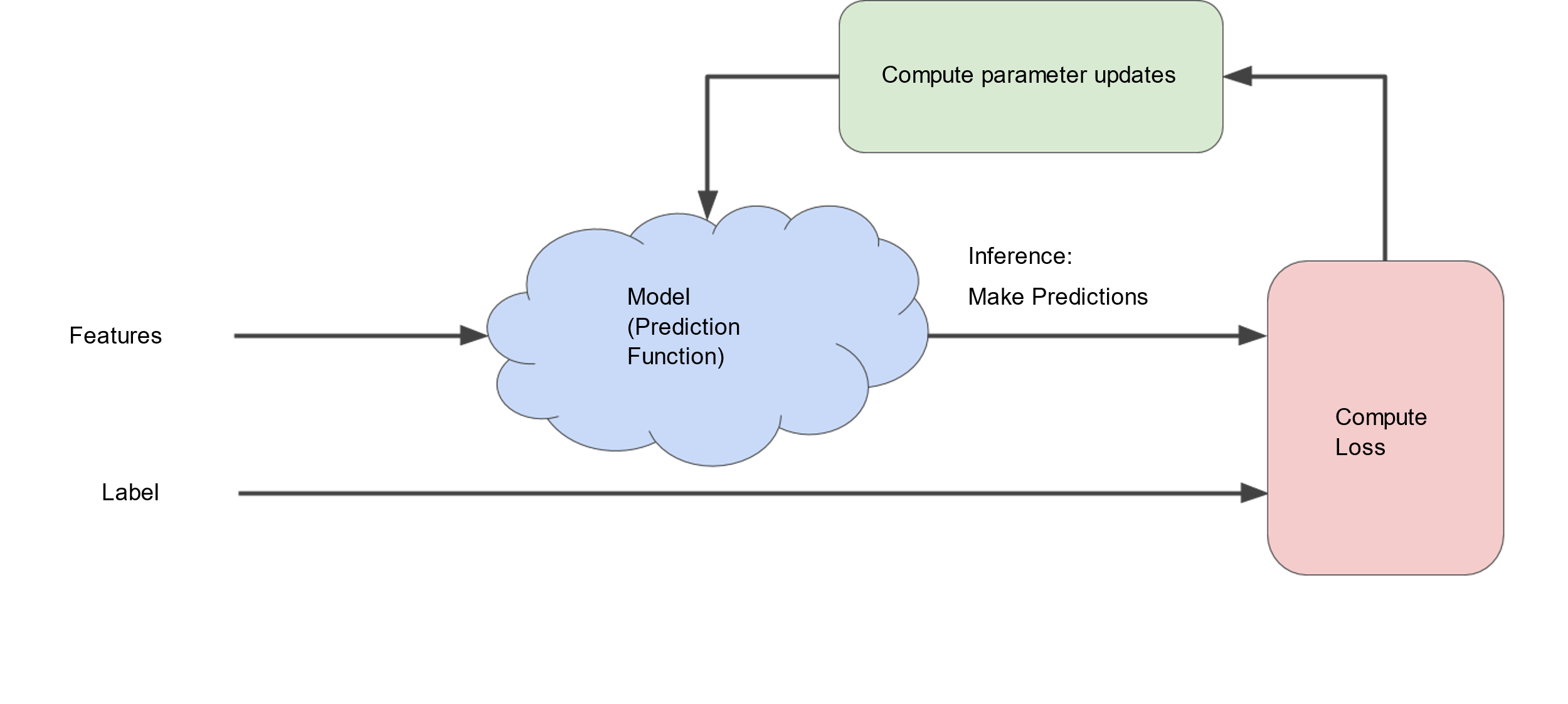
Reducing Loss: An Iterative Approach

**Estimated Time:** 10 minutes

The [previous module](https://developers.google.com/machine-learning/crash-course/descending-into-ml) introduced the concept of loss. Here, in this module, you'll learn how a machine learning model iteratively reduces loss.

Iterative learning might remind you of the ["Hot and Cold"](http://www.howcast.com/videos/258352-how-to-play-hot-and-cold/) kid's game for finding a hidden object like a thimble. In this game, the "hidden object" is the best possible model. You'll start with a wild guess ("The value of w1 is 0.") and wait for the system to tell you what the loss is. Then, you'll try another guess ("The value of w1 is 0.5.") and see what the loss is. Aah, you're getting warmer. Actually, if you play this game right, you'll usually be getting warmer. The real trick to the game is trying to find the best possible model as efficiently as possible.

The following figure suggests the iterative trial-and-error process that machine learning algorithms use to train a model:



**Figure 1. An iterative approach to training a model.**

We'll use this same iterative approach throughout Machine Learning Crash Course, detailing various complications, particularly within that stormy blue cloud. Iterative strategies are prevalent in machine learning, primarily because they scale so well to large data sets.

