# Week 1

### **What is a Neural Network?**

A neural network is a function which you give some parameters and itself, returns you some other value related. For example, if you give specifications of a house-like size, number of bedrooms, etc., then this function can calculate the price for this house. Between the parameters that you are giving and the result, there are connections between all of them. The function that we would like to use is implemented by the neurons (keep between the inputs and the output).

A ReLU function is a function that comes from a 0 and then increases to a linear function.

### **Supervised Learning with Neural Networks**

* Standard Neural Network: neural networks like calculating the price of a house with parameters or showing ads to a user depending on its information.
* Convolutional Neural Networks: for apps which use images
* Recurrent Neural Network: for sequence data, like audio or language (word comes one at a time, sequence)

Structured data: the features have a well-defined meaning.

Unstructured data: it works with images, audio or text. It is more difficult for computers to work with this data.

### **Why is Deep Learning taking off?**

Neural networks are bigger when they have more models to be trained. At first all the different sizes of NN are practically the same but when the quantity of data increases, depending on the NN it would do its performance.

The learning process had improved from a Sigmoid function to a ReLU function making Gradient descent works much faster.

Fast computation is also important because the process of training a neural network is very iterative and it is better to know if your idea works as fast as possible.

# Week 2

## Logistic Regression as a Neural Network

### **Logistic Regression**

To calculate which probability of something, we use this formula:

This is the sigmoid activation function. When is very small the probability would be almost 1, but when is very big the probability would be almost 0.

### **Logistic Regression Cost Function**

: prediction that is calculated

: result got in the training set

We want that and are the most equals as possible.

Loss function (single):

Cost function (for the entire training set):

### **Gradient Descent**

The objective is to get and that minimize :

We must initialize and at some value to get to the global optimum following the two next functions:

A colorful graph of a graph

Description automatically generated with medium confidence

learning rate

### **Derivatives**

Know how to derivate functions and know their slope (pendiente)

### **More Derivative Examples**

Know how to derivate functions and know their slope (pendiente)

### **Computation Graph**

The computation graph explains why in neural networks going forwards we first get the output but when we go backwards, we compute gradients or derivatives.

A math equation with arrows pointing to the sides

Description automatically generatedThis function has different steps shown in the drawing.

First, it’s calculated , then and finally . Going forwards we can get the output of the function, the result that the neural network gives us.

### **Derivatives with a Computation Graph**

The objective is to see how changes depending on the rest of the values.

A diagram of a diagram

Description automatically generatedFor example, if we change the increase is of:

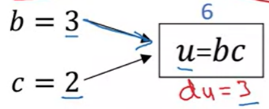
A diagram of a line with arrows and a red line

Description automatically generated with medium confidenceThis is the first step backwards the function, now we have two more derivatives in the second step:



We can also see the next equality:

And the same occurs with the next step:





Finally, if we see how affects :

A math equations and formulas

Description automatically generated with medium confidence



### **Logistic Regression Gradient Descent**

We want to modify , and to reduce the loss function .

A diagram of a mathematical equation

Description automatically generated

First, we calculate the derivative of :

Then the :

And finally, , and :

So, we finally get:

### **Gradient Descent on m Examples**

To do so, we must implement a for loop that repeats the process for all the m examples:

## Python and Vectorization

### **Vectorization**

Vectorization is much faster than for loops, so it’s very recommended to avoid the use of loops and instead use vectorization.

import numpy as np

import time

a = np.random.rand(1000000)

b = np.random.rand(1000000)

tic = time.time()

c = np.dot(a,b)

toc = time.time()

print("Vectorized version:"+str(1000\*(toc-tic)) + "ms")

c = 0

tic = time.time()

for i in range(1000000):

    c += a[i] \* b[i]

toc = time.time()

print("For loop:" + str(1000\*(toc-tic)) + "ms")

The code line shows how to calculate without a for loop and the code lines show how to do exactly the same but with a for loop. This code shows that the vectorized way is 300 times faster than the for loop, because of that it’s important to use vectorized functions.

### **More Vectorization Examples**

To do logistic regression with vectorization we must change all the variables to a vector named and make the appropriate changes to the rest of the code:

To see all the changes compare this code with the one from 10. Gradient Descent on m Examples.

### **Binary Classification**

Images are classified as 3 matrixes of numbers that are the colors red, green and blue. These matrixes become a feature vector with the values ordered.

