Gradient descent with Python

by **Adrian Rosebrock (https://www.pyimagesearch.com/author/adrian/)** *on* **October 10, 2016**

The gradient descent algorithm comes in two flavors:

1. The standard "vanilla" implementation.
2. The optimized "stochastic" version that is more commonly used.

Today well be reviewing the basic vanilla implementation to form a baseline for our understanding. Then next week I'll be discussing the stochastic version of gradient descent.

Gradient descent is an optimization algorithm

The gradient descent method is an *iterative optimization algorithm* that operates over a *loss landscape.*

We can visualize our loss landscape as a bowl, similar to the one you may eat cereal or soup out of:

Figure 1: A plot of our loss landscape. We typically see this landscape depicted as a "bowl". Our goal is to move towards the basin of this bowl where this is minimal loss.

The surface of our bowl is called our *loss landscape*, which is essentially a *plot* of our loss function.

The difference between our loss landscape and your cereal bowl is that your cereal bowl only exists in three dimensions, while your loss landscape exists in *many dimensions*, perhaps tens, hundreds, or even thousands of dimensions.

Each position along the surface of the bowl corresponds to a particular *loss value* given our set of parameters, *W* (weight matrix) and *b* (bias vector).

Our goal is to try different values of *W* and *b*, evaluate their loss, and then take a step towards more optimal values that will (ideally) have lower loss.

Iteratively repeating this process will allow us to navigate our loss landscape, following the gradient of the loss function (the bowl), and find a set of parameters that have minimum loss and high classification accuracy.

The “gradient” in gradient descent

To make our explanation of gradient descent a little more intuitive, let’s pretend that we have a robot — let’s name him Chad:

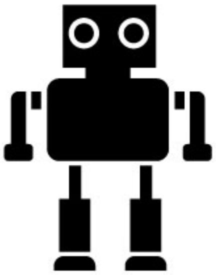


Figure 2: Introducing our robot, Chad, who will help us understand the concept of gradient descent.

We place Chad on a random position in our bowl (i.e., the loss landscape):

Figure 3: Chad is placed on a random position on the loss landscape. However, Chad has only one sensor — the loss value at the *exact position* he is standing at. Using this sensor (and this sensor alone), how is he going to get to the bottom of the basin?

It's now Chad's job to navigate to the bottom of the basin (where there is minimum loss).

Seems easy enough, right? All Chad has to do is orient himself such that he's facing "downhill" and then ride the slope until he reaches the bottom of the basin.

But we have a problem: Chad isn't a very smart robot.

Chad only has one sensor — this sensor allows him to take his weight matrix *W* and compute a loss function *L*.

Therefore, Chad is able to compute his (relative) position on the loss landscape, but he has *absolutely no idea* in which direction he should take a step to move himself closer to the bottom of the basin.

What is Chad to do?

The answer is to apply gradient descent.

All we need to do is follow the slope of the gradient *W*. We can compute the gradient of *W* across all dimensions using the following equation:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$$

In *> 1* dimensions, our gradient becomes a *vector* of *partial derivatives.*

The problem with this equation is that:

1. It's an *approximation* to the gradient.
2. It's very slow.

In practice, we use the *analytic gradient* instead. This method is exact, fast, but extremely challenging to implement due to partial derivatives and multivariable calculus. You can read more about the numeric and analytic gradients **here (https://www.youtube.com/watch?v=ruuW4-InUxM)**.

For the sake of this discussion, simply try to internalize what gradient descent is doing: attempting to optimize our parameters for low loss and high classification accuracy.

Pseudocode for gradient descent

Below I have included some Python-like pseudocode of the standard, vanilla gradient descent algorithm, inspired by the **CS231n slides (http://cs231n.stanford.edu/slides/winter1516_lecture3.pdf)**:

Gradient descent with Python

```
1. | while True:
2. |     Wgradient = evaluate_gradient(loss, data, W)
3. |     W += -alpha * Wgradient
```

This pseudocode is essentially what *all* variations of gradient descent are built off of.