The "model" takes one or more features as input and returns one prediction (y') as output. To simplify, consider a model that takes one feature and returns one prediction:

y′=b+w1x1

What initial values should we set for b and w1? For linear regression problems, it turns out that the starting values aren't important. We could pick random values, but we'll just take the following trivial values instead:

* b = 0
* w1 = 0

Suppose that the first feature value is 10. Plugging that feature value into the prediction function yields:

  y' = 0 + 0(10)  
  y' = 0

The "Compute Loss" part of the diagram is the [loss function](https://developers.google.com/machine-learning/crash-course/descending-into-ml/training-and-loss) that the model will use. Suppose we use the squared loss function. The loss function takes in two input values:

* *y'*: The model's prediction for features *x*
* *y*: The correct label corresponding to features *x*.

At last, we've reached the "Compute parameter updates" part of the diagram. It is here that the machine learning system examines the value of the loss function and generates new values for b and w1. For now, just assume that the mysterious green box devises new values and then the machine learning system re-evaluates all those features against all those labels, yielding a new value for the loss function, which yields new parameter values. And the learning continues iterating until the algorithm discovers the model parameters with the lowest possible loss. Usually, you iterate until overall loss stops changing or at least changes extremely slowly. When that happens, we say that the model has **converged**.

**Key Point:**

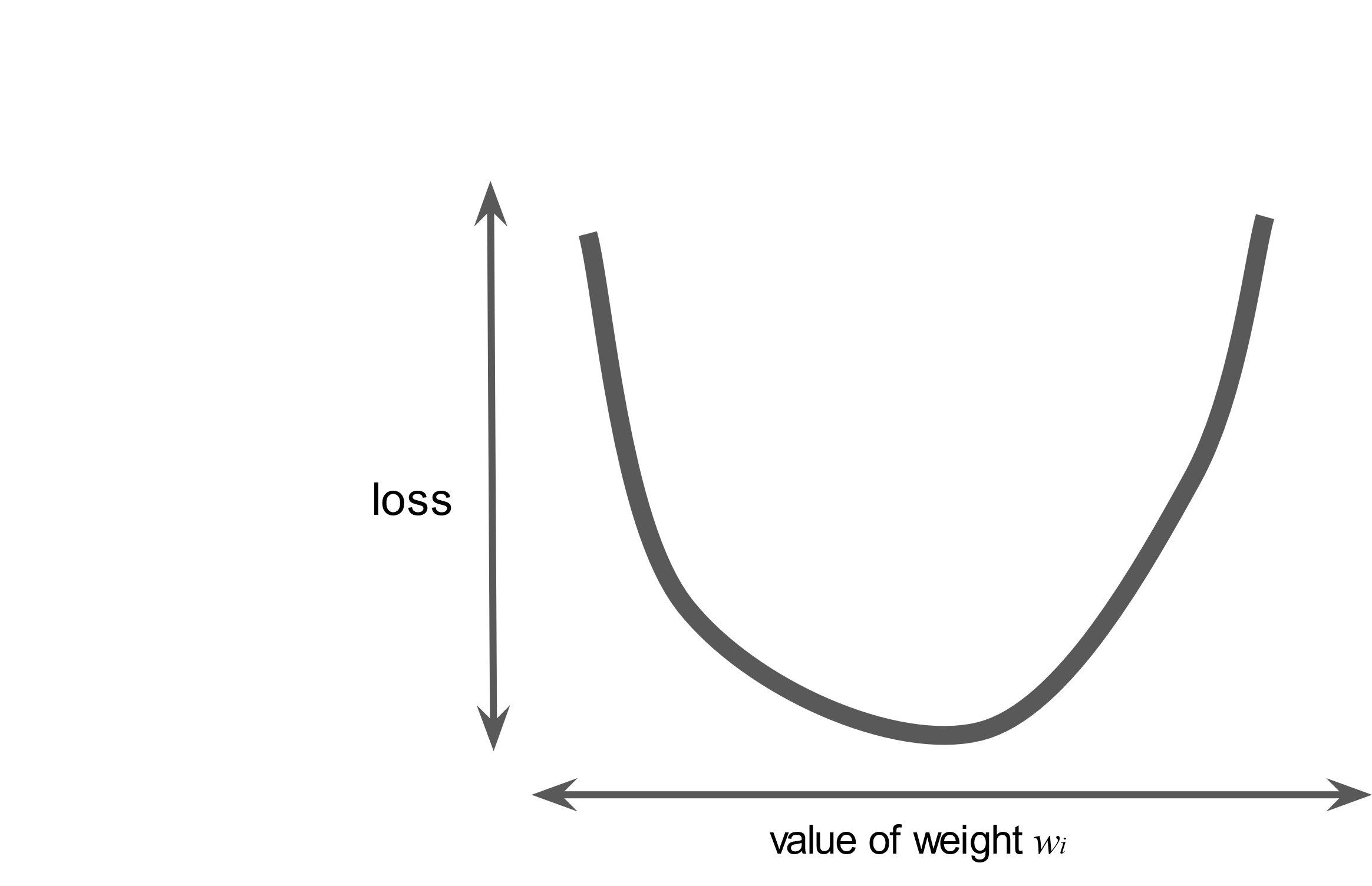
A Machine Learning model is trained by starting with an initial guess for the weights and bias and iteratively adjusting those guesses until learning the weights and bias with the lowest possible loss.

