "total_bill"]]
    y_obs = tips["tip"]
    return mse_loss(theta, X, y_obs)
```

So now, you can try out different values using one argument. And when you optimize this, you find that the optimal choice is around 92% and 0.1.

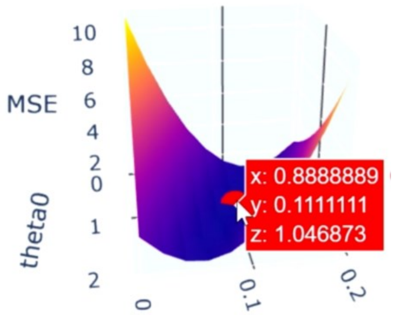
```
mse_loss_single_arg([0.92, 0.1])
```

```
1.0479490286885247
```

To optimize and find if these are the right values, you could plot this loss function as a function of its two parameters—that is, create a three-dimensional (3D) loss surface. This lets you see the model error as a function of the two parameters—the fixed offset tip, θ_0, and the percentage.



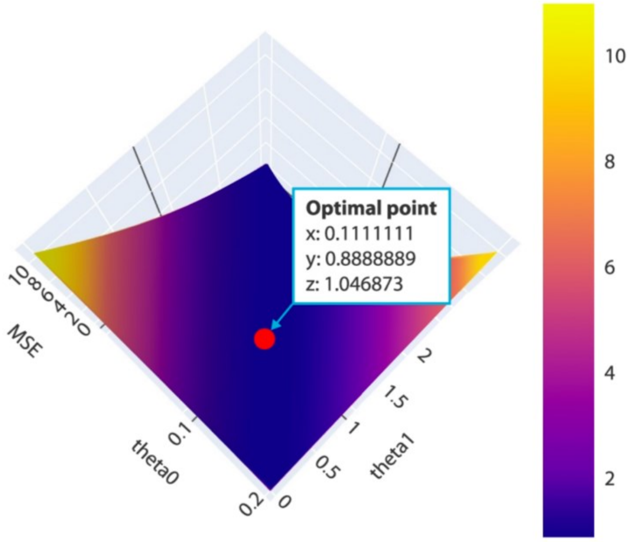
So, somewhere at the bottom of this bowl shape you see x of 0.8888 and y 0.111, which is the optimal point.



Next, you can use an optimization library, like scikit-learn, and give it the **mse_loss_single_arg** function. If you call that function, starting from a 0% tip with a $0 offset, it optimizes and finds $0.92 fixed offset and 10.5% tip. So, it used gradient descent to find the minimum of this function.

2D Gradient Descent

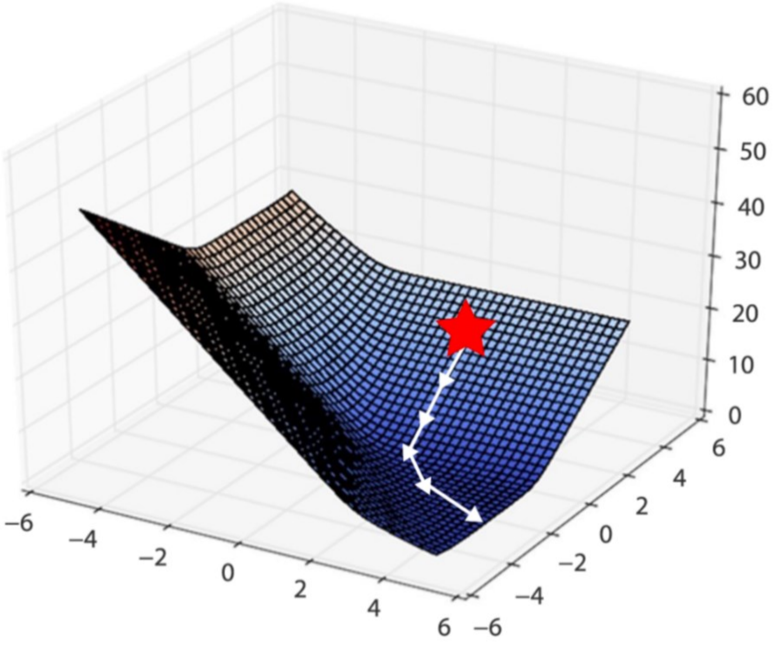
Now, you can use the **convex optimization on multivariable functions** procedure to minimize functions, such as the 2D loss function for the tips dataset, where the two inputs are the parameters θ_0 and θ_1.