We start off on **Line 1** by looping until some condition is met. Normally this condition is either:

1. A specified number of epochs has passed (meaning our learning algorithm has “seen” each of the training data points *N* times).
2. Our loss has become *sufficiently low* or training accuracy *satisfactorily high*.
3. Loss has not improved in *M* subsequent epochs.

Line 2 then calls a function named `evaluate_gradient` . This function requires three parameters:

- `loss` : A function used to compute the *loss* over our current parameters *W* and input `data` .
- `data` : Our training data where each training sample is represented by a feature vector.
- `W` : This is actually our weight matrix that we are optimizing over. Our goal is to apply gradient descent to find a *W* that yields minimal loss.

The `evaluate_gradient` function returns a vector that is *K*-dimensional, where *K* is the number of dimensions in our feature vector. The `Wgradient` variable is actually our *gradient*, where we have a gradient entry for each dimension.

We then apply the actual *gradient descent* on **Line 3**.

We multiply our `Wgradient` by `alpha` , which is our learning rate. **The learning rate controls the size of our step.**

In practice, you’ll spend a lot of time finding an optimal learning rate `alpha` — it is *by far* the most important parameter in your model.

If `alpha` is too large, we’ll end up spending all our time bouncing around our loss landscape and never actually “descending” to the bottom of our basin (unless our random bouncing takes us there by pure luck).

Conversely, if `alpha` is too small, then it will take *many* (perhaps prohibitively many) iterations to reach the bottom of the basin.

Because of this, `alpha` will cause you many headaches — and you’ll spend a considerable amount of your time trying to find an optimal value for your classifier and dataset.

Implementing gradient descent with Python

Now that we know the basics of gradient descent, let’s implement gradient descent in Python and use it to classify some data.

Open up a new file, name it `gradient_descent.py`, and insert the following code:

Gradient descent with Python

```
 1. # import the necessary packages
 2. import matplotlib.pyplot as plt
 3. from sklearn.datasets.samples_generator import make_blobs
 4. import numpy as np
 5. import argparse
 6.
 7. def sigmoid_activation(x):
 8.     # compute and return the sigmoid activation value for a
 9.     # given input value
10.     return 1.0 / (1 + np.exp(-x))
```

Lines 2-5 import our required Python packages.

We then define the `sigmoid_activation` function on **Line 7**. When plotted, this function will resemble an “S”-shaped curve:

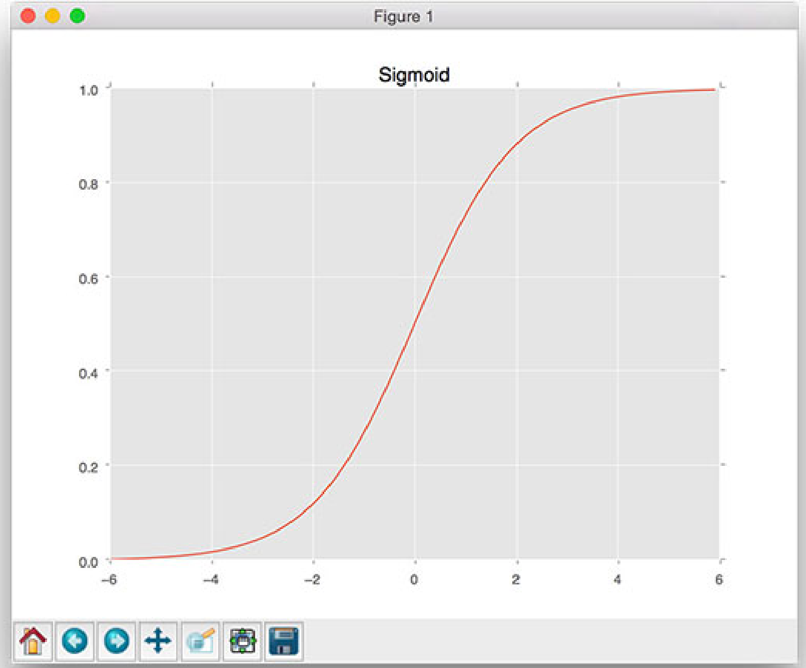


Figure 4: A plot of the sigmoid activation function. Notice how *y=0.5* when *x=0*.