# Reducing Loss: Gradient Descent

**Estimated Time:** 10 minutes

The iterative approach diagram ([Figure 1](https://developers.google.com/machine-learning/crash-course/reducing-loss/an-iterative-approach#ml-block-diagram)) contained a green hand-wavy box entitled "Compute parameter updates." We'll now replace that algorithmic fairy dust with something more substantial.

Suppose we had the time and the computing resources to calculate the loss for all possible values of w1 . For the kind of regression problems we've been examining, the resulting plot of loss vs. w1 will always be convex. In other words, the plot will always be bowl-shaped, kind of like this:

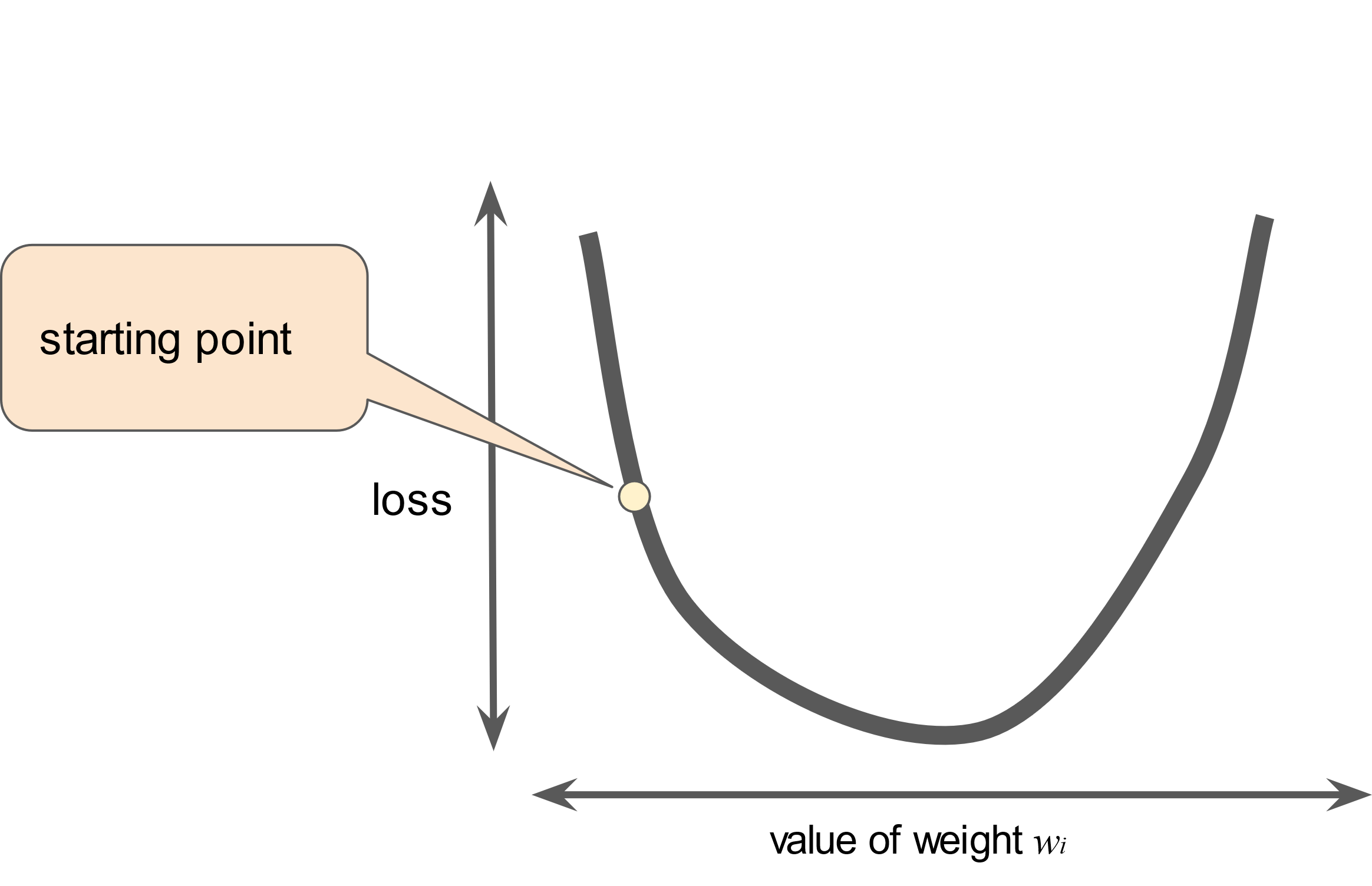


**Figure 2. Regression problems yield convex loss vs weight plots.**

Convex problems have only one minimum; that is, only one place where the slope is exactly 0. That minimum is where the loss function converges.

