Neural Networks and Deep Learning

Notes by José A. Espiño P. [1]

Summer Semester 2022–2023

[1]The content in these notes is sourced from what was covered in the MOOG the document is named after. I claim no autorship over any of the contents herein.

Contents

1 Introduction to Deep Learning

Deep Learning refers to training neural networks. The most essential component of a neural network is a **neuron**, a unit that take in certain inputs, operates on them, and outputs a result. In a NN, several neurons are stacked together in **layers**, where each neuron will be tasked with computing features out of certain inputs.
For instance, imagine your inputs for a housing price prediction NN are the house size, number of bedrooms, zip code, and wealth. Perhaps one of the neurons in the next layer will calculate the family size (from the size and number of bedrooms), another one will calculate the walkability (from zip code), and another one will calculate the school quality (from zip code and wealth). These three new features can then be fed as a vector to another neuron that will compute the price.
The feature that makes NN so attractive is that the features in the middle will be discovered by the algorithm during training; you only need to provide the network with x inputs paired with y outputs as training samples.
In a neural network, each neuron in a layer is called a **hidden unit**. **Dense** neural networks are characterised for providing every individual neuron in a layer with all the outputs of the previous one, even if the feature they compute only requires of a portion of the features.

2 Neural Networks Basics

Logistic regression, an algorithm for binary classification, will be the basis of our introduction. If we have an image as an input, we will *unroll* the red, green, and blue pixel matrices into a vector $\vec{x}$ of size $hxwx3$, where h and w are the height and width of the image in question. Usually, all the inputs will be stored in a matrix X where every train sample is a column. This matrix will be of n, m dimensions, where n is the amount of features per input and m is the amount of samples.
So, given an input feature vector $\vec{x}$, we want an algorithm that outputs a prediction $\hat{y}$, an estimate of the probability that $y =$ label 1. $\hat{y}$ is generated by calculating $\sigma(w^T x + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$ and $\vec{w}, b$ are parameters you want to estimate during training.
As you know, training is done by trying to minimise parameters $\vec{w}$ and b in a given cost function. The cost function we shall use for logistic regression is the average of the **the**

log–likelihood function, namely:

$$J(w,b) = -\frac{1}{m}\sum_{i=1}^{m}(y^i log(\hat{y}^i) + (1-y^i) log(1-\hat{y}^i))$$

Where m stands for the number of training samples.
Equipped with this, we can use an optimisation algorithm to find the parameters that most effectively diminish this cost. One of the most common functions for this purpose is **gradient descent**, which does the following:
Repeat{

$$w_i := w_i - \alpha \frac{\partial J(\vec{w}, b)}{\partial w_i}$$

$$b := b - \alpha \frac{\partial J(\vec{w}, b)}{\partial b}$$

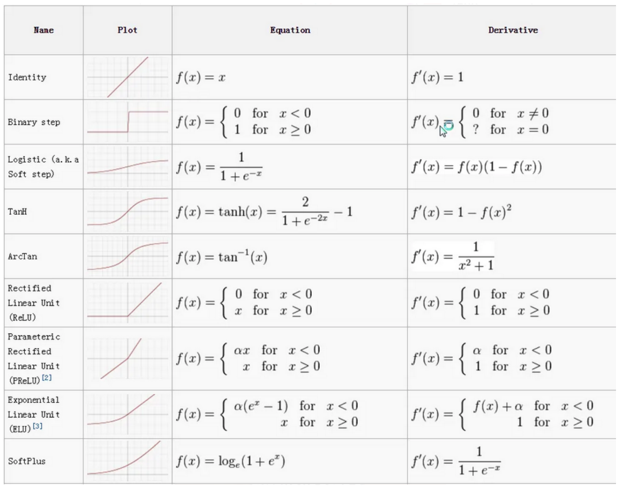
}
The w update must be applied for all m features. The updates must be done simultaneously, meaning that we will perform the operations in the curly brackets and then we will update all the parameters.
The computations of a neural network are organised in a **forward propagation step**, in which we compute the output of the neural network, followed by a **back propagation step**, in which we compute the gradients/derivatives.

Vectorisation can render gradient descent much more efficient than it currently is.