```
def mse_loss(theta, X, y_obs):
    y_hat = X @ theta
    return np.mean((y_hat – y_obs) ** 2)
```

Multi-dimensional gradient descent has some important differences from the one-dimensional (1D) case. Just like with 1D functions, you can optimize a two-dimensional (2D) function by following the slope. However, unlike a 1D function, the slope of a 2D function is described by a 2D vector. The best way down has two components, each corresponding to the slope with respect to the two input variables to the function.

Consider this wireframe picture of a 2D function.



If the star represents your starting point, the arrow shown gives the best way down. The formal term for the best way down is the **gradient** of a function. The gradient is the generalization of the idea of a derivative of a 1D function. Each component of the gradient represents the partial derivative of the function, with respect to one of the input variables to the function.

Consider this 2D function:

$$f(\theta_0, \theta_1) = 8\theta_0^2 + 3\theta_0\theta_1$$

This is not a linear regression loss function, as it has a nonlinear dependence on the θs.

Since this function is 2D, its gradient has two components given by the formula shown:

$$\nabla_{\vec{\theta}} f = \frac{\partial f}{\partial \theta_0}\vec{\imath} + \frac{\partial f}{\partial \theta_1}\vec{\jmath}$$

For a function of two variables $f(\theta_0, \theta_1)$, you define the gradient, where $\vec{\imath}$ and $\vec{\jmath}$ are the unit vectors in the $\theta_1$and θ_2 directions. The symbol used for the gradient is the nabla, ∇, or del.

Given this definition, you can compute the gradient of your function, f. First, you compute the partial derivative of f with respect to θ_0:

$$\frac{\partial f}{\partial \theta_0} = 16\theta_0 + 3\theta_1$$

Then you compute the partial derivative of f with respect to θ_1:

$$\frac{\partial f}{\partial \theta_1} = 3\theta_0$$

Given these two partial derivatives:

$$\nabla_{\vec{\theta}} f = (16\theta_0 + 3\theta_1)\vec{\imath} + 3\theta_0\vec{\jmath}$$

Consider the 2D function:

$$f(\theta_0, \theta_1) = 8\theta_0^2 + 3\theta_0\theta_1$$

Now for notational convenience, gradients are often written in column vector notation. For example, you can also write the gradient as:

$$\nabla_{\vec{\theta}} f(\vec{\theta}) = \begin{bmatrix} 16\theta_0 + 3\theta_1 \\ 3\theta_0 \end{bmatrix}$$

You can expand this definition of the gradient to p dimensions:

$$\nabla_{\vec{\theta}} f(\vec{\theta}) = \begin{bmatrix} \frac{\partial}{\partial\theta_0}(f) \\ \frac{\partial}{\partial\theta_1}(f) \\ \vdots \\ \frac{\partial}{\partial\theta_p}(f) \end{bmatrix}$$

You can use the gradient of a function to find its optimizing parameters. In the context of model optimization, you can think of the top value in the column vector as follows. If you slightly increase θ_0, what happens to the function, or what happens to the loss? The next row tells you if you slightly increase θ_1, what happens to the loss and so forth.

To reduce the loss, you can adjust the θ values:

- Increase all values of θ that have a negative partial derivative
- Decrease all values of θ that have a positive partial derivative

So, this finally brings you to the gradient descent algorithm. The algorithm is used to find the best next guess:

- Increasing θ proportionally to the magnitudes of the partial derivatives will result in the most improvement relative to the current guess
- This process adjusts the θ vector in a negative gradient direction until it converges

Now, written in mathematical notation, you can say that the next guess is:

$$\vec{\theta}^{(t+1)} = \vec{\theta}^{(t)} - \propto \nabla_{\vec{\theta}} L(\vec{\theta}, \mathrm{X}, \vec{y})$$

The gradient is given by ∇, $\propto$ is the learning rate, $\vec{\theta}^{(t)}$ represents the current guess for optimal parameters, and $\vec{\theta}^{(t+1)}$ represents the next guess for the optimal parameters. Note that the gradient of the loss function depends on your current guess for θ, the input features X, and the true value you are trying to predict, $\vec{y}$. When you optimize a function using this gradient descent procedure, you have to have some sort of starting guess. This could be all zeros, small random numbers, or some arbitrary starting guess provided as an input to this gradient descent procedure.

2D Gradient Descent for Linear Regression (Part 1)

Now that you know how to define the gradient of a function, to get a better understanding of just what that definition means and see why it is useful, you can write your own gradient descent code from scratch and use it to optimize a two-dimensional (2D) linear regression model.

First, you compute the gradient of the loss for linear regression. So, here you have a 2D linear regression model with no intercept term:

$$f_{\vec{\theta}}(\vec{x}) = \vec{x}^T \vec{\theta} = \theta_0 x_0 + \theta_1 x_1$$

The output of that model is $\theta_0 x_0 + \theta_1 x_1$. The loss for such a model—if you are using the squared error—is $(y_i - \theta_0 x_0 - \theta_1 x_1)^2$. That is a loss for just a single data point.

$$\ell(\vec{\theta}, \vec{x}, y_i) = (y_i - \theta_0 x_0 - \theta_1 x_1)^2$$

So, if you substitute in the model prediction, you have the loss for a model equal to $(y_i - \theta_0 x_0 - \theta_1 x_1)^2$. This equation gives you the loss only for data point number i.

What is the gradient of this loss function for data point number i?

$$\frac{\partial}{\partial \theta_0} \ell(\vec{\theta}, \vec{x}, y_i) =$$

To start, you compute the first component of your gradient, the partial derivative of the loss function with respect to θ_0.

$$2(y_i - \theta_0 x_0 - \theta_1 x_1)(-x_0).$$

Now, $(-x_0)$ appears because of the chain rule in calculus. So, for the second component of the gradient, the partial derivative of the loss function with respect to θ_1 is $2(y_i - \theta_0 x_0 - \theta_1 x_1)(-x_1)$.

In other words, this component of the gradient is exactly the same as before, only now with $(-x_1)$ instead of $(-x_0)$. You can also write that as a column vector:

$$\nabla_\theta \ell(\vec{\theta}, \vec{x}, y_i) = \begin{bmatrix} -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_0) \\ -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_1) \end{bmatrix}$$

The top row is the partial derivative with respect to θ_0 and the bottom row is the partial derivative with respect to θ_1.

2D Gradient Descent for Linear Regression (Part 2)

Here is the equation for the gradient with respect to θ_0 and θ_1 of your loss function.

$$\nabla_\theta \ell(\vec{\theta}, \vec{x}, y_i) = \begin{bmatrix} -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_0) \\ -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_1) \end{bmatrix}$$

The top component is the derivative with respect to θ_0, and the bottom is the derivative with respect to θ_1.

To write that out in code, you use this gradient descent function.

```
def mse_gradient(theta, X, y_obs):
      x0 = X.iloc[:, 0]
      x1 = X.ilocL:, 1]
```

So, you set aside x0, which is the 0th column of your array, and x1.

And then you write out the two expressions $\begin{bmatrix} -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_0) \\ -2(y_i - \theta_0 x_0 - \theta_1 x_1)(x_1) \end{bmatrix}$:

```
dth0 = np.mean(-2 * (y_obs - theta[0]*x0 - theta[1]*x1) * x0)
dth1 = np.mean(-2 * (y_obs - theta[0]*x0 - theta[1]*x1) * x1)
```

Once you have computed those two components, you return a NumPy array of those two components, θ_0 and θ_1.

And the average that you get if you take into account all the data points, is as follows. Basically, it tells you that if your original guess is a fixed offset of $0 and the tip percentage is 0% then you need to adjust both of these upward because the gradient is negative in both directions.

```
X = tips_with_bias[["bias", "total_bill"]]
y_obs = tips["tip"]
mse_gradient(np.array([0, 0]), X, y_obs)
```

It is convenient to write a single-argument version of the gradient descent function, as follows:

```
def mse_gradient_single_arg(theta):
    X = tips_with_bias[["bias", "total_bill"]]
    y_obs = tips["tip"]
    return mse_gradient(theta, X, y_obs)
```

You can hardcode in the X and y values and return the output as a function of a single argument, θ. Keep in mind this single argument is both of your θs, so it is an array of two numbers. So, now if you try out a value like [0, 0], you get back minus 5.99996 and minus 135.22.

```
mse_gradient_single_arg(np.array(0, 0]))
```

So this is the gradient of your two-dimensional (2D) loss function. Next, you plug in the value 0, 0 that tells you the 2D slope of that original loss surface, or the slope of the value once you are at 0, 0. If you plug in something close to the correct answer, such as 0.9 and 0.1, you should get a gradient that is pretty small.

Now you can use the gradient descent function that you wrote earlier.

