Linear Regression

Avinash A S

Linear regression is a statistical technique used to find the relationship between variables. In an ML context, linear regression finds the relationship between **features** and a **label**.

For example, suppose we want to predict a car's fuel efficiency in miles per gallon based on how heavy the car is, and we have the following dataset:

Pounds in 1000s (feature)	Miles per gallon (label)
3.5	18
3.69	15
3.44	18
3.43	16
4.34	15
4.42	14
2.37	24

If we plotted these points, we'd get the following graph:

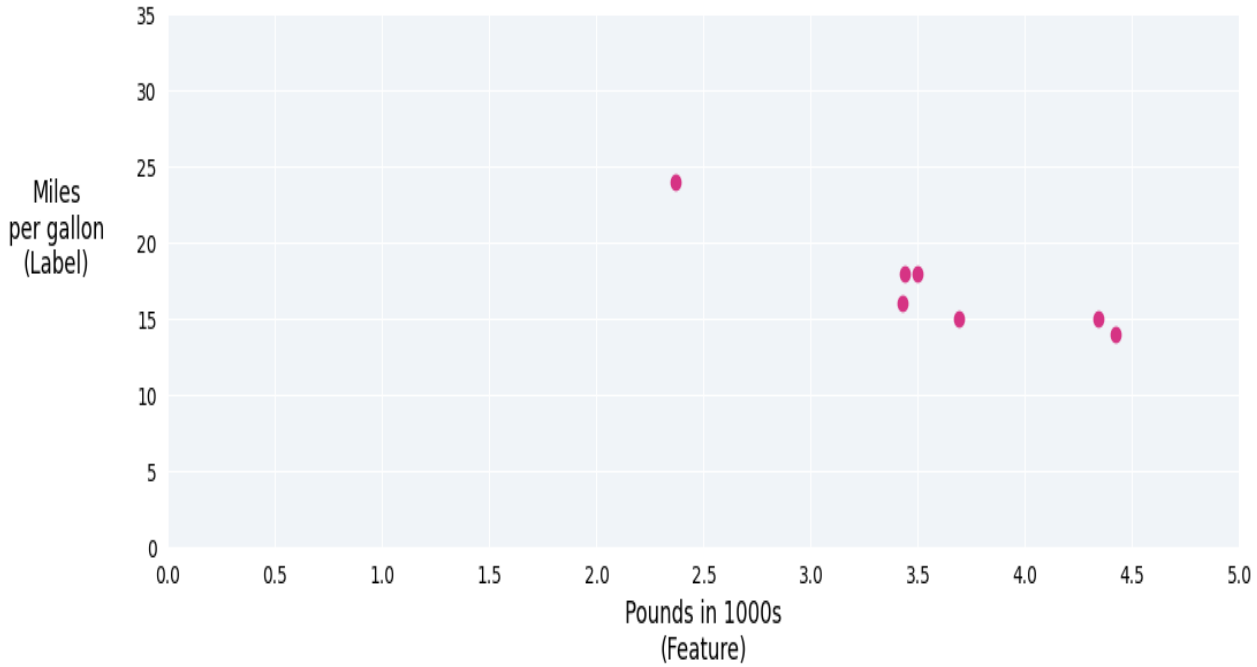

Figure 1. Car heaviness (in pounds) versus miles per gallon rating. As a car gets heavier, its miles per gallon rating generally decreases.

We could create our own model by drawing a best fit line through the points:

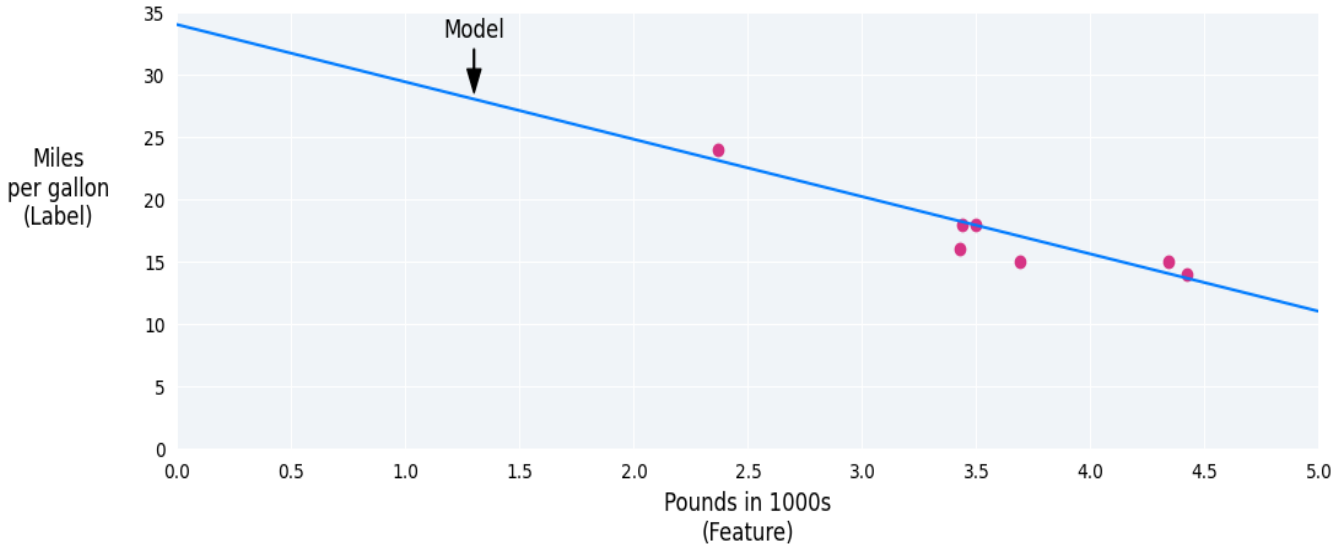

Figure 2. A best fit line drawn through the data from the previous figure.

Linear regression equation

In algebraic terms, the model would be defined as ,

$Y = mx + b$, where

- Y is miles per gallon—the value we want to predict.
- m is the slope of the line.
- x is pounds—our input value.
- b is the y-intercept.

In ML, we write the equation for a linear regression model as follows:

$$Y' = b + w_1 x_1$$

where:

- Y' is the predicted label—the output.
- b is the **bias** of the model. Bias is the same concept as the y-intercept in the algebraic equation for a line. In ML, bias is sometimes referred to as w_0. Bias is a **parameter** of the model and is calculated during training.
- w_1 is the **weight** of the feature. Weight is the same concept as the slope in the algebraic equation for a line. Weight is a **parameter** of the model and is calculated during training.

- x_1 is a **feature**—the input.

During training, the model calculates the weight and bias that produce the best model.

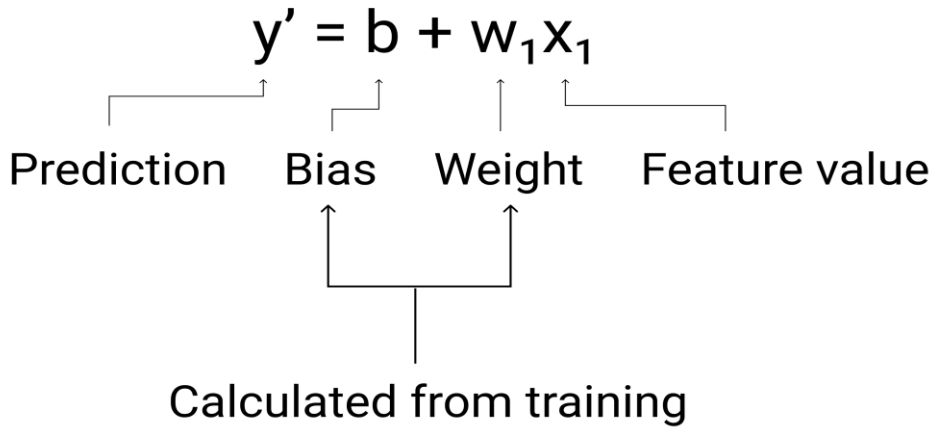

Figure 3. Mathematical representation of a linear model.

In our example, we'd calculate the weight and bias from the line we drew. The bias is 34 (where the line intersects the y-axis), and the weight is –4.6 (the slope of the line). The model would be defined as y'=34+(-4.6)(x1) , and we could use it to make predictions. For instance, using this model, a 4,000-pound car would have a predicted fuel efficiency of 15.6 miles per gallon.

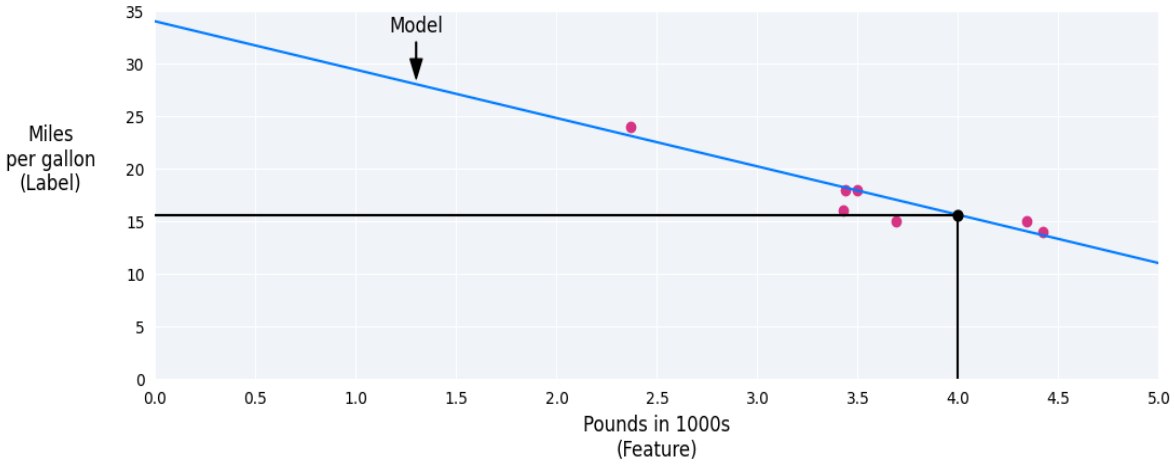

Figure 4. Using the model, a 4,000-pound car has a predicted fuel efficiency of 15.6 miles per gallon.

