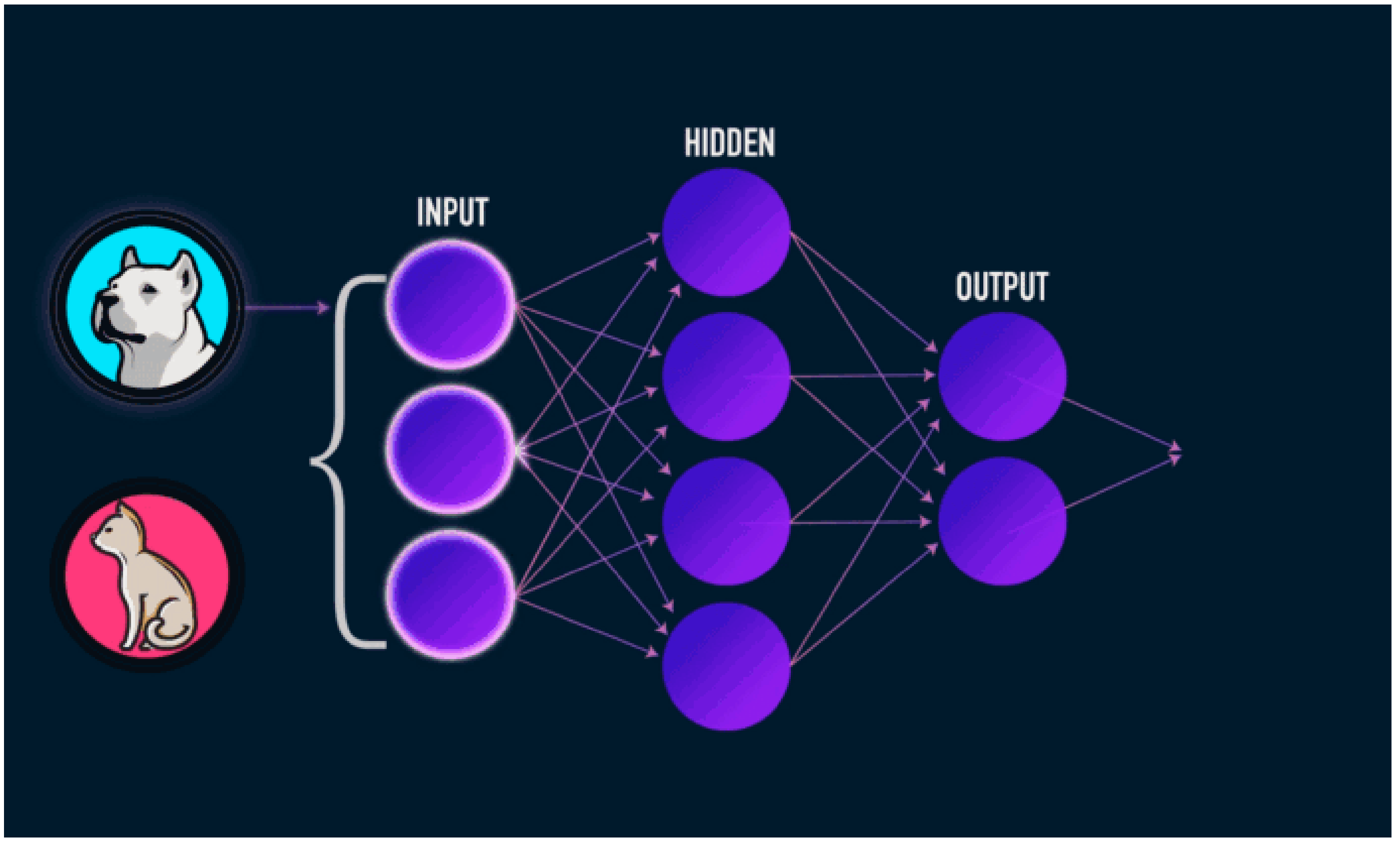
Everything you need to know about Neural Networks and Backpropagation — Machine Learning Easy and Fun

Neural Network explanation from the ground including understanding the math behind it



I find it hard to get step by step and detailed explanations about Neural Networks in one place. Always some part of the explanation was missing in courses or in the videos. So I tried to gather all the information and explanations in one blog post (step by step). I would separate this blog in 8 sections as I find it most relevant.

1. Model Representation
2. Model Representation Mathematics
3. Activation Functions
4. Bias Node

5. Cost Function
6. Forward Propagation Calculation
7. Backpropagation Algorithm
8. Code Implementation

So let's start…

Model Representation

Artificial Neural Network is computing system inspired by biological neural network that constitute animal brain. Such systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules.