```
def gradient_descent(df, initial_guess, alpha, n):
    guesses - [initial_guess]
    current_guess initial_guess
    while len(guesses) < n:
        current_guess = current_guess - alpha * df(current_guess)
        guesses.append(current_guess)
     return np.array(guesses)
```

It does not matter that before you were doing one-dimensional and now you are doing 2D. It is irrelevant because of the way that NumPy works in Python.

So, you can just run gradient_descent.

guesses = gradient_descent(mse_gradient_single_arg, np.array([0, 0]), 0.001, 10000)

After running it, if you look at the trajectory of the procedure of gradient descent, you see that it moves pretty quickly at the beginning. It starts with a $0.00 offset and a 0% slope.

pd.DataFrame(guesses).head(10)

	0	**1**
0	0.000000	0.000000
1	0.005997	0.135226
2	0.006630	0.142991
3	0.006955	0.143426
…	…	…

And it increases that offset and the slope jumps around. OK, so it is evolving over time. And then at the very bottom, once it is done, it is getting pretty close to converging to the correct answer.

```
pd.DataFrame(guesses).tail(10)
```

	0	1
9990	0.888098	0.106378
9991	0.888108	0.106378
9992	0.888119	0.106377
9993	0.888130	0.106377
...	...	...

It is still not quite there, but it is much closer. $0.888 offset and 10% tip.

Stochastic Gradient Descent

Real gradient descent implementations result in what's known as **stochastic gradient descent**. To see why this approach is used, you can explore a rudimentary implementation of this algorithm.

So, recall that the gradients are going to be a function of the entire dataset. To know the mean squared error (MSE) across the entire dataset, you have to compute the difference between the prediction and the actual tip for the entire dataset.

```
tips[["total_bill", "tip"]]
```

	total_bill	tip
0	16.99	1.01
1	10.34	1.66
2	21.01	3.50
3	23.68	3.61
…	…	…

Now here there are only 244 rows. But in a real dataset with a large number of datapoints, it is not practical to be computing the gradient for each data point. So you cannot run the code that looks at every single data point.

You can instead compute the gradient using a subset of the data. To do this, you create a function—here called **mse_gradient_batch_only**—where you give it a set of indices, the batch of data you care about. And that will give the gradient if you only take into consideration a subset of the data.

```
def mse_gradient_batch_only(theta, batch_indices, X, y_obs):
    x0 = X.iloc[batch_indices, 0]
    x1 = X.iloc[batch_indices, 1]
    dth0 = np.mean(-2 * (y_obs[batch_indices] - theta[0] * x0 - theta[1] * x1 * x0)
    dth1 = np.mean(-2 * (y_obs[batch_indices] - theta[0] * x0 - theta[1] * x1 * x1)
```

So, now you can write this function and ask it to give you the gradient for only the points between 0 and the length of the array.

```
X = tips_with_bias[["bias", "total_bill"]]
y_obs = tips["tip"]
mse_gradient_batch_only(np.array([0, 0]), np.arange(0, len(X)), X, y_obs)
```

```
array([   -5.99655738,   -135.22631803])
```

And if you want, you can ask it to determine only the loss on point number five.

```
X = tips_with_bias[["bias", "total_bill"]]
y_obs = tips["tip"]
mse_gradient_batch_only(np.array([0, 0]), 5, X, y_obs)
```

So that is much faster to compute compared to the entire dataset. Now you can also take a subset of data, like a batch of data. So, you can ask it to give you everything between row 0 and 4.

```
mse_gradient_batch_only(np.array([0, 0]), np.arange(0, 4), X, y_obs)
```

```
array([  -4.74 ,   -93.12005])
```

So, that is the average gradient you get back across only those data points. It might seem strange that this gradient would be useful, as it is considering only some of the data points. That is where stochastic gradient descent comes in. Now, one thing to do before you introduce the algorithm, is you want to create an **only_two_arg** version of **mse_gradient_batch_only**.

```
def mse_gradient_batch_only_two_arg(theta, batch_indices):
    X = tips_with_bias[["bias", "total_bill"]]
    y_obs = tips["tip"]
    return mse_gradient_batch_only(theta, batch_indices, X, y_obs)
```

So, rather than having to provide X and y_obs, you do the same old trick as before, where you hardcode those. And so now when you want to know the gradient, you provide the two thetas and the batch indices you care about. So, only indices 5, 6, 7, 8, 15, and 32. So, that lets you compute the average, the gradient, across only the values that you specify.

```
mse_gradient_batch_only_two_arg(np.array([0, 0]), [5, 6, 7, 8, 15, 32])
```

So, the next piece of code is going to hinge on the behavior of this NumPy **split** function. The NumPy split function divides an array of values into a specified number of equal parts.

```
np.split(np.array([1 2, 3, 4, 5, 6, 7, 8, 9]), 3)
```

You choose an array of values, say 1 through 9, and you give it the value 3, and it is going to split them into three equal parts. You can say split of a random permutation, of np.arrange, say 12. You can split that into three parts and get three arrays, which are: [11, 8, 10, 1], [6, 0, 5, 7] and [9, 2, 4, 3].

```
np.split(np.random.permutation(np.arange(12)), 3)
```

```
[array([11, 8, 10, 1]), array([6, 0, 5, 7]), array([9, 2, 4, 3])]
```

So this **mse_gradient_batch_only_two_arg** function returns the gradient, only taking into consideration a subset of the points.

This brings you to **stochastic gradient descent**. This algorithm is the same idea as before, except this time you keep track of your losses as you go.

```
def stochastic_gradient_descent(df, initialguess, alpha, n, num_dps,
number_of_batches):
    guesses = [initial_guess]
```

```
guess = initial_guess
losses = [mse loss single arg(guess)]
while len(guesses) < n:
    dp_indices = np.random.permutation(np.arange(num_dps))
     for batch indices in np.split(dp_indices, number of batches):
        guess = guess - alpha - df(guess, batch_indices)
        guesses.append(guess)
        losses.append(mse_loss_single_arg(guess))
 return np.array(guesses), np.array(losses)
```

And so, while the number of guesses is less than n, you create a bunch of indices—different arrays of indices—and set them aside. So, you say the number of batches and the number of data points. This function is going to create an array that is between the numbers, 0 and 12. So, say you have 12 data points:

```
np.random.permutation(np.arange(12))
```

This will create a random permutation of the data points. And then this line will split it into separate arrays.

```
np.split(np.random.permutation(np.arange(12)), 3)
```

So, rather than doing one gradient descent step across all 12 data points, it breaks it into three rounds. The first round considers only four of the data points. The next round considers only another four of the data points. Finally, the last four of the data points are considered.

So, what happens if you run this code? The result is as follows.

```
results = pd.DataFrame(guesses).rename(columns = {0: "theta0", 1: "theta1"})
results["loss"] = losses
results.tail(20)
```

	theta0	theta1	loss
9981	0.887516	0.103818	1.039341
9982	0.887552	0.105040	1.037070
9983	0.887382	0.098547	1.065269
...	...	...	...

You track the losses, so you can see the pattern of the losses as you go. Now, explore the top of this results frame.

