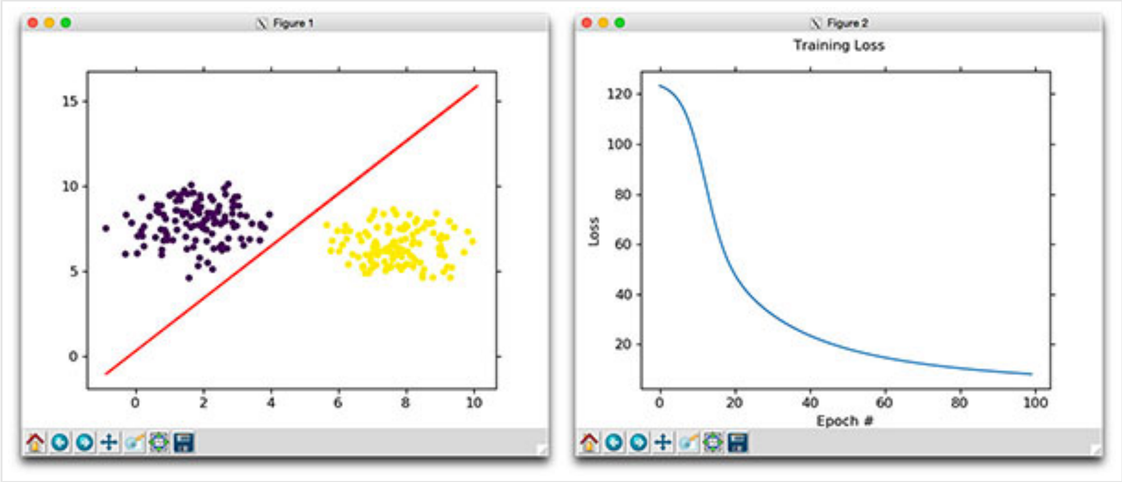
Gradient descent with Python

by **Adrian Rosebrock** on October 10, 2016 in **Deep Learning**, **Machine Learning**, **Tutorials**



Every relationship has its building blocks. Love. Trust. Mutual respect.

Yesterday, I asked my girlfriend of 7.5 years to marry me. *She said yes*.

It was quite literally the happiest day of my life. I feel like the luckiest guy in the world, not only because I have her, but also becaus PyImageSearch community has been so supportive over the past 3 years. **Thank you for being on this journey with me.**

And just like love and marriage have a set of building blocks, so do machine learning and neural network classifiers.

Over the past few weeks we opened our discussion of machine learning and neural networks with an introduction to linear classific discussed the concept of *parameterized learning*, and how this type of learning enables us to define a *scoring function* that maps output class labels.

This scoring function is defined in terms of *parameters*; specifically, our weight matrix *W* and our bias vector *b*. Our scoring functio parameters as inputs and returns a *predicted* class label for each input data point x_i.

From there, we discussed two common loss functions: Multi-class SVM loss and cross-entropy loss (commonly referred to in the s "Softmax classifiers"). Loss functions, at the most basic level, are used to quantify how "good" or "bad" a given predictor (i.e., a se are at classifying the input data points in our dataset.

Given these building blocks, we can now move on to arguably the most important aspect of machine learning, neural networks, ar ***— optimization.***

Throughout this discussion we've learned that high classification accuracy is *dependent* on finding a set of weights *W* such that ou correctly classified. Given *W*, can compute our output class labels via our *scoring function*. And finally, we can determine how goo classifications are given some *W* via our *loss function*.

But how do we go about *finding* and *obtaining* a weight matrix *W* that obtains high classification accuracy?

Do we randomly initialize *W*, evaluate, and repeat over and over again, ***hoping*** that at some point we land on a *W* that obtains rea classification accuracy?

Looking for the source code to this post?
Jump right to the downloads section.

Gradient descent with Python

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 discussin
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.

54 n.

The difference between our loss landscape and your cereal bowl is that your cereal bowl only exists in three dimensions, while you
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) a

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

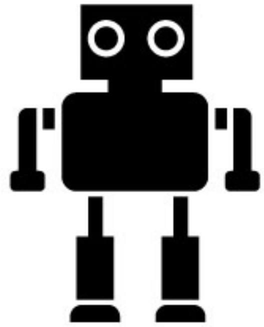


Figure 2: Introducing our robot, Chad, who will help us understand the concept of gradient descent.