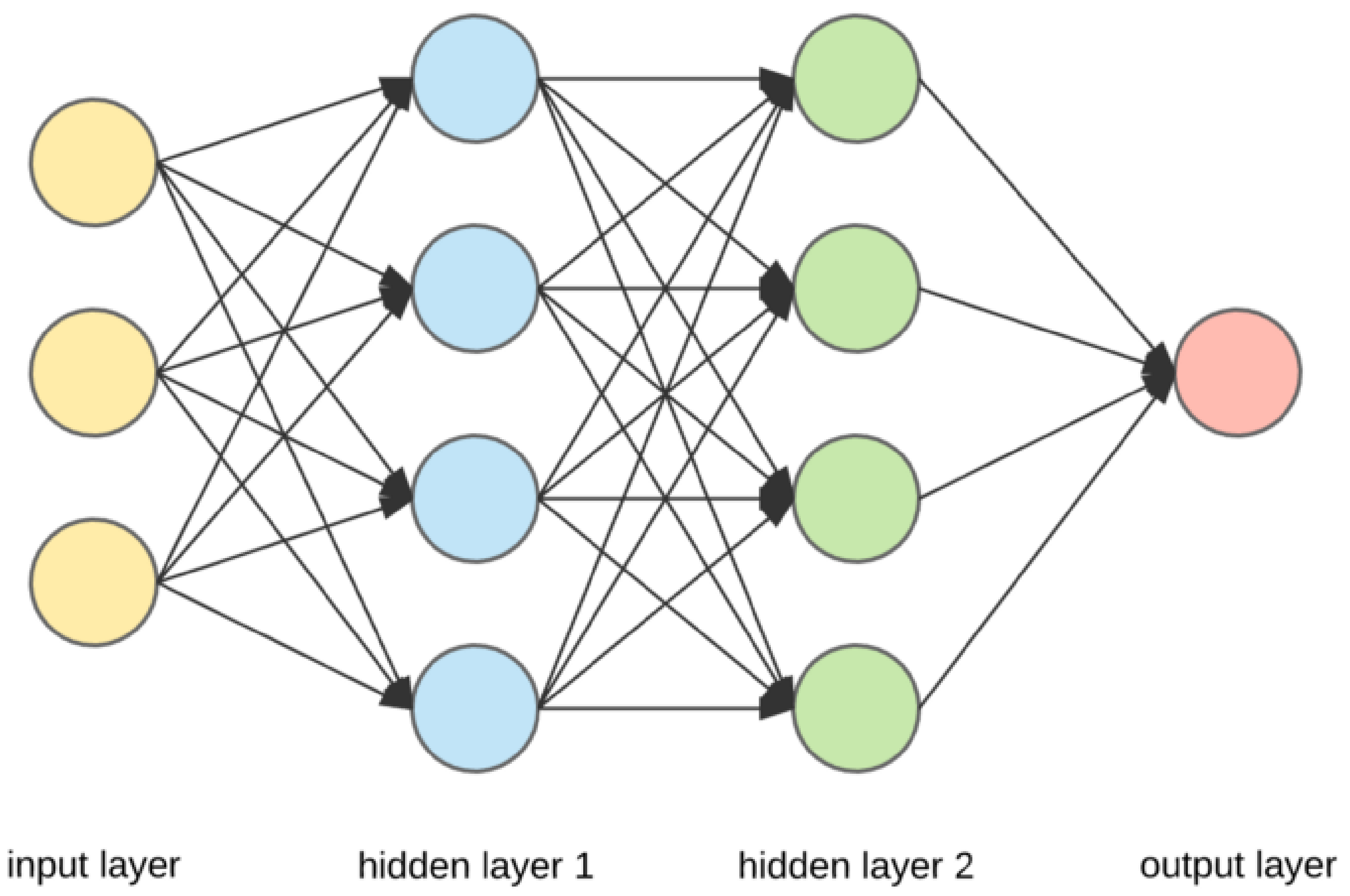


Image 1: Neural Network Architecture

The Neural Network is constructed from 3 type of layers:

1. Input layer — initial data for the neural network.

2. Hidden layers — intermediate layer between input and output layer and place where all the computation is done.
3. Output layer — produce the result for given inputs.

There are 3 yellow circles on the image above. They represent the input layer and usually are noted as vector ***X***. There are 4 blue and 4 green circles that represent the hidden layers. These circles represent the "activation" nodes and usually are noted as ***W*** or ***θ***. The red circle is the output layer or the predicted value (or values in case of multiple output classes/types).

