

<https://www.udacity.com/course/deep-learning--ud730>

# [730] (DEEP LEARNING)

ALBERT GARCIA

VINCENT VANHOUCKE • (WINTER) 2016 • UDACITY – GOOGLE RESEARCH

---

Last Revision: February 1, 2016

## Table of Contents

<b>1</b>	<b>Machine Learning to Deep Learning</b>	<b>1</b>
1.1	Linear Model . . . . .	1
1.2	Softmax . . . . .	1
1.3	One-Hot Encoding . . . . .	1
1.4	Cross Entropy . . . . .	2
1.5	Multinomial Logistic Classification . . . . .	2
1.6	Minimizing Cross Entropy . . . . .	3
1.7	Stochastic Gradient Descent . . . . .	3

---

**Abstract**

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. If you spot any errors or would like to contribute, please contact me.

**1 Machine Learning to Deep Learning****1.1 Linear Model**

The most simple linear model that allows us to make predictions is represented by the following equation:

$$wx + b = y,$$

where  $w$  is the weights matrix,  $b$  is the bias vector,  $x$  is the matrix corresponding to the values of the sample to be classified, and  $y$  is the output or scores vector – which has as many elements as classes can be classified. The training will change the values of the weights and biases to tune the prediction system so that for each input  $x$  its correct class is predicted in  $y$ .

**1.2 Softmax**

The way to turn scores (also known as logits) into probabilities is to use a softmax function  $S$  and apply it to all the elements of the classifier output vector  $Y$ . Assuming that the output of the classifier has  $n$  dimensions, the softmax function is defined as follows

$$S(y_i) = \frac{e^{y_i}}{\sum_{j=0}^n e^{y_j}}.$$

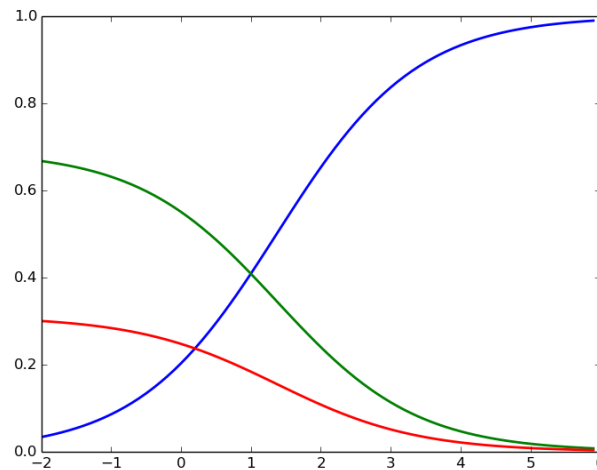
There are two important remarks about this softmax function. On the one hand, if the magnitude of the scores increases significantly, the probabilities get close to either 0 or 1. On the other hand, if the magnitude of the scores decreases, the probabilities tend to get close to the uniform distribution. In other words, increasing the magnitude of the classifier output makes the system more confident about the decision once softmax is applied and vice-versa (see Figure 1.1). For machine learning systems, we would like to progressively turn the system from unsure to confident.

**1.3 One-Hot Encoding**

We need a way to represent class labels mathematically. We can use some sort of one-hot encoding for that, representing each label with a vector that has as many elements as class labels our system is able to classify. Each one-hot class vector will have a 1 at the position of the corresponding class, and the rest of the elements will be 0.

a	0	0	0	1
b	0	0	1	0
c	1	0	0	0
d	0	1	0	0

**Table 1:** One-hot encoding example vectors for four classes  $a$ ,  $b$ ,  $c$ , and  $d$ .



**Figure 1.1:** Softmax outputs as the score of class  $x$ , in blue, increases.

By doing this, we can map the classifier output probability provided by softmax to the appropriate class label following a likeness criteria. For instance, for the output probability vector  $S(y) = [0.7 \ 0.0 \ 0.1 \ 0.2]$ , and the one-hot encoding example shown previously, the most likely class is  $c$ .

This approach works well for most of the cases, but in some situations we can have a vast number of classes so that we end up having huge class vectors with almost all positions filled up with zeroes. This turns out to be quite inefficient. However, using one-hot encoding provides us a way to determine how well our classifier is doing by comparing the probabilities vector provided by softmax and the correct class label one-hot encoded vector.