Models with multiple features

Although the example in this section uses only one feature—the heaviness of the car—a more sophisticated model might rely on multiple features, each having a separate weight (w_1, w_2 , etc.). For example, a model that relies on five features would be written as follows:

$$Y'=b+w_1x_1+w_2x_2+w_3x_3+w_4x_4+w_5x_5$$

For example, a model that predicts gas mileage could additionally use features such as the following:

- Engine displacement
- Acceleration
- Number of cylinders
- Horsepower

This model would be written as follows:

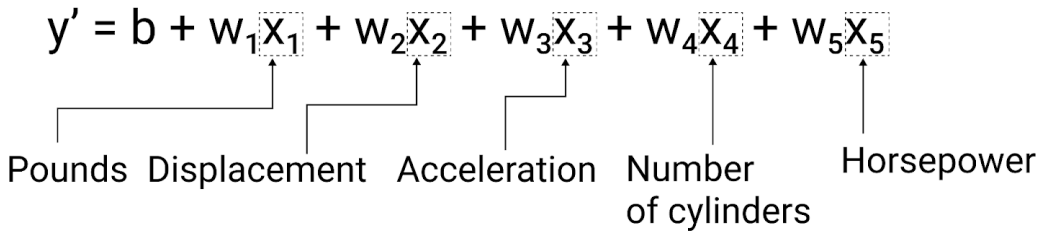

Figure 5. A model with five features to predict a car's miles per gallon rating.

By graphing some of these additional features, we can see that they also have a linear relationship to the label, miles per gallon:

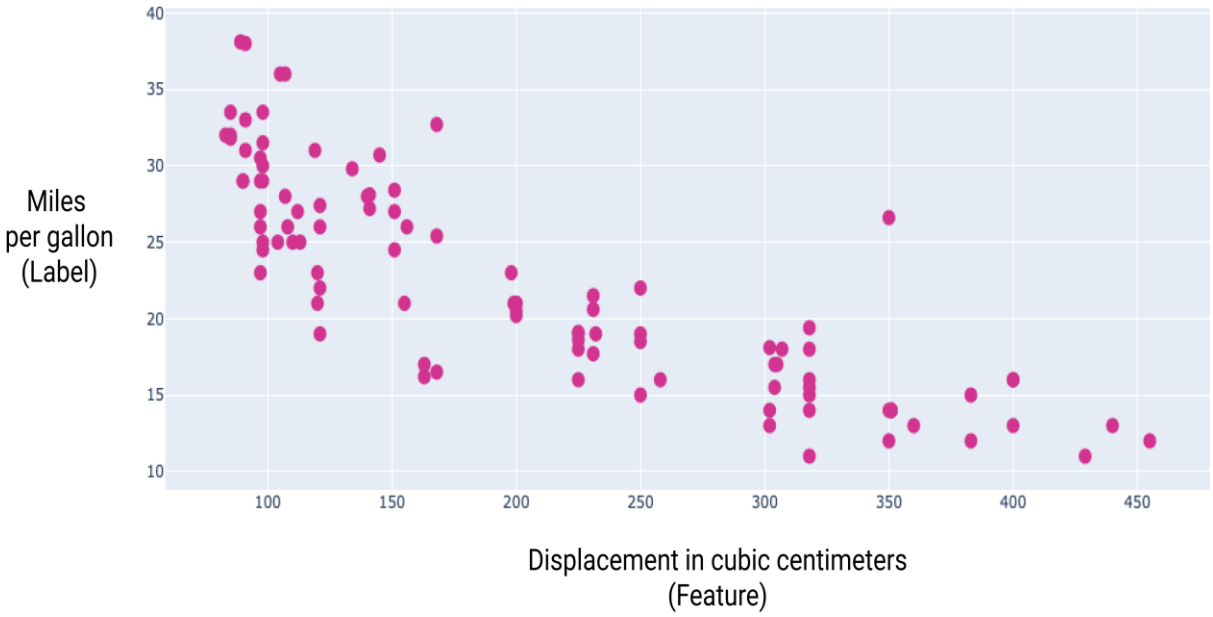

Figure 6. A car's displacement in cubic centimeters and its miles per gallon rating. As a car's engine gets bigger, its miles per gallon rating generally decreases.

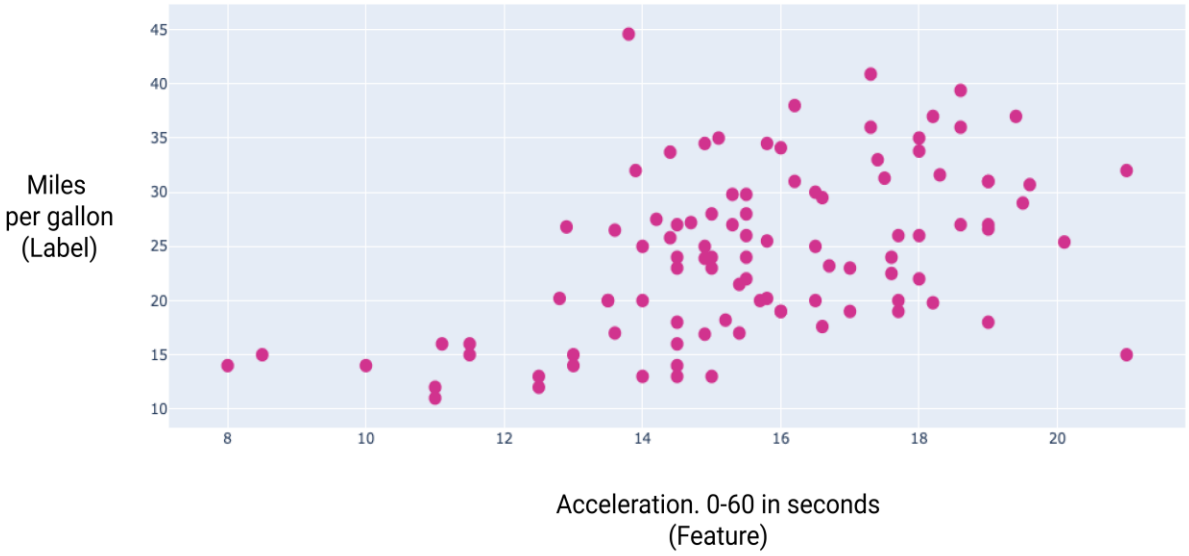

Figure 7. A car's acceleration and its miles per gallon rating. As a car's acceleration takes longer, the miles per gallon rating generally increases.

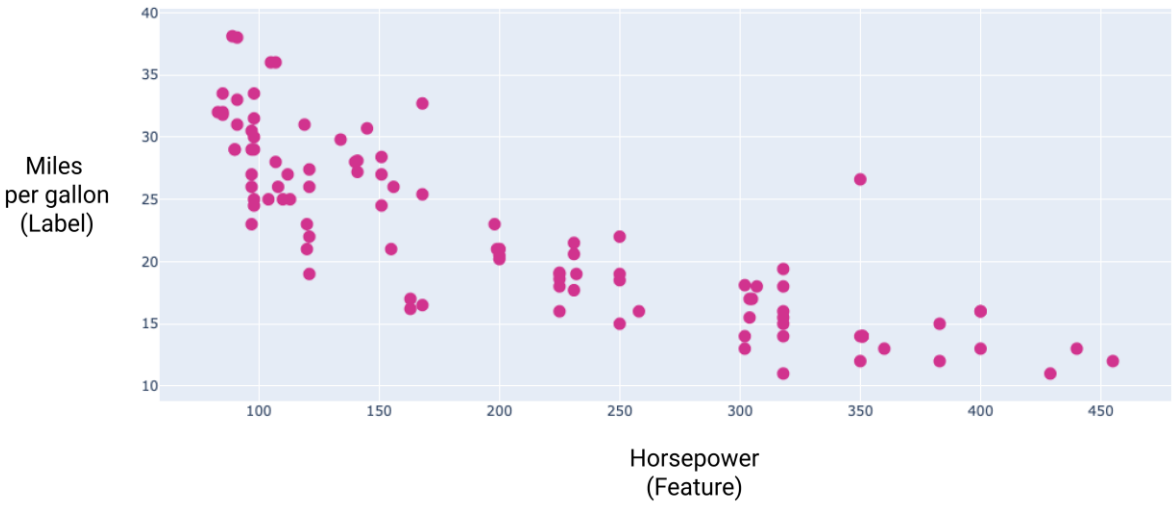

Figure 8. A car's horsepower and its miles per gallon rating. As a car's horsepower increases, the miles per gallon rating generally decreases.

LOSS

Loss is a numerical metric that describes how wrong a model's predictions are. Loss measures the distance between the model's predictions and the actual labels. The goal of training a model is to minimize the loss, reducing it to its lowest possible value.

In the following image, you can visualize loss as arrows drawn from the data points to the model. The arrows show how far the model's predictions are from the actual values.