Calculating the loss function for every conceivable value of w1 over the entire data set would be an inefficient way of finding the convergence point. Let's examine a better mechanism—very popular in machine learning—called **gradient descent**.

The first stage in gradient descent is to pick a starting value (a starting point) for w1 . The starting point doesn't matter much; therefore, many algorithms simply set w1 to 0 or pick a random value. The following figure shows that we've picked a starting point slightly greater than 0:



**Figure 3. A starting point for gradient descent.**

The gradient descent algorithm then calculates the gradient of the loss curve at the starting point. In brief, a **gradient** is a vector of partial derivatives; it tells you which way is "warmer" or "colder." Note that the gradient of loss with respect to a single weight (as in Figure 3) is equivalent to the derivative.

▸

#### Learn more about partial derivatives and gradients.

The math around machine learning is fascinating and we're delighted that you clicked the link to learn more. Please note, however, that TensorFlow handles all the gradient computations for you, so you don't actually have to understand the calculus provided here.

### Partial derivatives

A **multivariable function** is a function with more than one number argument, such as:

f(x,y)=e2ysin(x)

The **partial derivative of f with respect to x**, denoted as follows:

∂f∂x

is the derivative of *f* considered as a function of **x** alone. To find the following:

∂f∂x

you must hold *y* constant (so *f* is now a function of one variable *x*), and take the regular derivative of *f* with respect to *x*. For example, when *y* is fixed at 1, the preceding function becomes:

f(x)=e2sin(x)

This is just a function of one variable *x*, whose derivative is:

e2cos(x)

In general, thinking of *y* as fixed, the partial derivative of f with respect to x is calculated as follows:

∂f∂x(x,y)=e2ycos(x)

Similarly, if we hold *x* fixed instead, the partial derivative of f with respect to y is:

∂f∂y(x,y)=2e2ysin(x)

Intuitively, a partial derivative tells you how much the function changes when you perturb one variable a bit. In the preceding example:

∂f∂x(0,1)=e2≈7.4

So when you start at (0,1), hold *y* constant, and move *x* a little, *f* changes by about 7.4 times the amount that you changed *x*.

In machine learning, partial derivatives are mostly used in conjunction with the gradient of a function.

### Gradients

The **gradient** of a function, denoted as follows, is the vector of partial derivatives with respect to all of the independent variables:

∇f

For instance, if:

f(x,y)=e2ysin(x)

then:

∇f(x,y)=(∂f∂x(x,y),∂f∂y(x,y))=(e2ycos(x),2e2ysin(x))

Note the following:

|  |  |
| --- | --- |
| ∇f | Points in the direction of greatest increase of the function. |
| −∇f | Points in the direction of greatest decrease of the function. |

The number of dimensions in the vector is equal to the number of variables in the formula for *f*; in other words, the vector falls within the domain space of the function. For instance, the graph of the following function f(x,y):

f(x,y)=4+(x−2)2+2y2

when viewed in three dimensions with z = f(x,y) looks like a valley with a minimum at (2,0,4):

A three-dimensional plot of z = 4 + (x - 2)^2 + y^2, which produces a paraboloid with a minimum at (2,0,4)