We call this an **activation function (https://en.wikipedia.org/wiki/Activation_function)** because the function will "activate" and fire "ON" (*output value >= 0.5)* or "OFF" (*output vale < 0.5)* based on the inputs `x` .

While there are other (better) alternatives to the sigmoid activation function, it makes for an excellent "starting point" in our discussion of machine learning, neural networks, and deep learning.

I'll also be discussing activation functions in more detail in a future blog post, so for the time being, simply keep in mind that this is a non-linear activation function that we can use to "threshold" our predictions.

Next, let's parse our command line arguments:

Gradient descent with Python

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-e", "--epochs", type=float, default=100,
    help="# of epochs")
ap.add_argument("-a", "--alpha", type=float, default=0.01,
    help="learning rate")
args = vars(ap.parse_args())
```

We can provide two (optional) command line arguments to our script:

We can provide two (optional) command line arguments to our script:

- `--epochs` : The number of epochs that we'll use when training our classifier using gradient descent.
- `--alpha` : The *learning rate* for gradient descent. We typically see *0.1*, *0.01*, and *0.001* as initial learning rate values, but again, you'll want to **tune this hyperparameter (https://pyimagesearch.com/2016/08/15/how-to-tune-hyperparameters-with-python-and-scikit-learn/)** for your own classification problems.

Now that our command line arguments are parsed, let's generate some data to classify:

Gradient descent with Python

```
20. # generate a 2-class classification problem with 250 data points,
21. # where each data point is a 2D feature vector
22. (X, y) = make_blobs(n_samples=250, n_features=2, centers=2,
23.    cluster_std=1.05, random_state=20)
24.
25. # insert a column of 1's as the first entry in the feature
26. # vector -- this is a little trick that allows us to treat
27. # the bias as a trainable parameter *within* the weight matrix
28. # rather than an entirely separate variable
29. X = np.c_[np.ones((X.shape[0])), X]
30.
31. # initialize our weight matrix such it has the same number of
32. # columns as our input features
33. print("[INFO] starting training...")
34. W = np.random.uniform(size=(X.shape[1],))
35.
36. # initialize a list to store the loss value for each epoch
37. lossHistory = []
```

On **Line 22** we make a call to `make_blobs` which generates 250 data points. These data points are 2D, implying that the "feature vectors" are of length *2*.

Furthermore, 125 of these data points belong to *class 0* and the other 125 to *class 1*. Our goal is to train a classifier that correctly predicts each data point as being *class 0* or *class 1*.

Line 29 applies a neat little trick that allows us to skip *explicitly* keeping track of our bias vector *b*. To accomplish this, we insert a brand new column of *1's* as the first entry in our feature vector. This addition of a column containing a constant value across *all* feature vectors allows us to treat our bias as a *trainable parameter* that is ***within*** the weight matrix *W* rather than an entirely separate variable. You can learn more about this trick **here (http://book.caltech.edu/bookforum/showthread.php?t=845)** and **here (http://stackoverflow.com/questions/3775032/how-to-update-the-bias-in-neural-network-backpropagation)**.

Line 34 (randomly) initializes our weight matrix such that it has the same number of dimensions as our input features.

It's also common to see both *zero* and *one* weight initialization, but I tend to prefer random initialization better. Weight initialization methods will be discussed in further detail inside future neural network and deep learning blog posts.

Finally, **Line 37** initializes a list to keep track of our loss after each epoch. At the end of our Python script, we'll plot the loss which should ideally decrease over time.

All of our variables are now initialized, so we can move on to the actual training and gradient descent procedure:

Gradient descent with Python