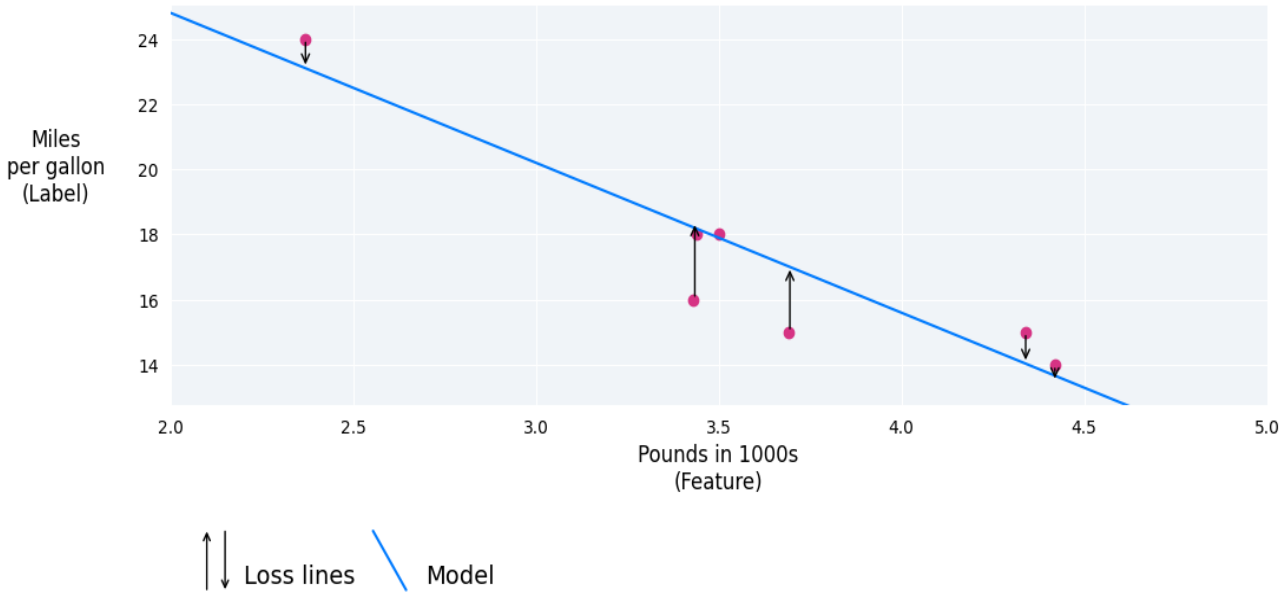

Figure 9. Loss is measured from the actual value to the predicted value.

Distance of loss

In statistics and machine learning, loss measures the difference between the predicted and actual values. Loss focuses on the *distance* between the values, not the direction. For example, if a model predicts 2, but the actual value is 5, we don't care that the loss is negative (2-5=-3). Instead, we care that the *distance* between the values is 3. Thus, all methods for calculating loss remove the sign.

The two most common methods to remove the sign are the following:

- **Take the absolute value of the difference between the actual value and the prediction.**
- **Square the difference between the actual value and the prediction.**

Types of loss

In linear regression, there are four main types of loss, which are outlined in the following table.

Loss type	Definition	Equation
L_1 loss	**The sum of the absolute values of the difference between the predicted values and the actual values.**	$\sum \lvert actual\ value - predicted\ value \rvert$
Mean absolute error (MAE)	**The average of L_1 losses across a set of *N* examples.**	$\frac{1}{N} \sum \lvert actual\ value - predicted\ value \rvert$
L_2 loss	**The sum of the squared difference between the predicted values and the actual values.**	$\sum (actual\ value - predicted\ value)^2$
Mean squared error (MSE)	**The average of L_2 losses across a set of *N* examples.**	$\frac{1}{N} \sum (actual\ value - predicted\ value)^2$

The functional difference between L_1 loss and L_2 loss (or between MAE and MSE) is squaring. When the difference between the prediction and label is large, squaring makes the loss even larger. When the difference is small (less than 1), squaring makes the loss even smaller.

When processing multiple examples at once, we recommend averaging the losses across all the examples, whether using MAE or MSE.

Calculating loss example

Using the previous best fit line, we'll calculate L_2 loss for a single example. From the best fit line, we had the following values for weight and bias:

- ***Weight : -4.6***
- ***Bias : 34***

If the model predicts that a 2,370-pound car gets 23.1 miles per gallon, but it actually gets 26 miles per gallon, we would calculate the L_2 loss as follows:

Value	Equation	Result
Prediction	$bias + (weight * feature\ value)$ $34 + (-4.6 * 2.37)$	**23.1**
Actual value	$label$	**26**

L_2 loss	$(actual\ value - predicted\ value)^2$ $(26 - 23.1)^2$	**8.41**

In this example, the L_2 loss for that single data point is 8.41.

Choosing a loss

Deciding whether to use MAE or MSE can depend on the dataset and the way you want to handle certain predictions. Most feature values in a dataset typically fall within a distinct range. For example, cars are normally between 2000 and 5000 pounds and get between 8 to 50 miles per gallon. An 8,000-pound car, or a car that gets 100 miles per gallon, is outside the typical range and would be considered an outlier.

An outlier can also refer to how far off a model's predictions are from the real values. For instance, 3,000 pounds is within the typical car-weight range, and 40 miles per gallon is within the typical fuel-efficiency range. However, a 3,000-pound car that gets 40 miles per gallon would be an outlier in terms of the model's prediction because the model would predict that a 3,000-pound car would get around 20 miles per gallon.

When choosing the best loss function, consider how you want the model to treat outliers. For instance, MSE moves the model more toward the outliers, while MAE doesn't. L_2 loss incurs a much higher penalty for an outlier than L_1 loss. For example, the following images show a model trained using MAE and a model trained using MSE. The red line represents a fully trained model that will be used to make predictions. The outliers are closer to the model trained with MSE than to the model trained with MAE.

Figure 10. A model trained with MSE moves the model closer to the outliers.

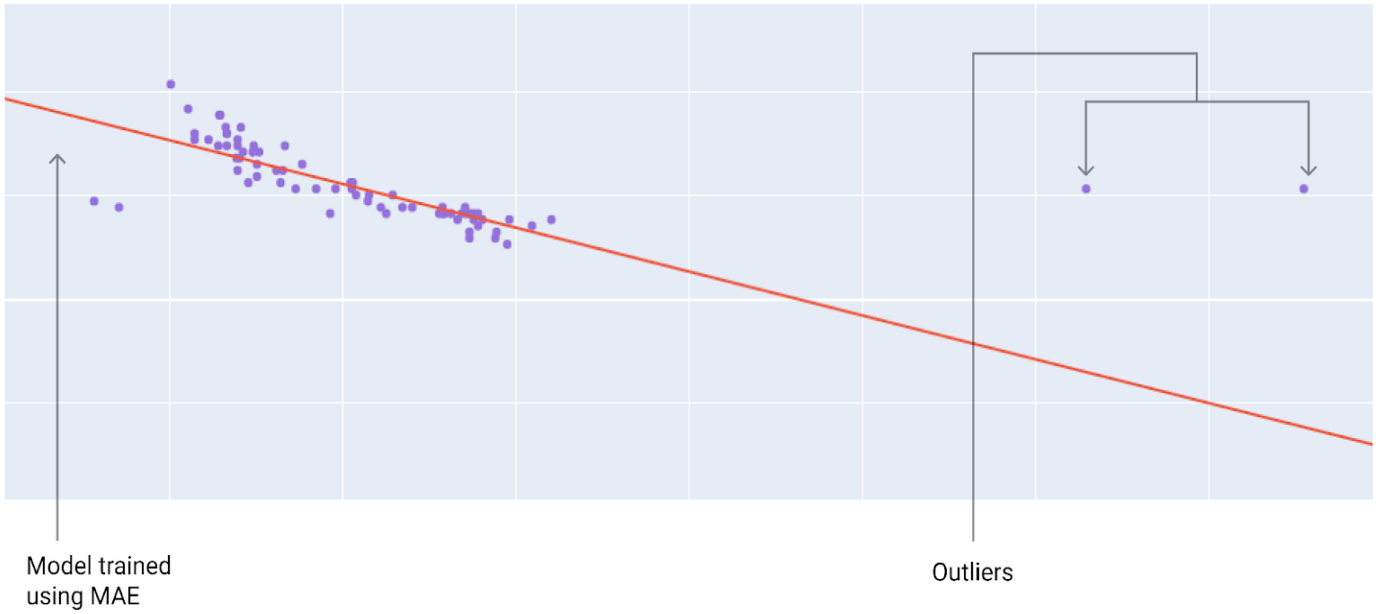

Figure 11. A model trained with MAE is farther from the outliers.

Note the relationship between the model and the data:

- **MSE. The model is closer to the outliers but further away from most of the other data points.**
- **MAE. The model is further away from the outliers but closer to most of the other data points.**

Gradient descent

Gradient descent is a mathematical technique that iteratively finds the weights and bias that produce the model with the lowest loss. Gradient descent finds the best weight and bias by repeating the following process for a number of user-defined iterations.

The model begins training with randomized weights and biases near zero, and then repeats the following steps:

1. Calculate the loss with the current weight and bias.
2. Determine the direction to move the weights and bias that reduce loss.
3. Move the weight and bias values a small amount in the direction that reduces loss.
4. Return to step one and repeat the process until the model can't reduce the loss any further.

The diagram below outlines the iterative steps gradient descent performs to find the weights and bias that produce the model with the lowest loss.