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 [illegible] 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 rea
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.

All we need to do is follow the slope of the gradient *W*. We can compute the gradient of *W* acr

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

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 chal multivariable calculus. You can read more about the numeric and analytic gradients here.

For the sake of this discussion, simply try to internalize what gradient descent is doing: attemp 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 s

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

The `evaluate_gradient` function returns a vector that is *K*-dimensional, where *K* is the number of dimensions in our feature vecto `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 m

If `alpha` is too large, we'll end up spending all our time bouncing around our loss landscape and never actually "descending" to t basin (unless our random bouncing takes us there by pure luck).

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:

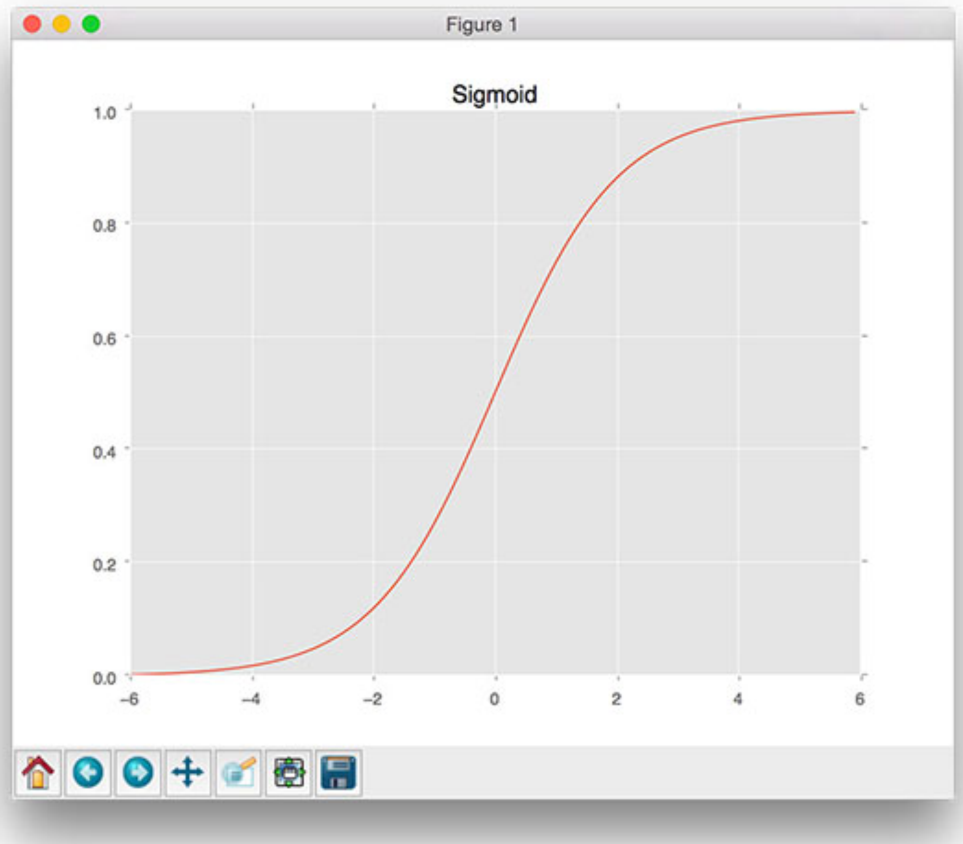


Figure 4: A plot of the sigmoid activation function. Notice how *y=0.5* when *x=0.*

We call this an activation function because the function will "activate" and fire "ON" (*output value >= 0.5)* or "OFF" (*output vale < 0.*
inputs `x` .

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

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

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 agai tune this hyperparameter 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 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 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 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 treat our bias as a *trainable parameter* that is ***within*** the weight matrix *W* rather than an entirely separate variable. You can learn m trick here and here.

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 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 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
```

```
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 o 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 produ 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 *predi* the *true values* (**Line 50**).

