

Mathematical Notes

Contents

1	Deep Learning: Basics	3
1.1	McCulloch Pitts Neuron	4
1.2	Perceptron	7
1.3	Sigmoid Neuron	10
1.4	Feedforward Neural Networks	11
1.4.1	Motivation: XOR Function With A Network Of Perceptrons	11
1.4.2	Forward Propagation	14
1.4.3	Backward Propagation	16
1.4.4	Extra: Generalize To Full Dataset	19
1.5	Modified Gradient Descent Algorithms	20
1.5.1	Mini-Batch & Stochastic Gradient Descent	20
1.5.2	Gradient Descent With Momentum	21
1.5.3	Nesterov Accelerated Gradient Descent	22
1.5.4	Adaptive Gradient Descent (AdaGrad)	23
1.5.5	Root Mean Square Propagation (RMSProp)	24
1.5.6	Adaptive Moment Estimation (Adam)	24
1.5.7	Learning Rate Decay	27
1.6	Errors, Evaluation & Regularization	27
1.6.1	L2 Regularization	28
1.6.2	Dropout	28
1.6.3	Dataset Augmentation	29
1.6.4	Early Stopping	29
1.7	Improvements On A Feedforward Neural Network	29
1.7.1	Activation Functions	29
1.7.2	Initialization Strategies	31
1.7.3	Batch Normalization	31

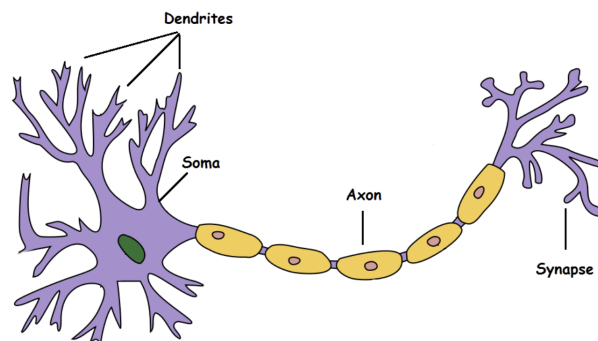
Chapter 1

Deep Learning: Basics

Definition 1.1 (Deep Learning). *Deep learning* is a class of machine learning algorithms that uses multiple layers (hence the adjective “deep”) to progressively extract higher-level features from the raw input. It is part of a broader family of machine learning methods based on artificial neural networks with representation learning.

Deep learning architectures such as deep neural networks, deep belief networks, recurrent neural networks and convolutional neural networks have been applied to fields including computer vision, machine vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics, drug design, medical image analysis, material inspection and board game programs, where they have produced results comparable to and in some cases surpassing human expert performance.

The most fundamental unit of a deep neural network is called an artificial neuron. The inspiration comes from the brain where the biological neurons are the neural processing units.



The layers of a biological neuron are:

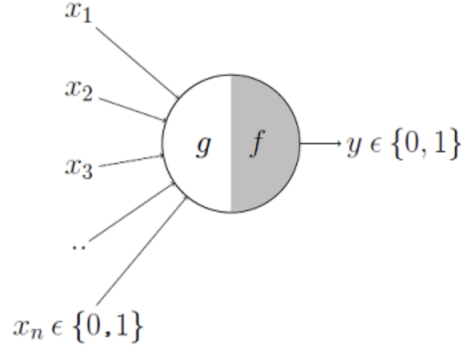
- Dendrite: receives signals from other neurons.
- Synapse: point of connection to other neurons.
- Soma: processes the information.
- Axon: transmits the output of this neuron.

Basically, a neuron takes an input signal (dendrite), processes it like the CPU (soma), passes the output through a cable like structure to other connected neurons (axon to synapse to other neuron's dendrite). Now, this might be biologically inaccurate as there is a lot more going on out there but on a higher level, this is what is going on with a neuron in our brain.

In our brains there is a massively parallel interconnected network of neurons (an average human brain has around 10^{11} neurons). Briefly what happens is that our sense organs interact with the outside world, they relay information to the lowest layer of neurons, some of these neurons may fire in response to this information and in turn relay information to other neurons they are connected to. These neurons may also fire and the process continues eventually resulting in a response.

1.1 McCulloch Pitts Neuron

The first computational model of a neuron was proposed by Warren McCulloch (neuroscientist) and Walter Pitts (logician) in 1943.



In McCulloch Pitts neuron (MP), the inputs are all binary (either 0 or 1), g is a function that aggregates the inputs and the function f takes a decision based on this aggregation following this very simple format:

$$g(x_1, x_2, \dots, x_n) = \sum_{i=1}^n x_i$$

and:

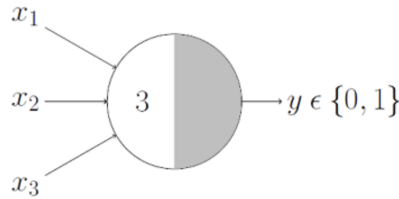
$$y = f(g(x_1, x_2, \dots, x_n)) = \begin{cases} 1, & \text{if } \sum_{i=1}^n x_i \geq \theta \\ 0, & \text{if } \sum_{i=1}^n x_i < \theta \end{cases}$$

where θ is called the “thresholding parameter” and represents the decision boundary.

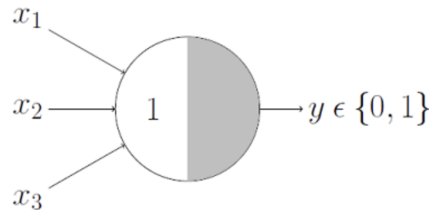
The inputs of MP can either be excitatory or inhibitory. Inhibitory inputs are those that have maximum effect on the decision making irrespective of other inputs. In other words if an inhibitory input is on then it defines the decision. Excitatory inputs are not the ones that will make the neuron fire on their own but they might fire it when combined together.

So far we have seen how the MP neuron works. Now let's look at how this very neuron can be used to represent a few boolean functions. Mind you that our inputs are all boolean and the output is also boolean so essentially, the neuron is just trying to learn a boolean function. A lot of boolean decision problems can be represented by the MP neuron, based on appropriate input variables.

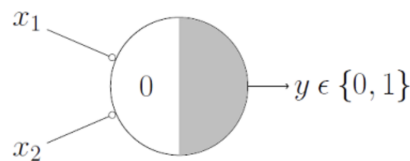
An AND function neuron would only fire when all the inputs are on i.e: $\sum_{i=1}^n x_i \geq 3$ here.



An OR function neuron would fire if any of the inputs is on i.e: $\sum_{i=1}^n x_i \geq 1$ here.



For a NOR neuron to fire, we want all the inputs to be 0 so the thresholding parameter should also be 0.

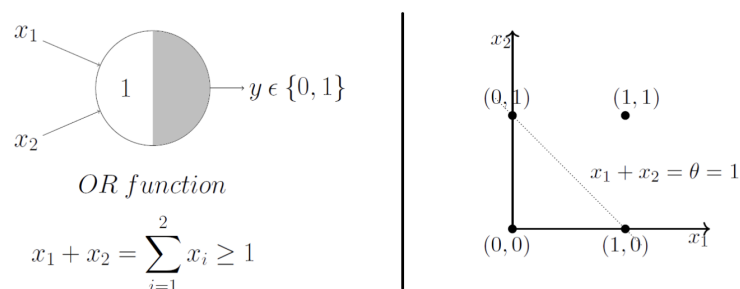


For a NOT neuron, 1 outputs 0 and 0 outputs 1. So we take the input as an inhibitory input and set the thresholding parameter to 0.



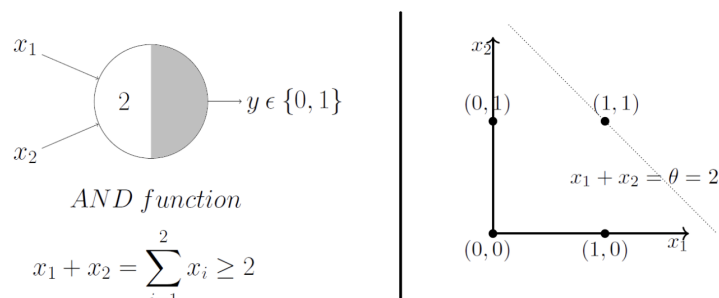
It is very useful to understand what MP neuron is doing geometrically. Let's use as an example the OR case.

We already discussed that the OR function's thresholding parameter θ is 1. The inputs are obviously boolean, so only 4 combinations are possible: (0,0), (0,1), (1,0) and (1,1). Now plotting them on a two dimensional graph and making use of the OR function's aggregation equation $x_1 + x_2 \geq 1$ using which we can draw the decision boundary as shown in the graph below.

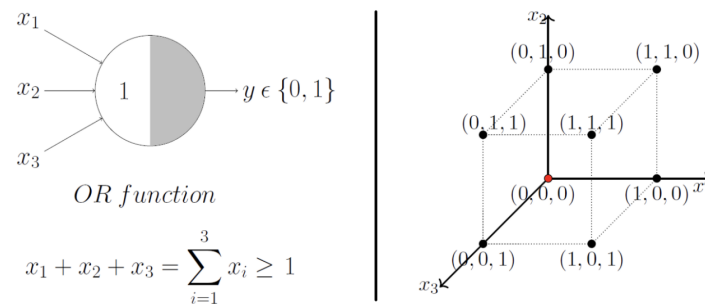


We just used the aggregation equation i.e $x_1 + x_2 = 1$ to graphically show that all those inputs whose output when passed through the OR function MP neuron lie either on or above that line (due to \geq) and all the input points that lie below that line (due to $<$) are going to output 0. This means that the MP neuron just learnt a linear decision boundary! The MP neuron is splitting the input sets into two classes: positive and negative. Positive ones (which output 1) are those that lie on or above the decision boundary and negative ones (which output 0) are those that lie below the decision boundary. Let's convince ourselves that the MP unit is doing the same also for the AND boolean function.

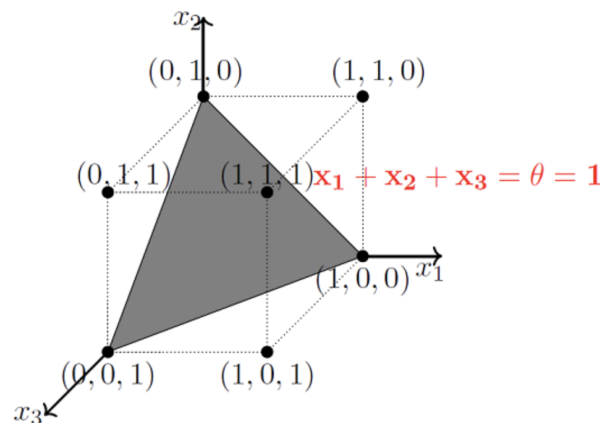
For an AND function the decision boundary equation is $x_1 + x_2 = 2$ Here, all the input points that lie on or above, just (1,1), output 1 when passed through the AND function MP neuron. It fits! The decision boundary works!



Of course this generalizes to higher dimensions. As an example let's take a look at a 3 input OR function MP unit. In this case, the possible inputs are 8 points: $(0, 0, 0)$, $(0, 0, 1)$, $(0, 1, 0)$, $(1, 0, 0)$, $(1, 0, 1)$, $(1, 1, 0)$, $(0, 1, 1)$ and $(1, 1, 1)$.



We can map these on a three dimensional graph and this time we draw a decision boundary in 3 dimensions which is of course a plane.



With a little effort we can see that all the points that lie on or above that plane (positive half space) will result in output 1 when passed through the OR function MP unit and all the points that lie below that plane (negative half space) will result in output 0.

Subsequently for even higher dimensions the decision boundary is a hyperplane.

Now that we introduced the MP neuron, an inevitable question arises: “can any boolean function be represented using the MP neuron”? The answer is no! By hand coding a thresholding parameter, MP neuron is able to conveniently represent the boolean functions which are linearly separable, i.e when there exists a line (hyperplane) such that all inputs which produce a 1 lie on one side of the line (hyperplane) and all inputs which produce a 0 lie on other side of the line (hyperplane).