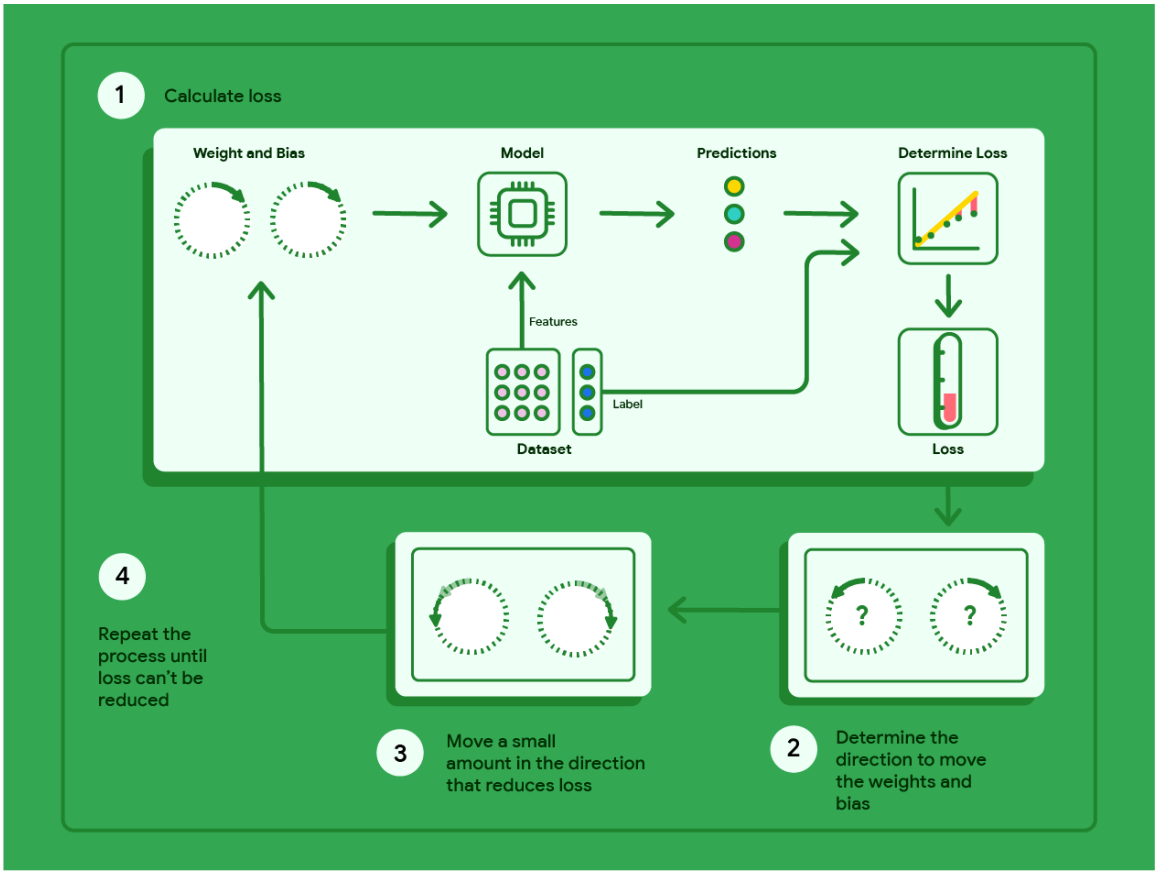

Figure 12. Gradient descent is an iterative process that finds the weights and bias that produce the model with the lowest loss.

learn more about the math behind gradient descent.

At a concrete level, we can walk through the gradient descent steps using a small dataset with seven examples for a car's heaviness in pounds and its miles per gallon rating:

Pounds in 1000s (feature)	Miles per gallon (label)
3.5	18
3.69	15
3.44	18
3.43	16
4.34	15
4.42	14
2.37	24

1. The model starts training by setting the weight and bias to zero:

   Weight : 0

   Bias : 0

Y=0+0(x_1)

2. Calculate MSE loss with the current model parameters:

$$Loss = \frac{(18-0)^2 + (15-0)^2 + (18-0)^2 + (16-0)^2 + (15-0)^2 + (14-0)^2 + (24-0)^2}{7}$$

$$Loss = 303.71$$

3. Calculate the slope of the tangent to the loss function at each weight and the bias:

   Weight slope : -119.7

   Bias slope : -34.3

4. Move a small amount in the direction of the negative slope to get the next weight and bias. For now, we'll arbitrarily define the "small amount" as 0.01:

$$New\ weight = old\ weight - (small\ amount * weight\ slope)$$

$$New\ bias = old\ bias - (small\ amount * bias\ slope)$$

$$New\ weight = 0 - (0.01) * (-119.7)$$

$$New\ bias = 0 - (0.01) * (-34.3)$$

$$New\ weight = 1.2$$

$$New\ bias = 0.34$$

Use the new weight and bias to calculate the loss and repeat. Completing the process for six iterations, we'd get the following weights, biases, and losses:

Iteration	Weight	Bias	Loss (MSE)
1	0	0	303.71
2	1.20	0.34	170.84
3	2.05	0.59	103.17
4	2.66	0.78	68.70
5	3.09	0.91	51.13
6	3.40	1.01	42.17

You can see that the loss gets lower with each updated weight and bias. In this example, we stopped after six iterations. In practice, a model trains until it **converges**. When a model converges, additional iterations don't reduce loss more because gradient descent has found the weights and bias that nearly minimize the loss.

If the model continues to train past convergence, loss begins to fluctuate in small amounts as the model continually updates the parameters around their lowest values. This can make it hard to verify that the model has actually converged. To confirm the model has converged, you'll want to continue training until the loss has stabilized.

Model convergence and loss curves

When training a model, you'll often look at a **loss curve** to determine if the model has **converged**. The loss curve shows how the loss changes as the model trains. The following is what a typical loss curve looks like. Loss is on the y-axis and iterations are on the x-axis:

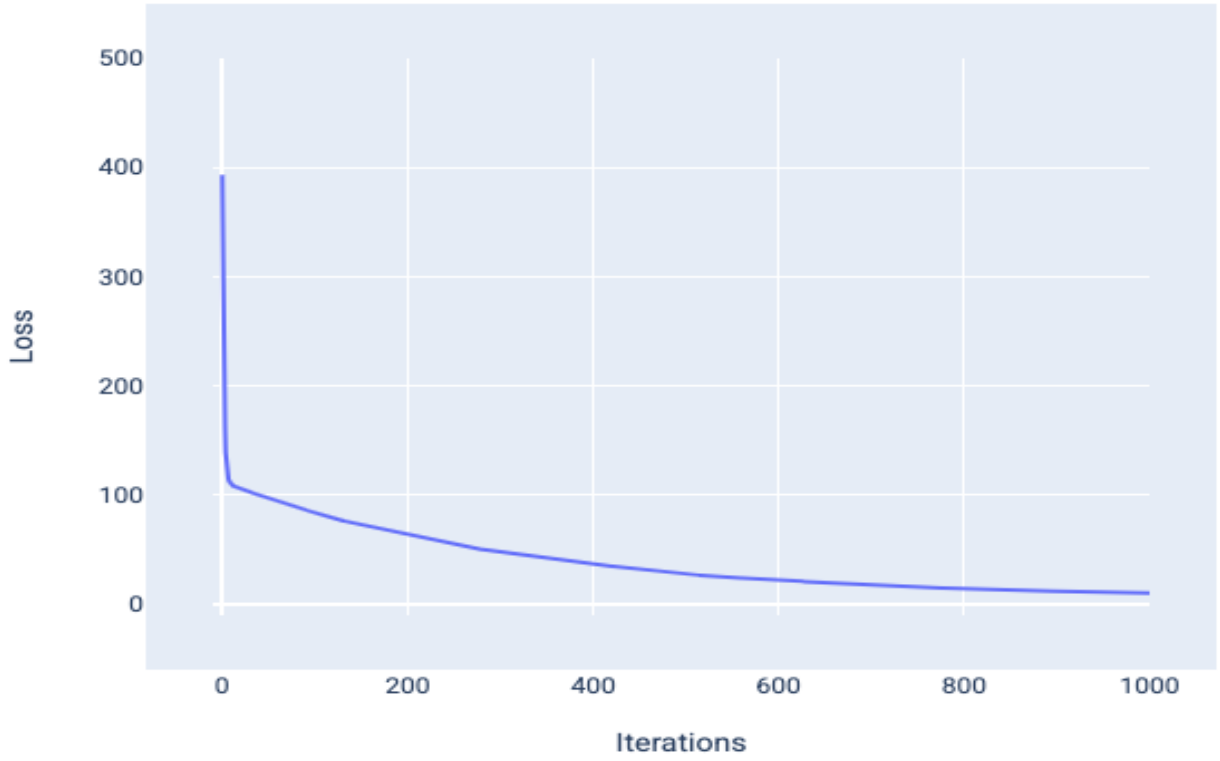

Figure 13. Loss curve showing the model converging around the 1,000th-iteration mark.

You can see that loss dramatically decreases during the first few iterations, then gradually decreases before flattening out around the 1,000th-iteration mark. After 1,000 iterations, we can be mostly certain that the model has converged.

In the following figures, we draw the model at three points during the training process: the beginning, the middle, and the end. Visualizing the model's state at snapshots during the training process solidifies the link between updating the weights and bias, reducing loss, and model convergence.

In the figures, we use the derived weights and bias at a particular iteration to represent the model. In the graph with the data points and the model snapshot, blue loss lines from the model to the data points show the amount of loss. The longer the lines, the more loss there is.

In the following figure, we can see that around the second iteration the model would not be good at making predictions because of the high amount of loss.

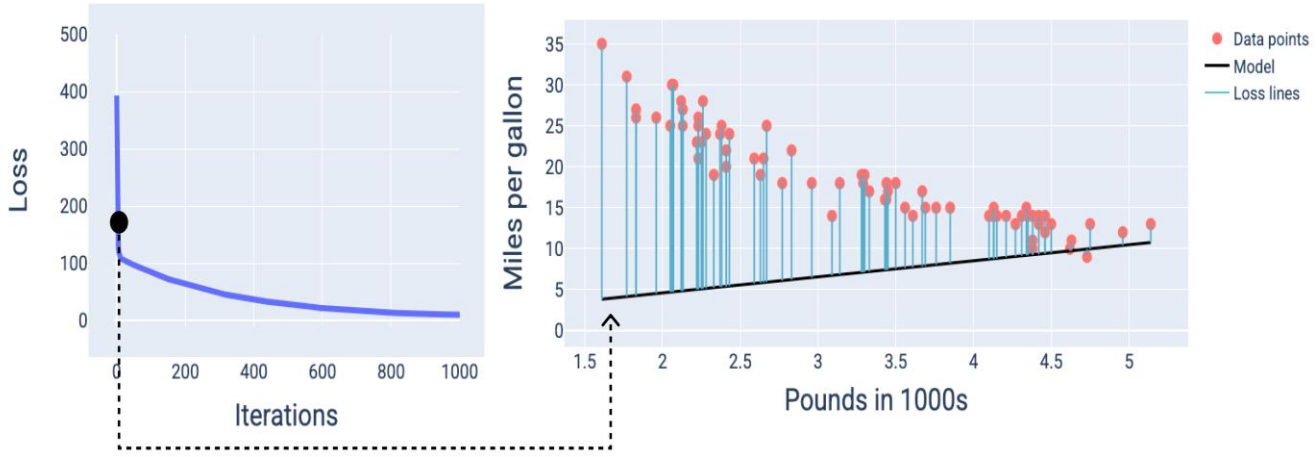

Figure 14. Loss curve and snapshot of the model at the beginning of the training process.

At around the 400th-iteration, we can see that gradient descent has found the weight and bias that produce a better model.