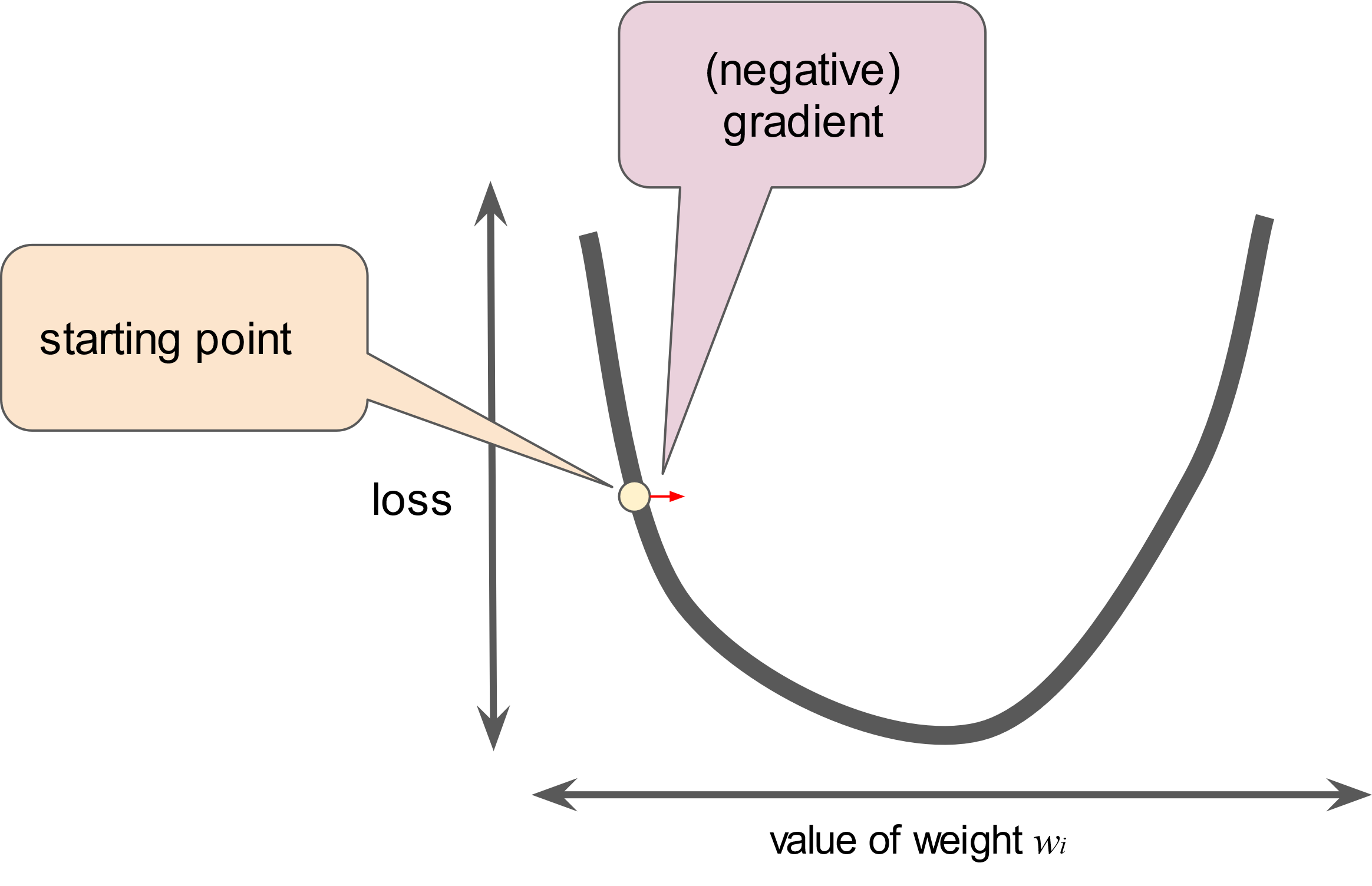
The gradient of f(x,y) is a two-dimenional vector that tells you in which (x,y) direction to move for the maximum reduction in height. In other words, the gradient vector points toward the valley.

In machine learning, gradients are used in gradient descent. We often have a loss function of many variables that we are trying to minimize, and we try to do this by following the negative of the gradient of the function.

Note that a gradient is a vector, so it has both of the following characteristics:

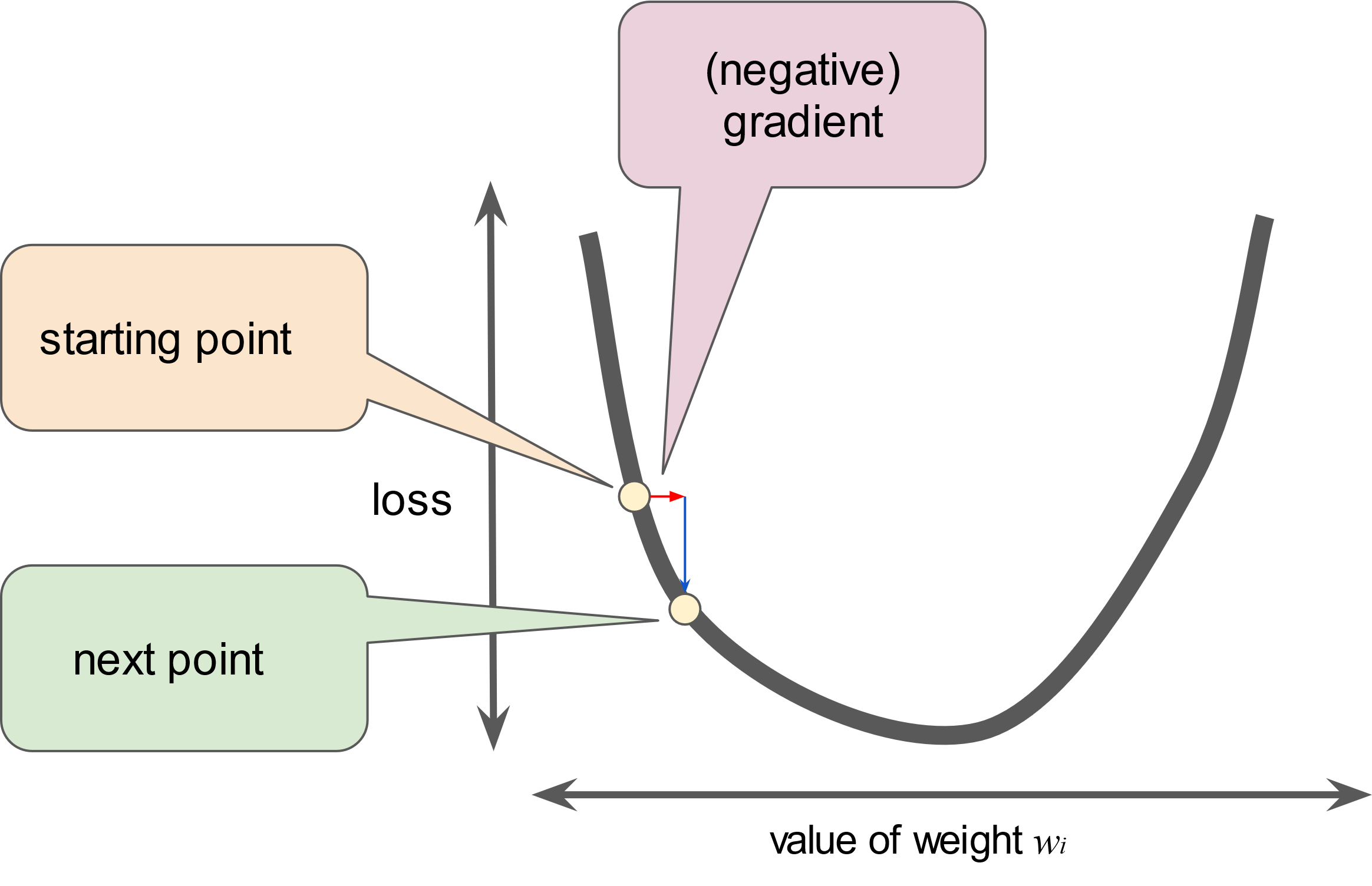
* a direction
* a magnitude

The gradient always points in the direction of steepest increase in the loss function. The gradient descent algorithm takes a step in the direction of the negative gradient in order to reduce loss as quickly as possible.



**Figure 4. Gradient descent relies on negative gradients.**

To determine the next point along the loss function curve, the gradient descent algorithm adds some fraction of the gradient's magnitude to the starting point as shown in the following figure:



**Figure 5. A gradient step moves us to the next point on the loss curve.**

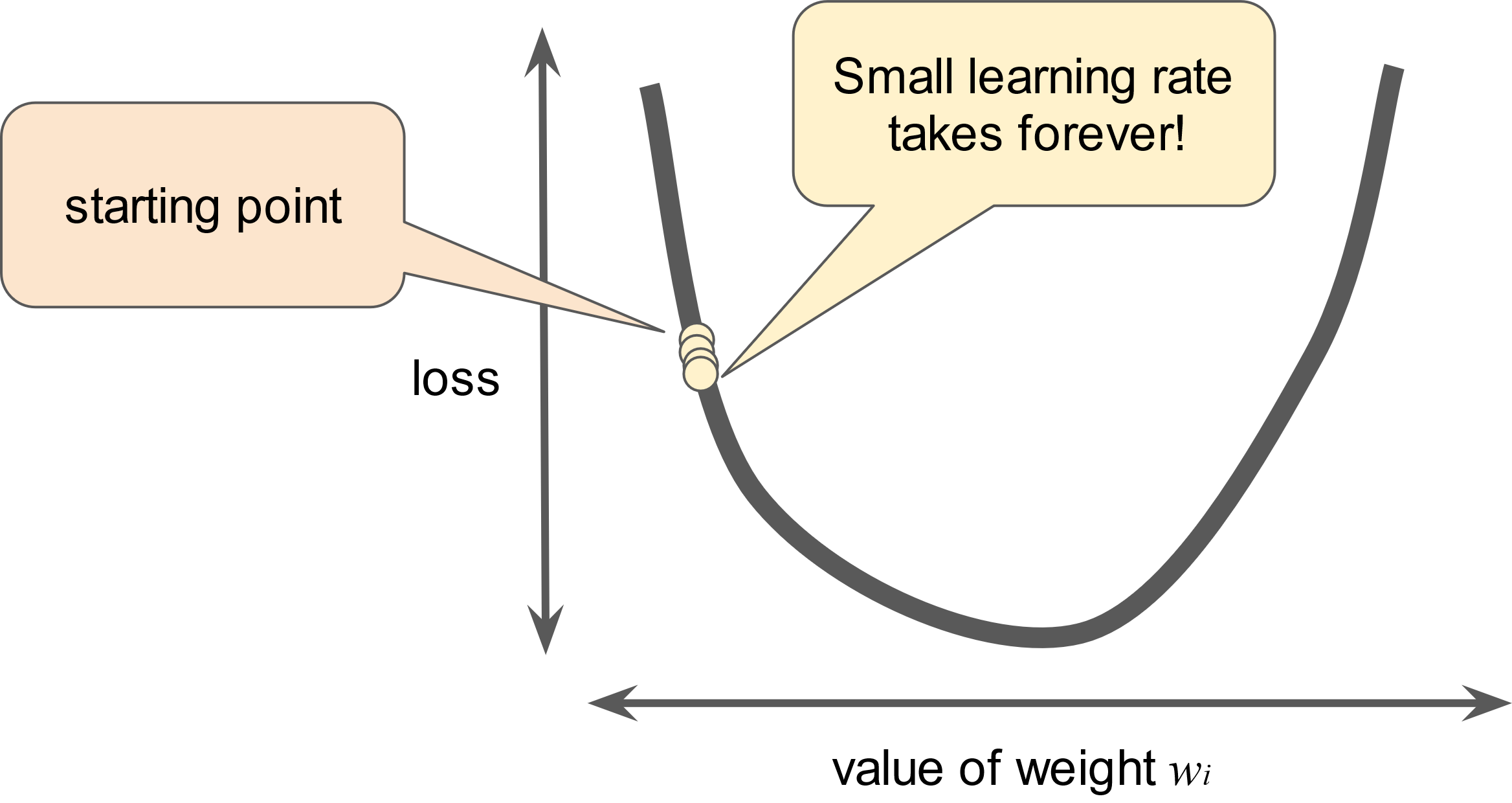
The gradient descent then repeats this process, edging ever closer to the minimum.