We will postpone for now the “why MP fails to represent any boolean function” and we will illustrate the problem that arises with non linear separable data in the next section which is more suited.

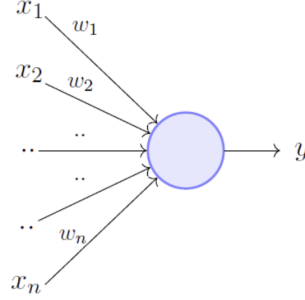
So given an MP the following questions remain unanswered:

- What about non-boolean, real inputs?
- Do we always need to hand code the threshold?
- Are all inputs equal? What if we want to assign more importance to some inputs?
- What about functions which are not linearly separable?

All of these questions will be answered gradually in the next chapters. For now we will introduce a slight advancement of MP neuron called “perceptron” which is a fundamental block of neural nets and deep learning.

1.2 Perceptron

The perceptron model, proposed by Minsky and Papert, is a more general computational model than the MP neuron. It overcomes some of the limitations of the MP neuron by introducing the concept of numerical weights (a measure of importance) for inputs, and a mechanism for learning those weights. Inputs are no longer limited to boolean values like in the case of an MP neuron since it supports real inputs as well which makes it more useful and generalized.



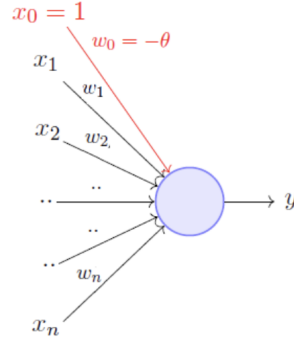
In perceptron the function g that aggregates the inputs and the function f that takes a decision based on this aggregation follow this very simple format:

$$g(x_1, x_2, \dots, x_n) = \sum_{i=1}^n w_i x_i$$

and:

$$y = f(g(x_1, x_2, \dots, x_n)) = \begin{cases} 1, & \text{if } \sum_{i=1}^n w_i x_i \geq \theta \\ 0, & \text{if } \sum_{i=1}^n w_i x_i < \theta \end{cases}$$

In perceptron it is quite common to set $w_0 = -\theta$ and $x_0 = 1$ (as we have already done in supervised learning). We call w_0 the “bias” term while the rest of w_i ’s the “weights”. We refer to the whole collection of weights and bias as the “parameters”.



Using this convention, the functions g and f can be rewritten as:

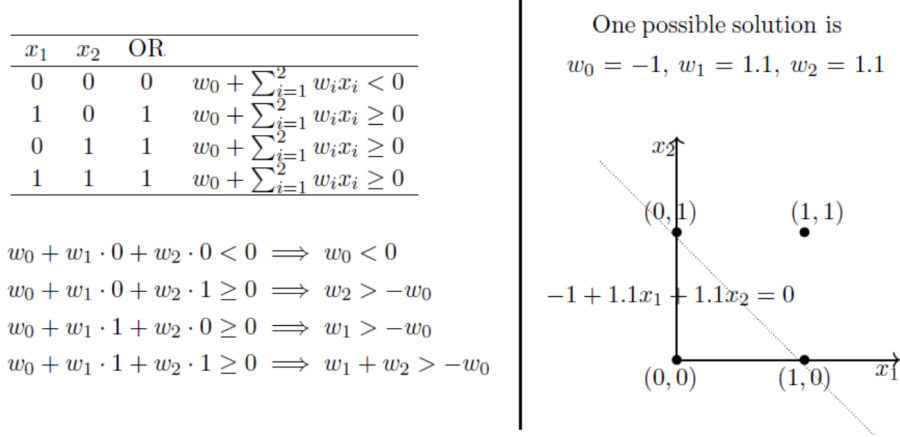
$$g(x_0, x_1, x_2, \dots, x_n) = \sum_{i=0}^n w_i x_i$$

and:

$$\begin{aligned} y = f(g(x_0, x_1, x_2, \dots, x_n)) &= \begin{cases} 1, & \text{if } \sum_{i=1}^n w_i x_i - \theta \geq 0 \\ 0, & \text{if } \sum_{i=1}^n w_i x_i - \theta < 0 \end{cases} \\ &= \begin{cases} 1, & \text{if } \sum_{i=0}^n w_i x_i \geq 0 \\ 0, & \text{if } \sum_{i=0}^n w_i x_i < 0 \end{cases} \end{aligned}$$

From the equations, it is clear that even a perceptron separates the input space into two halves, positive and negative. All the inputs that produce an output 1 lie on one side (positive half space) and all the inputs that produce an output 0 lie on the other side (negative half space). In other words, a single perceptron can only be used to implement linearly separable functions, just like the MP neuron. The difference with MP is that the weights and the bias can be learned and the inputs can be real values.

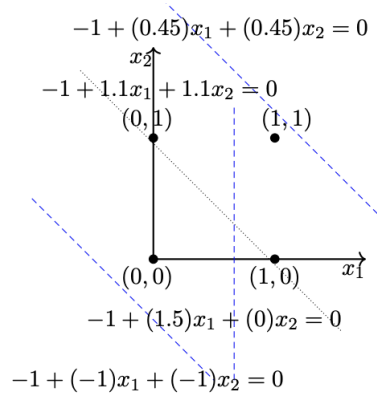
Now we will illustrate the way to obtain the weights and the bias using again an OR example.



The above “possible solution” was obtained by solving the linear system of equations on the left. It is clear that the solution separates the input space into two spaces, negative and positive half spaces. Note that we can come up with a similar set of inequalities and find the value of θ for a MP neuron also.

Note that the linear equations above, has of course multiple solutions. In general for different values of the parameters we obtain different lines that separate the points in different ways. Some of them work (i.e they separate the points correctly) and some of the don't (i.e they misclassify some of the points) hence they produce errors. Different values of the parameters produce different number of errors.

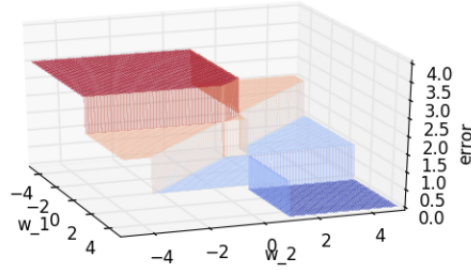
For example, let us fix the bias $w_0 = -1$ and try some random values for the weights w_1, w_2 .



We observe that the specific choices we did for the weights produce different number of errors.

w_1	w_2	errors
-1	-1	3
1.5	0	1
0.45	0.45	3

In general we are interested in those values of the parameters which result in 0 error. In the plot below we can see the error surface corresponding to different values of w_1, w_2 by keeping $w_0 = -1$ fixed. All the combinations of w_1, w_2 in the dark blue area would work for an OR function with $w_0 = -1$.

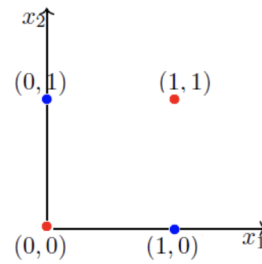


Now that we introduced the perceptron, the same inevitable question as the one we had in MP neuron arises: “can any boolean function be represented using the perceptron”? The answer is again no! Now is the time to illustrate why not!

As we said in the first chapter of the fundamental mathematics part “axiomatic set theory”, there are 16 boolean functions one can design from 2 inputs. Out of these 16, only 14 are linearly separable, and 2 of them (XOR and !XOR) are not. Let’s take as an example the XOR function and see why we cannot draw a line to separate positive inputs from the negative ones. If we try to solve the system of equation that arise from a XOR function we obtain the following.

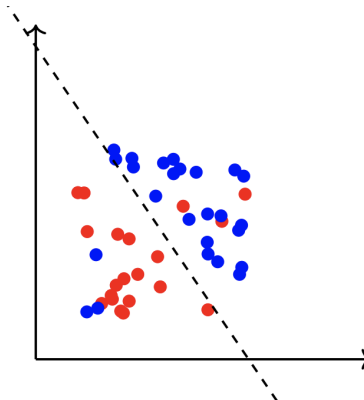
x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^2 w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^2 w_i x_i < 0$

$$\begin{aligned}
 w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 &\implies w_0 < 0 \\
 w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 &\implies w_2 > -w_0 \\
 w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 &\implies w_1 > -w_0 \\
 w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 &\implies w_1 + w_2 < -w_0
 \end{aligned}$$



Notice that the fourth equation contradicts the second and the third equation. Point is, there are no perceptron solutions for non-linearly separated data. So the key take away is that a single perceptron cannot learn to separate the data that are non-linear in nature. Hence perceptron, similarly to an MP neuron, is able to conveniently represent the boolean functions which are again linearly separable.

However it is worth mentioning the following. What does “perceptron cannot deal with data not linearly separable” mean? It means that the final result would inevitably misclassify some of the observations. In other words the final choice of parameters would produce a number of errors different than zero. The thing is that most real world data is not linearly separable and will always contain some outliers. In fact, sometimes there may not be any outliers but still the data may not be linearly separable. On top of that in most of the cases we could live with some errors. Hence, from now on, we will accept that it is hard to drive the error to 0 in most cases and will instead aim to reach the minimum possible error.



It is worth mentioning that there exists an algorithm able to find the values of the parameters which minimize the error, called simply “perceptron learning algorithm“. However since is not of use any more we will not get into details. Just for the sake completeness here is a sketch of how perceptron learning algorithm works.

Algorithm: Perceptron Learning Algorithm

```

P ← inputs with label 1;
N ← inputs with label 0;
Initialize w randomly;
while !convergence do
    Pick random x ∈ P ∪ N ;
    if x ∈ P and  $\sum_{i=0}^n w_i * x_i < 0$  then
        | w = w + x ;
    end
    if x ∈ N and  $\sum_{i=0}^n w_i * x_i \geq 0$  then
        | w = w - x ;
    end
end
//the algorithm converges when all the
inputs are classified correctly

```

One can prove that this algorithm always converges, hence it finds the parameters that minimize the error. We will skip the proof.

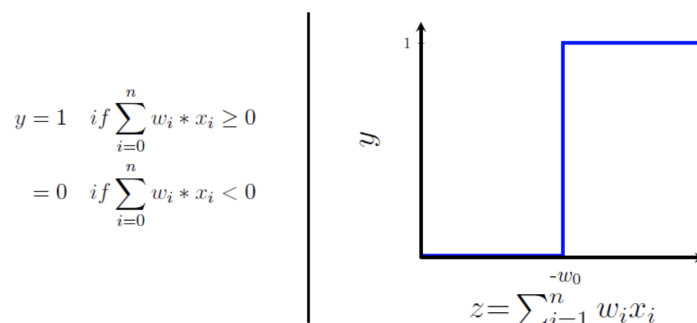
Now let's go back to the questions we posed in the previous section.

- What about non-boolean, real inputs? **Real valued inputs are allowed in perceptron!**
- Do we always need to hand code the threshold? **No, we can learn both the weights and the bias (i.e the threshold)!**
- Are all inputs equal? What if we want to assign more importance to some inputs? **A perceptron allows weights to be assigned to inputs!**
- What about functions which are not linearly separable? **Not possible with a single perceptron!**

Hence we solved all the problems but the non linearly separable data! We showed that a single perceptron cannot deal with such data, however in what follows we will show that a network of perceptrons can indeed deal with such data. Before that, we will introduce yet another advancement on the models we developed called the “sigmoid neuron”.