Each node is connected with each node from the next layer and each connection (black arrow) has particular weight. Weight can be seen as impact that that node has on the node from the next layer. So if we take a look on one node it would look like this

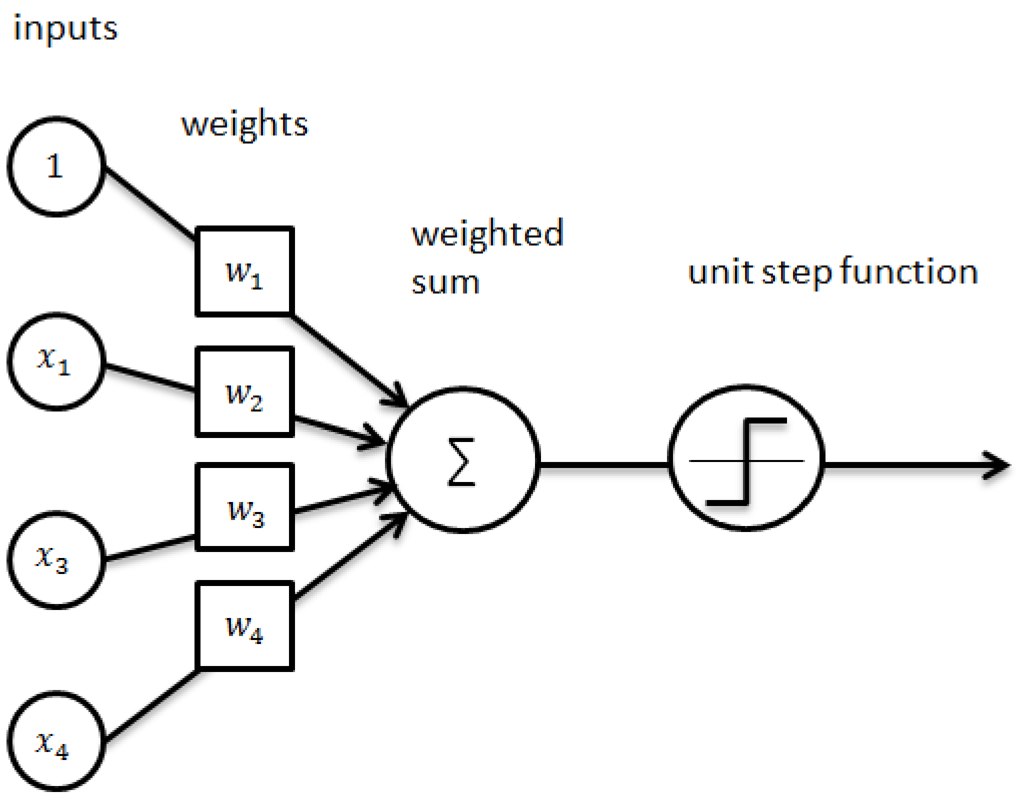


Image 2: Node from Neural Network

Let's look at the top blue node (*"Image 1"*). All the nodes from the previous layer (yellow) are connected with it. All these connections represent the weights (impact). When all the node values from the yellow layer are multiplied with their weight and all this is summarized it gives some value for the top blue node. The blue node has predefined "activation" function (*unit step function* on *"Image 2"*) which defines if this node will be "activated" or how "active" it will be, based on the summarized value. The additional node with value 1 is called "bias" node.

Model Representation Mathematics

In order to understand the mathematical equations I will use a simpler Neural Network model. This model will have 4 input nodes (3 + 1 "bias"). One hidden layer with 4 nodes (3 + 1 "bias") and one output node.