# Reducing Loss: Learning Rate

**Estimated Time:** 5 minutes

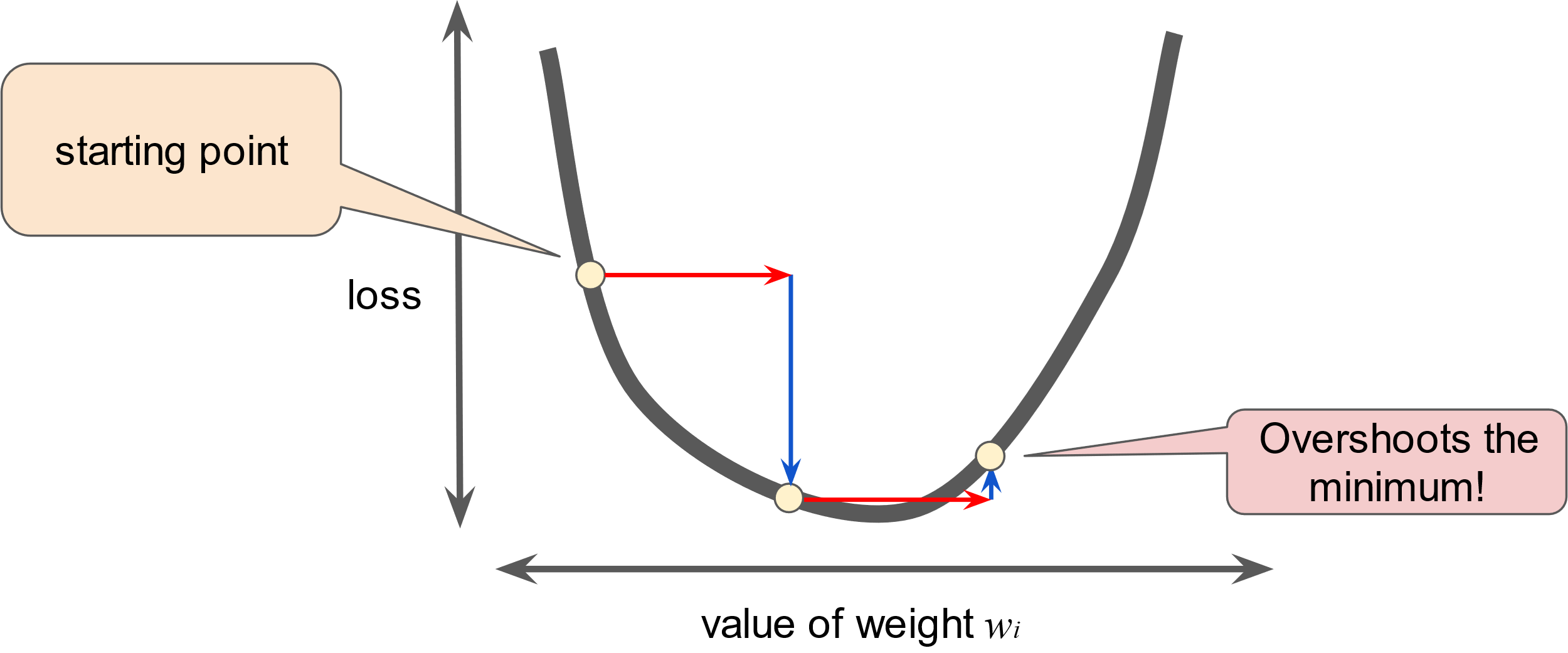
As noted, the gradient vector has both a direction and a magnitude. Gradient descent algorithms multiply the gradient by a scalar known as the **learning rate** (also sometimes called **step size**) to determine the next point. For example, if the gradient magnitude is 2.5 and the learning rate is 0.01,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