1.3 Sigmoid Neuron

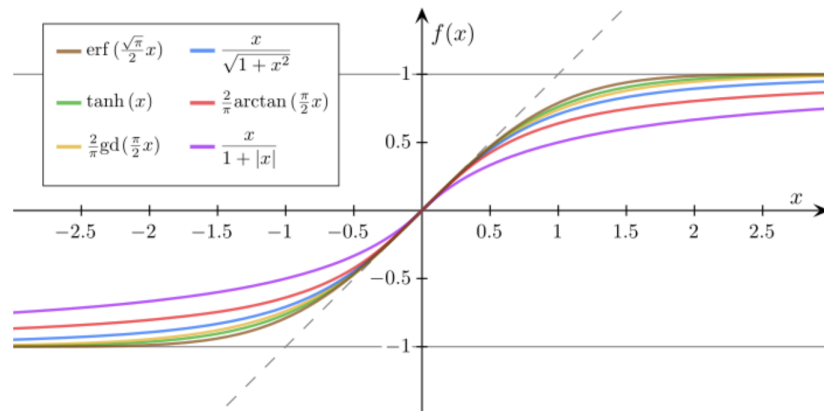
As we saw a perceptron will fire if the weighted sum of its inputs is greater than the bias w_0 . This thresholding logic used by a perceptron though is very harsh since it behaves like a step function. In other words there will always be this sudden change in the decision (from 0 to 1) when we cross the bias w_0 . For most real world applications we would expect a smoother decision function which gradually changes from 0 to 1.



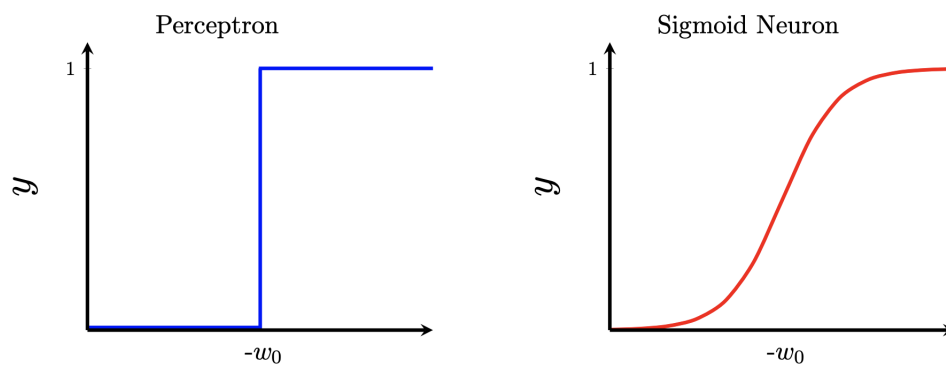
In order to fix this problem we need to introduce the so called “sigmoid functions” that we briefly mentioned in the logistic regression section of unsupervised learning chapter.

Definition 1.2 (Sigmoid Function). A **sigmoid function** is a bounded, differentiable, real function that is defined for all real input values and has a non-negative derivative at each point and exactly one inflection point.

There are many different functions that can be characterized as sigmoid function such as the logistic function, the hyperbolic tangent function, the arctangent function, and many more.



A sigmoid neurons uses as an output function a sigmoid function hence it is much smoother than a perceptron that uses the step function. As a consequence, we no longer see a sharp transition around the w_0 . Also, the output is no longer binary but a real value between 0 and 1 which can be interpreted as a probability. So instead of yes/no decision, we get the probability of yes. The output here is smooth, continuous and differentiable and just how any learning algorithm likes it.



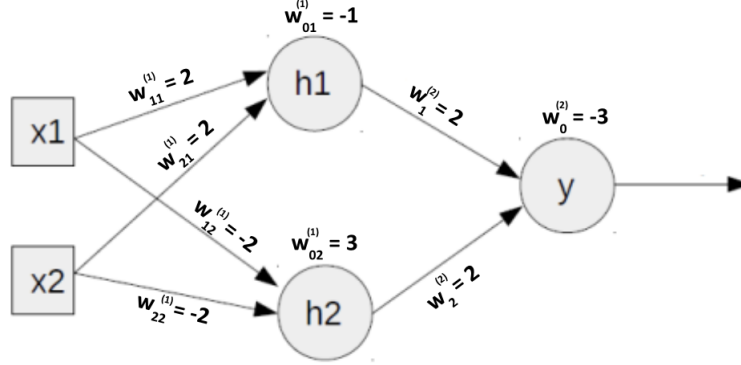
Observe that one could use the logistic function as the sigmoid function. In that case the final sigmoid neuron would be exactly the same as logistic regression. Hence we see that a sigmoid logistic neuron is just another representation of logistic regression. The advantage of this representation is that it can be generalized to a network by stacking together a lot of these neurons. As we will see in the section that follows this idea of a network of neurons can deal with the non linear separable data problem.

1.4 Feedforward Neural Networks

1.4.1 Motivation: XOR Function With A Network Of Perceptrons

In the perceptron section we showed that a single perceptron cannot deal with non linearly separable data. Now we will consider a network of perceptrons and see what can we achieve through that. We will work again with the XOR function (that we know is not linearly separable) and we will create the simplest possible network of perceptrons (see figure).

Our simple network will contain 3 layers: the layer containing the inputs x_1 and x_2 called the “input layer”, the middle layer containing the 2 perceptrons called the “hidden layer”, and the final layer containing one output neuron called the “output layer”. The terminology is not important right now, we will properly introduce the terms in the next section.



Notice that we need 10 parameters for the middle layer (8 weights and 2 biases) and 4 parameters for the output layer (2 weights and 2 biases). The notation of the variables of the parameters and the corresponding values are as denoted in the graph. The outputs of the 2 perceptrons in the hidden layer are denoted by h_1 and h_2 .

Remember that the XOR function satisfies the following results:

x_1	x_2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Remember also that for the perceptron the aggregation function g is:

$$g(x_0, x_1, x_2, \dots, x_n) = \sum_{i=0}^n w_i x_i$$

and the decision function y is the step function:

$$y = f(g(x_0, x_1, x_2, \dots, x_n)) = \begin{cases} 1, & \text{if } \sum_{i=0}^n w_i x_i \geq 0 \\ 0, & \text{if } \sum_{i=0}^n w_i x_i < 0 \end{cases}$$

Now let's see what this network predicts for the XOR function. Remember that x_0 is always equal to 1 by definition, so we will skip writing it in the calculations.

- For $(x_1 = 0, x_2 = 0)$:

$$\begin{aligned} h_1 &= f(g(x_1 = 0, x_2 = 0)) = f(w_{11}^{(1)} \cdot x_1 + w_{21}^{(1)} \cdot x_2 + w_{01}^{(1)}) = f(2 \cdot 0 + 2 \cdot 0 - 1) = f(-1) = 0 \\ h_2 &= f(g(x_1 = 0, x_2 = 0)) = f(w_{12}^{(1)} \cdot x_1 + w_{22}^{(1)} \cdot x_2 + w_{02}^{(1)}) = f(-2 \cdot 0 - 2 \cdot 0 + 3) = f(3) = 1 \\ y &= f(g(h_1, h_2)) = f(w_1^{(2)} \cdot h_1 + w_2^{(2)} \cdot h_2 + w_0^{(2)}) = f(2 \cdot 0 + 2 \cdot 1 - 3) = f(-2) = 0 \end{aligned}$$

- For $(x_1 = 1, x_2 = 0)$:

$$\begin{aligned} h_1 &= f(g(x_1 = 1, x_2 = 0)) = f(w_{11}^{(1)} \cdot x_1 + w_{21}^{(1)} \cdot x_2 + w_{01}^{(1)}) = f(2 \cdot 1 + 2 \cdot 0 - 1) = f(1) = 1 \\ h_2 &= f(g(x_1 = 1, x_2 = 0)) = f(w_{12}^{(1)} \cdot x_1 + w_{22}^{(1)} \cdot x_2 + w_{02}^{(1)}) = f(-2 \cdot 1 - 2 \cdot 0 + 3) = f(1) = 1 \\ y &= f(g(h_1, h_2)) = f(w_1^{(2)} \cdot h_1 + w_2^{(2)} \cdot h_2 + w_0^{(2)}) = f(2 \cdot 1 + 2 \cdot 1 - 3) = f(1) = 1 \end{aligned}$$

- For $(x_1 = 0, x_2 = 1)$:

$$h_1 = f(g(x_1 = 0, x_2 = 1)) = f(w_{11}^{(1)} \cdot x_1 + w_{21}^{(1)} \cdot x_2 + w_{01}^{(1)}) = f(2 \cdot 0 + 2 \cdot 1 - 1) = f(1) = 1$$

$$h_2 = f(g(x_1 = 0, x_2 = 1)) = f(w_{12}^{(1)} \cdot x_1 + w_{22}^{(1)} \cdot x_2 + w_{02}^{(1)}) = f(-2 \cdot 0 - 2 \cdot 1 + 3) = f(1) = 1$$

$$y = f(g(h_1, h_2)) = f(w_1^{(2)} \cdot h_1 + w_2^{(2)} \cdot h_2 + w_0^{(2)}) = f(2 \cdot 1 + 2 \cdot 1 - 3) = f(1) = 1$$

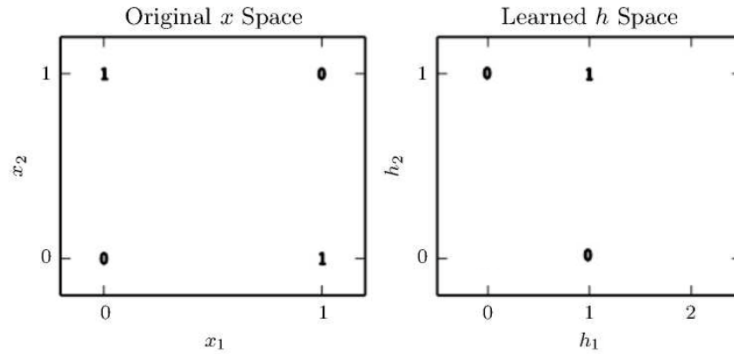
- For $(x_1 = 1, x_2 = 1)$:

$$h_1 = f(g(x_1 = 1, x_2 = 1)) = f(w_{11}^{(1)} \cdot x_1 + w_{21}^{(1)} \cdot x_2 + w_{01}^{(1)}) = f(2 \cdot 1 + 2 \cdot 1 - 1) = f(3) = 1$$

$$h_2 = f(g(x_1 = 1, x_2 = 1)) = f(w_{12}^{(1)} \cdot x_1 + w_{22}^{(1)} \cdot x_2 + w_{02}^{(1)}) = f(-2 \cdot 1 - 2 \cdot 1 + 3) = f(-1) = 0$$

$$y = f(g(h_1, h_2)) = f(w_1^{(2)} \cdot h_1 + w_2^{(2)} \cdot h_2 + w_0^{(2)}) = f(2 \cdot 1 + 2 \cdot 0 - 3) = f(-1) = 0$$

It works! The neural network has obtained the correct answer for every example in the batch. The idea behind this network is quite simple.



What we actually did here is to solve the XOR problem by learning a representation. The bold numbers printed on the plot indicate the value that the learned function must output at each point. A linear model applied directly to the original input cannot implement the XOR function. When $x_1 = 0$, the model's output must increase as x_2 increases. When $x_1 = 1$, the model's output must decrease as x_2 increases. A linear model must apply a fixed coefficient w_2 to x_2 . The linear model therefore cannot use the value of x_1 to change the coefficient on x_2 and cannot solve this problem (left part of the figure). In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem. In our example solution, the two points that must have output 1 have been collapsed into a single point in feature space. In other words, the non linear features have mapped both $(1, 0)$ and $(0, 1)$, to a single point in feature space $(1, 1)$. The linear model can now describe the function (right part of the figure).

In the example, above we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here. Instead, a gradient-based optimization algorithm can find parameters that produce very little error. The solution we described to the XOR problem is at a global minimum of the loss function, so gradient descent could converge to this point. There are other equivalent solutions to the XOR problem that gradient descent could also find. The convergence point of gradient descent depends on the initial values of the parameters. In practice, gradient descent would usually not find clean, easily understood, integer-valued solutions like the one we presented here.