Line 55 computes the least squares error over our predictions (our loss value). The goal of this training procedure is thus to minimi 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 m taking a `--step` in the negative direction of the gradient, thereby allowing us to move towards the *bottom* of the basin of the loss (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 ar *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`
```

For each training point `X[i]` we compute the dot product between `X[i]` and the weight matrix `W` , then feed the value throug
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 outp

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 poi
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

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 accu
after each epoch:

```
[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:

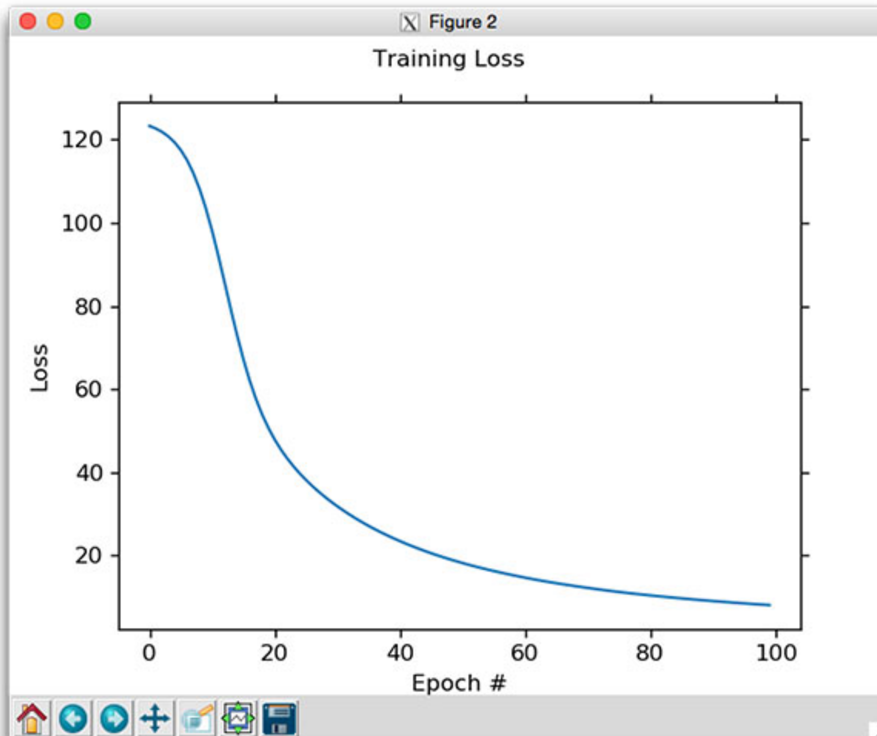


Figure 6: Plotting loss over time using gradient descent. Notice how loss sharply drops and then levels out towards later epochs.

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

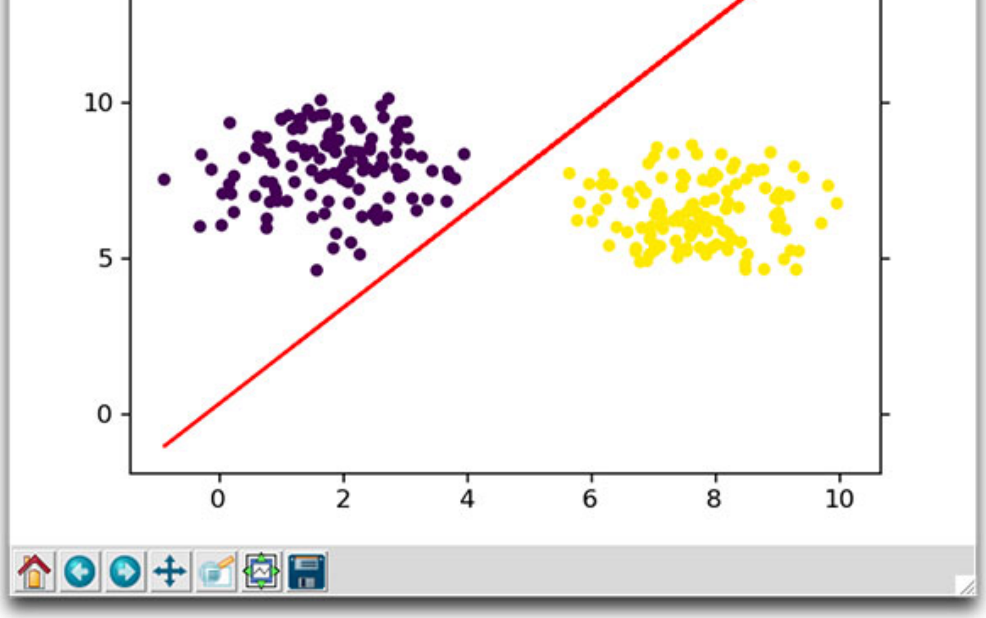


Figure 7: Plotting the decision boundary learned by our gradient descent classifier.