3 Shallow Neural Networks

The three core components of a neural network are the **input layer**, the **hidden layer**, and the **output layer**. In a training set, we have the values for the inputs and outputs, but not the ones for the layers in the middle — which is why we call it hidden in the first place. The values that each layer passes to the next layer are called the **activation**. The hidden layers and the output layer will have parameters $\vec{w}$ and $\vec{b}$ associated to them. $\vec{w}$ will be a (n, m) matrix, where n refers to the number of nodes in the hidden layer and m to the number of inputs. Similarly, $\vec{b}$ will be a vector of $(n, 1)$ dimension.
In a neural network, every neuron in a layer will perform a certain operation (e.g. logistic regression) on the input vector it receives and the entire layer will output an activation vector $\vec{a}$. With vectorisation, we might be able to compute the output of the neural network for several samples at a time: we create a matrix X where every training sample $\vec{x}$ is stacked as a column. We will then compute each activation as $\sigma(W^T X + \vec{b})$, which will then be stacked on a matrix, where each column will have the activation vector generated by each training sample.
So far we have been using the Sigmoid function as our activation function. This is not the most ideal option in most cases. A function that usually outperforms the Sigmoid function is the **hyperbolic tangent** function, defined as $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. The one exception in

this case is the output layer of binary classification, where the Sigmoid activation function proves the most efficient. One of the downside of these two is that if z is very large or small, the slope will be close to zero — which significantly slows down gradient descent. Another function that avoids this issue is **Rectified Linear Unit**, also known as **ReLU**. This function is defined as $Q(z) = max(0, z)$. As a rule of thumb, if you do not know which activation function to use, opt to use ReLU. A downside of this function is the fact that any negative derivatives will be zeroed. This is addressed by another function, **Leaky ReLU**, which is defined as $QL(z) = max(0.01z, z)$.
Here are some ways to compute the derivative of these common activation functions during back propagation:



Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Figure 1: Sourced from here

When it comes to gradient descent, it is the following way:

```
Repeat until convergence{
        Compute predictions $(\hat{y}^{(i)}, i - 1..., m)$
        $\partial w^{[i]} = \frac{\partial J}{\partial w^{[i]}}, \partial b^{[i]} = \frac{\partial J}{\partial b^{[i]}}$
        $w^{[i]} = w^{[i]} - \alpha \partial w^{[i]}$
        $b^{[i]} = b^{[i]} - \alpha \partial b^{[i]}$
    }
```

In short, the process is:

1. Forward Propagation:

$$
\begin{aligned}
Z^{[i]} &= W^{[i]}X + b^{[i]} \\
A^{[i]} &= g^{[i]}(Z^{[i]}) \\
Z^{[i+1]} &= W^{[i+1]}A^{[i]} + b^{[i+1]} \\
A^{[i+1]} &= g^{[i+1]}(Z^{[i+1]})
\end{aligned}
$$

   Until we get to the last layer.

2. Back Propagation (let u be the last layer):

$$
\begin{aligned}
\partial Z^{[u]} &= A^{[u]} - Y \qquad\qquad Y = [y^{(1)}, y^{(2)}, ..., y^{(m)}] \\
\partial W^{[u]} &= \frac{1}{m}\partial Z^{[u]} A^{[u-1]T} \\
\partial b^{[u]} &= \frac{1}{m}\text{np.sum}(\partial Z^{[u]}, axis = 1, keepdims = True) \\
\partial Z^{[u-1]} &= W^{[u]T}\partial Z^{[u]} * g^{[u-1]}(Z^{[u-1]})
\end{aligned}
$$

   continue until first layer

$$
\begin{aligned}
\partial W^{[i]} &= \frac{1}{m}\partial Z^{[i+1]} X^T \\
\partial b^{[i]} &= \frac{1}{m}\text{np.sum}(\partial Z^{[i]}, axis = 1, keepdims = True)
\end{aligned}
$$

When training the neural network, it is important to initialise the weights **randomly**; initialising them to zero and then applying gradient descent will be unsuccessful. This is because this renders the activation of every neuron and thus their derivatives through back propagation identical, a problem called *symmetry breaking*.