```
results.head(5 )
```

	theta0	theta1	loss
0	0.000000	0.000000	10.896284
1	0.006111	0.132319	1.234781
2	0.006831	0.139755	1.182501
...	...	...	...

At the very beginning, the first θ_0 and θ_1 you have, are 0 and 0 and the loss is relatively large. Thereafter, you make some jumps and the loss is better.

Over time, as you run this algorithm further and further and further, you end up with different θ_0 and θ_1 values.

Now, the pattern here is a little different than before because each step of the algorithm is only using a subset of the data. And so, it seems plausible that maybe that subset is missing some outliers somewhere. So, the gradient descent algorithm may make steps that are not really as globally optimal as the gradient descent where you are using the entire gradient.

Gradient Descent vs. Stochastic Gradient Descent

Here is the formula for gradient descent in mathematical notation:

$$\theta^{(\tau+1)} \leftarrow \theta^{(\tau)} - \rho(\tau)\left(\frac{1}{n}\ \sum_{i=1}^{n} \nabla_{\theta} L_i(\theta)\,\Big|_{\,\theta=\theta^{(\tau)}}\right)$$

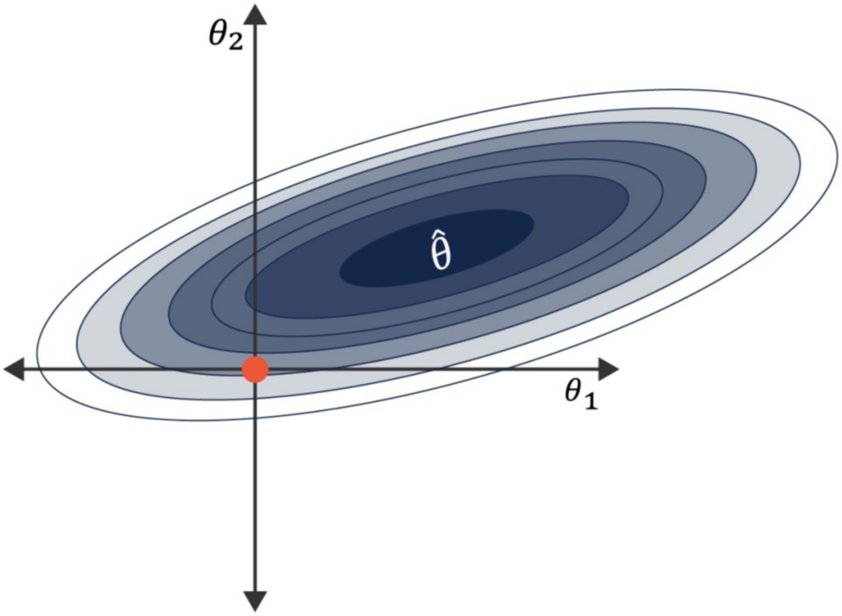
Computing the gradient requires iterating over the entire dataset. So, in the example, this is not such a big deal since the tips dataset is so small. However, for larger real-world datasets of millions of data points, computing the gradient can be extremely costly. So to accommodate that, you can instead compute the gradient for a batch of data where the batch is smaller than the entire dataset.

In the case where the batch size is 1, you have stochastic gradient descent, where the optimizing algorithm considers a single data point at a time.

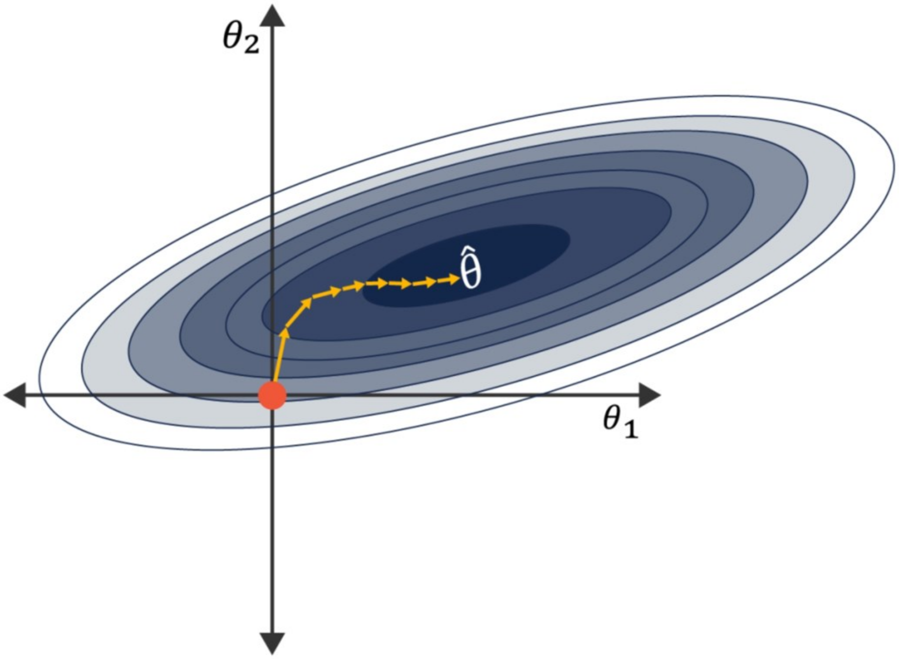
Imagine that you are trying to train an algorithm to recognize pictures of dogs. Stochastic gradient descent adjusts what might be an absolutely, massive number of model parameters, based on only a single training image at a time. And yet, stochastic gradient descent works. The rough idea is: If you iterate over the entire dataset multiple times, then on average you will end up in roughly the same place as if you had been computing the true gradient based on the entire dataset.

In effect, batch size is a parameter that gives you the ability to tradeoff the quality of your gradient approximation against the runtime to compute that gradient approximation. If the batch size is one, the quality is minimum, but the calculation is very fast. And if the batch size is the entire dataset, the quality of the gradient is extremely good, but the calculation may be very slow. So many rules of thumb developed, such as picking a batch size.