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 p

```
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 boundar after each epoch:

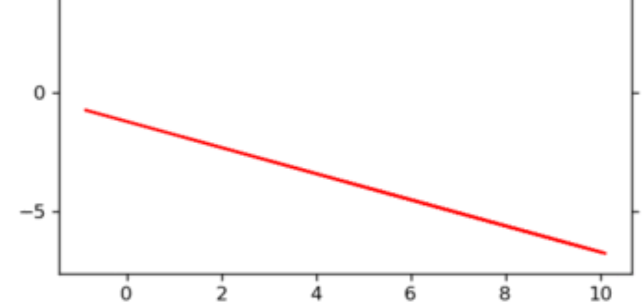


Figure 8: An animation depicting how the decision boundary is learned via gradient descent.

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

Want to learn more about gradient descent?

In next week's blog post, I'll be discussing a slight modification to gradient descent called *Stochastic Gradient Descent* (SGD).

In the meantime, if you want to learn more about gradient descent, you should absolutely refer to Andrew Ng's gradient descent le
Coursera Machine Learning course.

I would also recommend Andrej Karpathy's excellent slides from the CS231n course.

Summary

In this blog post we learned about *gradient descent*, a first-order optimization algorithm that can be used to learn a set of paramet
(ideally) obtain low loss and high classification accuracy on a given problem.

I then demonstrated how to implement a basic gradient descent algorithm using Python. Using this implementation, we were able
actually *visualize* how gradient descent can be used to learn and optimize our weight matrix *W*.

In next week's blog post, I'll be discussing a modification to the vanilla gradient descent implementation called *Stochastic Gradient*
The SGD flavor of gradient descent is more commonly used than the one we introduced today, but I'll save a more thorough discu
week.

See you then!

***Before you go, be sure to use the form below to sign up for the PyImageSearch Newsletter — you'll then be notified w
blog posts are published.***

Downloads:

If you would like to download the code and images used in this post, please enter your email address in the fo
only will you get a .zip of the code, I'll also send you a **FREE 11-page Resource Guide** on Computer Vision a
Engines, including **exclusive techniques** that I don't post on this blog! Sound good? If so, enter your email a
send you the code immediately!

Email address:

Your email address

Enter your email address below to get my **free 11-page Image Search Engine Resource Guide PDF** ... **exclusive techniques** that I don't publish on this blog and start building image search engines of you

Your email address

DOWNLOAD THE GUIDE!

classification, **deep learning**, **gradient descent**, **machine learning**, **neural nets**

< Bubble sheet multiple choice scanner and test grader using OMR, Python and OpenCV

Stochastic Gradient Descent

38 Responses to *Gradient descent with Python*

joe minichino October 10, 2016 at 11:26 am #

Congratulations Adrian. And yes, the article is also very interesting.

Adrian Rosebrock October 11, 2016 at 12:59 pm #

Thank you very much Joe!

Jason October 10, 2016 at 11:31 am #

Didn't have time to read this but CONGRATULATIONS! I hope your experience of marriage is as fulfilling and wonderful as mine

Adrian Rosebrock October 11, 2016 at 12:59 pm #

Thank you Jason! 🙂

Sujit Pal October 10, 2016 at 11:40 am #

Nice article. Just wanted to point out a typo. In your formula for d(f(x))/dx it should be limit of h tends to 0 not infinity.

Max Kostka October 10, 2016 at 12:17 pm #

Thanks for the informative post and the link to the slides, and I can totally recommend the machine learning course by andrew
too. It's a real good introduction to different machine learning techniques. The hands on approach with the homework is worth every mir

Johnny October 10, 2016 at 12:44 pm #

