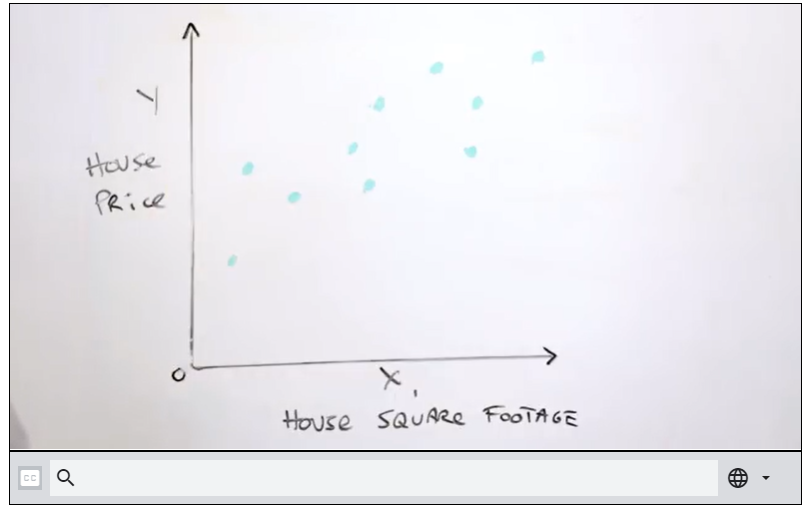
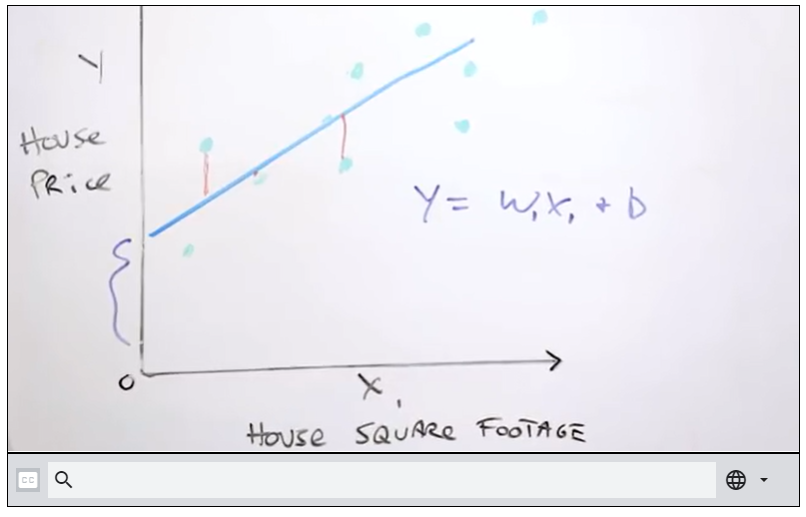
Descending into ML

**Linear regression** is a method for finding the straight line or hyperplane that best fits a set of points. This module explores linear regression intuitively before laying the groundwork for a machine learning approach to linear regression.

There are lots of ways to learn from data.

The simplest of these ways is fitting a line through the data points.





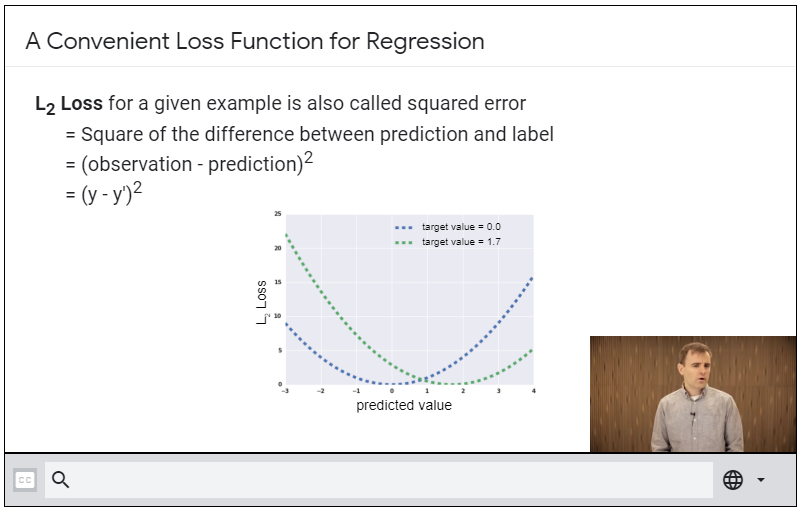
Here, the loss tells us how well our model is good at predicting the result.