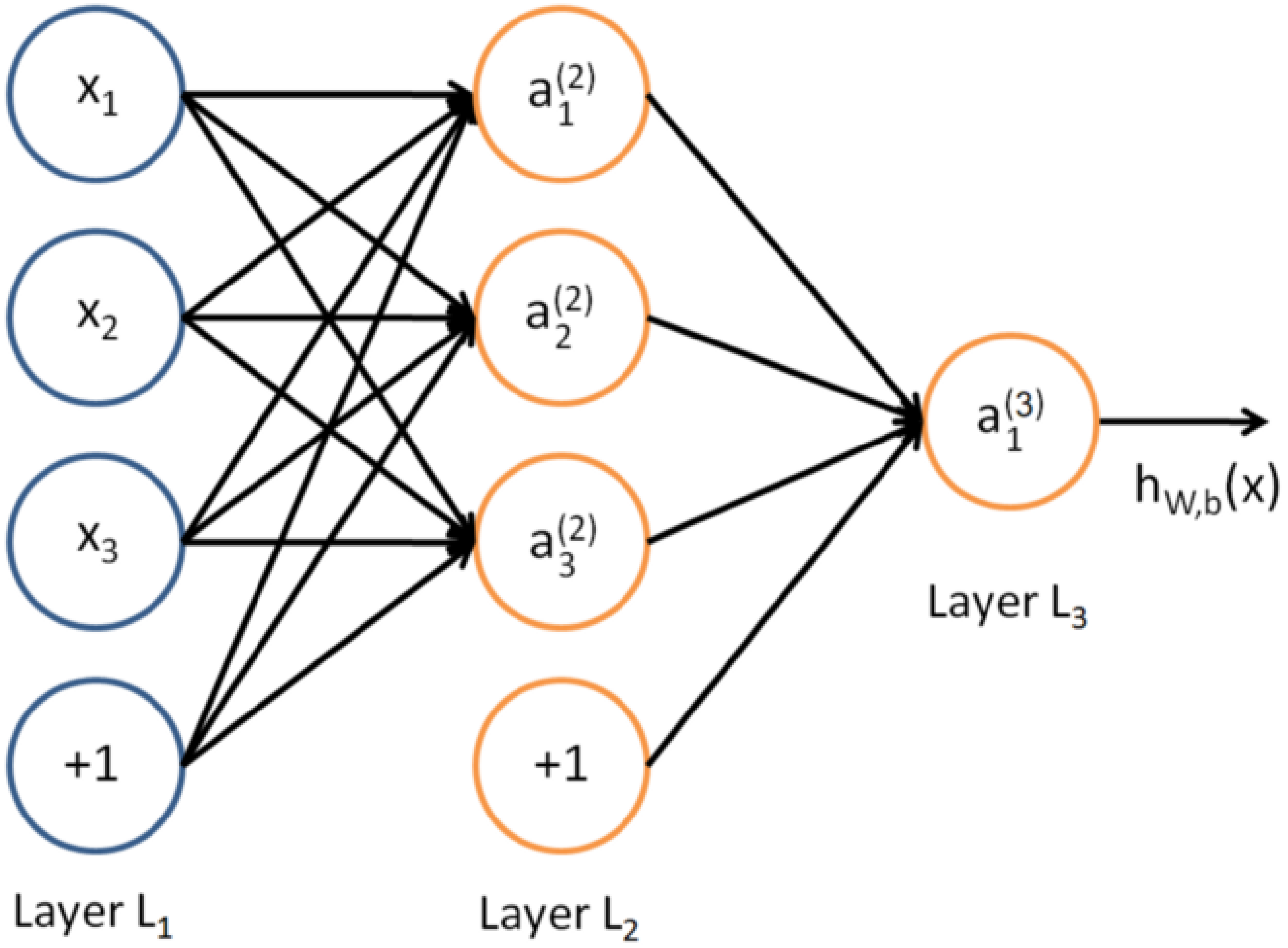


Image 3: Simple Neural Network

We are going to mark the "bias" nodes as xo and ao respectively. So, the input nodes can be placed in one vector X and the nodes from the hidden layer in vector A.

$$X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad A = \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix}$$

Image 4: X (input layer) and A (hidden layer) vector

The weights (arrows) are usually noted as θ or *W.* In this case I will note them as θ. The weights between the input and hidden layer will represent *3x4* matrix. And the weights between the hidden layer and the output layer will represent *1x4* matrix.

If network has ***a*** *units in layer* j *and* ***b*** *units in layer* j+1*, then* θ_j *will be of dimension* ***b*** $\times$***(a+1)***.

$$\theta^{(1)} = \begin{bmatrix} \theta_{10} & \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{20} & \theta_{21} & \theta_{22} & \theta_{23} \\ \theta_{30} & \theta_{31} & \theta_{32} & \theta_{33} \end{bmatrix}$$

Image 5: Layer 1 Weights Matrix (θ)

Next, what we want is to compute the "activation" nodes for the hidden layer. In order to do that we need to multiply the input vector ***X*** and weights matrix $\boldsymbol{\theta^1}$ for the first layer ***(X****$\boldsymbol{\theta^1}$***)***and then apply the activation function ***g***. What we get is :

$$
\begin{aligned}
a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\
a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\
a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)
\end{aligned}
$$

Image 6: Compute activation nodes

And by multiplying hidden layer vector with weights matrix $\boldsymbol{\theta}$ for the second layer($\boldsymbol{A}$*$\boldsymbol{\theta}$) we get output for the hypothesis function:

$$
h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})
$$

Image 7: Compute output node value (hypothesis)

This example is with only one hidden layer and 4 nodes there. If we try to generalize for Neural Network with multiple hidden layers and multiple nodes in each of the layers we would get next formula.

$$
\mathrm{a}_n^L = \left[\sigma\left(\sum_m \theta_{nm}^L \left[\cdots\left[\sigma\left(\sum_j \theta_{kj}^2 \left[\sigma\left(\sum_i \theta_{ji}^1 x_i + b_j^1\right)\right] + b_k^2\right)\right]\cdots\right]_m + b_n^L\right)\right]_n
$$

Image 8: Generalized Compute node value function

Where we have *L* layers with *n* nodes and *L-1* layer with *m* nodes.