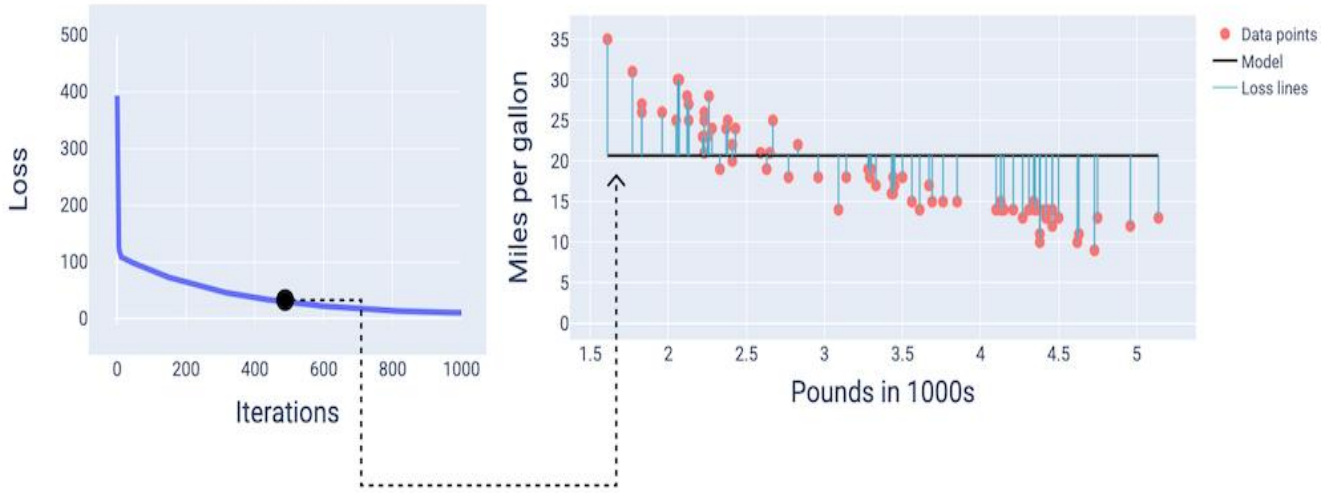

Figure 15. Loss curve and snapshot of model about midway through training.

And at around the 1,000th-iteration, we can see that the model has converged, producing a model with the lowest possible loss.

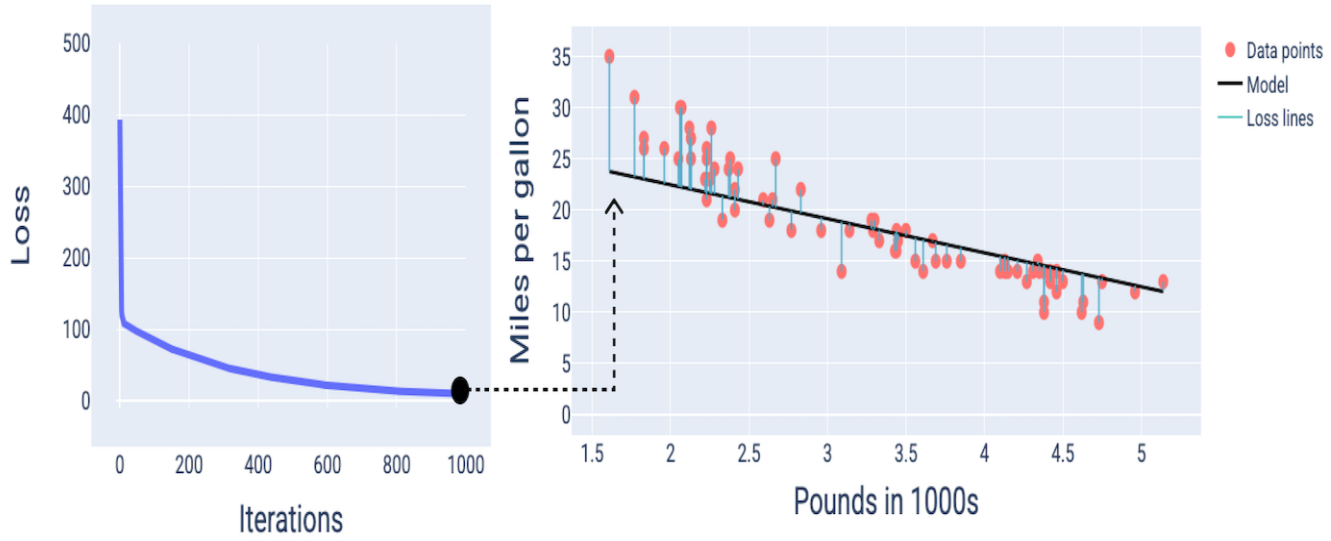

Figure 16. Loss curve and snapshot of the model near the end of the training process.

Convergence and convex functions

The loss functions for linear models always produce a **convex** surface. As a result of this property, when a linear regression model converges, we know the model has found the weights and bias that produce the lowest loss.

If we graph the loss surface for a model with one feature, we can see its convex shape. The following is the loss surface for a hypothetical miles per gallon dataset. Weight is on the x-axis, bias is on the y-axis, and loss is on the z-axis:

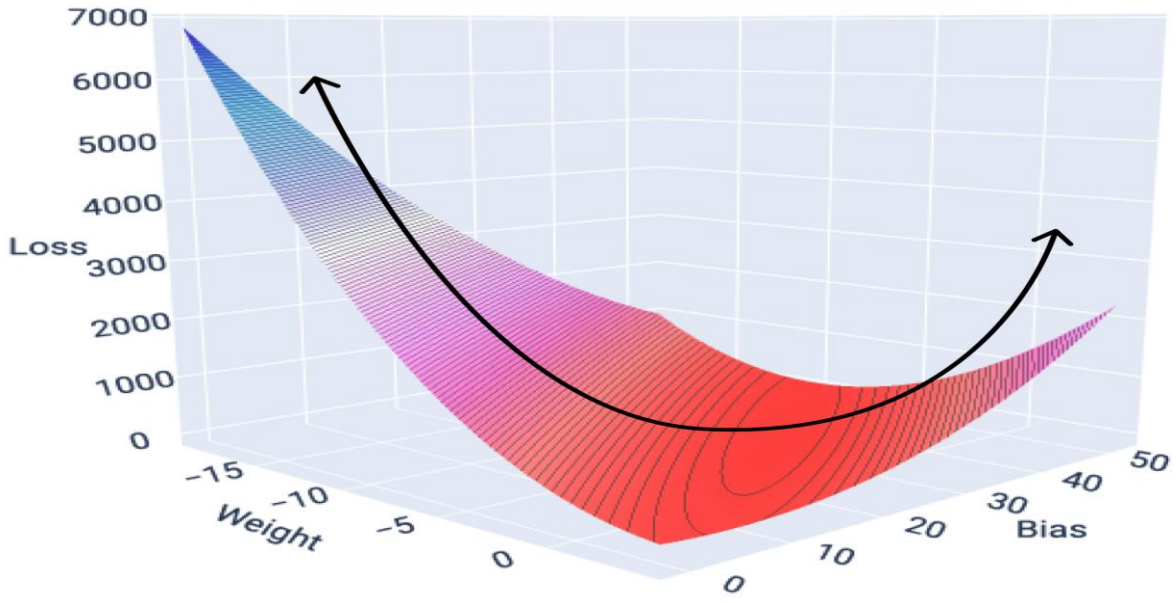

Figure 17. Loss surface that shows its convex shape.

In this example, a weight of -5.44 and bias of 35.94 produce the lowest loss at 5.54:

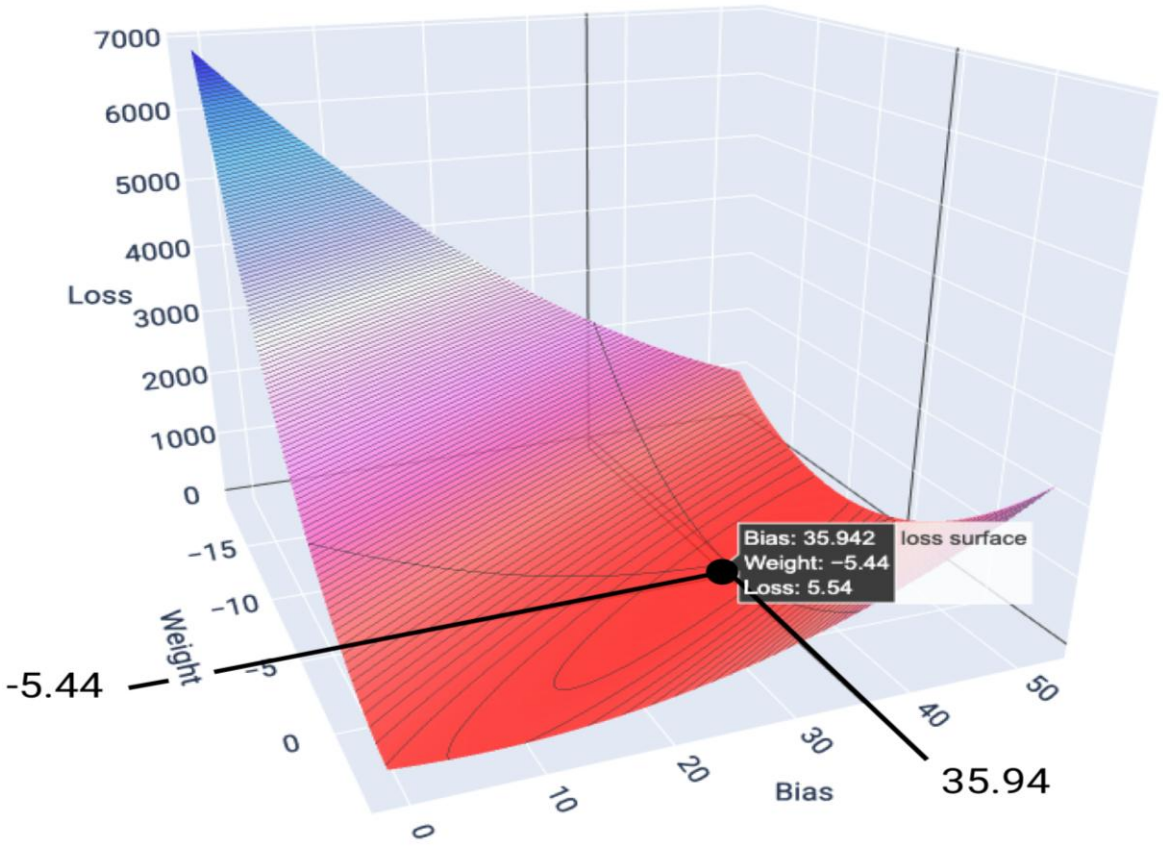

Figure 18. Loss surface showing the weight and bias values that produce the lowest loss.