Fascinating! I love gradient descent studies like this one. I love Adrian's teaching style. And mostly, love that you shared the new
Congratulations you two, many blessings and prayers coming at you!

Adrian Rosebrock October 11, 2016 at 12:56 pm #

Thank you Johnny! 🙂

Ranjeet Singh October 11, 2016 at 2:19 am #

Great tutorial
Kidnly put some Backpropagation tutorials too.

Adrian Rosebrock October 11, 2016 at 12:52 pm #

I will certainly be doing backpropagation tutorials, likely 2-3 of them. Right now it looks like I should be able to publish them
November/December following my current schedule.

Wajih October 11, 2016 at 4:10 am #

This explanation is so beautiful, simple and elegant…

Adrian Rosebrock October 11, 2016 at 12:51 pm #

Thank you Wajih!

abkul October 12, 2016 at 2:43 am #

Thanks for the crystal clear explanation of the topic.

Congratulations for quiting bachelors club.

Moeen October 13, 2016 at 5:13 pm #

Hey Adrian- Big Congrats. I wish you the best!

Adrian Rosebrock October 15, 2016 at 9:57 am #

Thanks Moeen!

Arasch U Lagies October 14, 2016 at 1:20 am #

Congratulations Adrian.

Adrian Rosebrock October 15, 2016 at 9:56 am #

Thank you Arasch!

vantruong57 October 16, 2016 at 10:51 am #

Congratulations Adrian.

Adrian Rosebrock October 17, 2016 at 4:07 pm #

Thank you!

Srinivasan Babu October 21, 2016 at 9:46 am #

nice article.

Thanks lot

Adrian Rosebrock October 23, 2016 at 10:19 am #

Thank you Srinivasan!

Abkul October 31, 2016 at 2:18 am #

kindly do tutorials on “feature selection” in relation to deep learning

zauron November 2, 2016 at 6:49 pm #

Nice article! It's my fault but I don't understand how you calculate the decision boundary:

Y = (-W[0] – (W[1] * X)) / W[2]

Could you elaborate a little more or give me some reference?

Thanks in advance

dpereira December 1, 2016 at 8:38 am #

Hi zauron,

you can think of it like this:

In order to draw the decision boundary, you need to draw only the points (x,y) which lie right over the boundary.

According to the sigmoid function, the boundary is the value 0.5. So, in order to obtain a 0.5, you need to provide a zero value as in sigmoid (That is, a zero value as output from the scoring function).

Thus, if the scoring function equals zero:

0 = w0 + w1*x + w2*y ==> y = (-w0 – w1*x)/w2

You can use any x's coordinates you want, and you'll get the proper y's coordinates to draw the boundary

Chahrazad January 23, 2017 at 2:45 am #

hello, it is a bit confusiong to me how the gradient was computed :

gradient = X.T.dot(error) / X.shape[0] ,

shouldn't this computation be true if the loss function was derived using maximum likelihood estimation not the squared error ?

foobar April 12, 2017 at 5:42 am #

Great stuff, thanks!

"In the meantime, if you want to learn more about gradient descent, you should absolutely refer to Andrew Ng's gradient descent lesson Machine Learning course.

I would also recommend Andrej Karpathy's excellent slides from the CS231n course."

Both links are dead.

Adrian Rosebrock April 12, 2017 at 12:58 pm #

Thank you for letting me know! I have updated both links so they are now working.

Niki July 13, 2017 at 5:06 pm #

Thank you, an interesting tutorial! I'm a little bit confused though. When calculating the gradient, we try to minimize the loss fun
means we need to take the derivative of the loss function. The loss function is the sum of the square of the errors, with error being define
label minus the predicted label. Here the predicted labels are calculated using Sigmoid function. This means the gradient will include the