Activation Functions

In Neural Network the activation function defines if given node should be "activated" or not based on the weighted sum. Let's define this weighted sum value as $\boldsymbol{z}$. In this section I would explain why "Step Function" and "Linear Function" won't work and talk about "*Sigmoid Function*" one of the most popular activation functions. There are also other functions which I will leave aside for now.

Step Function

One of the first ideas would be to use so called "*Step Function*" (discrete output values) where we define threshold value and:

if(z > threshold) — "activate" the node (value 1)
if(z < threshold) — don't "activate" the node (value 0)

This looks nice but it has drawback since the node can only have value 1 or 0 as output. In case when we would want to map multiple output classes (nodes) we got a problem. The problem is that it is possible multiple output classes/nodes to be activated (to have the value 1). So we are not able to properly classify/decide.

Linear Function

Another possibility would be to define "*Linear Function*" and get a range of output values.

$$y = ax$$

However using only linear function in the Neural Network would cause the output layer to be linear function, so we are not able to map any ***non-linear*** data. The proof for this is given by:

$$f(x) = x + 3$$

$$g(x) = 2x + 5$$

then by function composition we get

$$g(f(x)) = 2(x + 3) + 5 = 2x + 11$$

which is also a linear function.

Sigmoid Function

It is one of the most widely used activation function today. It equation is given with the formula below.

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

Image 9: Sigmoid Equation. source: wikipedia

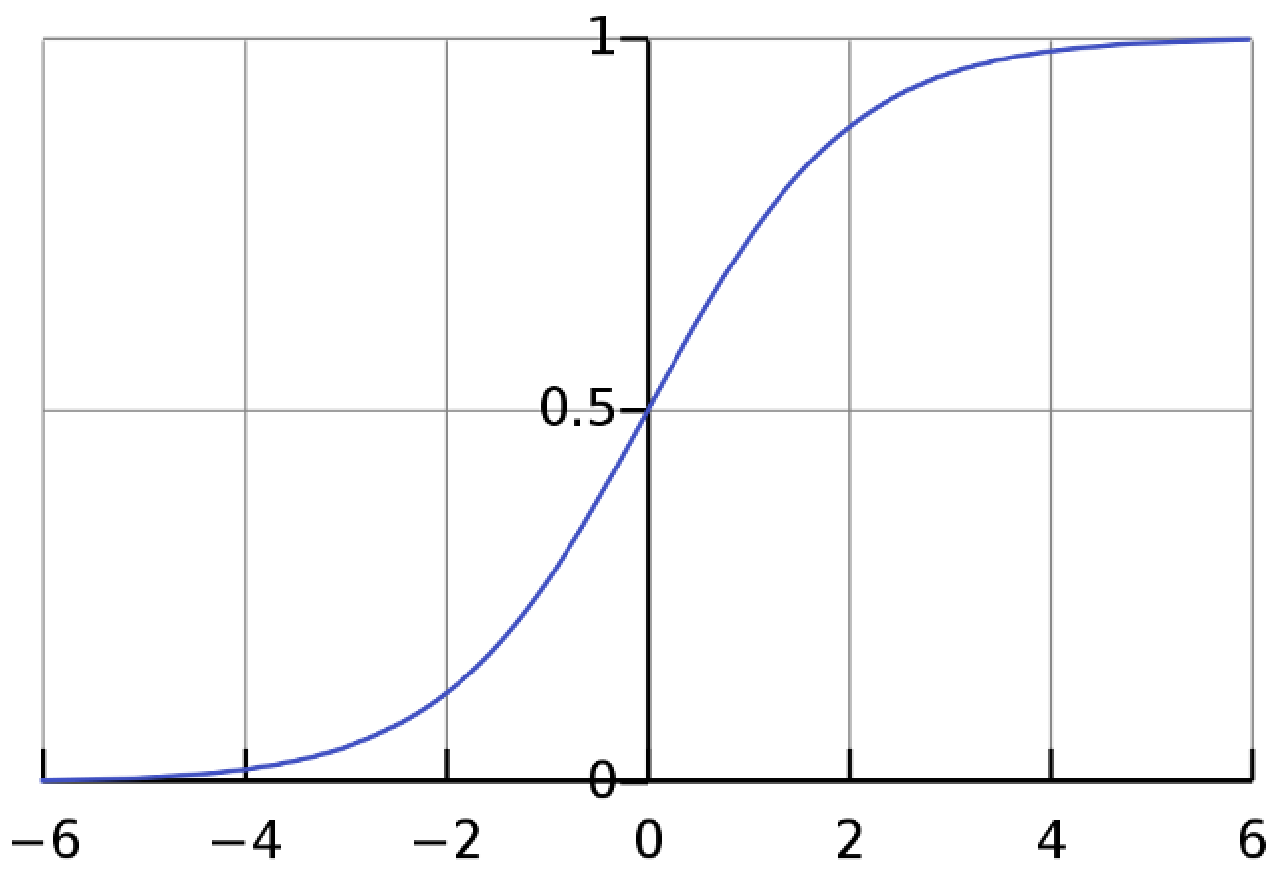


Image 10: Sigmoid Function. source: wikipedia

It has multiple properties which makes it so popular:

- It's non-linear function
- Range values are between (0,1)
- Between (-2,2) on x-axis the function is very steep, that cause function to tend to classify values ether 1 or 0

Because of this properties it allows the nodes to take any values between 0 and 1. In the end, in case of multiple output classes, this would result with different probabilities of *"activation"* for each output class. And we will choose the one with the highest "activation"(probability) value.

Bias Node

Using "bias" node is usually critical for creating successful learning model. In short, ***a bias value allows to shift the activation function to the left or right*** and it helps getting **better fit for the data** (better prediction function as output).

Below there are 3 Sigmoid functions that I draw where you can notice how multiplication/add/subtract the variable x by some value can influence the function.

- Multiplying $\boldsymbol{x}$ — makes the function steeper
- Add/Subtract $\boldsymbol{x}$ — shift the function left/right

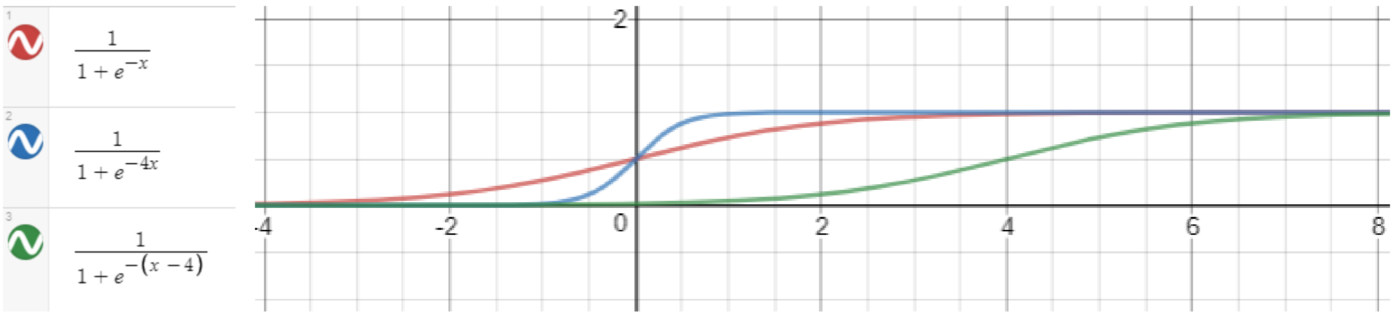


Image 11: Sigmoid Functions. source: desmos.com

Cost Function

Let’s start with defining the general equation for the cost function. This function represent the sum of the error, difference between the predicted value and the real (labeled) value.

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

Image 12: General Cost functoin. source: coursera.org

Since this is type of a classification problem $\boldsymbol{y}$ can only take discrete values {0,1}. It can only be in one type of class. For example if we classify images of dogs (class 1), cats (class 2) and birds (class 3). If the input image is dog. The output classes will be value 1 for dog class and value 0 for the other classes.

This means that we want our hypothesis to satisfy

$$0 \leq h_\theta(x) \leq 1$$

Image 13: Hypothesis function range values

So that's why we will define our hypothesis as

$$h_\theta(x) = g(\theta^T x)$$

Image 14: Hypothesis function

Where g in this case will be Sigmoid function, since this function has range values between (0,1).

Our goal is to optimize the cost function so we need to find min ***J(θ)***. But Sigmoid function is a "non-convex" function ("*Image 15*") which means that there are multiple local minimums. So it's not guaranteed to converge (find) to the global minimum. What we need is "convex" function in order gradient descent algorithm to be able to find the global minimum (minimize J(θ)). In order to do that we use ***log*** function.