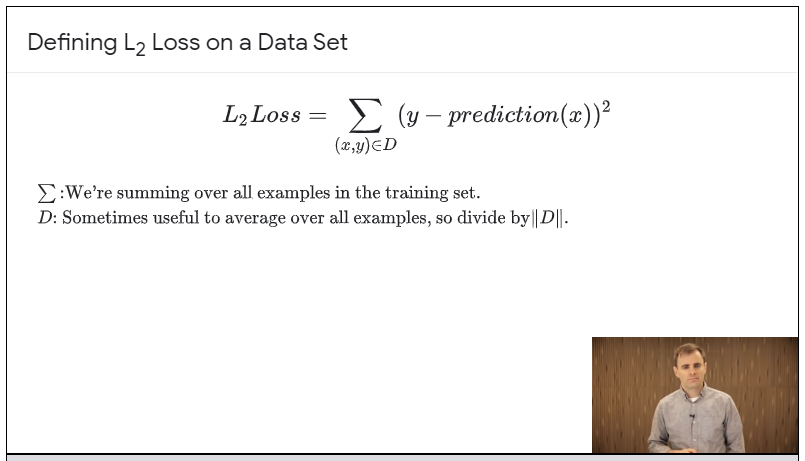
So, lesser loss means there is lesser difference between predicted value and actual value (which is the difference between the line and the point for a given X value – marked in red in the above picture.)

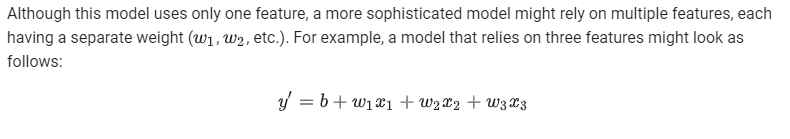
Loss is always positive

## Defining L2 loss on a data set

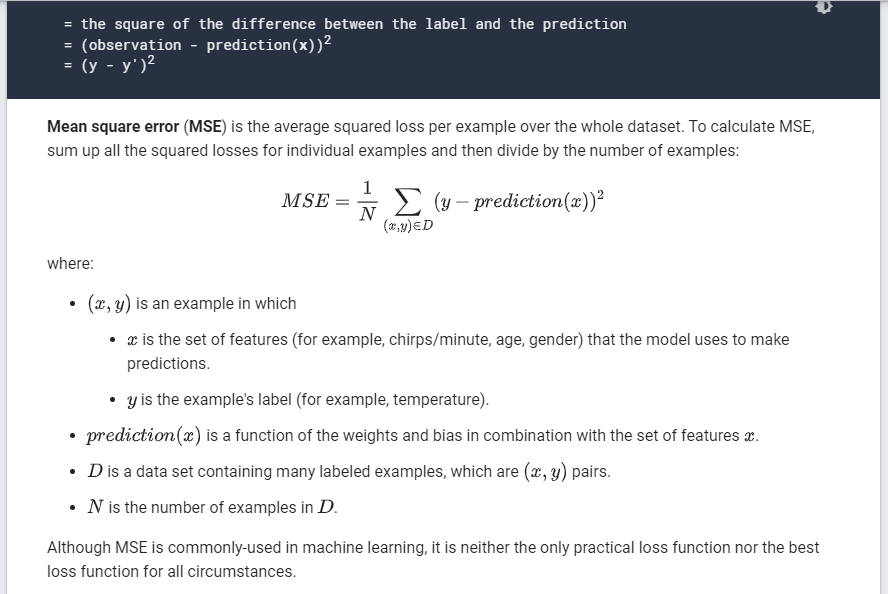
A convenient loss function (among many other functions) for depicting the loss in a model is the L2 loss function.