```
39.     # loop over the desired number of epochs
40.     for epoch in np.arange(0, args["epochs"]):
41.         # take the dot product between our features `X` and the
42.         # weight matrix `W`, then pass this value through the
43.         # sigmoid activation function, thereby giving us our
44.         # predictions on the dataset
45.         preds = sigmoid_activation(X.dot(W))
46. 
47.         # now that we have our predictions, we need to determine
48.         # our `error`, which is the difference between our predictions
49.         # and the true values
50.         error = preds - y
51. 
52.         # given our `error`, we can compute the total loss value as
53.         # the sum of squared loss -- ideally, our loss should
54.         # decrease as we continue training
55.         loss = np.sum(error ** 2)
56.         lossHistory.append(loss)
57.         print("[INFO] epoch #{}, loss={:.7f}".format(epoch + 1, loss))
```

On **Line 40** we start looping over the supplied number of `--epochs` . By default, we'll allow our training procedure to "see" each of the training points a total of 100 times (thus, 100 epochs).

Line 45 takes the dot product between our *entire* training data `X` and our weight matrix `W` . We take the output of this dot product and feed the values through the sigmoid activation function, giving us our predictions.

Given our predictions, the next step is to determine the "error" of the predictions, or more simply, the difference between our *predictions* and the *true values* (**Line 50**).

Line 55 computes the **least squares error (http://www.anc.ed.ac.uk/rbf/intro/node9.html)** over our predictions (our loss value). The goal of this training procedure is thus to minimize the least squares error.

Now that we have our `error` , we can compute the `gradient` and then use it to update our weight matrix `W` :

Gradient descent with Python

```
59.         # the gradient update is therefore the dot product between
60.         # the transpose of `X` and our error, scaled by the total
61.         # number of data points in `X`
62.         gradient = X.T.dot(error) / X.shape[0]
63. 
64.         # in the update stage, all we need to do is nudge our weight
65.         # matrix in the negative direction of the gradient (hence the
66.         # term "gradient descent" by taking a small step towards a
67.         # set of "more optimal" parameters
68.         W += -args["alpha"] * gradient
```

Line 62 handles computing the actual gradient, which is the dot product between our data points `X` and the `error` .

Line 68 is the most critical step in our algorithm and where the actual gradient descent takes place. Here we update our weight matrix `W` by taking a `--step` in the negative direction of the gradient, thereby allowing us to move towards the *bottom* of the basin of the loss landscape (hence the term, *gradient descent*).

After updating our weight matrix, we keep looping until the desired number of epochs has been met — gradient descent is thus an *iterative algorithm.*

To actually demonstrate how we can use our weight matrix *W* as a classifier, take a look at the following code block:

Gradient descent with Python

```
70. # to demonstrate how to use our weight matrix as a classifier,
71. # let's look over our a sample of training examples
72. for i in np.random.choice(250, 10):
73.     # compute the prediction by taking the dot product of the
74.     # current feature vector with the weight matrix W, then
75.     # passing it through the sigmoid activation function
76.     activation = sigmoid_activation(X[i].dot(W))
77.
78.     # the sigmoid function is defined over the range y=[0, 1],
79.     # so we can use 0.5 as our threshold -- if `activation` is
80.     # below 0.5, it's class `0`; otherwise it's class `1`
81.     label = 0 if activation < 0.5 else 1
82.
83.     # show our output classification
84.     print("activation={:.4f}; predicted_label={}, true_label={}".format(
85.         activation, label, y[i]))
```

We start on on **Line 72** by looping over a sample of our training data.

For each training point `X[i]` we compute the dot product between `X[i]` and the weight matrix `W`, then feed the value through our activation function.

On **Line 81**, we compute the actual output class label. If the `activation` is *< 0.5*, then the output is *class 0*; otherwise, the output is *class 1*.

Our last code block is used to plot our training data along with the *decision boundary* that is used to determine if a given data point is *class 0* or *class 1*:

Gradient descent with Python