Sigmoid function, but here I see the gradient of a linear predictor function.
Could you elaborate more on what has been done. Thank you!

Niki July 13, 2017 at 7:21 pm #

My bad! I got error for W in the gradient formula.

Adrian Rosebrock July 14, 2017 at 7:24 am #

Congrats on figuring it out Niki, nice job 🙂

Kiro September 20, 2017 at 4:33 am #

I think you are right. There is a mistake. Shouldn't it be like that:

gradient = X.T.dot(error) * preds*(1-preds)

where preds*(1-preds) is the derivative of the sigmoid function?

Thanks!

Ben August 17, 2017 at 10:27 am #

Congratulations on your pending nuptial.

Adrian Rosebrock August 17, 2017 at 10:31 am #

Thanks Ben!

Trackbacks/Pingbacks

Stochastic Gradient Descent (SGD) with Python - PyImageSearch - October 17, 2016
[…] last week's blog post, we discussed gradient descent, a first-order optimization algorithm that can be used to learn a set of classifier co

Leave a Reply

Name (required)

Email (will not be published) (required)

Website

SUBMIT COMMENT

Search...

Resource Guide (it’s totally free).

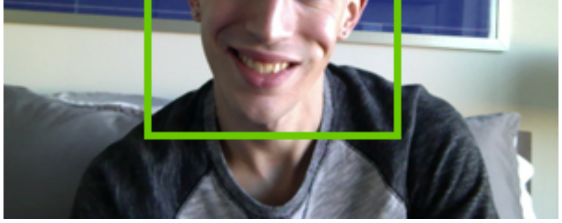
Click the button below to get my **free 11-page Image Search Engine Resource Guide PDF**. Uncover **exclusive techniques** t
on this blog and start building image search engines of your own.

Download for Free!

Deep Learning for Computer Vision with Python Book — OUT NOW!

You're interested in deep learning and computer vision, *but you don't know how to get started.* Let me help. **My new book will teach you all you ne
deep learning.**

CLICK HERE TO MASTER DEEP LEARNING



Are you interested in **detecting faces in images & video?** But **tired of Googling for tutorials** that *never work?* Then let me help! I guarantee that r
turn you into a **face detection ninja** by the end of this weekend. Click here to give it a shot yourself.

CLICK HERE TO MASTER FACE DETECTION

PyImageSearch Gurus: NOW ENROLLING!

The PyImageSearch Gurus course is *now enrolling!* Inside the course you'll learn how to perform:

- Automatic License Plate Recognition (ANPR)
- Deep Learning
- Face Recognition
- *and much more!*

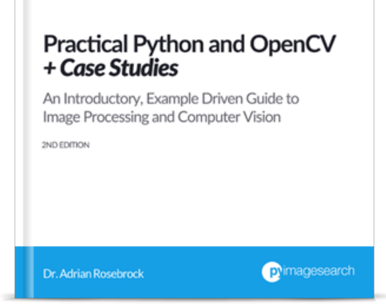
Click the button below to learn more about the course, take a tour, and get 10 (FREE) sample lessons.

TAKE A TOUR & GET 10 (FREE) LESSONS

Hello! I'm Adrian Rosebrock.

I'm an entrepreneur and Ph.D who has launched two successful image search engines, ID My Pill and Chic Engine. I'm here to sha
and hacks I've learned along the way.

Learn computer vision in a single weekend.



Want to learn computer vision & OpenCV? I can teach you in a **single weekend**. I know. It sounds crazy, but it's no joke. My new book is your **guara**
guide to becoming an OpenCV Ninja. So why not give it a try? Click here to become a computer vision ninja.

CLICK HERE TO BECOME AN OPENCV NINJA

Subscribe via RSS

Never miss a post! Subscribe to the PyImageSearch RSS Feed and keep up to date with my image search engine tutorials, tips, and tric

POPULAR

Install OpenCV and Python on your Raspberry Pi 2 and B+
FEBRUARY 23, 2015

Install guide: Raspberry Pi 3 + Raspbian Jessie + OpenCV 3
APRIL 18, 2016

Home surveillance and motion detection with the Raspberry Pi, Python, OpenCV, and Dropbox
JUNE 1, 2015

How to install OpenCV 3 on Raspbian Jessie
OCTOBER 26, 2015

Basic motion detection and tracking with Python and OpenCV
MAY 25, 2015

Ubuntu 16.04: How to install OpenCV
OCTOBER 24, 2016

Accessing the Raspberry Pi Camera with OpenCV and Python
MARCH 30, 2015

Find me on **Twitter**, **Facebook**, **Google+**, and **LinkedIn**.