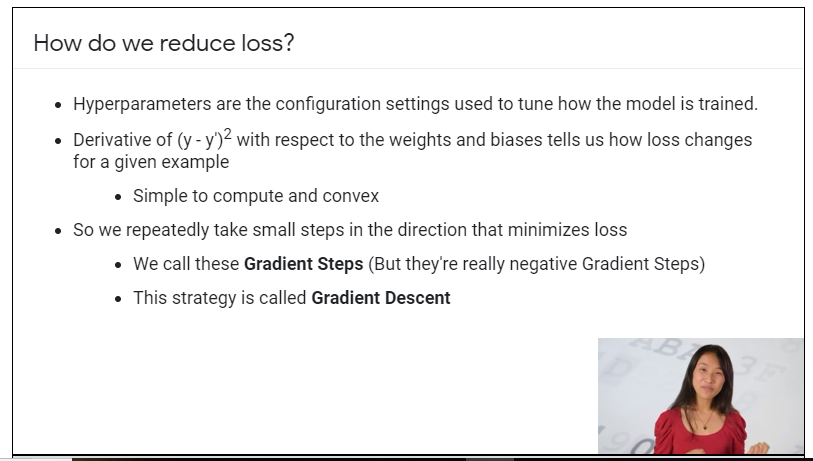


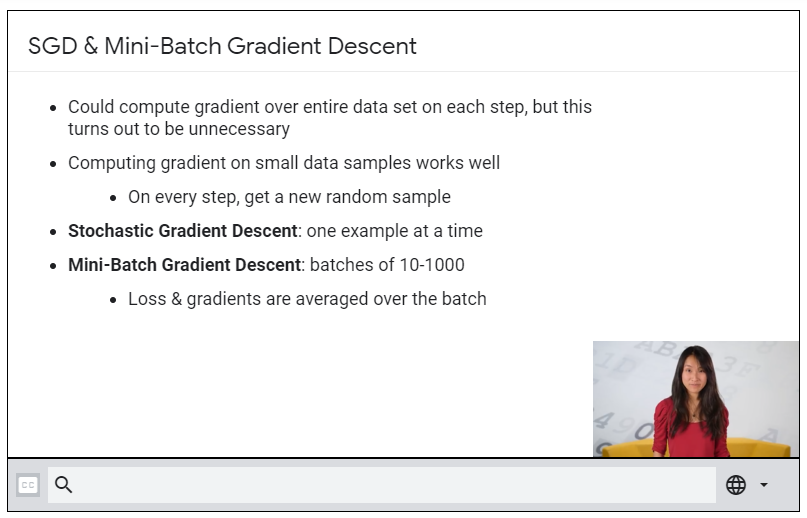


**Training** a model simply means learning (determining) good values for all the weights and the bias from labeled examples. In supervised learning, a machine learning algorithm builds a model by examining many examples and attempting to find a model that minimizes loss; this process is called **empirical risk minimization**.