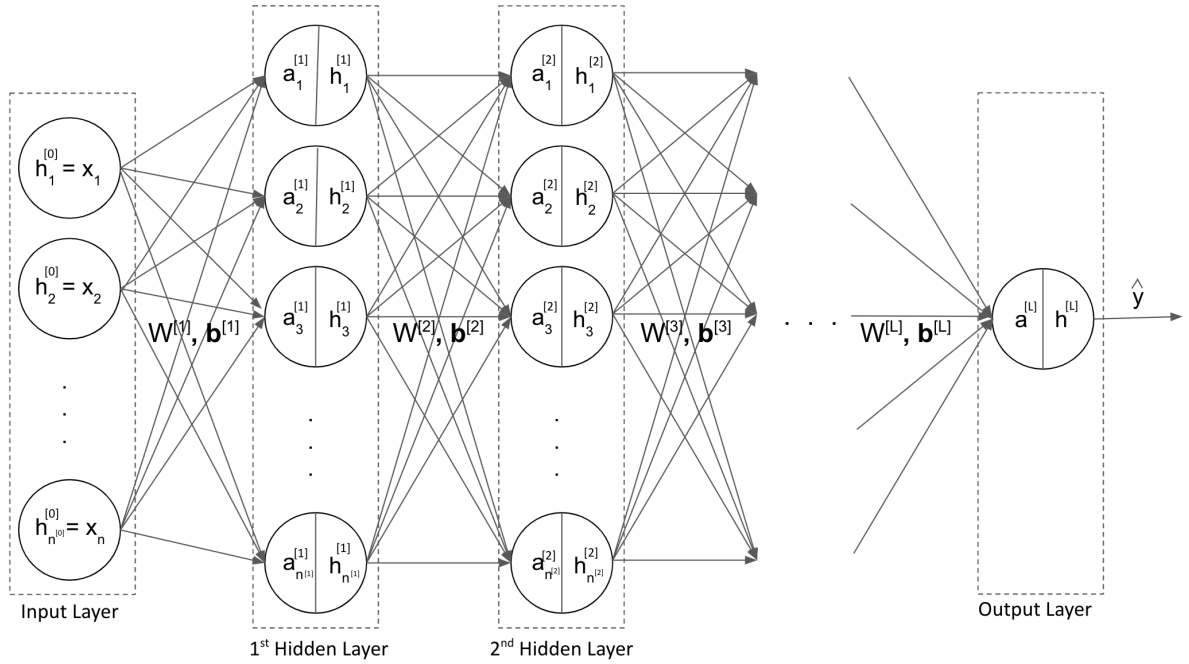
One can prove that a multilayer network of perceptrons with a single hidden layer can be used to represent any boolean function precisely (i.e with no errors). Moreover if one substitutes the perceptrons with sigmoid neurons then this multilayer network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision. In other words, there is a guarantee that for any function $f(x) : R^n \rightarrow R^m$ we can always find a neural network (with 1 hidden layer containing enough neurons) whose output $g(x)$ satisfies $|g(x) - f(x)| < \epsilon$ for some ϵ .

In the next subsection we will develop the most generic network of sigmoid neurons called the “feedforward neural network”.

1.4.2 Forward Propagation

Feedforward neural networks are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f by defining a mapping and learning the value of the parameters that result in the best function approximation. These models are called feedforward because information flows through the function being evaluated from x , through the intermediate computations and finally to the output y . There are no feedback connections in which outputs of the model are fed back into itself. Feedforward neural networks are of extreme importance to machine learning practitioners since they form the basis of many important commercial applications.

In this part we will give all the fundamental definitions of deep learning and feedforward neural networks and we will develop the most generic feedforward neural networks which is in the core of whatever will follow! Let's begin.



The input to the network is an $(n \times 1)$ dimensional feature vector \mathbf{x} :

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

From now on, as it is common in deep learning, we will separate the bias from the weights, hence there is no need for the extra $x_0 = 1$ feature and this is why it does not appear in \mathbf{x} . We will keep the notation w for the weights but we will notate the bias as b instead of w_0 . We usually refer to weights and bias as the “parameters” of the model.

We call the first layer of the network (the one with the inputs) the “input layer” or the “0th layer”. Each subsequent layer with neurons is called “1st hidden layer”, “2nd hidden layer” and so on up to the final “L hidden layer” which is usually called the “output layer”. In other words the network contains one input layer, $L - 1$ hidden layers and one output layer.

Each layer l contains a number of neurons $n^{[l]}$ (we usually use the bracket notation $[l]$ to indicate a variable for a specific layer l). It is obvious that for the input layer $n^{[0]} = n$, i.e the input layer has one neuron for each feature.

Each neuron in the hidden layers (and output layer) can be split into two parts: pre-activation and activation. For each layer l the pre-activation function a aggregates the activations from the previous layer (or the features in the case of input layer) with the weights and the biases into a $(n^{[l]} \times 1)$ dimensional

vector $\mathbf{a}^{[l]}$ as:

$$\mathbf{a}^{[l]} = a(\mathbf{h}^{[l-1]}; W^{[l]}, \mathbf{b}^{[l]}) = W^{[l]} \mathbf{h}^{[l-1]} + \mathbf{b}^{[l]}$$

where $W^{[l]}$ is a $(n^{[l]} \times n^{[l-1]})$ dimensional matrix that carries all the weights for the l layer and $\mathbf{b}^{[l]}$ is a $(n^{[l]} \times 1)$ dimensional vector that carries all the biases for the l layer, defined as:

$$W^{[l]} = \begin{bmatrix} w_{11}^{[l]} & w_{12}^{[l]} & \dots & w_{1n^{[l-1]}}^{[l]} \\ w_{21}^{[l]} & w_{22}^{[l]} & \dots & w_{2n^{[l-1]}}^{[l]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^{[l]}1}^{[l]} & w_{n^{[l]}2}^{[l]} & \dots & w_{n^{[l]}n^{[l-1]}}^{[l]} \end{bmatrix}, \quad \mathbf{b}^{[l]} = \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix}$$

We usually denote the full set of parameters of a layer l as $\theta^{[l]}$ (in other words $\theta^{[l]}$ can either be $W^{[l]}$ for the weights or $\mathbf{b}^{[l]}$ for the biases depending on the occasion). In that way we can write just one equation for both weights and biases and avoid repetitions when needed.

Finally, in each layer the activation function g (for now a sigmoid function) acts on the pre-activations and spits the activations of each neuron i.e a $(n^{[l]} \times 1)$ dimensional vector $\mathbf{h}^{[l]}$ defined as:

$$\mathbf{h}^{[l]} = g(\mathbf{a}^{[l]})$$

The activation functions define the output of the nodes given the set of inputs. They can be seen as a digital network that can be “on” (1) or “off” (0), depending on input (this is similar to the behaviour of the linear perceptron). For the activation at the input layer holds: $\mathbf{h}^{[0]} = \mathbf{x}$.

The activation function at the output layer is usually called “output activation function” and it is denoted by O instead of g . Of course its output is the prediction of the neural network:

$$\hat{y} = h^{[L]} = O(a^{[L]})$$

The role of the output activation function of the output layer is to provide some additional transformation from the features to complete the task that the network must perform. Of course the choice of the output activation function depends on the nature of the problem (e.g: linear function, logistic function, softmax function, etc). Any kind of neural network unit that may be used as an output can also be used as a hidden unit.

Notice that in the way we constructed the feedforward neural network, the final predicted value \hat{y} of the network is simply a sequence of compositions:

$$\hat{y} = O(W^{[L]} \cdot g(\dots g(W^{[3]} \cdot g(W^{[2]} \cdot \underbrace{g(W^{[1]} \mathbf{h}^{[0]} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]}) + \mathbf{b}^{[3]}) \dots) + \mathbf{b}^{[L]})$$

$\underbrace{\hspace{10em}}_{\text{output of first layer}}$
 $\underbrace{\hspace{10em}}_{\text{output of second layer}}$
 $\underbrace{\hspace{10em}}_{\text{output of third layer}}$
 \vdots
 $\underbrace{\hspace{10em}}_{\text{output of } l-1 \text{ layer}}$

Having in mind the equation above, we can at this point elaborate on why a sigmoid activation function (or any other non-linear function that we will introduce later) is essential for a feedforward neural network (hence we introduced it), and deep learning models in general.

Consider, for example, a feedforward neural network where we replace all the sigmoid activation functions g in each layer by a simple linear transformation of the form $y = mx + c$. Then the output of the first layer would simply be:

$$g(W^{[1]} \mathbf{h}^{[0]} + \mathbf{b}^{[1]}) = M^{[1]}(W^{[1]} \mathbf{h}^{[0]} + \mathbf{b}^{[1]}) + C^{[1]}$$

$$\begin{aligned}
&= M^{[1]}W^{[1]}\mathbf{h}^{[0]} + M^{[1]}\mathbf{b}^{[1]} + M^{[1]}C^{[1]} \\
&= (M^{[1]}W^{[1]})\mathbf{h}^{[0]} + (M^{[1]}\mathbf{b}^{[1]} + M^{[1]}C^{[1]}) \\
&= W'^{[1]}\mathbf{h}^{[0]} + \mathbf{b}'^{[1]}
\end{aligned}$$

By feeding this output of the first layer to the second layer we would obtain:

$$\begin{aligned}
g(W'^{[1]}\mathbf{h}^{[0]} + \mathbf{b}'^{[1]}) &= M^{[2]}(W'^{[1]}\mathbf{h}^{[0]} + \mathbf{b}'^{[1]}) + C^{[2]} \\
&= M^{[2]}W'^{[1]}\mathbf{h}^{[0]} + M^{[2]}\mathbf{b}'^{[1]} + M^{[2]}C^{[2]} \\
&= (M^{[2]}W'^{[1]})\mathbf{h}^{[0]} + M^{[2]}\mathbf{b}'^{[1]} + (M^{[2]}C^{[2]}) \\
&= W'^{[2]}\mathbf{h}^{[0]} + \mathbf{b}'^{[2]}
\end{aligned}$$

By continuing like this in the end the prediction of the network will be:

$$\hat{y} = O(W'^{[L]}\mathbf{h}^{[0]} + \mathbf{b}'^{[L]}) = O(A\mathbf{x} + B)$$

for some matrices A and B . In other words we just learn a linear transformation of the inputs hence we are constrained to learning linear decision boundaries. We cannot learn arbitrary decision boundaries. Hence, only non-linear activation functions allow such networks to compute non-trivial problems using only a small number of nodes. Actually one can prove (we will skip the proof though) that neural networks with non-linear activation functions can represent a wide variety of interesting functions when given appropriate weights. This last theorem is usually called the “universal approximation theorem”.

The process we have described so far that builds the network is usually called “forward propagation”. Notice that during building the network one needs to take several decisions regarding the architecture (usually called “topology”) of the network as for example its number of units L and its number of neurons in each unit $n^{[l]}$. We refer to these kind of parameters as “hyperparameters”. A hyperparameter is a parameter whose value is used to control the learning process in contrast to the values of the parameters of the model (weights and biases) that are derived via training. Thus, hyperparameters cannot be inferred while fitting the machine to the training set because they refer to the model selection task that in principle have no influence on the performance of the model but affect the speed and quality of the learning process. Keep in mind that L and $n^{[l]}$ are not the only hyperparameters of a network. As we move on we will meet more and more hyperparameters.

Once one pass of forward propagation is over we have both y and the predicted value \hat{y} , and one can equip a loss function $J(y, \hat{y})$ that will quantify the error of the neural network. It turns out that one of the most important aspects of the design of a neural network is actually the choice of this loss function. Fortunately, the loss functions for neural networks are more or less the same as those for other parametric models, such as linear models (e.g: MSE and cross entropy loss functions). In most cases, our parametric model defines a probability distribution and we simply use the principle of maximum likelihood as we did in supervised learning. Sometimes, we take a simpler approach, where rather than predicting a complete probability distribution over y , we merely predict some statistic of y conditioned on x . Specialized loss functions allow us to train a predictor of these estimates.