A linear model converges when it's found the minimum loss. Therefore, additional iterations only cause gradient descent to move the weight and bias values in very small amounts around the minimum. If we graphed the weights and bias points during gradient descent, the points would look like a ball rolling down a hill, finally stopping at the point where there's no more downward slope.

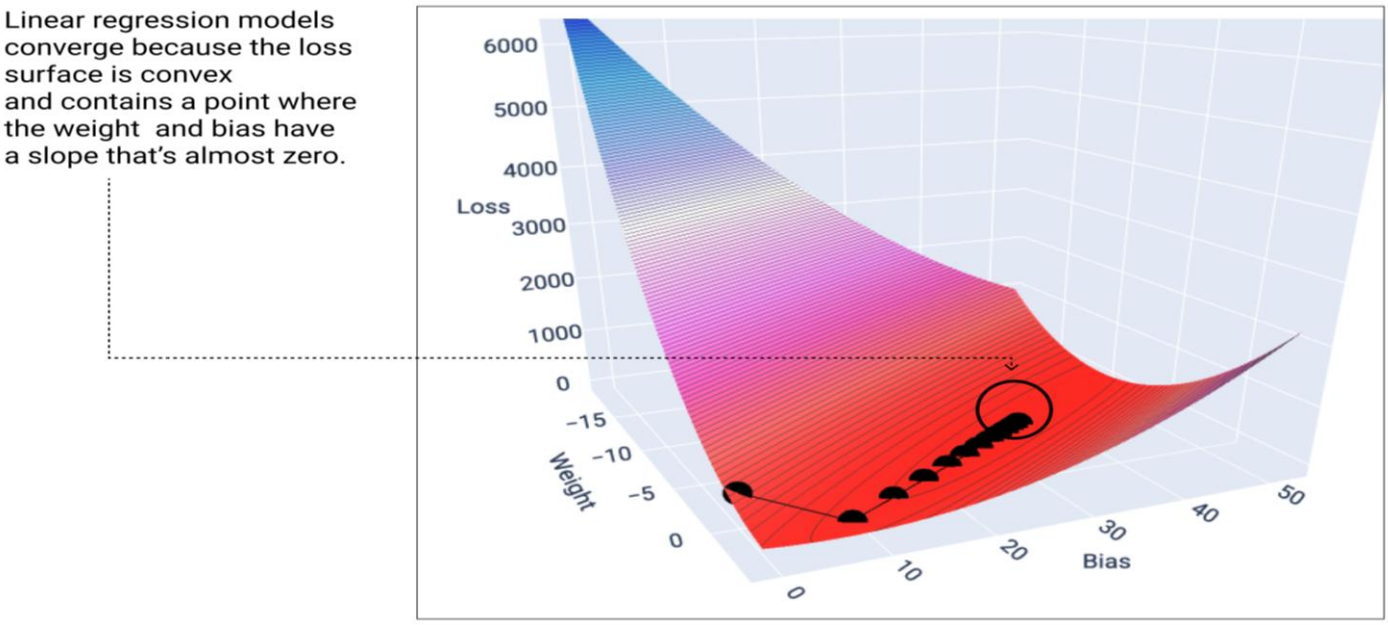

Figure 19. Loss graph showing gradient descent points stopping at the lowest point on the graph.

Notice that the black loss points create the exact shape of the loss curve: a steep decline before gradually sloping down until they've reached the lowest point on the loss surface.

It's important to note that the model almost never finds the exact minimum for each weight and bias, but instead finds a value very close to it. It's also important to note that the minimum for the weights and bias don't correspond to zero loss, only a value that produces the lowest loss for that parameter.

Using the weight and bias values that produce the lowest loss—in this case a weight of -5.44 and a bias of 35.94—we can graph the model to see how well it fits the data:

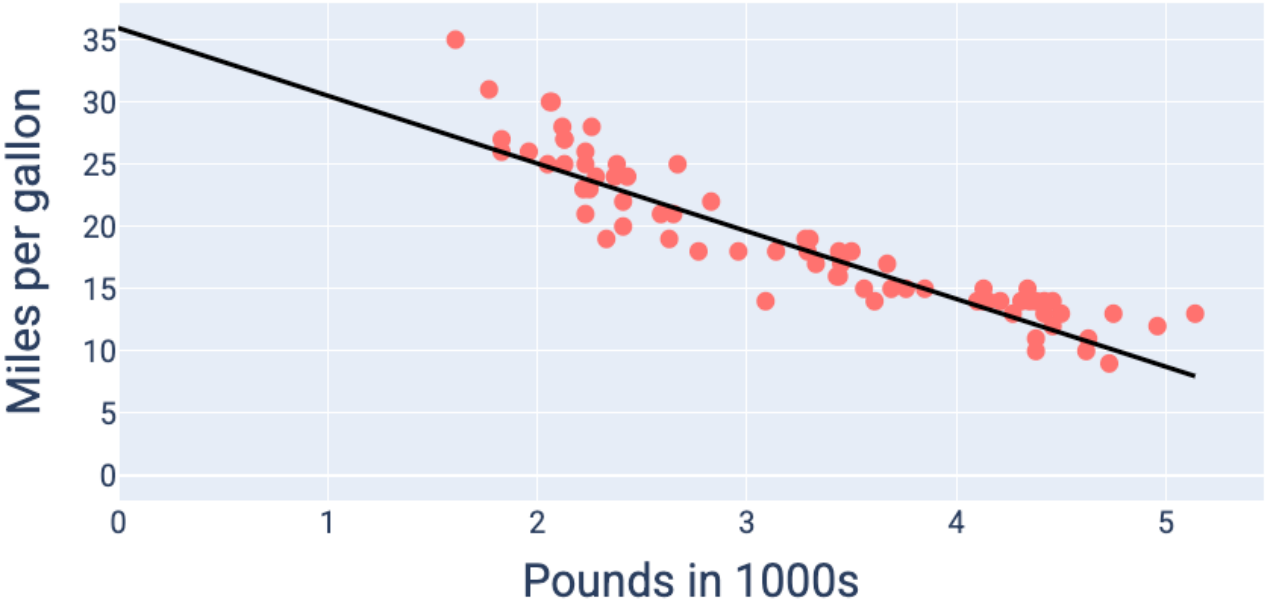

Figure 20. Model graphed using the weight and bias values that produce the lowest loss.

This would be the best model for this dataset because no other weight and bias values produce a model with lower loss.

Hyperparameters

Hyperparameters are variables that control different aspects of training. Three common hyperparameters are:

- **Learning rate**
- **Batch size**
- **Epochs**

In contrast, **parameters** are the variables, like the weights and bias, that are part of the model itself. In other words, hyperparameters are values that you control; parameters are values that the model calculates during training.

Learning rate

Learning rate is a floating point number you set that influences how quickly the model converges. If the learning rate is too low, the model can take a long time to converge. However, if the learning rate is too high, the model never converges, but instead bounces around the weights and bias that minimize the loss. The goal is to pick a learning rate that's not too high nor too low so that the model converges quickly.

The learning rate determines the magnitude of the changes to make to the weights and bias during each step of the gradient descent process. The model multiplies the gradient by the learning rate to determine the model's parameters (weight and bias values) for the next iteration. In the third step of gradient descent, the "small amount" to move in the direction of negative slope refers to the learning rate.

The difference between the old model parameters and the new model parameters is proportional to the slope of the loss function. For example, if the slope is large, the model takes a large step. If small, it takes a small step. For example, if the gradient's magnitude is 2.5 and the learning rate is 0.01, then the model will change the parameter by 0.025.

The ideal learning rate helps the model to converge within a reasonable number of iterations. In Figure 21, the loss curve shows the model significantly improving during the first 20 iterations before beginning to converge:

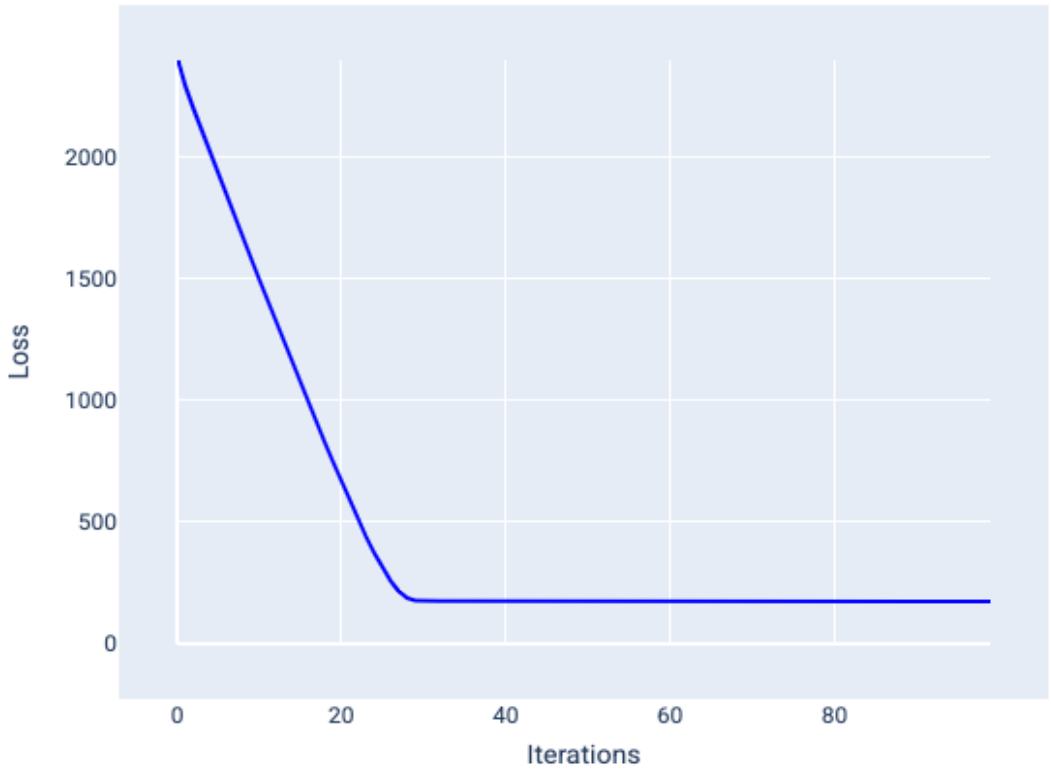

Figure 21. Loss graph showing a model trained with a learning rate that converges quickly.