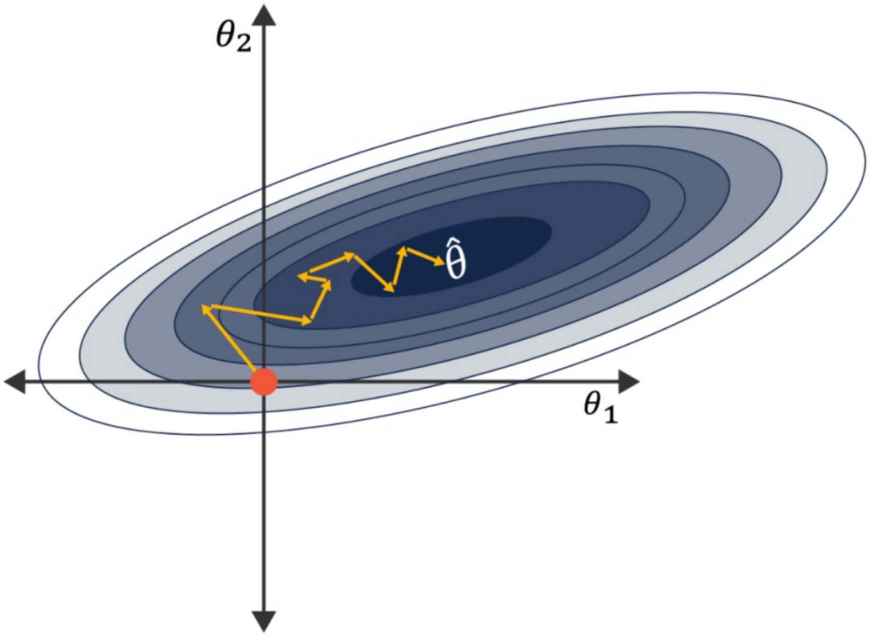
Consider another perspective. This is a visual picture of gradient descent in two dimensions, where the batch size is the size of the dataset.



Here, the darker the color, the lower the loss. If you start at the origin, the gradient descent follows the steepest path toward the minimum and it is the true steepest path. Now consider a visual depiction of gradient descent where the batch size is small.



It is the same figure, because it is the same loss surface, but you do not compute the exact loss surface as you go. This time—since you are only considering a small subset of the data—you only get a crude approximation of the gradient.



Eventually for convex functions, like the mean squared error (MSE), you will end up converging somewhere very close to the true minimum.

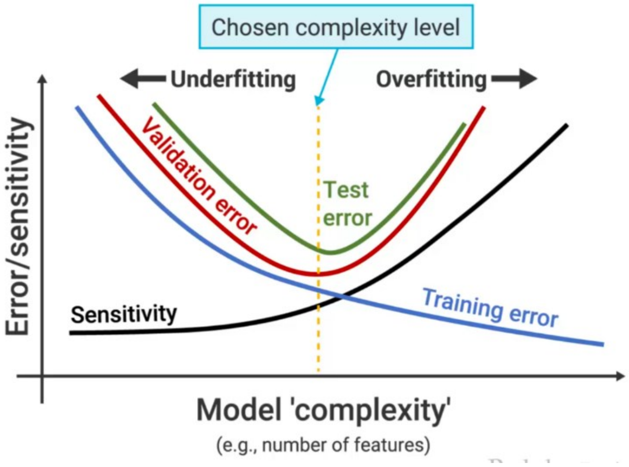
In practice, both techniques are used:

- Gradient descent—used by scikit-learn
- Mini-batch gradient descent—is very common when dealing with large amounts of data

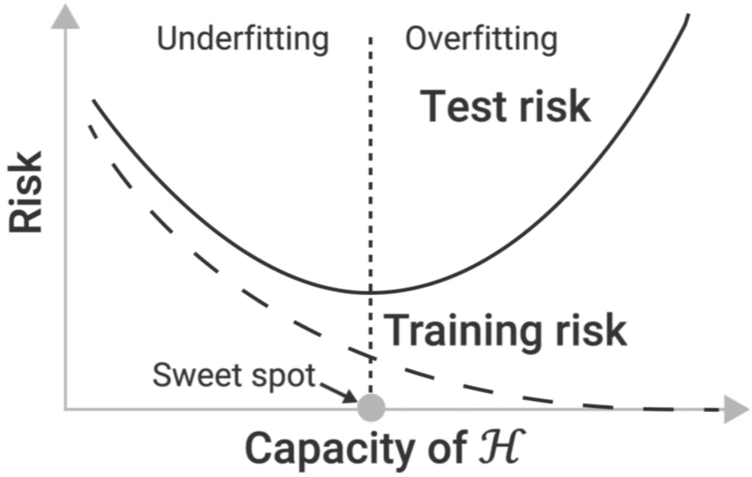
In practice, you will almost never write your own gradient descent algorithms and instead will be using function optimization libraries.

Implicit Regularization and Stochastic Gradient Descent

According to the classical picture of machine learning models, as model complexity goes up, training error tends to decrease more and more.

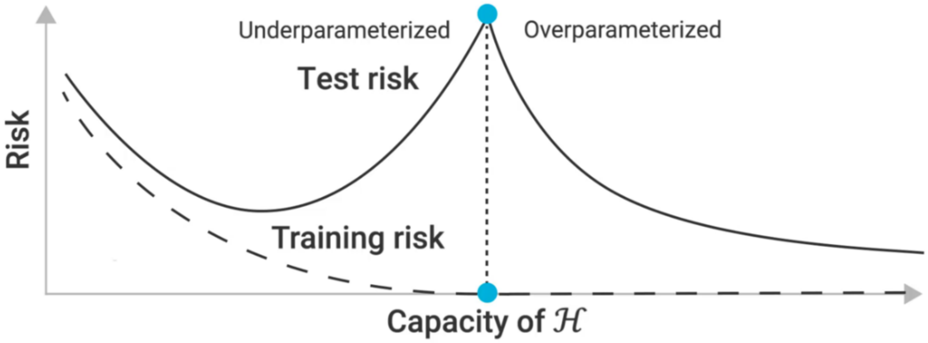


By contrast, test and validation errors go down for a while and then go back up as the model begins to overfit. Here is the same picture from a paper called *Reconciling Modern Machine-Learning Practice and the Classical Bias–Variance Trade-Off*.



Slightly different terminology is used. The error is called the **risk** and the complexity is called the **capacity**. In the late 2010s, it was observed that some models exhibited a surprising relationship between the test error and the model complexity.

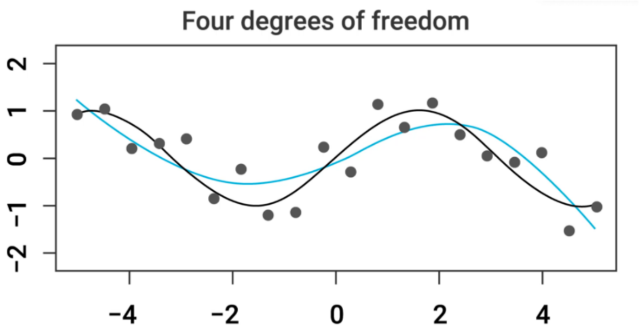
This figure, from that same paper, shows a common pattern observed when training large neural networks. Right around the level of complexity that results in zero training error, the test error spikes.



But, if you push the model to even higher levels of complexity, the test error starts to decrease again. In many real-world cases, that test error actually ends up reaching an even lower value than the 'sweet spot.' In other words, in the world of practical machine learning, an entirely new regime was discovered for large models, called the 'Modern' interpolating regime. The word **interpolating** represents situations where the training set has zero error.