It makes sense that the choice of loss function is tightly coupled with the choice of the output activation function since the choice of how to represent the output determines the form of the loss function. For example it’s logical that a linear output activation function justifies an MSE loss function while a sigmoid (like for example softmax) output activation function justifies a cross entropy loss function.

1.4.3 Backward Propagation

We have introduced feedforward neural networks and its corresponding loss function that quantifies the errors the network makes. We are now interested in finding an algorithm for learning the parameters of this model. Normal equations are out of the picture for neural networks, so gradient descent is the way to go.

Training a neural network is not much different from training any other machine learning model with gradient descent. The largest difference between the linear models we have seen so far and neural networks is that the non-linearity of a neural network causes most interesting loss functions to become non-convex.

This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the loss function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs. Convex optimization converges starting from any initial parameters (in theory - in practice it is very robust but can encounter numerical problems). Stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters. For feedforward neural networks, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values.

For the moment, it suffices to understand that the training algorithm is almost always based on using the gradient to descend the loss function in one way or another. The specific algorithms are improvements and refinements on the ideas of gradient descent, will be explored in the next section.

Remember that in gradient descent the weights are updated as:

$$\mathbf{w} := \mathbf{w} - \alpha \nabla_{\mathbf{w}} J(\mathbf{w}, b)$$

and similarly the bias:

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

Coming to neural networks the difference is that now we have a collection of weights $W^{[l]}$ and biases $\mathbf{b}^{[l]}$ hence the update rules turn to:

$$W^{[l]} := W^{[l]} - \alpha \nabla_{W^{[l]}} J$$

and similarly for the bias:

$$\mathbf{b}^{[l]} := \mathbf{b}^{[l]} - \alpha \nabla_{\mathbf{b}^{[l]}} J$$

where J is a function of y, \hat{y} and the parameters:

$$J = J(y, \hat{y}; \theta^{[l]}) = J(y, \hat{y}; W^{[1]}, \mathbf{b}^{[1]}, W^{[2]}, \mathbf{b}^{[2]}, \dots, W^{[L]}, \mathbf{b}^{[L]})$$

Hence the problem now is to calculate the derivative of the loss function with respect to all the parameters of the network. The idea here is that we will start from the last parameters $W^{[L]}$ and biases $\mathbf{b}^{[L]}$ of the output layer, by making use of the chain rule we will compute those and gradually we will propagate backwards and by making further use of the chain rule, we will compute all the derivatives with respect to the parameters until we reach the beginning thus all the derivatives are found and we can perform the updates in gradient descent. Let's see this in practice.

For the weights:

$$\begin{aligned} \nabla_{W^{[L]}} J &= \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial W^{[L]}} \\ &= \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial h^{[L]}}{\partial a^{[L]}} \cdot \frac{\partial (W^{[L]} \mathbf{h}^{[L-1]} + \mathbf{b}^{[L]})}{\partial W^{[L]}} \\ &= \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial g(a^{[L]})}{\partial a^{[L]}} \cdot \mathbf{h}^{[L-1]^\top} \\ &= \frac{\partial J}{\partial \hat{y}} \cdot g'(a^{[L]}) \cdot \mathbf{h}^{[L-1]^\top} \\ &= \delta^{[L]} \cdot \mathbf{h}^{[L-1]^\top} \end{aligned}$$

For the biases:

$$\begin{aligned} \nabla_{\mathbf{b}^{[L]}} J &= \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a^{[L]}} \cdot \frac{\partial a^{[L]}}{\partial \mathbf{b}^{[L]}} \\ &= \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial h^{[L]}}{\partial a^{[L]}} \cdot \frac{\partial (W^{[L]} \mathbf{h}^{[L-1]} + \mathbf{b}^{[L]})}{\partial \mathbf{b}^{[L]}} \end{aligned}$$

$$\begin{aligned}
&= \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial g(a^{[L]})}{\partial a^{[L]}} \cdot \mathbb{I} \\
&= \frac{\partial J}{\partial \hat{y}} \cdot g'(a^{[L]}) \\
&= \delta^{[L]}
\end{aligned}$$

where in both cases $\delta^{[l]}$ is a $(n^{[l]} \times 1)$ dimensional vector defined as :

$$\delta^{[L]} = \frac{\partial J}{\partial \hat{y}} \cdot g'(a^{[L]})$$

In the case where $l = L \Rightarrow n^{[l]} = n^{[L]} = 1$, $\delta^{[l]}$ has dimensions $(n^{[L]} \times 1) = (1 \times 1)$ and this is why (only in this case) $\delta^{[L]}$ is not a vector but a scalar.

In a similar way, by using the chain rule we can propagate back and calculate all the derivatives. If one do that we can show that we can generalize the results as:

$$\nabla_{W^{[l]}} J = \delta^{[l]} \cdot \mathbf{h}^{[l-1]\top}$$

and:

$$\nabla_{\mathbf{b}^{[l]}} J = \delta^{[l]}$$

where each $\delta^{[l]}$ depends on the next $\delta^{[l+1]}$ (thus the need of starting from the end) through the relation:

$$\delta^{[l]} = \delta^{[l+1]} \cdot W^{[l]} \cdot g'(a^{[l]})$$

with:

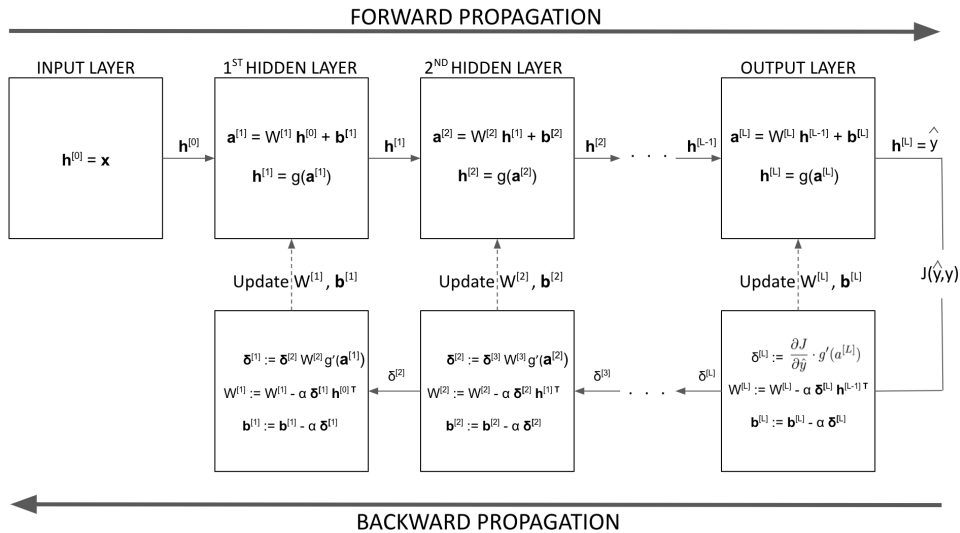
$$\delta^{[L]} = \frac{\partial J}{\partial \hat{y}} \cdot g'(a^{[L]})$$

Hence the update rules for gradient descent for the weights turn to:

$$W^{[l]} := W^{[l]} - \alpha \cdot \delta^{[l]} \cdot \mathbf{h}^{[l-1]\top}$$

The process we have described in this section is usually called “backward propagation” or more commonly “backpropagation”.

Now let’s summarize what we have so far. In the previous section we showed that using the process of forward propagation one can pick a specific choice of hyperparameters in order to define the topology of their feedforward neural network and build it. Once the network is built then we use the method of backward propagation in order to train it, i.e to learn the parameters of the model that minimize the loss. The following figure summarizes all we have developed so far on forward and backward propagation.



1.4.4 Extra: Generalize To Full Dataset

We can generalize the notation a bit more for the case that someone has m training examples instead of just one that we have seen so far:

$$\{\mathbf{x}^{(i)}, y^{(i)}\} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(m)}, y^{(m)})\}$$

Of course nothing really changes, in theory, one must pass all the training examples through the network in order to obtain the prediction, however when it comes to notation one can equip a handy way of combining together all the training examples and quantities that appear during forward propagation in vectors and matrices.

The usefulness of this subsection is more practical than theoretical, since once someone wants to use programming to actually train a model switching to vectors and matrices is inevitable.

Starting from the features one can stack all the feature vectors $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(m)}\}$ next to each other in one big $(n \times m)$ dimensional matrix X and all the targets $\{y^{(1)}, y^{(2)}, \dots, y^{(m)}\}$ next to each other into one $(1 \times m)$ dimensional row vector \mathbf{y} defined as:

$$X = [\mathbf{x}^{(1)} \quad \mathbf{x}^{(2)} \quad \dots \quad \mathbf{x}^{(m)}] = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(m)} \\ \vdots & \vdots & \ddots & \vdots \\ x_n^{(1)} & x_n^{(2)} & \dots & x_n^{(m)} \end{bmatrix}, \quad \mathbf{y} = [y^{(1)} \quad y^{(2)} \quad \dots \quad y^{(m)}]$$

Then for each layer we will have one pre-activation vector $\mathbf{a}^{[l](i)}$ per training example that we can further stack next to each other into one $(n^{[l]} \times m)$ dimensional matrix $A^{[l]}$ defined as:

$$A^{[l]} = [\mathbf{a}^{[l](1)} \quad \mathbf{a}^{[l](2)} \quad \dots \quad \mathbf{a}^{[l](m)}] = \begin{bmatrix} a_1^{[l](1)} & a_1^{[l](2)} & \dots & a_1^{[l](m)} \\ a_2^{[l](1)} & a_2^{[l](2)} & \dots & a_2^{[l](m)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n^{[l]}}^{[l](1)} & a_{n^{[l]}}^{[l](2)} & \dots & a_{n^{[l]}}^{[l](m)} \end{bmatrix}$$

where for $A^{[l]}$ holds:

$$A^{[l]} = a(H^{[l-1]}; W^{[l]}, B^{[l]}) = W^{[l]}H^{[l-1]} + B^{[l]}$$

with $H^{[l]}$ being again one $(n^{[l]} \times m)$ matrix that stacks together all the activation vectors $\mathbf{h}^{[l](i)}$ for each training example as:

$$H^{[l]} = [\mathbf{h}^{[l](1)} \quad \mathbf{h}^{[l](2)} \quad \dots \quad \mathbf{h}^{[l](m)}] = \begin{bmatrix} h_1^{[l](1)} & h_1^{[l](2)} & \dots & h_1^{[l](m)} \\ h_2^{[l](1)} & h_2^{[l](2)} & \dots & h_2^{[l](m)} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n^{[l]}}^{[l](1)} & h_{n^{[l]}}^{[l](2)} & \dots & h_{n^{[l]}}^{[l](m)} \end{bmatrix}$$

where $W^{[l]}$ is the same $(n^{[l]} \times n^{[l-1]})$ dimensional matrix of weights as before, and the matrix $B^{[l]}$ is simply a $(n^{[l]} \times m)$ matrix of biases which is created by stacking together $\mathbf{b}^{[l]}$ m times in order for the dimensions to work out:

$$W^{[l]} = \begin{bmatrix} w_{11}^{[l]} & w_{12}^{[l]} & \dots & w_{1n^{[l-1]}}^{[l]} \\ w_{21}^{[l]} & w_{22}^{[l]} & \dots & w_{2n^{[l-1]}}^{[l]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n^{[l]}1}^{[l]} & w_{n^{[l]}2}^{[l]} & \dots & w_{n^{[l]}n^{[l-1]}}^{[l]} \end{bmatrix}, \quad B^{[l]} = \underbrace{\begin{bmatrix} \mathbf{b}^{[l]} & \mathbf{b}^{[l]} & \dots & \mathbf{b}^{[l]} \end{bmatrix}}_{m \text{ times}} = \begin{bmatrix} b_1^{[l]} & b_1^{[l]} & \dots & b_1^{[l]} \\ b_2^{[l]} & b_2^{[l]} & \dots & b_2^{[l]} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n^{[l]}}^{[l]} & b_{n^{[l]}}^{[l]} & \dots & b_{n^{[l]}}^{[l]} \end{bmatrix}$$

Following this notation, the activation at the output layer now is itself a $(1 \times m)$ dimensional row vector $\hat{\mathbf{y}}$ given by:

$$\hat{\mathbf{y}} = H^{[L]} = O(A^{[L]})$$

which gives one prediction for each entry of the $(1 \times m)$ dimensional target row vector \mathbf{y} .