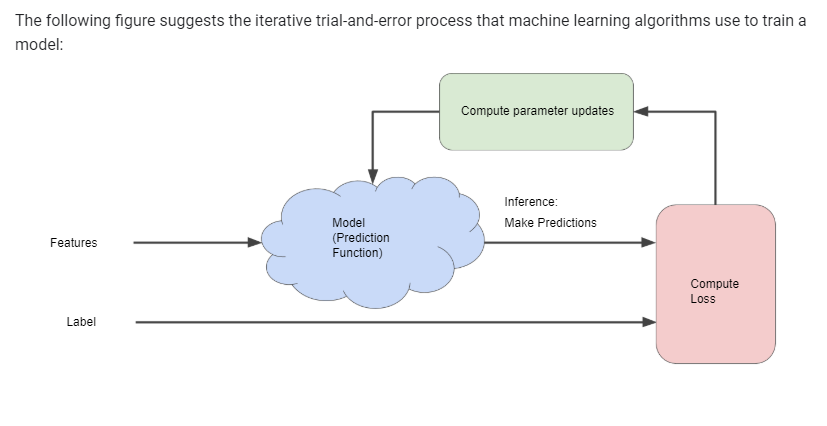


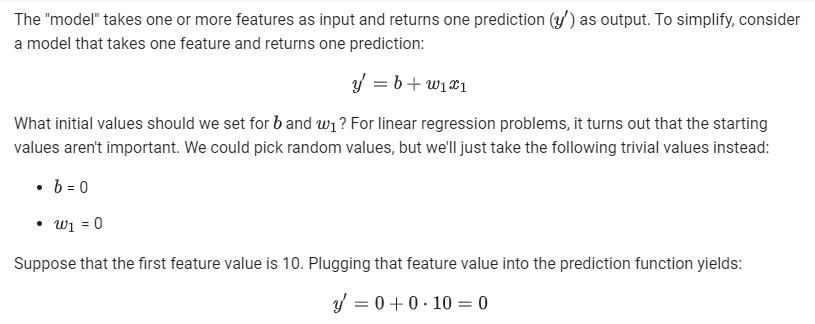
## **Reducing Loss**

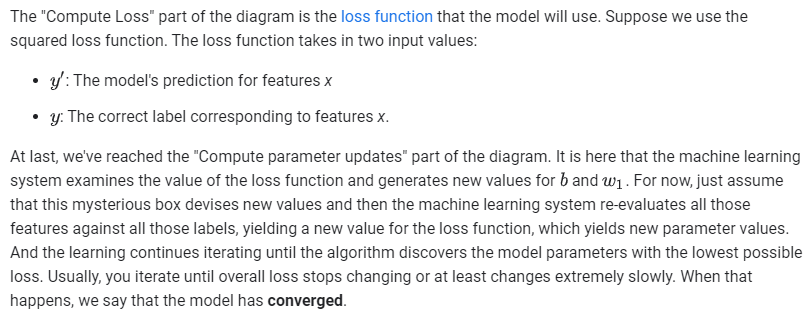




## **Reducing Loss: An Iterative Approach**



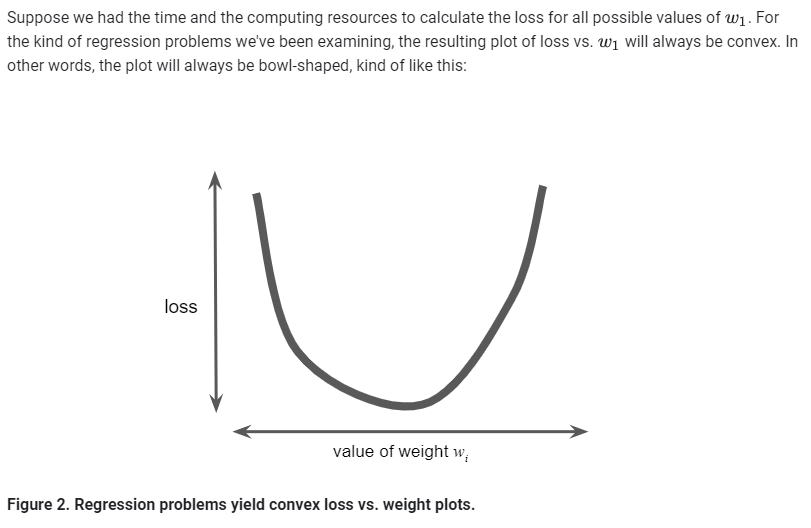




**Key Point:**

A Machine Learning model is trained by starting with an initial guess for the weights and bias and iteratively adjusting those guesses until learning the weights and bias with the lowest possible loss.