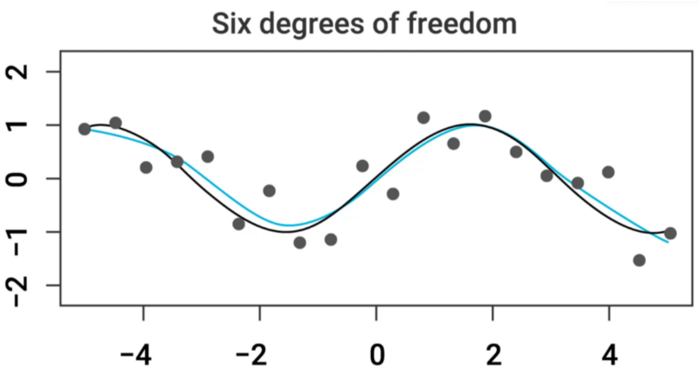
A series of figures, taken from tweets by Dr. Daniela Witten in August 2020, portray the situation quite nicely.

Here is a degree for a linear regression model fit to 20 data points:

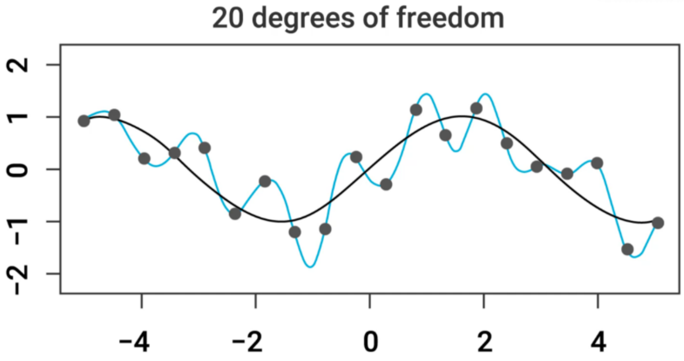


Here Dr. Witten is using **spline regression**, which is similar to but not quite the same thing as polynomial regression. So, here the original data points are in gray. The blue line is the fourth-order model fit. And the black line is the real process which generated the data, which also had some noise added in that gave you the gray data points.

Now, when you move to a degree-six model, you would do even better:

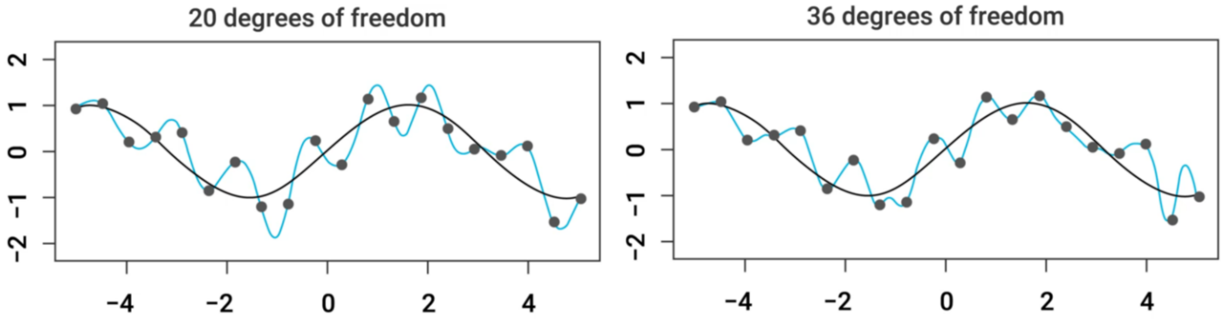


If you now move to a degree-20 model, you see that you have zero training error, but the model is grossly overfit:



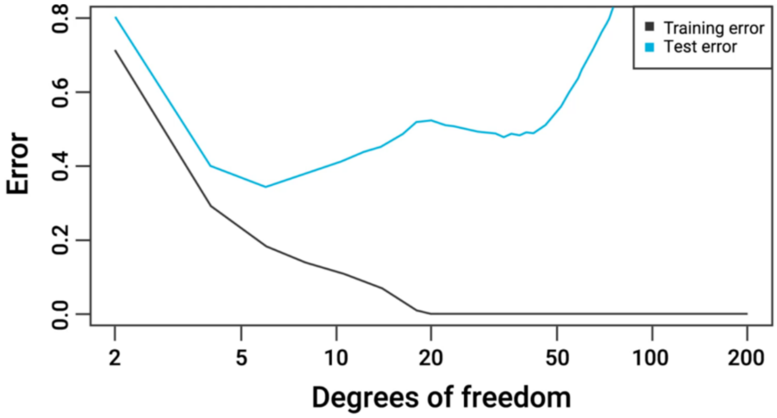
The model is interpolating the observed training data points perfectly, but for points adjacent to the training data, the model is failing to capture the true data generation process.

What if you have even more than 20 parameters? Imagine fitting a degree-36 polynomial, to 20 data points. If you think about it, you basically have 36 linear equations and 20 unknowns. And thus, there is an infinite number of possible solutions that all yield zero training error.



This figure, also by Dr. Witten, shows the side-by-side results for a 20-parameter and a 36-parameter model. The 36-degree model is slightly better. By moving deeper into the modern interpolating regime, you observe **double descent**, where the test error actually drops again.

If you plot Dr. Witten's training and test error versus complexity for this example, you get the figure shown:



Note that the test error in this case spikes back up for even more degrees of freedom. In other contexts, like in neural networks, this spike back up often does not occur. Also note that the test error after the second descent does not reach a new lower value. However, in some contexts, the test error is actually lower after the double descent and the interpolating regime.

Understanding **stochastic gradient descent** gives some insight into why double descent occurs. As Dr. Witten notes, there are an infinite number of models, which are over-parameterized and have zero training error. If you have n data points and a model with n parameters, then there exists exactly one set of parameters that fits the data perfectly. But, if you increase the number of parameters higher, you end up with a situation where the model is over-parameterized and there are an infinite number of valid choices of parameter that give you zero training error. How do you know which model you will end up with from among this infinite sea of possibilities? That is where knowledge of stochastic gradient descent becomes useful.