In contrast, a learning rate that's too small can take too many iterations to converge. In Figure 22, the loss curve shows the model making only minor improvements after each iteration:

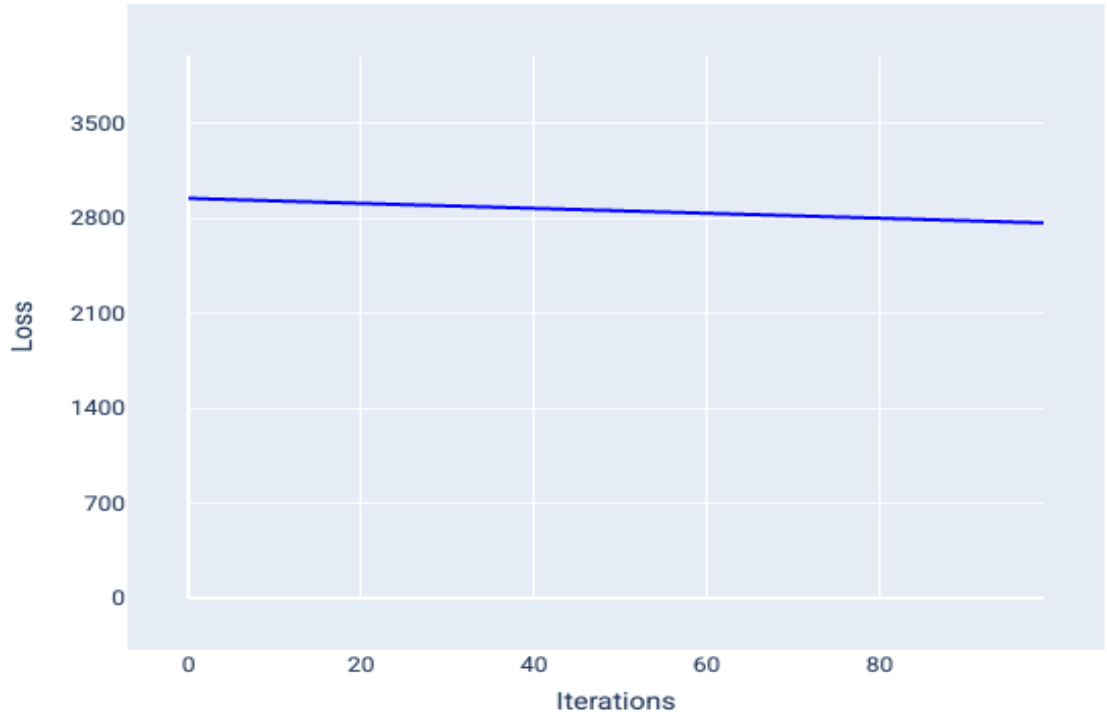

Figure 22. Loss graph showing a model trained with a small learning rate.

A learning rate that's too large never converges because each iteration either causes the loss to bounce around or continually increase. In Figure 23, the loss curve shows the model decreasing and then increasing loss after each iteration, and in Figure 24 the loss increases at later iterations:

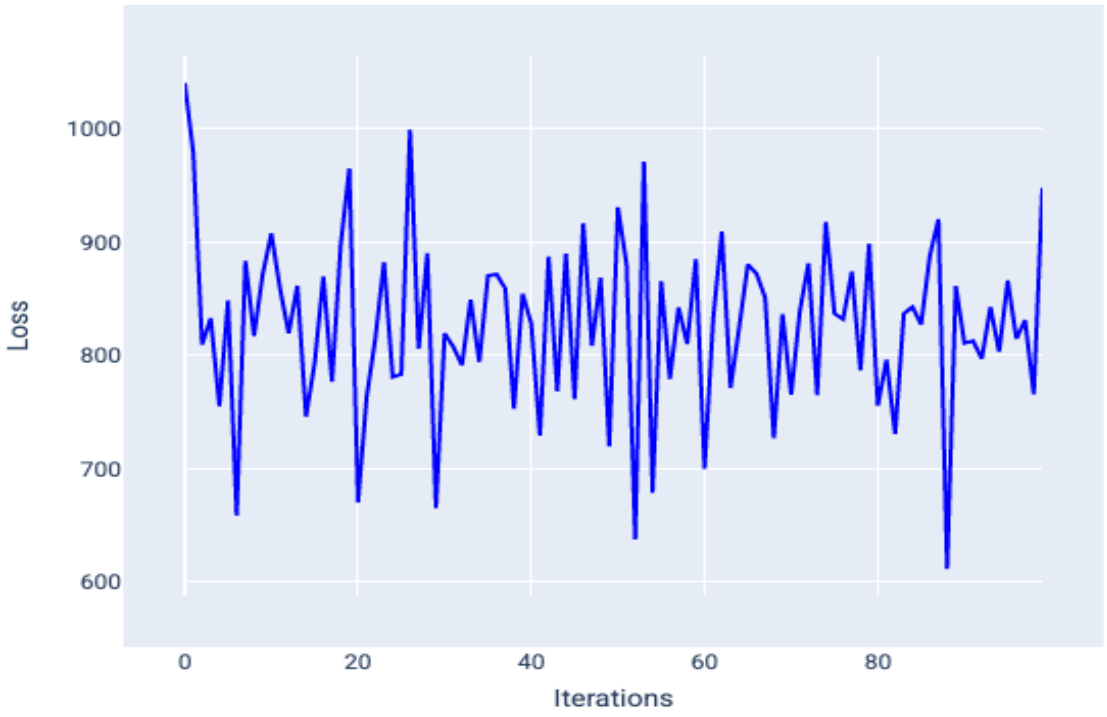

Figure 23. Loss graph showing a model trained with a learning rate that's too big, where the loss curve fluctuates wildly, going up and down as the iterations increase.

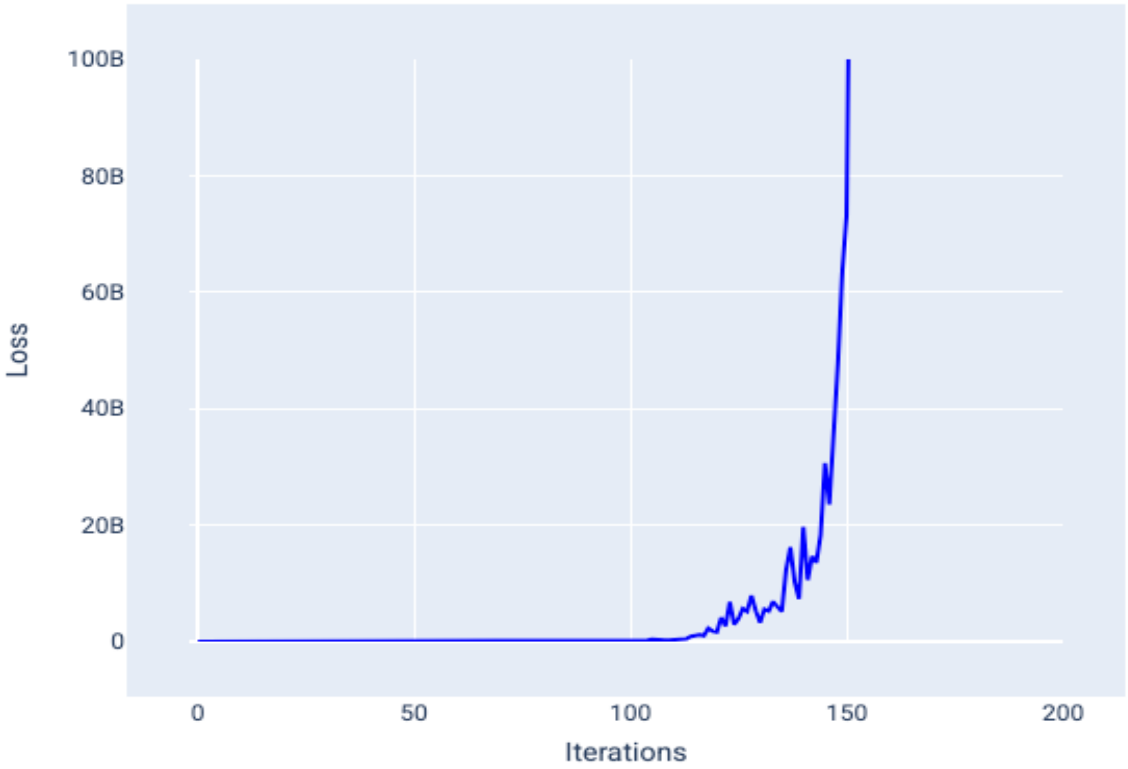

Figure 24. Loss graph showing a model trained with a learning rate that's too big, where the loss curve drastically increases in later iterations.

Batch size

Batch size is a hyperparameter that refers to the number of **examples** the model processes before updating its weights and bias. You might think that the model should calculate the loss for *every* example in the dataset before updating the weights and bias. However, when a dataset contains hundreds of thousands or even millions of examples, using the full batch isn't practical.

Two common techniques to get the right gradient on *average* without needing to look at every example in the dataset before updating the weights and bias are **stochastic gradient descent** and **mini-batch stochastic gradient descent**:

- **Stochastic gradient descent (SGD)**: Stochastic gradient descent uses only a single example (a batch size of one) per iteration. Given enough iterations, SGD works but is very noisy. "Noise" refers to variations during training that cause the loss to increase rather than decrease during an iteration. The term "stochastic" indicates that the one example comprising each batch is chosen at random.