## 1.4 Cross Entropy

The natural way to measure the difference between two probability vectors is the cross-entropy  $D$ . Given a probability vector from softmax  $S(y)$ , and a class label one-hot encoded vector  $L$ , their difference can be expressed as follows

$$D(S, L) = - \sum_{i=0}^n L_i \log(S_i).$$

**Be wary,**  $D(S, L) \neq D(L, S)$ .

## 1.5 Multinomial Logistic Classification

The setting that we have described throughout the previous sections takes an input  $x$ , which is turned into a logit  $y$  by means of a linear model  $wx + b = y$ . Then, that logit is fed to a softmax layer  $S(y)$  which converts those scores into probabilities. At last, the probabilities are compared to the different classes one-hot encoded labels using the cross-entropy function to determine the prediction. This setup is called multinomial logistic classification and it is represented by the following expression:

$$D(S(wx + b), L)$$

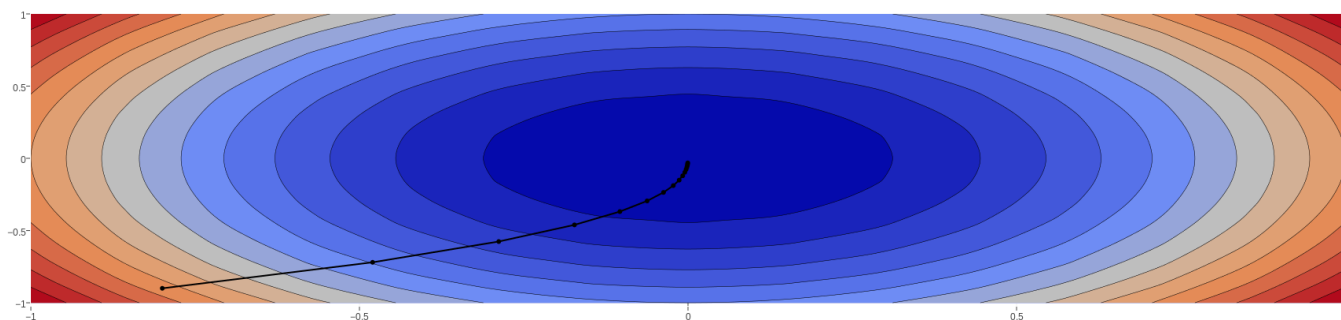
## 1.6 Minimizing Cross Entropy

In order to find the weights and biases that will produce low distances for the correct classes, e.g.,  $D(A, a)$ , and high distances for incorrect ones, e.g.,  $D(A, b)$ , we will minimize the loss  $\zeta$  which is expressed as

$$\zeta(w, b) = \frac{1}{N} \sum_{i=0}^N D(S(wx_i + b), L_i),$$

where  $N$  is the number of samples  $x$  and labels  $L$  in the training set. In other words, the loss is the average cross-entropy over the entire training set. This loss, is a function of multiple parameters: the weights and the biases. This function will be large in some areas and small in others. We will find the parameters that cause the loss to be the smallest as possible. By doing this, our problem is turned into a numerical optimization one.

The simplest way to solve this problem is Gradient Descent (GD). This will take the derivative of the loss with respect to the parameters and follow that derivative by taking steps backwards until we get to a minimum, so the step will be  $-\alpha \Delta \zeta(w, b)$ . Figure 1.2 represents this procedure for a simple loss function with two parameters.



**Figure 1.2:** GD from a larger area (red) to a smaller one (blue) taking steps following the derivative backwards (black) until the minimum is reached.

## 1.7 Stochastic Gradient Descent

As we have seen earlier, the loss is huge function which depends on every single element of the training set, which means a lot of computing power if we want to train with big datasets. In practice, we will get more gains the more data we use. In addition, GD is iterative, so we will have to compute that loss for many steps, so this will not scale well.

Instead of computing the loss, we will compute an estimate of it. It will consists of computing the average loss for a tiny random fraction of the training set for each step we take. Then we will compute that estimate, the derivative of that estimate, and pretend that it is the right direction to follow to perform GD.

It might not be the right direction, and sometimes it will increase the loss instead of decreasing it, but this fact will be compensated by doing much more steps of a smaller size. In balance, each step is much cheaper to compute, but we will have to take many more of them to arrive to a solution. In the end, we win by a lot, since this technique called Stochastic Gradient Descent (SGD) is vastly more efficient for larger datasets than simple GD.