4 Deep Neural Networks

A **deep neural network** is a neural network which has several layers. When we are covering DNNs, L will refer to the number of layers in the network, and $n^{[l]}$ refers to the number of units in a specific layer l. Similarly, a^l represents the activation produced by each individual layer.
In this type of network, forward propagation takes a slightly different shape: each layer will be computed as:

$$
\begin{aligned}
Z^{[n]} &= W^{[n]}A^{[n-1]} + B^{[n]} \\
A^{[n]} &= g^{[n]}(Z^{[n]})
\end{aligned}
$$

Remember that $A^{[0]}$ is the input vector X.
A very crucial factor to be considered in the implementation of a DNN is the dimension

of the matrices that are being handled. For every step of forward propagation, we must consider W and A. $W^{[l]}$ must have the dimensions $[n^{[l]}, n^{[l-1]}]$ while $A^{[l-1]}$ will have the dimensions $[n^{[l-1]}, 1]$. The dimensions of the vector B must match that of the vector WA; thus, $B^{[l]}$ will have the dimensions $[n^{[l]}, 1]$. Even for vectorised implementations, the dimensions of these three elements will remain consistent. The only point to note is that B will be broadcasted to match the dimensions of the matrix it will be summed to.

Take a look at the usual flow of information in a deep neural network iteration:

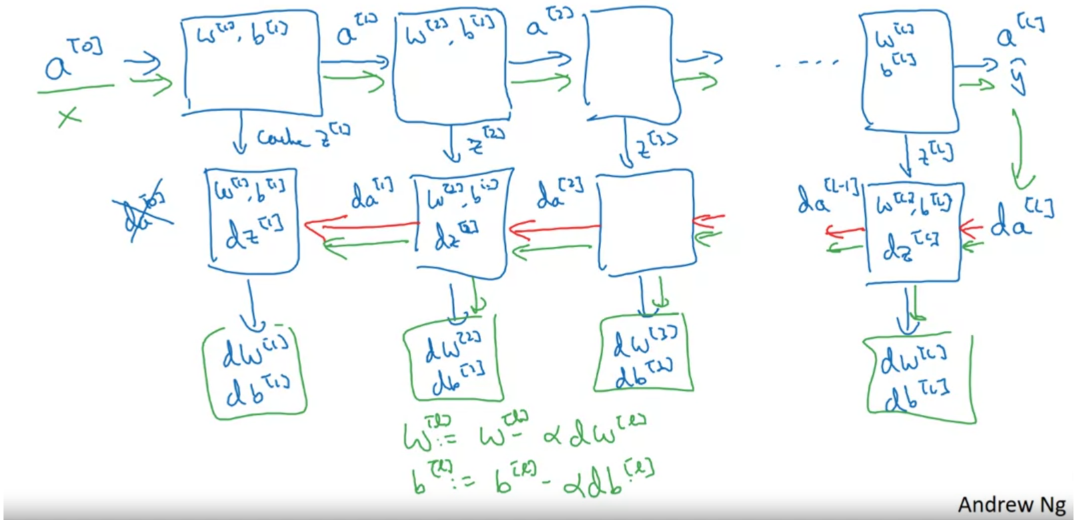


Figure 2: Sourced from here

Let us see the implementation of forward propagation:
As you can see in the above diagram, our goal is to obtain an output $A^{[l]}$ and to cache the value of $Z^{[L]}$ from an input $A^{[l-1]}$. You might already be familiar with the equations needed to realise this:

$$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

Similarly, in the case of backward propagation, our goal is to obtain a outputs $da^{[l-1]}, dW^{[l]}, db^{[l]}$ from an input $da^{[l]}$. This is realised through:

$$dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$$
$$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot dA^{[l]}$$
$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis = 1, keepdims = True)$$
$$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$$

For the final layer, the formula we use in this case is $dA^{[1]} =$ (Do this for every element) $-\frac{y}{a} + \frac{(1-y)}{(1-a)}$.

Hyperparameters are parameters that control the final value of the regular parameters, such as W and b. It includes the learning rate α, the number of iterations, the number of hidden layers L, the number of hidden units $n^{[x]}$, etc. Later on, we will cover some ways to systematically find the best values for hyperparameters, but it often involves a large amount of trial and error.

```
sample!
```