Notice in the following image how loss slightly fluctuates as the model updates its weights and bias using SGD, which can lead to noise in the loss graph:

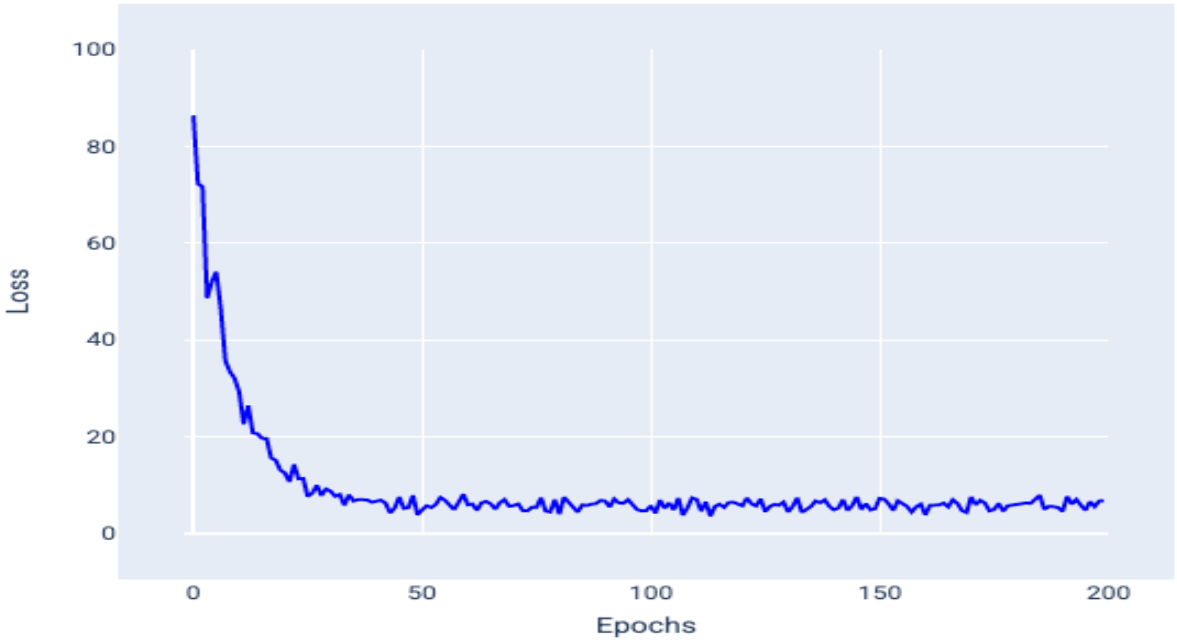

Figure 25. Model trained with stochastic gradient descent (SGD) showing noise in the loss curve.

Note that using stochastic gradient descent can produce noise throughout the entire loss curve, not just near convergence.

- **Mini-batch stochastic gradient descent (mini-batch SGD)**: Mini-batch stochastic gradient descent is a compromise between full-batch and SGD. For "N" number of data points, the batch size can be any number greater than 1 and less than "N". The model chooses the examples included in each batch at random, averages their gradients, and then updates the weights and bias once per iteration.

Determining the number of examples for each batch depends on the dataset and the available compute resources. In general, small batch sizes behaves like SGD, and larger batch sizes behaves like full-batch gradient descent.

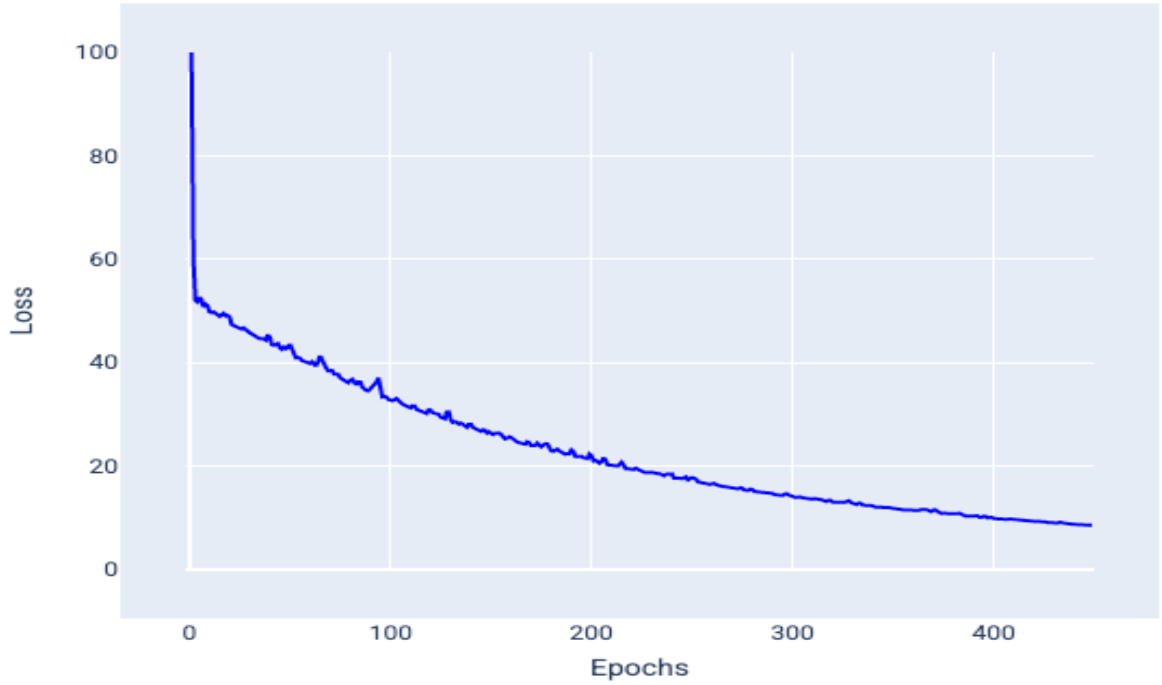

Figure 26. Model trained with mini-batch SGD.

When training a model, you might think that noise is an undesirable characteristic that should be eliminated. However, a certain amount of noise can be a good thing. In later modules, you'll learn how noise can help a model **generalize** better and find the optimal weights and bias in a **neural network**.

Epochs

During training, an **epoch** means that the model has processed every example in the training set *once*. For example, given a training set with 1,000 examples and a mini-batch size of 100 examples, it will take the model 10 **iterations** to complete one epoch.

Training typically requires many epochs. That is, the system needs to process every example in the training set multiple times.

The number of epochs is a hyperparameter you set before the model begins training. In many cases, you'll need to experiment with how many epochs it takes for the model to converge. In general, more epochs produces a better model, but also takes more time to train.

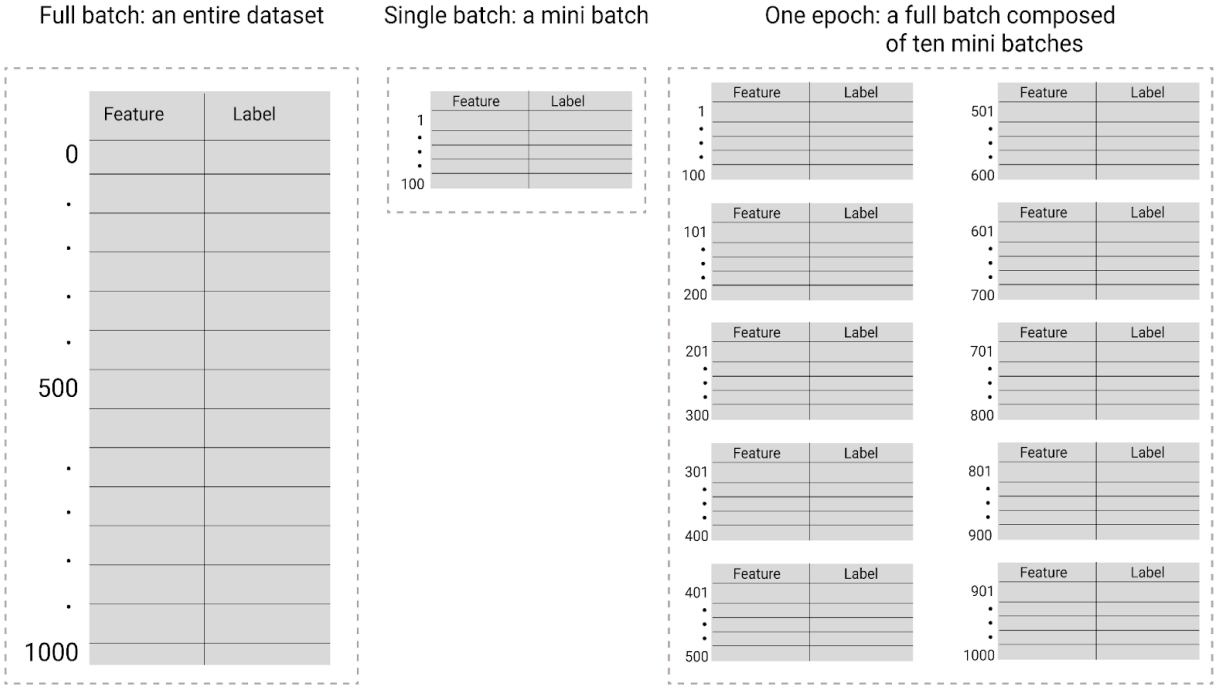

Figure 27. Full batch versus mini batch.

The following table describes how batch size and epochs relate to the number of times a model updates its parameters.

Batch type	When weights and bias updates occur
Full batch	After the model looks at all the examples in the dataset. For instance, if a dataset contains 1,000 examples and the model trains for 20 epochs, the model updates the weights and bias 20 times, once per epoch.
Stochastic gradient descent	After the model looks at a single example from the dataset. For instance, if a dataset contains 1,000 examples and trains for 20 epochs, the model updates the weights and bias 20,000 times.
Mini-batch stochastic gradient descent	After the model looks at the examples in each batch. For instance, if a

	dataset contains 1,000 examples, and the batch size is 100, and the model trains for 20 epochs, the model updates the weights and bias 200 times.