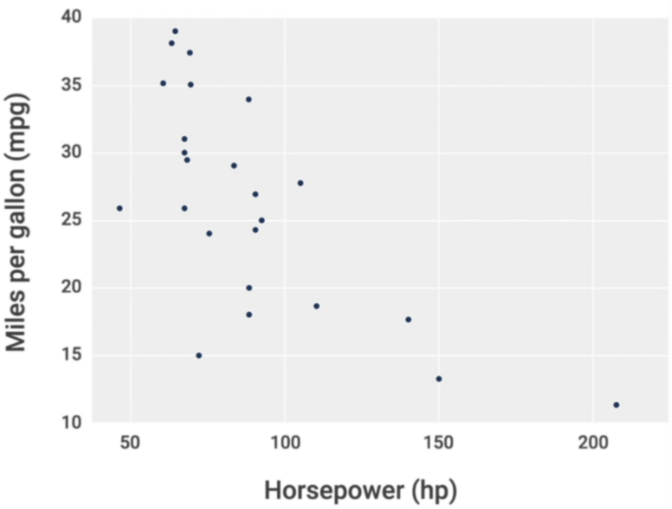
Stochastic gradient descent is a specific procedure that you can use to experimentally explore where it will land among all of the infinite possible models with zero training error. And stochastic gradient descent yields a solution, which is implicitly regularized. In other words, even without an explicit regularization penalty, over-parameterized models that are trained with stochastic gradient descent act like they are regularized.

As the number of parameters in a model grows, the impact of this effect increases. The net result resembles Dr. Witten's post; stochastic gradient descent results in a model that is implicitly regularized, yielding less wiggly behavior. Counterintuitively, because the 36-degree model has a greater implicit regularization strength, it is actually less free than a 20-degree model.

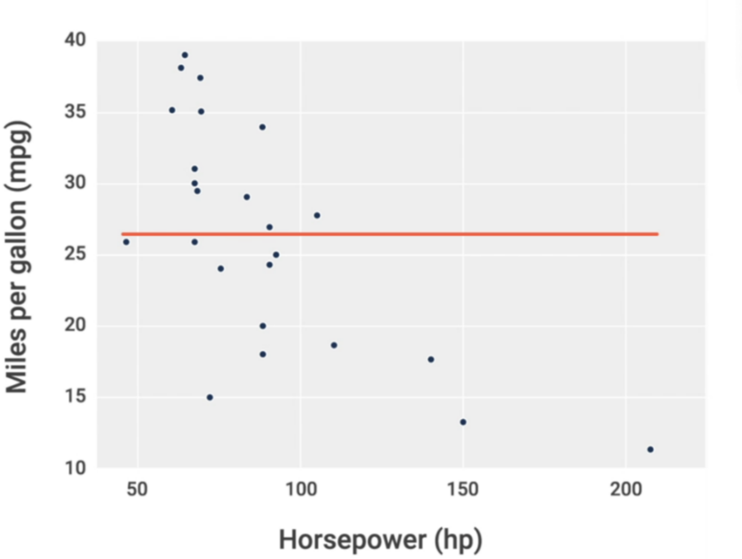
The Bias–Variance Tradeoff

You can think of the error of a model as stemming from two sources: The **bias** and the **variance**. The bias represents the fundamental inability of a model to fit the data, no matter what parameters are provided.

For example, consider the Miles per gallon dataset shown.

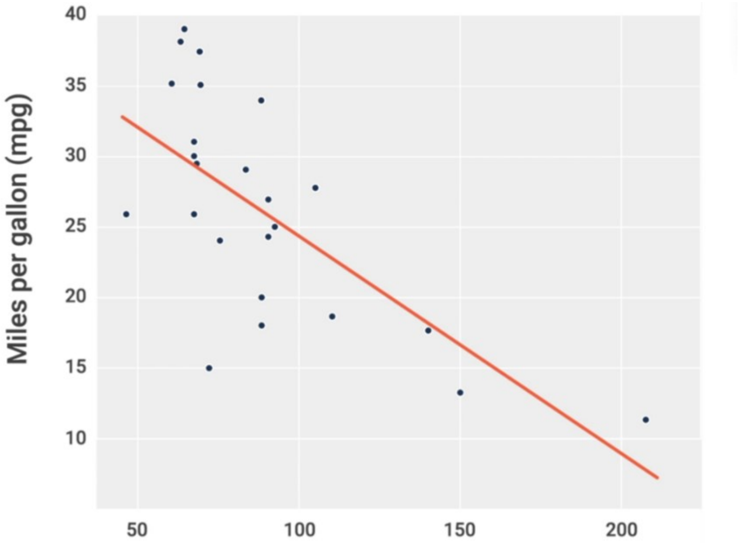


What happens if you fit a model which only has an intercept term?

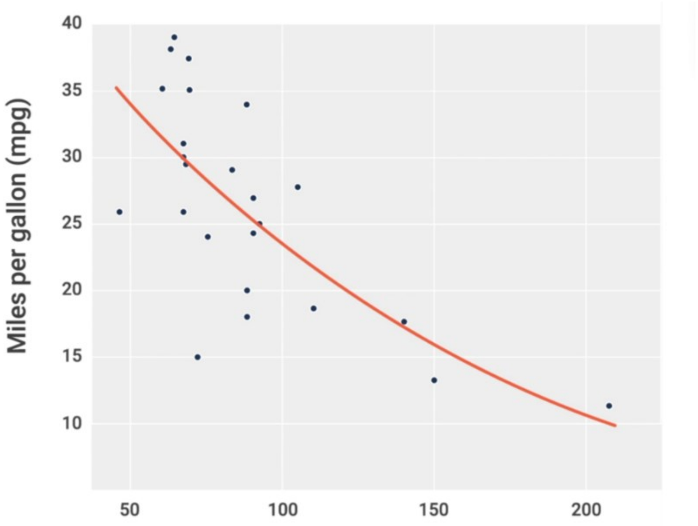


It believes all vehicles have the same fuel efficiency. Such a model is unable to capture the dependency that is apparent on the plot with the independent variable. The bias here is **high**. Such a model has a preconceived notion—or bias—that all vehicles have the same fuel efficiency.

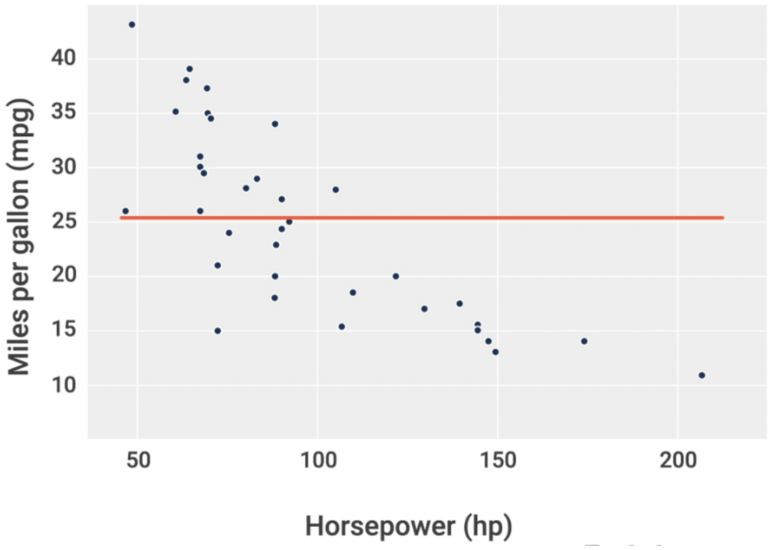
Now, if you fit a model with an intercept and a slope, you have a model like the one shown, a linear regression line. Here, the bias is **lower**.



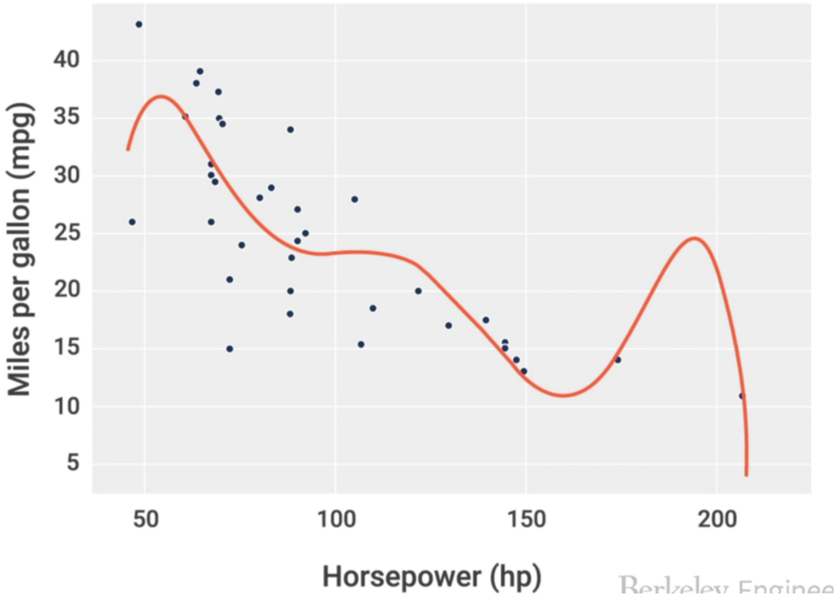
If you then fit a degree two polynomial, you have a model that is able to capture this roughly parabolic shape observed in the data.



Now the bias is even lower. If you were to fit a degree 25 polynomial—not shown—bias will go to zero. Your model would be able to perfectly fit the training data; a very high degree model with low or even zero bias. But it has a serious problem: **variance**. Variance represents how sensitive the model is to the data. This model, the first one with only an intercept term, has low variance. Small changes to the data result in small changes to the model.