In exactly the same way one can define a loss function $J(\mathbf{y}, \hat{\mathbf{y}})$ that quantifies the error of the network which would be simply the average of all errors from all training examples:

$$J(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{m} \sum_{i=1}^m J(y^{(i)}, \hat{y}^{(i)})$$

where $J(y^{(i)}, \hat{y}^{(i)})$ is the loss of the training example $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$, i.e exactly what we developed in the previous section of forward and backward propagation with one single observation.

Finally one can use gradient descent for backward propagation in exactly the same way with the update rules being:

$$W^{[l]} := W^{[l]} - \alpha \cdot \Delta^{[l]} \cdot \mathbf{H}^{[l-1]\top}$$

and:

$$B^{[l]} := B^{[l]} - \alpha \cdot \Delta^{[l]}$$

where $\Delta^{[l]}$ is a $(n^{[l]} \times m)$ matrix acting as the generalization of $\delta^{[l]}$ defined as:

$$\Delta^{[l]} = \Delta^{[l+1]} \cdot W^{[l]} \cdot g'(A^{[l]})$$

with:

$$\Delta^{[L]} = \frac{\partial J}{\partial \hat{\mathbf{y}}} \cdot g'(A^{[L]})$$

1.5 Modified Gradient Descent Algorithms

Feedforward neural networks, and many other deep learning models that we will introduce after, tend to be quite heavy in calculations. Passing through the network just one single observations is not as fast as passing it from a supervised learning model. On top of that in deep learning most of the time the datasets are huge (much bigger than datasets used in supervised and unsupervised learning) with millions of observations. As a consequence, training a neural network can be tedious.

For this reasons it is always a good practice to try to make the optimization process as light as possible in order to be efficient. As a result we will introduce some modified gradient descent algorithms heavily used in deep learning.

1.5.1 Mini-Batch & Stochastic Gradient Descent

We have already introduced the concepts of mini-match & stochastic gradient descent in supervised learning. Back in that chapter we mentioned that in most supervised learning models batch gradient descent works just fine so we don't need these alternative techniques. However in deep learning models both of them are heavily used in order to train neural networks.

Let us briefly remind ourselves of the algorithms.

- **Mini-Batch Gradient Descent**

In mini-batch gradient descent we divide the whole dataset to b subsets of $\frac{m}{b}$ training examples each, called “mini-batches”, and we update the parameters using each of the mini-batches in each iteration.

- **Stochastic Gradient Descent**

In stochastic gradient descent, which can be seen as an extreme case of mini-batch gradient descent where $b = m$, we only use one training example per iteration to update the parameters. In every iteration we are estimating the total gradient based on just one single data point that we pick randomly hence the name “stochastic”. One has to keep in mind that since this is an approximation there is no guarantee that each step will decrease the loss function.

In most of the cases stochastic gradient descent (and its modifications that we will introduce in a while) is the main optimizer used in deep learning. A very useful and heavily “jargon” of deep learning is the concept of the “epoch”.

Definition 1.3 (Epoch). *An **epoch** is one full pass over the entire dataset.*

For (batch) gradient descent 1 epoch is equal to 1 parameters’ update since we use the whole dataset in order to perform the update, while for stochastic gradient descent 1 epoch is equal to m parameters’ updates (where m is the number of training examples / observations) since we update the parameters in every single pass of one training example. Finally in mini-batch gradient descent in 1 epoch happen $\frac{m}{b}$ parameters’ update.

1.5.2 Gradient Descent With Momentum

Another problem of plain gradient descent is that it takes a lot of time to navigate regions having a gentle slope due to the fact that the gradient in these regions is very small. This again makes the optimization slow. Gradient descent with momentum deals with this problem by making use of exponentially weighted moving averages.

Definition 1.4 (Exponentially Weighted Moving Average). *An exponentially weighted moving average (EWMA) is a first-order infinite impulse response filter that applies weighting factors which decrease exponentially. The weighting for each older datum decreases exponentially, never reaching zero. The EWMA S for a series Y may be calculated recursively as:*

$$S_t = \begin{cases} (1 - \beta)Y_1, & t = 1 \\ (1 - \beta)Y_t + \beta \cdot S_{t-1}, & t > 1 \end{cases}$$

where

- The coefficient β represents the degree of weighting decrease, a constant smoothing factor between 0 and 1. A higher β discounts older observations faster.
- Y_t is the value at a time period t .
- S_t is the value of the EWMA at any time period t .

By using the EWMA we define the updates $[V_{\theta^{[i]}}]_t$ we want to implement to our parameters at each step t as:

$$[V_{\theta^{[i]}}]_t = \begin{cases} (1 - \beta)\nabla_{\theta_1^{[i]}} J & t = 1 \\ (1 - \beta)\nabla_{\theta_t^{[i]}} J + \beta[V_{\theta^{[i]}}]_{t-1}, & t > 1 \end{cases}$$

Lately, in most of the cases, in gradient descent with momentum we slightly modify the above formula by discarding the $(1 - \beta)$ factor:

$$[V_{\theta^{[i]}}]_t = \begin{cases} \nabla_{\theta_1^{[i]}} J & t = 1 \\ \nabla_{\theta_t^{[i]}} J + \beta[V_{\theta^{[i]}}]_{t-1}, & t > 1 \end{cases}$$

For now we will use this second version since it's the most likely version one could find in literature. The update rule for gradient descent with momentum for the parameters turns to:

$$\theta_{t+1}^{[l]} = \theta_t^{[l]} - \alpha[V_{\theta^{[l]}}]_t$$

Notice that now in the update rule we do not use the gradient of the loss function but the update we defined earlier.

It would be easier to demonstrate what this algorithm does by calculating the first few steps for the parameters by making use of the equations we just wrote.

For the first step $t = 1$, straight from the definition we get $[V_{\theta^{[l]}}]_1 = \nabla_{\theta_1^{[l]}} J$ hence the update rule:

$$\theta_2^{[l]} = \theta_1^{[l]} - \alpha \nabla_{\theta_1^{[l]}} J$$

As we see nothing really changed, this is exactly the update rule of the gradient descent without momentum.

For the second step $t = 2$ we obtain:

$$\begin{aligned} [V_{\theta^{[l]}}]_2 &= \nabla_{\theta_2^{[l]}} J + \beta[V_{\theta^{[l]}}]_1 \\ &= \nabla_{\theta_2^{[l]}} J + \beta \nabla_{\theta_1^{[l]}} J \end{aligned}$$

Hence:

$$\theta_3^{[l]} = \theta_2^{[l]} - \alpha(\nabla_{\theta_2^{[l]}} J + \beta \nabla_{\theta_1^{[l]}} J)$$

Using the same calculations one can show that for the third update it is:

$$\theta_4^{[l]} = \theta_3^{[l]} - \alpha(\nabla_{\theta_3^{[l]}} J + \beta \nabla_{\theta_2^{[l]}} J + \beta^2 \nabla_{\theta_1^{[l]}} J)$$

and so on.

As we see in gradient descent with momentum at every step t in addition to the update $\nabla_{\theta_t^{[l]}} J$ of plain gradient descent, we also look at the history of updates with diminishing weight factors β (remember that $0 \leq \beta \leq 1$). In other words the current update $\nabla_{\theta_t^{[l]}} J$ is the one with the most weight (i.e 1), $\nabla_{\theta_{t-1}^{[l]}} J$ is trusted a bit less (i.e β), $\nabla_{\theta_{t-2}^{[l]}} J$ is trusted even lesser (β^2), and so on.

Intuitively this means that if the descent repeatedly moves in the same direction then gradient descent with momentum “adds” some confidence and starts taking bigger steps in that direction, just as a ball gains momentum while rolling down a slope. As a result even in the regions having gentle slopes, gradient descent with momentum is able to take large steps because the momentum carries it along!

Observe that in gradient descent with momentum (and all the other algorithms that will follow in this section) we need to keep a history of all the updates. Previously (in plain gradient descent) once we updated the parameters we forgot about the update and we moved on. Notation wise this is the reason why in plain gradient descent we used $:=$ for denoting the update, while now we use the steps t to denote the update at each point t . Nothing really changes in practice, but the notation indicates the fact that we actually need the history of the updates for future updates!

Of course gradient descent with momentum carries its own drawbacks. The main problem is that towards the end (i.e when we are very close to the minimum) the algorithm has gained a lot of momentum towards the right direction and it oscillates in and out of the minima valley as the momentum carries it out of the valley. This oscillations in the end slow down gradient descent with momentum a bit.

Fortunately we can deal with this issue. The final result is called “Nesterov accelerated gradient descent” and we will introduce it in the next subsection.

1.5.3 Nesterov Accelerated Gradient Descent

Nesterov accelerated gradient descent deals with the problem of oscillations of gradient descent with momentum by “looking before taking a leap”. Let's see what this means.

Recall that in gradient descent with momentum we update the parameters as:

$$\theta_{t+1}^{[l]} = \theta_t^{[l]} - \alpha[V_{\theta^{[l]}}]_t$$

By substituting $[V_{\theta^{[l]}}]_t$ and rearranging the terms we have:

$$\begin{aligned}\theta_{t+1}^{[l]} &= \theta_t^{[l]} - \alpha[V_{\theta^{[l]}}]_t \\ &= \theta_t^{[l]} - \alpha\nabla_{\theta_t^{[l]}} J - \alpha\beta[V_{\theta^{[l]}}]_{t-1} \\ &= (\theta_t^{[l]} - \alpha\beta[V_{\theta^{[l]}}]_{t-1}) - \alpha\nabla_{\theta_t^{[l]}} J \\ &= \theta'_t{}^{[l]} - \alpha\nabla_{\theta_t^{[l]}} J\end{aligned}$$

where we defined the so called “look ahead” parameters prime $\theta'_t{}^{[l]}$ as:

$$\theta'_t{}^{[l]} = \theta_t^{[l]} - \alpha\beta[V_{\theta^{[l]}}]_{t-1}$$

In Nesterov accelerated gradient descent we update the parameters by computing the derivative of the loss function with respect to those “look ahead” parameters instead of the parameters at time t . In other words the updates for the parameters turn to:

$$[V_{\theta^{[l]}}]_t = \begin{cases} \nabla_{\theta'_1{}^{[l]}} J & t = 1 \\ \nabla_{\theta'_t{}^{[l]}} J + \beta[V_{\theta^{[l]}}]_{t-1}, & t > 1 \end{cases}$$

While the update rules stay the same:

$$\theta_{t+1}^{[l]} = \theta_t^{[l]} - \alpha[V_{\theta^{[l]}}]_t$$

Looking ahead helps Nesterov accelerated gradient descent in correcting its course quicker than momentum based gradient descent, hence the oscillations are smaller and the chances of escaping the minima valley also smaller!

1.5.4 Adaptive Gradient Descent (AdaGrad)

Another important drawback of gradient descent is the following. As we proved, the updates of the weights in gradient descent follow:

$$W^{[l]} := W^{[l]} - \alpha \cdot \delta^{[l]} \cdot \mathbf{h}^{[l-1]\top}$$

and for the biases:

$$\mathbf{b}^{[l]} := \mathbf{b}^{[l]} - \alpha \cdot \delta^{[l]}$$

Recall that both $\mathbf{h}^{[l-1]}$ and $\delta^{[l]}$ are function of the inputs $\mathbf{h}^{[0]} = \mathbf{x}$ and actually it holds:

$$\mathbf{h}^{[l]}, \delta^{[l]} \sim \mathbf{x}, \quad \forall l$$

In the case where some specific feature x is very sparse (e.g if its value is 0 for most inputs) then the terms $\mathbf{h}^{[l]}$ and $\delta^{[l]}$ will be 0 for most inputs and hence the parameters that correspond to this specific feature will not get enough updates. On top of that if the feature x happens to be sparse as well as important we would want to take the updates to the parameters more seriously. For this reason we want to have a different learning rate for each parameter which takes care of the frequency of features.