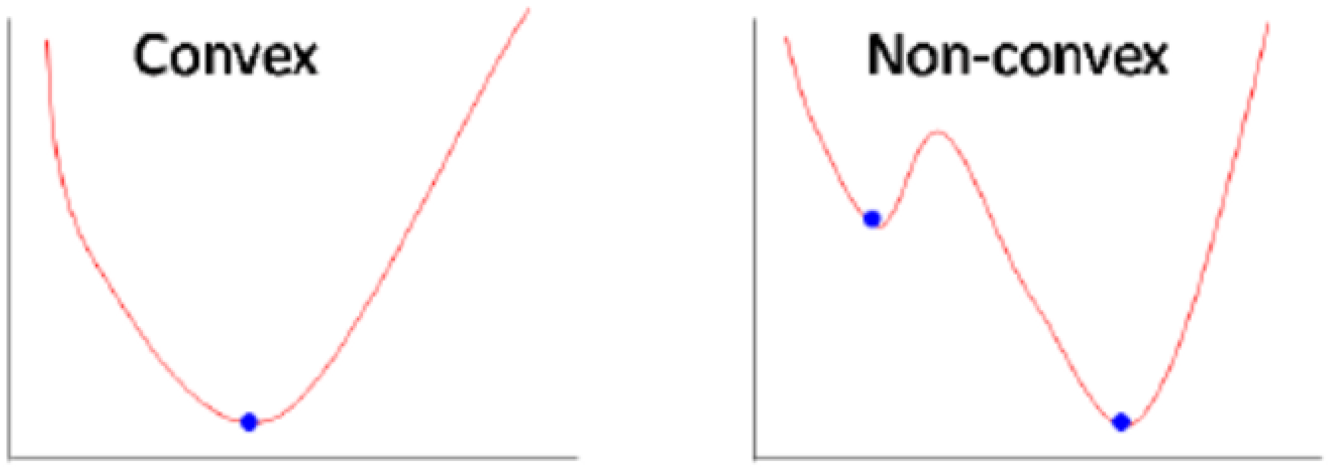


Image 15: Convex vs Non-convex function. source: researchgate.com

So that's why we use following cost function for neural networks

$$\mathrm{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

Image 16: Neural Network cost function. source: coursera.org

In case where labeled value *y* is equal to ***1*** the hypothesis is ***-log(h(x))*** or ***-log(1-h(x))*** otherwise.

The intuition is pretty simple if we look at the function graphs. Let first look at the case where ***y=1***. Then ***-log(h(x))*** would look like the graph below. And we are only interested in the (0,1) x-axis interval since hypothesis can only take values in that range (*"Image 13"*)

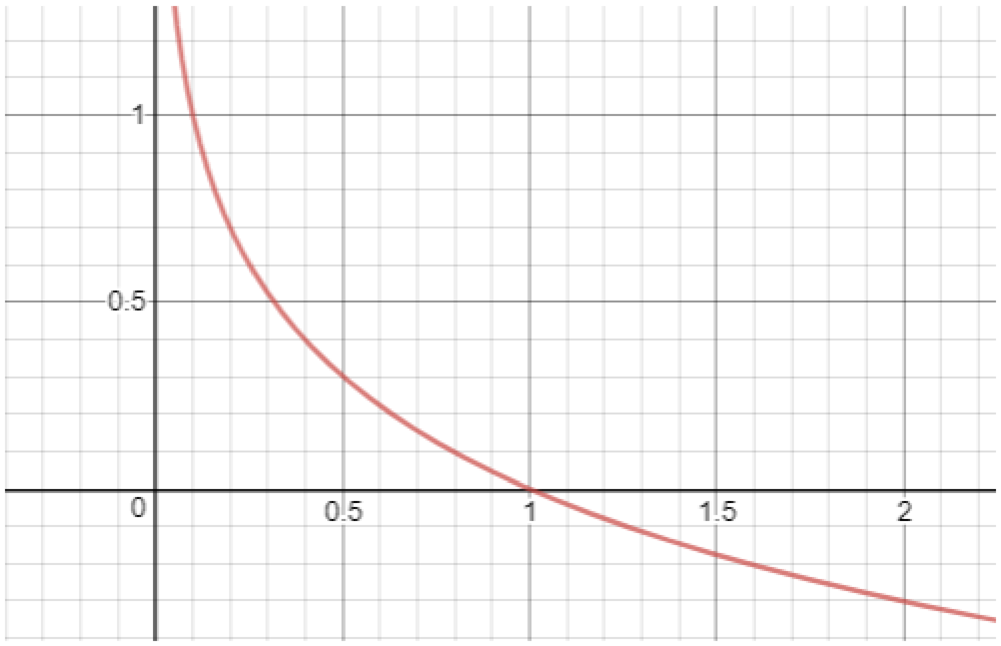


Image 17: Cost function -log(h(x)) . source: desmos.com

What we can see from the graph is that if ***y=1*** and ***h(x)*** approaches value of ***1*** (*x-axis*) the cost approaches the value ***0*** (*h(x)-y* would be *0*) since it's the right prediction. Otherwise if ***h(x)*** approaches ***0*** the cost function goes to infinity (very large cost).