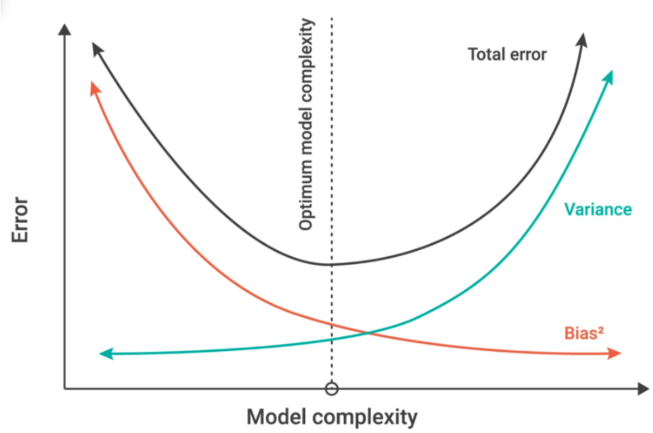
Now by contrast, a higher-order model like the degree-6 model shown, has high variance. Even a small change to a single data point can result in a dramatically different model.



Bias and variance can be given formal mathematical definitions. When formally defined, the expected loss—also known as the risk—is equal to the square of the bias plus the variance plus another term you will not describe.

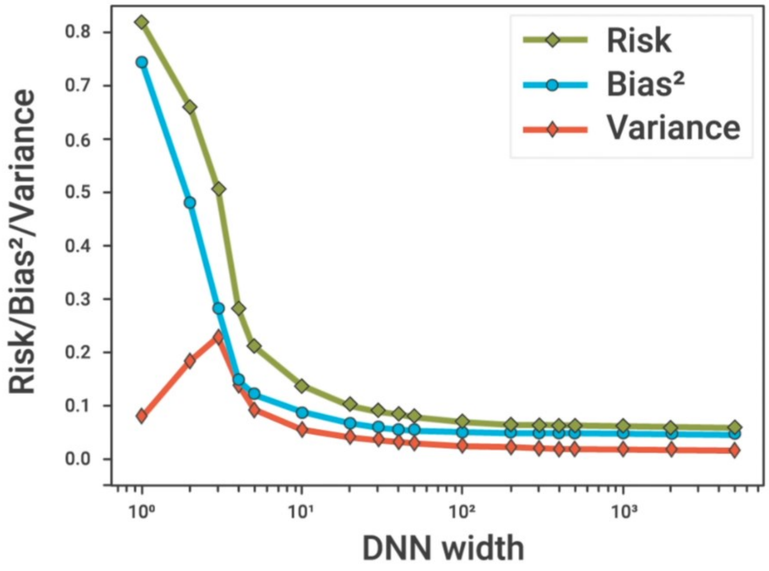
$$model\,risk = \sigma^2 + (model\,bias)^2 + model\,variance$$

As model complexity increases, bias decreases; variance tends to increase.



So, visually, there is the error, the bias^2 and the variance of a model, versus the complexity in the classical regime. In this regime, an ideal model sits at the sweet spot where the bias has gotten fairly low, but the variance has not yet taken off. By contrast, in the modern interpolating regime, the variance often increases, but starts to decrease as you go deeper into the interpolating regime.

The figure below is from *Rethinking Bias–Variance Trade-Off for Generalization of Neural Networks*, a paper by Zitong Yang and other authors. In this paper, the authors experimentally compute the bias^2, the variance, and the expected loss for a model as a function of the model's complexity.



The classical picture of the bias–variance tradeoff is shown at the leftmost part of this figure. Bias is decreasing, but variance is increasing. The sweet spot, representing the optimal model, is classically assumed to exist in this space.

However, as you move into the interpolating regime, the variance begins to drop again, eventually reaching lower levels than for simpler models. At the same time, the bias continues to decrease as well. As a result, the optimal model is out toward the right-hand side of the figure. In this case, you might even say there is no real bias–variance tradeoff. Instead, you might want to build your model as big as possible, as such a model minimizes both bias and variance.

In common practice today, many of the world's most advanced machine learning models are these absolutely gigantic neural networks with hundreds of billions of parameters or even more. For such models, the number of parameters vastly exceeds the amount of available data. Then, stochastic gradient descent chooses from among the infinite number of models, with zero or very low training error.