A training model is represented by a pair where is a feature vector of size and the result of this, that can be 0 or 1.

We can represent x in a more compact form with a matrix:

are the different training examples; is the quantity of these training examples and is the size of the examples.

And we can do the same with y:

y are the results, and m is the quantity of results.

### **Vectorizing Logistic Regression**

What we want to do is calculate the and the for all the examples. For that we must calculate and :

To calculate we use this in Python:

Although b is a , Python transforms it into a matrix automatically, this is named Broadcasting.

To calculate A is the same process:

### **Vectorizing Logistic Regression’s Gradient Output**

First, we vectorize :

We can reduce even more the quantity of for loops in the code for logistic regression (although there would be one left):

### **Broadcasting in Python**

To explain the broadcasting in Python, we create an array of 3x4:

import numpy as np

A = np.array([[56.0, 0.0, 4.4, 68.0],

            [1.2, 104.0, 52.0, 8.0],

            [1.8, 135.0, 99.0, 0.9]])

With the next function, we are adding vertically. If we put, then it would add it horizontally.

cal = A.sum(axis = 0)

Here is where broadcasting occurs, we are dividing a matrix of 3x4 by 1x4. We can also write this code without , we put it to make sure that the matrix is 1x4.

percentage = 100\*A/cal.reshape(1,4)

Broadcasting works like it’s shown next:

Written in general it would look like this:

With real numbers too:

### **A Note on Python/Numpy Vectors**

When creating a random matrix, be careful and do not create an array of shape (5,), because could lead to errors or different matrix shapes. This type of matrix is call “rank 1 array”.

a = np.random.randn(5) // wrong!!

a = np.random.randn(5,1) // correct!!

Furthermore, we can see that the function also has the property of broadcasting. In the next example it multiplies a matrix (5,1) by another one of (1,5) (it’s its transposed matrix) and it returns a (5,5):

a = np.random.randn(5,1)

print(np.dot(a,a.T))

To establish the size of the matrix if we get a rank 1 array we can resize it:

a = a.reshape((5,1))

### **Quick tour of Jupyter/Python Notebooks**

Notebooks where code with python small code:

<https://jupyter.org/try-jupyter/notebooks/?path=Untitled1.ipynb>

# Week 3

### **Neural Networks Overview**

A diagram of a complex function

Description automatically generatedWe would refer to the nodes with the superscript , nodes are also called layers.



The order to follow to calculate, finally, the loss function is the next one:

A diagram of mathematical equations

Description automatically generated



### **Neural Network Representation**

A diagram of a network

Description automatically generatedThere are 3 layers in a neural network:

* Input layer: where we see and introduce the inputs. It’s not officially a layer.
* Hidden layer: as its name says, we can’t see what is in this layer.
* Output layer: where we see and get the output.

The hidden layer is computed as shown next:

For each layer we have different values and . For example, for layer 1: is a matrix (4,3) because it has 3 inputs and 4 nodes.

### **Computing a Neural Network’s Output**

To get the value of and we must calculate the separate values:

Or we can’t vectorize the inputs, and :

To obtain the value of we must follow the next steps:

### **Vectorizing Across Multiple Examples**

Firstly, we could do it with the same formulas in the previous video with an extra for loop. But we want to vectorize this to avoid the loop.

Having as the examples:

The formula with the for loop:

So, if we vectorize this loop we’ll get:

Where , and look like this:

In these three arrays we have that the columns correspond to the number of examples and the rows correspond to the hidden units. But the rows of correspond to its different features.

### **Explanation for Vectorized Implementation**

(explanation of the previous video explanation, nothing new)

### **Activation Functions**

Since now we have only been using one activation function, but there are four that we can use:

|  |  |  |
| --- | --- | --- |
|  | Sigmoid | |
|  | Never use it except for the output layer if you are doing binary classification. |
|  | tanh | |
|  | Much superior than the sigmoid function. |
|  | ReLU | |
|  | The most common function, if you are not sure which is better, use this. |
|  | Leaky ReLU | |
|  | Works better than ReLU but is not that used. |

### **Why do you need Non-Linear Activation Functions?**

If we get rid of the g function in the next combination of formulas:

If we compute a neural network with a linear function, we will be outputting exactly what we are getting in the input. As shown in the next equality:

Where we may use a linear activation function is in the output layer, but in the hidden layers we would be using a ReLU or a tanh function.