Adaptive gradient descent (or AdaGrad) is a modified gradient descent algorithm with per-parameter learning rate, first published in 2011. Informally, this increases the learning rate for sparser parameters and decreases the learning rate for ones that are less sparse. This strategy often improves convergence performance over standard gradient descent in settings where data is sparse and sparse parameters are more informative.

The updates for AdaGrad are:

$$[V_{\theta^{[l]}}]_t = \begin{cases} (\nabla_{\theta_1^{[l]}} J)^2 & t = 1 \\ (\nabla_{\theta_t^{[l]}} J)^2 + [V_{\theta^{[l]}}]_{t-1}, & t > 1 \end{cases}$$

where the square term is just for getting rid of the sign of the gradient. The important part is coming from the update rule for AdaGrad which is:

$$\theta_{t+1}^{[l]} = \theta_t^{[l]} - \frac{\alpha}{\sqrt{[V_{\theta^{[l]}}]_t}} \nabla_{\theta_t^{[l]}} J$$

Notice that now in the update rule we use the gradient of the loss function and we use the update we defined for AdaGrad only in the denominator.

Using a parameter specific learning rate ensures that despite sparsity the parameters get a higher learning rates and hence larger updates. Further, it also ensures that if the parameters undergo a lot of updates its effective learning rate decreases because of the growing denominator. In practice, this does not work so well if we remove the square root from the denominator.

As usual, AdaGrad carries its own flipside. Namely, Adagrad decays the learning rate very aggressively (as the denominator grows) having as a result for the frequent parameters to start receiving very small updates because of the decayed learning rate and over time the effective learning rate for the parameters will decay to an extent that there will be no further updates to them.

To avoid this we have yet another modified gradient descent algorithm called “root mean square propagation”.

1.5.5 Root Mean Square Propagation (RMSProp)

As we already said AdaGrad decays the learning rate very aggressively (as the denominator grows) having as a result after a while the frequent parameters will start receiving very small updates because of the decayed learning rate. To avoid this we will decay the denominator and prevent its rapid growth.

Root mean square propagation (RMSProp) is also a method in which the learning rate is adapted for each of the parameters. The idea is to divide the learning rate for a weight by an exponentially weighted moving average of the magnitudes of recent gradients for that weight, hence the updates for RMSProp reads:

$$[V_{\theta^{[l]}}]_t = \begin{cases} (1 - \beta) (\nabla_{\theta_1^{[l]}} J)^2 & t = 1 \\ (1 - \beta) (\nabla_{\theta_t^{[l]}} J)^2 + \beta [V_{\theta^{[l]}}]_{t-1}, & t > 1 \end{cases}$$

while the update rule for RMSProp has the same form:

$$\theta_{t+1}^{[l]} = \theta_t^{[l]} - \frac{\alpha}{\sqrt{[V_{\theta^{[l]}}]_t}} \nabla_{\theta_t^{[l]}} J$$

1.5.6 Adaptive Moment Estimation (Adam)

Adaptive moment estimation (Adam) is one of the most used optimization techniques in deep learning and it's actually a combination of almost every algorithm that we developed earlier, hence it deals with all the problems of gradient descent we mentioned at the same time (slow updates due to learning rate, oscillations at minimum valley, sparse features, etc). In a way Adam can be seen as RMSProp optimizer with momentum thus the updates and the update rule is simply a combination of the updates and the updates rules of RMSProp and gradient descent with momentum.

Hence, more specifially, in Adam we take the updates from gradient descent with momentum (we notate them with m now to avoid confusion):

$$[m_{\theta^{[l]}}]_t = \begin{cases} (1 - \beta_m) \nabla_{\theta_1^{[l]}} J & t = 1 \\ (1 - \beta_m) \nabla_{\theta_t^{[l]}} J + \beta_m [m_{\theta^{[l]}}]_{t-1}, & t > 1 \end{cases}$$

and we rescale them by dividing with a bias correction term as:

$$[\hat{m}_{\theta^{[l]}}]_t = \frac{[m_{\theta^{[l]}}]_t}{1 - \beta_m^t}$$

Next we take the updates from RMSProp (we notate them with u now to avoid confusion):

$$[u_{\theta^{[l]}}]_t = \begin{cases} (1 - \beta_u) \left(\nabla_{\theta_1^{[l]}} J \right)^2 & t = 1 \\ (1 - \beta_u) \left(\nabla_{\theta_t^{[l]}} J \right)^2 + \beta_u [u_{\theta^{[l]}}]_{t-1}, & t > 1 \end{cases}$$

and we rescale them by dividing with a bias correction term as:

$$[\hat{u}_{\theta^{[l]}}]_t = \frac{[u_{\theta^{[l]}}]_t}{1 - \beta_u^t}$$

Now that we have the updates, we equip the update rules of gradient descent with momentum:

$$\theta_{t+1}^{[l]} = \theta_t^{[l]} - \alpha [m_{\theta^{[l]}}]_t$$

and the update rules for RMSProp:

$$\theta_{t+1}^{[l]} = \theta_t^{[l]} - \frac{\alpha}{\sqrt{[u_{\theta^{[l]}}]_t}} \nabla_{\theta_t^{[l]}} J$$

and we mix them together by making use of the rescaled updates to yield the update rule for Adam:

$$\theta_{t+1}^{[l]} = \theta_t^{[l]} - \frac{\alpha}{\sqrt{[\hat{u}_{\theta^{[l]}}]_t}} [\hat{m}_{\theta^{[l]}}]_t$$

As we said this Adam's update rules treats many problems at the same time. Adam seems to be more or less the default choice now. Having said that, many papers report that stochastic gradient descent with momentum (both the plain one or Nesterov one) with a simple annealing learning rate schedule also works well in practice. However, Adam might just be the best choice overall.

Before we close the section on Adam, let us provide an explanation on why we need to rescale the updates by dividing with those bias correction terms.

Let's start by focusing on the momentum part of Adam. Note that we are taking a running average of the gradients of $[m_{\theta^{[l]}}]_t$. The reason we are doing this is that we don't want to rely too much on the current gradient and instead rely on the overall behaviour of the gradients over many timesteps. One way of looking at this is that we are interested in the expected value of the gradients and not on a single point estimate computed at time t . However, instead of computing $E[\nabla_{\theta_t^{[l]}} J]$ we are computing the exponentially moving average $E[m_{\theta^{[l]}}]_t$. Ideally we would want to hold:

$$E[m_{\theta^{[l]}}]_t = E[\nabla_{\theta_t^{[l]}} J]$$

Let us see if that is the case. For the first step $t = 1$, straight from the definition of $[m_{\theta^{[l]}}]_t$ we get:

$$[m_{\theta^{[l]}}]_1 = (1 - \beta_m) \nabla_{\theta_1^{[l]}} J$$

For the second step $t = 2$ we obtain:

$$\begin{aligned} [m_{\theta^{[l]}}]_2 &= (1 - \beta_m) \nabla_{\theta_2^{[l]}} J + \beta_m [m_{\theta^{[l]}}]_1 \\ &= (1 - \beta_m) \nabla_{\theta_2^{[l]}} J + \beta_m (1 - \beta_m) \nabla_{\theta_1^{[l]}} J \\ &= (1 - \beta_m) \left(\nabla_{\theta_2^{[l]}} J + \beta_m \nabla_{\theta_1^{[l]}} J \right) \end{aligned}$$

For the third step $t = 3$ we obtain:

$$\begin{aligned}
[m_{\theta^{[t]}}]_3 &= (1 - \beta_m) \nabla_{\theta_3^{[t]}} J + \beta_m [m_{\theta^{[t]}}]_2 \\
&= (1 - \beta_m) \nabla_{\theta_2^{[t]}} J + \beta_m \left((1 - \beta_m) \nabla_{\theta_2^{[t]}} J + \beta_m (1 - \beta_m) \nabla_{\theta_1^{[t]}} J \right) \\
&= (1 - \beta_m) \nabla_{\theta_2^{[t]}} J + \beta_m (1 - \beta_m) \nabla_{\theta_2^{[t]}} J + \beta_m^2 (1 - \beta_m) \nabla_{\theta_1^{[t]}} J \\
&= (1 - \beta_m) \left(\nabla_{\theta_2^{[t]}} J + \beta_m \nabla_{\theta_2^{[t]}} J + \beta_m^2 \nabla_{\theta_1^{[t]}} J \right)
\end{aligned}$$

Observing the pattern we can write that for any step t :

$$[m_{\theta^{[t]}}]_t = (1 - \beta_m) \sum_{i=1}^t \beta_m^{t-i} \cdot \nabla_{\theta_i^{[t]}} J$$

Subsequently for the expected value:

$$\begin{aligned}
E[m_{\theta^{[t]}}]_t &= E \left[(1 - \beta_m) \sum_{i=1}^t \beta_m^{t-i} \cdot \nabla_{\theta_i^{[t]}} J \right] \\
&= (1 - \beta_m) E \left[\sum_{i=1}^t \beta_m^{t-i} \cdot \nabla_{\theta_i^{[t]}} J \right] \\
&= (1 - \beta_m) \sum_{i=1}^t E \left[\beta_m^{t-i} \cdot \nabla_{\theta_i^{[t]}} J \right] \\
&= (1 - \beta_m) \sum_{i=1}^t \beta_m^{t-i} \cdot E \left[\nabla_{\theta_i^{[t]}} J \right]
\end{aligned}$$

At this point we make the assumption that all $\nabla_{\theta_i^{[t]}}$ are coming from the same distribution hence $E[\nabla_{\theta_i^{[t]}}] = E[\nabla_{\theta_t^{[t]}}]$, $\forall i$. Thus:

$$\begin{aligned}
E[m_{\theta^{[t]}}]_t &= (1 - \beta_m) \sum_{i=1}^t \beta_m^{t-i} \cdot E[\nabla_{\theta_i^{[t]}} J] \\
&= (1 - \beta_m) \cdot E[\nabla_{\theta_t^{[t]}} J] \sum_{i=1}^t \beta_m^{t-i} \\
&= (1 - \beta_m) \cdot E[\nabla_{\theta_t^{[t]}} J] \cdot \frac{1 - \beta_m^t}{1 - \beta_m} \\
&= E[\nabla_{\theta_t^{[t]}} J] \cdot (1 - \beta_m^t)
\end{aligned}$$

Hence we see that $E[m_{\theta^{[t]}}]_t \neq E[\nabla_{\theta_t^{[t]}} J]$ due to this extra $(1 - \beta_m^t)$ factor. By dividing with the factor this last equation we obtain:

$$\begin{aligned}
\frac{1}{1 - \beta_m^t} E[m_{\theta^{[t]}}]_t &= E[\nabla_{\theta_t^{[t]}} J] \Rightarrow \\
E \left[\frac{[m_{\theta^{[t]}}]_t}{1 - \beta_m^t} \right] &= E[\nabla_{\theta_t^{[t]}} J] \Rightarrow \\
E[\hat{m}_{\theta^{[t]}}]_t &= E[\nabla_{\theta_t^{[t]}} J]
\end{aligned}$$

Thus, while $[m_{\theta^{[t]}}]_t$ does not satisfy the equality, the rescaled by the bias correction one $[\hat{m}_{\theta^{[t]}}]_t$ does. This is the reason for the bias correction. Similarly we do the same for the RMSProp update of Adam.