```
# compute the line of best fit by setting the sigmoid function
# to 0 and solving for X2 in terms of X1
Y = (-W[0] - (W[1] * X)) / W[2]

# plot the original data along with our line of best fit
plt.figure()
plt.scatter(X[:, 1], X[:, 2], marker="o", c=y)
plt.plot(X, Y, "r-")

# construct a figure that plots the loss over time
fig = plt.figure()
plt.plot(np.arange(0, args["epochs"]), lossHistory)
fig.suptitle("Training Loss")
plt.xlabel("Epoch #")
plt.ylabel("Loss")
plt.show()
```

Visualizing gradient descent

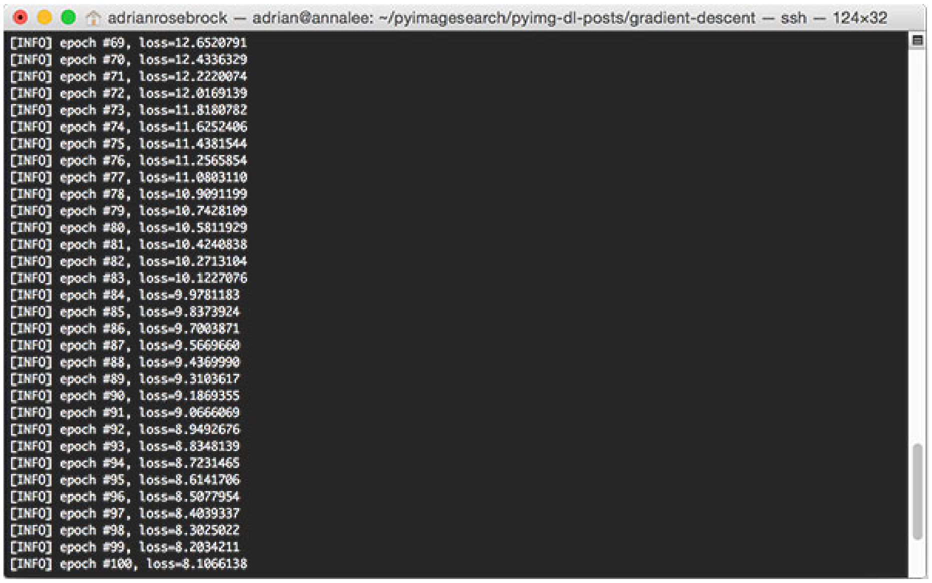
To test our gradient descent classifier, be sure to download the source code using the ***"Downloads"*** section at the bottom of this tutorial.

From there, execute the following command:

Gradient descent with Python

```
$ python gradient_descent.py
```

Examining the output, you'll notice that our classifier runs for a total of 100 epochs with the loss *decreasing* and classification accuracy *increasing* after each epoch:


adrianrosebrock — adrian@annalee: ~/pyimagesearch/pyimg-dl-posts/gradient-descent — ssh — 124×32