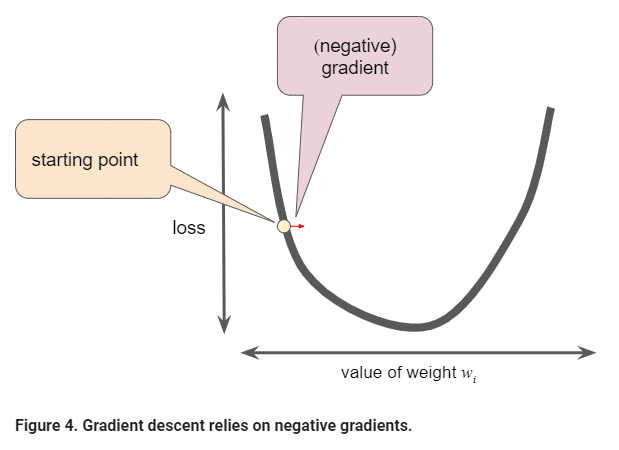
## **Reducing Loss: Gradient Descent**



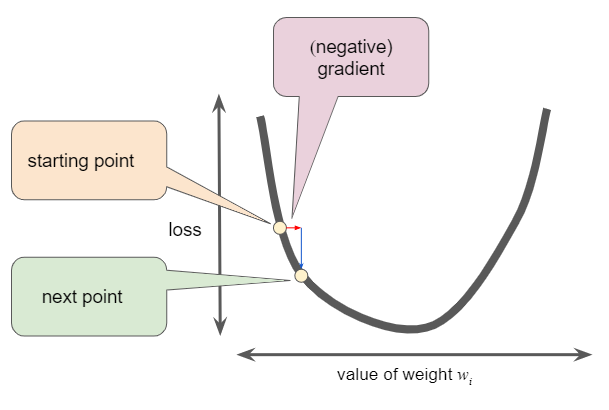
What we do is, we take a random starting point for W1, say 0 & calculate the gradient for the loss function at this point. The gradient will be a vector, i.e it will have a magnitude and direction. Since the gradient will point in the direction of steepest increase in loss function.

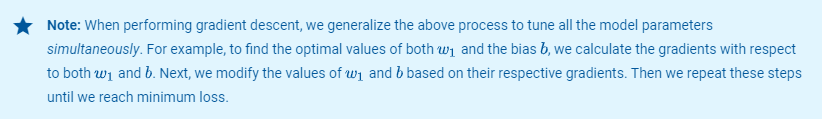
So, we need to go the opposite way in order to reduce loss.

When there are multiple weights, the gradient is a vector of partial derivates with respect to the weights.



We calculate the next value for Wi  by adding the *some fraction* of the gradient’s magnitude to the starting point/current point as shown in the figure:

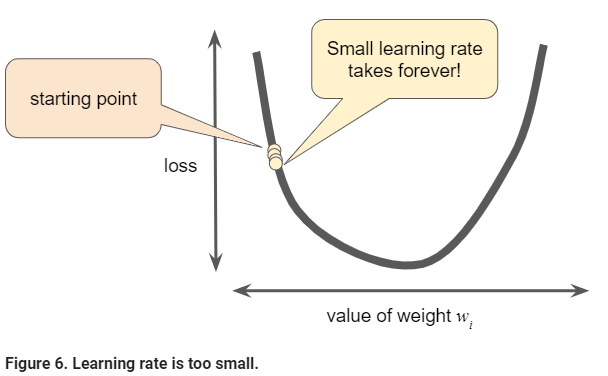


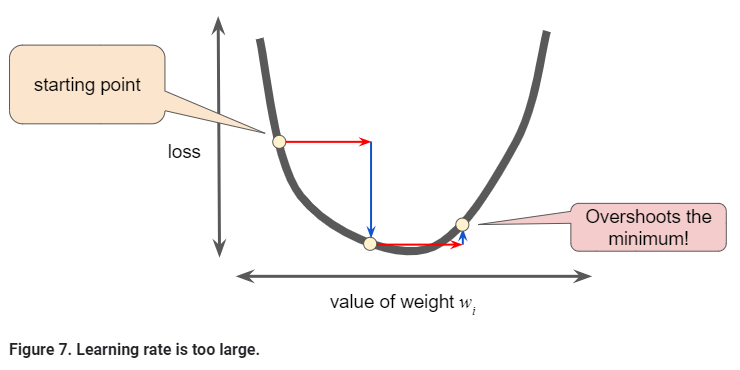


## **Reducing Loss: Learning Rate**

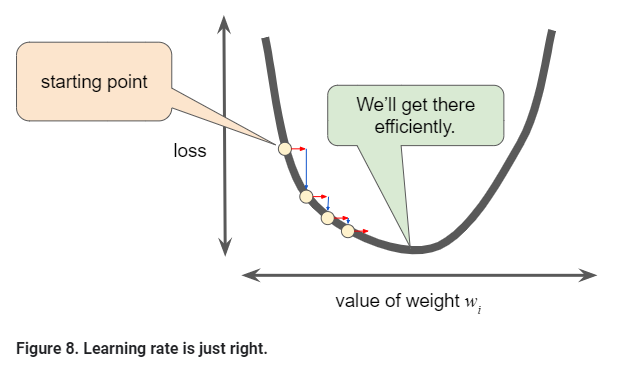
As noted, the gradient vector has both a direction and a magnitude. Gradient descent algorithms multiply the gradient by a scalar known as the **learning rate** (also sometimes called **step size**) to determine the next point. For example, if the gradient magnitude is 2.5 and the learning rate is 0.01, then the gradient descent algorithm will pick the next point 0.025 away from the previous point.

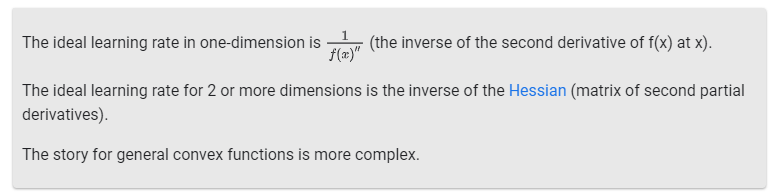
**Hyperparameters** are the knobs that programmers tweak in machine learning algorithms. Most machine learning programmers spend a fair amount of time tuning the learning rate. If you pick a learning rate that is too small, learning will take too long:





There's a [Goldilocks](https://wikipedia.org/wiki/Goldilocks_principle) learning rate for every regression problem. The Goldilocks value is related to how flat the loss function is. If you know the gradient of the loss function is small then you can safely try a larger learning rate, which compensates for the small gradient and results in a larger step size.





## **Reducing Loss: Stochastic Gradient Descent**

In gradient descent, a **batch** is the total number of examples you use to calculate the gradient in a single iteration. So far, we've assumed that the batch has been the entire data set. When working at Google scale, data sets often contain billions or even hundreds of billions of examples. Furthermore, Google data sets often contain huge numbers of features. Consequently, a batch can be enormous. A very large batch may cause even a single iteration to take a very long time to compute.

A large data set with randomly sampled examples probably contains redundant data. In fact, redundancy becomes more likely as the batch size grows. Some redundancy can be useful to smooth out noisy gradients, but enormous batches tend not to carry much more predictive value than large batches.

What if we could get the right gradient on average for much less computation? By choosing examples at random from our data set, we could estimate (albeit, noisily) a big average from a much smaller one. **Stochastic gradient descent** (**SGD**) takes this idea to the extreme--it uses only a single example (a batch size of 1) per iteration. Given enough iterations, SGD works but is very noisy. The term "stochastic" indicates that the one example comprising each batch is chosen at random.

**Mini-batch stochastic gradient descent** (**mini-batch SGD**) is a compromise between full-batch iteration and SGD. A mini-batch is typically between 10 and 1,000 examples, chosen at random. Mini-batch SGD reduces the amount of noise in SGD but is still more efficient than full-batch.

To simplify the explanation, we focused on gradient descent for a single feature. Rest assured that gradient descent also works on feature sets that contain multiple features.