1.5.7 Learning Rate Decay

As a final way to improve the optimization of a neural network we are going to talk about “learning rate decay”. As we have already discussed when the learning rate α is too big, optimization algorithm fails to find the minimum due to very big oscillations around it while, on the other hand, if we set α to a really small value the steps will be so small that optimization will be very slow.

The best of 2 worlds is if we start with a large learning rate (so we move fast towards the minimum) and then as epochs pass we gradually decrease learning rate so we are able to obtain the minimum. This is the idea of “learning rate decay”.

There are many different learning rate schedules but the most common are the following:

- **Time-based learning schedules** alter the learning rate depending on the learning rate of the previous time iteration. Factoring in the decay the mathematical formula for the learning rate is:

$$\alpha_{t+1} = \frac{\alpha_t}{1 + d_t}$$

- **Step-based learning schedules** changes the learning rate according to some pre defined steps. The decay application formula is here defined as:

$$\alpha_t = \alpha_0 \cdot d^{\text{floor}(\frac{1+t}{r})}$$

- **Exponential learning schedules** are similar to step-based but instead of steps a decreasing exponential function is used. The mathematical formula for factoring in the decay is:

$$\alpha_t = \alpha_0 \cdot e^{-d \cdot t}$$

where t is the iteration step, α_t is the learning rate at iteration t , α_0 is the initial learning rate, d_t is the decay parameter at time t (i.e how much the learning rate should change at each drop) and r corresponds to the droprate, or how often the rate should be dropped (i.e drop every r iterations).

Notice that decay parameter is yet another hyperparameter that needs to be fixed by us.

1.6 Errors, Evaluation & Regularization

Notice that despite its complexity a feedforward neural network is simply just another supervised learning model as the ones we developed in the supervised learning chapter. Everything we developed in the supervised learning chapter around errors, errors analysis, train/cross-validation/test split, bias variance trade-off and underfitting/overfitting hold true also for feedforward neural network, and all the other neural networks that we will develop.

One of the things that slightly change in neural networks is the way to deal with high bias (underfitting) and high variance (overfitting). In contrast to what we said for supervised learning models, for neural networks here are some of the techniques that we follow:

- **For high bias (underfitting)**
 - Increase network size (layers and node).
 - Train the network for longer time.
 - Use different network topology (architecture).
- **For high variance (overfitting)**
 - Find more training examples.
 - Use different network topology (architecture).
 - Regularization.

In what follows we will show some of the regularization techniques used in neural networks.

1.6.1 L2 Regularization

L2 regularization works in the exact same way as we described in supervised learning with some small adjustments in order to fit the more complex nature of a neural network. Let's do a small review of L2 regularization and introduce these adjustments.

As we said in neural networks the loss function J depends on the weights and biases:

$$J = J(y, \hat{y}; \theta^{[l]}) = J(y, \hat{y}; W^{[1]}, \mathbf{b}^{[1]}, W^{[2]}, \mathbf{b}^{[2]}, \dots, W^{[L]}, \mathbf{b}^{[L]})$$

Recall from the theory of L2 regularization that we developed, the reason of overfitting is that the parameters appear in the loss function are free to get any value. With regularization we penalize the parameters by imposing an extra constraint on $W^{[l]}$ of each layer, and by making use of the theory of constrained optimization of Appendix ?? the Lagrangian for L2 regularization for neural networks turns to:

$$\mathcal{L}(y, \hat{y}; \theta^{[l]}) = J(y, \hat{y}; \theta^{[l]}) + \frac{\lambda}{2m} \sum_{l=1}^L \|W^{[l]}\|_F^2$$

where $\|W^{[l]}\|_F^2$ is the so called "Frobenius Norm" given by:

$$\|W^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (W_{ij}^{[l]})^2$$

At this point we can consider the Lagrangian \mathcal{L} as a "new" loss function J_{new} that we need to optimize hence we can completely forget about regularization and solve this problem with gradient descent where in order to update the weights we use:

$$W^{[l]} := W^{[l]} - \alpha \nabla_{W^{[l]}} J_{new}$$

and then we follow backward propagation for iterations.

Notice that we only regularize the weights and not the biases. Since overfitting usually requires the output of the model to be sensitive to small changes in the input data (i.e to exactly interpolate the target values, you tend to need a lot of curvature in the fitted function), the bias parameters don't contribute to the curvature of the model, so there is usually little point in regularising them as well.

One can prove that L2 regularization is simply adding some Gaussian noise to the inputs. We will skip the proof for now since it is quite simple and quite unnecessary at the same time.

1.6.2 Dropout

Dropout (also called dilution) is yet another, heavily used, regularization method that approximates training a large number of neural networks with different architectures in parallel. During training, some number of layer outputs are randomly ignored or "dropped out", hence the naming. This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different view of the configured layer.

Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs. This conceptualization suggests that perhaps dropout breaks-up situations where network layers co-adapt to correct mistakes from prior layers, in turn making the model more robust.

Dropout simulates a sparse activation from a given layer, which interestingly, in turn, encourages the network to actually learn a sparse representation as a side-effect. As such, it may be used as an alternative to activity regularization for encouraging sparse representations in autoencoder models.

Because the outputs of a layer under dropout are randomly subsampled, it has the effect of reducing the capacity or thinning the network during training. As such, a wider network, e.g more nodes, may be required when using dropout.

1.6.3 Dataset Augmentation

Dataset augmentation is a very simple technique based on the idea “more data = better learning”, where it applies transformations to the training examples in order to create more for them. The transformations can be as simple as flipping an image, or as complicated as applying neural style transfer. The idea is that by changing the makeup of the data, one can improve the performance of the neural network by increasing the training set size. Dataset augmentation works well for image classification / object recognition tasks and it has also been shown to work well for speech. However, for some tasks it may not be clear how to generate such data.

1.6.4 Early Stopping

We have already introduced early stopping as a regularization method back in supervised learning chapter. Let us briefly remind ourselves of it.

In general as the neural network learns, its prediction error on the training set goes down, along with its prediction error on the validation set. After a while though, the validation error stops decreasing and starts to go back up. This indicates that the model has started to overfit the training data. Early stopping just stops training as soon as the validation error reaches the minimum.

Early stopping is more frequently used in deep learning than usual, simpler, supervised learning models.

1.7 Improvements On A Feedforward Neural Network

In this section we will introduce some of the latest advancements in deep learning that can improve a feedforward neural network.

1.7.1 Activation Functions

Up to this point we have considered as activation functions only functions from the broad family of sigmoid functions. Recall, for example, that in the perceptron section we said that one could use the logistic function as the sigmoid function playing the part of activation function which would lead to logistic regression. Indeed, a logistic function used to be the most widely used sigmoid function. However it carries three huge drawbacks.

Starting with the first one, recall that the updates of the weights in gradient descent follow:

$$W^{[l]} := W^{[l]} - \alpha \cdot \delta^{[l]} \cdot \mathbf{h}^{[l-1]\top}$$

and similarly the biases:

$$\mathbf{b}^{[l]} := \mathbf{b}^{[l]} - \alpha \cdot \delta^{[l]}$$

where:

$$\delta^{[l]} = \delta^{[l+1]} \cdot W^{[l]} \cdot g'(\mathbf{a}^{[l]})$$

Hence the derivative of the activation function is also part of the update rule of the parameters. Since the logistic function is of the form:

$$S(x) = \frac{1}{1 + e^{-x}}$$

its derivative is:

$$S'(x) = -\frac{(-e^{-x})}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} \cdot \frac{e^{-x}}{1 + e^{-x}} = \frac{1}{1 + e^{-x}} \cdot \left(1 - \frac{1}{1 + e^{-x}}\right) = S(x) \cdot (1 - S(x))$$

Hence $\delta^{[l]}$ can be written as:

$$\delta^{[l]} = \delta^{[l+1]} \cdot W^{[l]} \cdot g(\mathbf{a}^{[l]}) \cdot (1 - g(\mathbf{a}^{[l]}))$$

Definition 1.5 (Saturated Neuron). A **saturated neuron** is a neuron where its outputs values are close to the asymptotic ends of the bounded activation function.

In the case of the logistic function a saturate neuron would have:

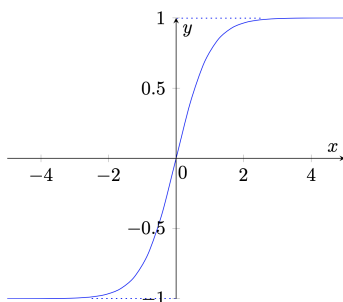
$$g(\mathbf{a}^{[l]}) = 0 \quad \text{or} \quad g(\mathbf{a}^{[l]}) = 1$$

The problem with logistic function is similar to the problem we described in AdaGrad where we said that since $\mathbf{h}^{[l]}, \delta^{[l]} \sim \mathbf{x}$, $\forall l$, in the case where some specific feature x is very sparse (e.g if its value is 0 for most inputs) then the terms $\mathbf{h}^{[l]}$ and $\delta^{[l]}$ will be 0 for most inputs and hence the parameters that correspond to this specific feature will not get enough updates. Exactly the same behaviour one has for the case of saturated neurons where the update terms get to zero and the update of the parameters do not happen any more. In other words saturated neurons cause the gradient to vanish.

The second problem with logistic function is that it is not zero centered. What this means is that since the output of a logistic function is always positive (since logistic function compresses all its inputs to the range $[0, 1]$), essentially, either all the gradients at a layer are positive or all the gradients at a layer are negative having as a result the restriction of the possible update directions.

Finally the third and last problem with logistic function is that it is computationally expensive due to the exponential.

As an alternative to the logistic function, it was proposed the hyperbolic tangent function \tanh which is also part of the sigmoid family.



Hyperbolic tangent function solves the problem of the logistic function not being zero centered since it compresses all its inputs to the range $[-1, 1]$ hence it is zero centered. However its gradient still vanishes at saturation since:

$$(\tanh(x))' = (1 - \tanh^2(x))$$

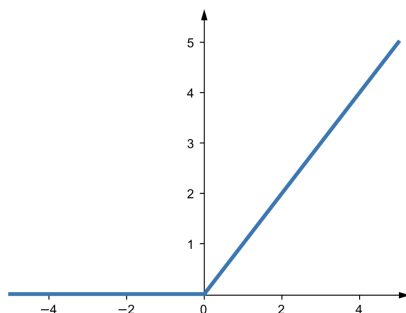
Finally it is also computationally expensive. Despite all its problems hyperbolic tangent function is still used in LSTM's and RNN's model that we will introduce later.

In general all sigmoid functions, due to its shapes, will have the problems of saturation. Thus there was a need for some alternatives to sigmoid functions.

Definition 1.6. *Rectified Linear Unit (ReLU)* The **rectified linear unit (ReLU)** is an activation function defined as the positive part of its argument:

$$f(x) = \max(0, x)$$

where x is the input to a neuron.



ReLU activation function was first introduced to a dynamical network with strong biological motivations and mathematical justifications. It was demonstrated for the first time in 2011 to enable better training of deeper networks, compared to the widely used logistic function and hyperbolic tangent. ReLU is, as of 2017, the most popular activation function for deep neural networks and it is more or less the standard unit for convolutional neural networks that we will introduce later.

The advantages of ReLU is that it does not saturate in the positive region, it is computationally efficient and it converges much faster than the sigmoid functions we have mentioned. In practice one should keep in mind that a large fraction of ReLU units can die if the learning rate is set too high so it is advised to initialize the bias to a positive value.

Finally it is worth mentioning two variation of ReLU.

Definition 1.7. *Leaky Rectified Linear Unit (Leaky ReLU)* The **leaky rectified linear unit** (leaky ReLU) is an activation function defined as:

$$f(x) = \max(0.01x, x)$$

Definition 1.8. *Parametric Rectified Linear Unit (Leaky ReLU)* The **parametric rectified linear unit** (parametric ReLU) is an activation function defined as:

$$f(x) = \max(\alpha x, x)$$

where α is a parameter of the model that gets updated during backpropagation.

1.7.2 Initialization Strategies

1.7.3 Batch Normalization