```
[INFO] epoch #69, loss=12.6520791
[INFO] epoch #70, loss=12.4336329
[INFO] epoch #71, loss=12.2220074
[INFO] epoch #72, loss=12.0169139
[INFO] epoch #73, loss=11.8180782
[INFO] epoch #74, loss=11.6252406
[INFO] epoch #75, loss=11.4381544
[INFO] epoch #76, loss=11.2565854
[INFO] epoch #77, loss=11.0803110
[INFO] epoch #78, loss=10.9091199
[INFO] epoch #79, loss=10.7428109
[INFO] epoch #80, loss=10.5811929
[INFO] epoch #81, loss=10.4240838
[INFO] epoch #82, loss=10.2713104
[INFO] epoch #83, loss=10.1227076
[INFO] epoch #84, loss=9.9781183
[INFO] epoch #85, loss=9.8373924
[INFO] epoch #86, loss=9.7003871
[INFO] epoch #87, loss=9.5669660
[INFO] epoch #88, loss=9.4369990
[INFO] epoch #89, loss=9.3103617
[INFO] epoch #90, loss=9.1869355
[INFO] epoch #91, loss=9.0666069
[INFO] epoch #92, loss=8.9492676
[INFO] epoch #93, loss=8.8348139
[INFO] epoch #94, loss=8.7231465
[INFO] epoch #95, loss=8.6141706
[INFO] epoch #96, loss=8.5077954
[INFO] epoch #97, loss=8.4039337
[INFO] epoch #98, loss=8.3025022
[INFO] epoch #99, loss=8.2034211
[INFO] epoch #100, loss=8.1066138
```

Figure 5: When applying gradient descent, our loss decreases and classification accuracy increases after each epoch.

To visualize this better, take a look at the plot below which demonstrates how our loss over time has decreased dramatically:

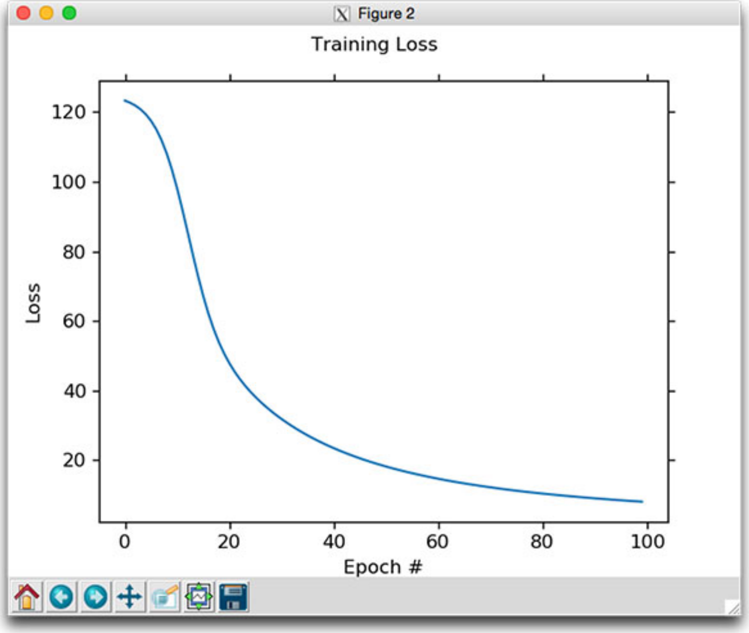


Figure 6: Plotting loss over time using gradient descent. Notice how loss sharply drops and then levels out towards later epochs.

We can then see a plot of our training data points along with the decision boundary learned by our gradient descent classifier:

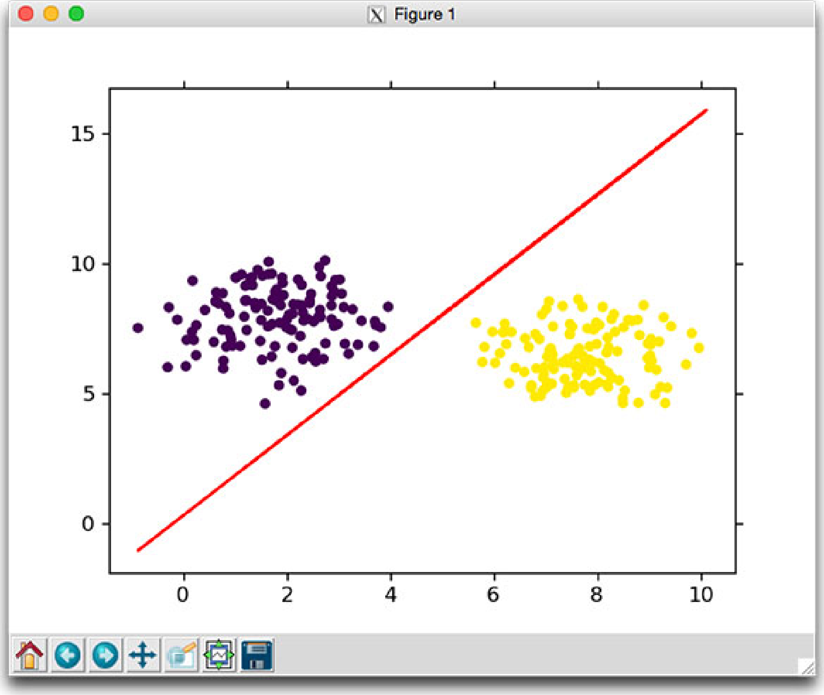
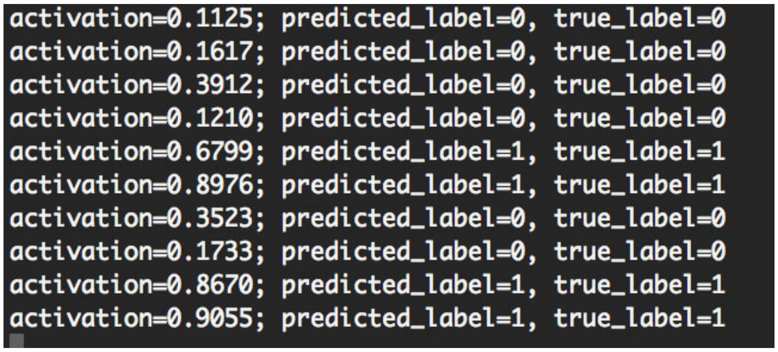


Figure 7: Plotting the decision boundary learned by our gradient descent classifier.

Notice how the decision boundary learned by our gradient descent classifier neatly divides data points of the two classes.

We then then manually investigate the classifications made by our gradient descent model. In each case, we are able to correctly predict the class:



```
activation=0.1125; predicted_label=0, true_label=0
activation=0.1617; predicted_label=0, true_label=0
activation=0.3912; predicted_label=0, true_label=0
activation=0.1210; predicted_label=0, true_label=0
activation=0.6799; predicted_label=1, true_label=1
activation=0.8976; predicted_label=1, true_label=1
activation=0.3523; predicted_label=0, true_label=0
activation=0.1733; predicted_label=0, true_label=0
activation=0.8670; predicted_label=1, true_label=1
activation=0.9055; predicted_label=1, true_label=1
```

Figure 8: Making predictions using our gradient descent classifier.

To visualize and demonstrate gradient descent in action, I have created the following animation which shows the decision boundary being “learned” after each epoch:

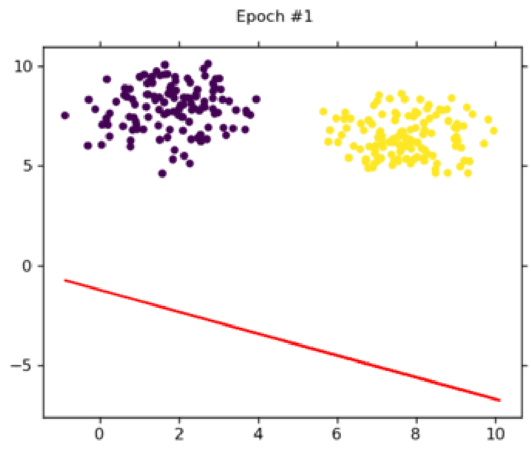


Figure 8: An animation depicting how the decision boundary is learned via gradient descent.

As you can see, our decision boundary starts off widely inaccurate due to the random initialization. But as time passes, we are able to apply gradient descent, update our weight matrix *W*, and eventually learn an accurate model.

Want to learn more about gradient descent?

In the meantime, if you want to learn more about gradient descent, you should absolutely refer to Andrew Ng's gradient descent lesson in the **Coursera Machine Learning course (https://www.youtube.com/watch?v=UfNU3Vhv5CA)**.

I would also recommend Andrej Karpathy's **excellent slides (http://cs231n.stanford.edu/slides/2016/winter1516_lecture3.pdf)** from the CS231n course.