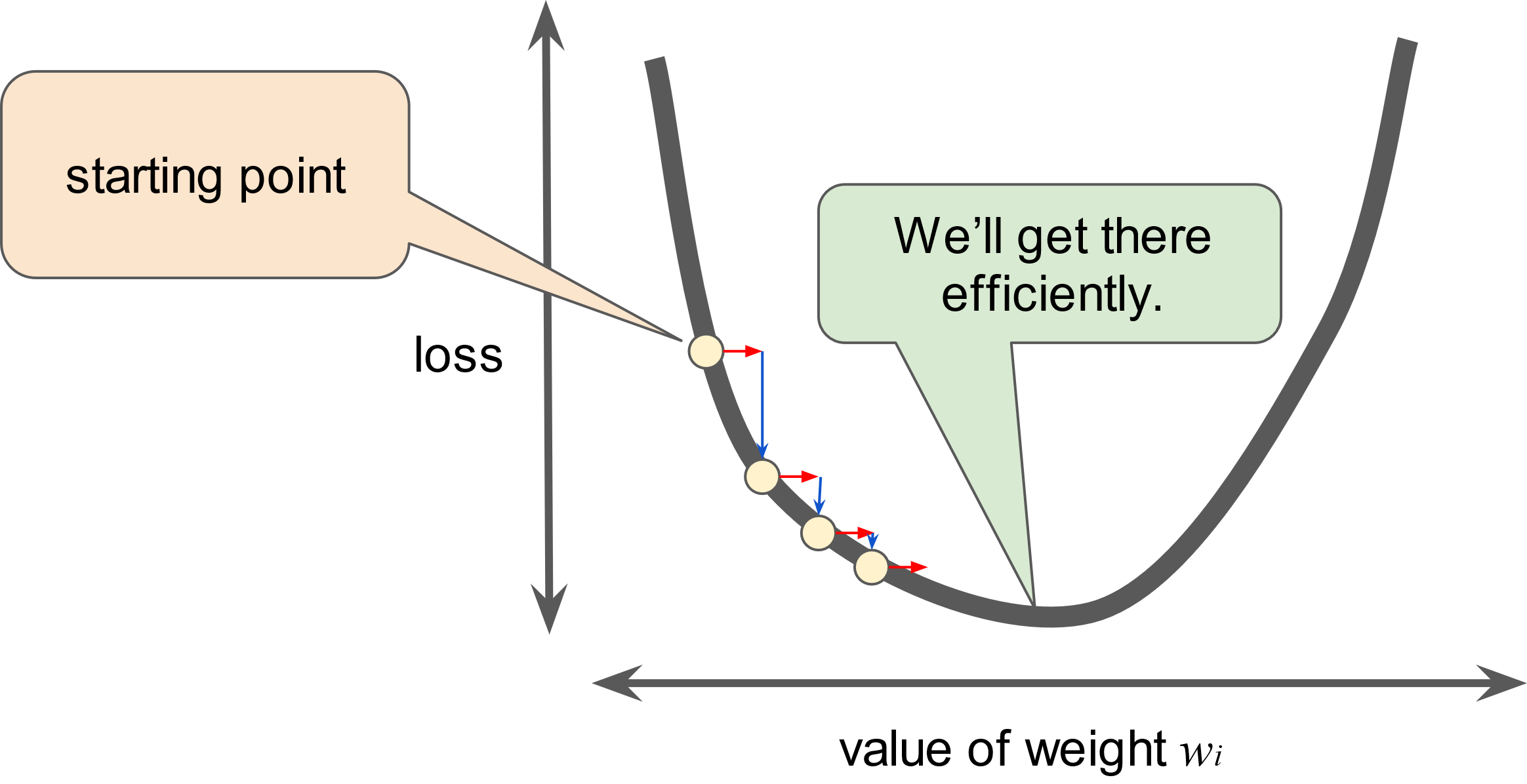


**Figure 6. Learning rate is too small.**

Conversely, if you specify a learning rate that is too large, the next point will perpetually bounce haphazardly across the bottom of the well like a quantum mechanics experiment gone horribly wrong:

**Figure 7. Learning rate is too large.**

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.

**Figure 8. Learning rate is just right.**

#### **Learn more about the ideal learning rate.**

The ideal learning rate in one-dimension is 1f(x)″ (the inverse of the second derivative of f(x) at x).

The ideal learning rate for 2 or more dimensions is the inverse of the [Hessian](https://wikipedia.org/wiki/Hessian_matrix) (matrix of second partial derivatives).

The story for general convex functions is more complex.

# Reducing Loss: Stochastic Gradient Descent

**Estimated Time:** 3 minutes

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.

Reducing Loss: Playground Exercise

<https://developers.google.com/machine-learning/crash-course/reducing-loss/playground-exercise>