In some cases, we would only use linear function if we want that the output be the same or a variant of the input.

### **Derivatives of Activation Functions**

To know the derivative

|  |  |
| --- | --- |
| Sigmoid |  |
| A black line with a black line and a black line  Description automatically generated |
| If  If  These two are the extremes of the functions that are practically horizontal.  If  This last point is when the function crosses the y axis. |
| tanh |  |
| A graph of a function  Description automatically generated |
| If  If  These two are the extremes of the functions that are practically horizontal.  If  This last point is when the function crosses the y axis. |
| ReLU |  |
| A line with a point and a line with a point  Description automatically generated with medium confidence |
|  |
| Leaky ReLU |  |
| A line with a point and a line with a point  Description automatically generated with medium confidence |
|  |

### **Gradient Descent for Neural Networks**

Parameters:

Cost function:

We need to compute the derivatives:

And we will update the variables to:

Forward propagation:

Back propagation:

### **Backpropagation Intuition**

A diagram of a mathematical equation

Description automatically generated



### **Random Initialization**

If we initialize all the hidden layers with the same value, for example 0, all the hidden layers will be computing the same function and what we want is that every hidden layer computes a different function.

We can initialize the values of the matrix to a random value but for the values of there’s no need to initialize it randomly, it can be perfectly 0.

The number must be a small number to have the smallest values of so the activation function won’t saturate and compute slow.

# Week 4

### **Deep L-layer Neural Network**

Here are a few different neural networks:

A diagram of a logistic regressing

Description automatically generatedA diagram of a network

Description automatically generatedA diagram of a network

Description automatically generatedA diagram of a network

Description automatically generated

* Logistic regression: 1 layer, shallow.
* 1 hidden layer: 2 layers.
* 2 hidden layers: 3 layers.
* 5 hidden layers: 6 layers, deep.

A diagram of a network

Description automatically generatedWe’ll take the example of the next NN:

### **Forward Propagation in a Deep Network**

Forward propagation in a deep network would look like this:

It should be noted that and .

If we want to do it vectorized, we have this:

Likewise, and .

To go through all the layers, we will use a for loop, there is no other way to do it without a for.

### **Getting your Matrix Dimensions Right**

To know the dimensions of a neural network we must follow these rules:

The derivatives for all these vectors also have the same dimensions.

For the vectorized option is very similar, remember that are the number of examples:

### **Why Deep Representation?**

Deep neural networks work much better than the shallow ones because the more layers it has, the more proves it can do to the given input.

For example, if we wanted to make a facial recognition IA, we would create a neural network with one layer to recognize the edges of the face, another one to recognize the different parts of the face and another one to recognize a whole face. We first begin with basic things and then we continue to more complex things.

There are functions you can compute with a small L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

|  |  |
| --- | --- |
| Small deep | More hidden units |
| XOR  XOR  XOR  XOR  XOR  XOR  XOR |  |

### **Building Blocks of Deep Neural Networks**

Representation of how forward and backward functions work:

Layer l

Cache

### **Forward and Backward Propagation**

Implementation of forward and backward propagation explained in the previous video.

|  |  |  |
| --- | --- | --- |
|  | Non-vectorized | Vectorized |
| Forward |  |  |
| Backward |  |  |

### **Parameters vs Hyperparameters**

* Parameters:
* Hyperparameters: learning rate , nº iterations, nº hidden layers (), nº hidden units (), choice of activation function

You need to try out the possible settings for your hyperparameters following the schema: Idea Code Experiment and again Idea. For example, with the learning rate, we could try different values till we get the one that gives us better results with less cost as the next graphic:

A diagram of a cycle

Description automatically generated A blue line drawing of a fish

Description automatically generated

It is necessary to try a lot of different values for the hyperparameters to see if it works or try again and again till it does. In the next course it will be explained how to do this in a systematic way with a range of values.

The perfect value that you choose today may not be the better option in a large period because of the infrastructure of computers like CPUs. This means that you would have to find the new better option that works on your neural network.

### **What does this have to do with the brain?**

It is sometimes compared to the brain because we could say that a neuron looks like a neural network and computes something to get a result.

A drawing of a nerve cell

Description automatically generated

There are some similarities with the brain but as even neuroscientists have no idea about what a neuron is doing, we can’t make a lot of similarities.