In the other case where ***y=0***, the cost function is ***-log(1-h(x))***

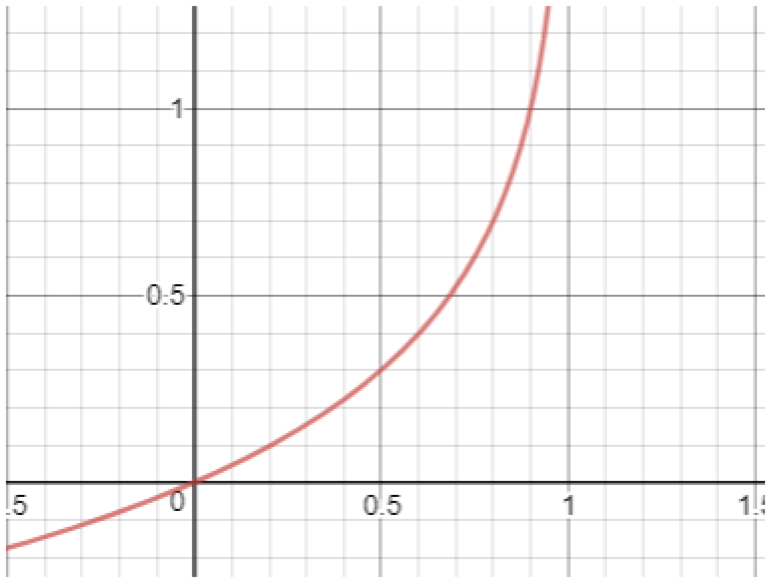


Image 18: -log(1-h) cost function. source: desmos.com

From the graph here we can see that if ***h(x)*** approaches value of ***0*** the cost would approach ***0*** since it's also the right prediction in this case.

Since ***y*** (labeled value) is always equal to ***0*** or ***1*** we can write cost function in one equation.

$$\mathrm{Cost}(h_\theta(x), y) = -y \ \log(h_\theta(x)) - (1-y)\log(1-h_\theta(x))$$

Image 19: Cost function equation. source: coursera.org

If we fully write our cost function with the summation we would get:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} \ \log(h_\theta(x^{(i)})) + (1-y^{(i)}) \ \log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Image 20: Cost function in case of one output node. source: coursera.org

And this is for the case where there is only one node in the output layer of Neural Network. If we generalize this for multiple output nodes (multiclass classification) what we get is:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

Image 21: Generalized Cost function. source: coursera.org

The right parts of the equations represent cost function "regularization". This regularization prevent the data from "overfitting", by reducing the magnitude/values of θ.

Forward Propagation Calculation

This process of Forward propagation is actually getting the Neural Network output value based on a given input. This algorithm is used to calculate the cost value. What it does is the same mathematical process as the one described in section 2 "Model Representation Mathematics". Where in the end we get our hypothesis value *"Image 7"*.

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

After we got the *h(x)* value (hypothesis) we use the Cost function equation (*"Image 21"*) to calculate the cost for the given set of inputs.

$$x = a^{(1)} \tag{1}$$

$$z^{(j+1)} = \theta^{(j)} a^{(j)} \tag{2}$$

$$a^{(j+1)} = \sigma(z^{(j+1)}) \tag{3}$$

$$h_\theta(x) = a^{(L)} = \sigma(z^{(L)}) \tag{4}$$

Image 22: Calculate Forward propagation

Here we can notice how forward propagation works and how a Neural Network **generates the predictions**.

Backpropagation Algorithm

What we want to do is minimize the cost function ***J(θ)*** using the optimal set of values for ***θ*** (weights). Backpropagation is a method we use in order to **compute the partial derivative of *J(θ)***.

This partial derivative value is then used in Gradient descent algorithm (*"Image 23"*) for calculating the ***θ*** values for the Neural Network that minimize the cost function ***J(θ)***.

$$\begin{aligned}&Repeat\ \{\\&\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)\\&\}\end{aligned}$$

Image 23: General form of gradient descent. source: coursera.org

Backpropagation algorithm has 5 steps:

1. Set ***a(1) = X;*** for the training examples
2. Perform forward propagation and compute ***a(l)*** for the other layers ***(l = 2…L)***
3. Use ***y*** and compute the delta value for the last layer ***δ(L) = h(x) — y***
4. Compute the ***δ(l)*** values backwards for each layer (described in "Math behind Backpropagation" section)
5. Calculate derivative values ***Δ(l) = (a(l))^T ∘ δ(l+1)*** for each layer, which represent the derivative of cost ***J(θ)*** with respect to ***θ(l)*** for layer ***l***

> Backpropagation is about determining how changing the weights impact the overall cost in the neural network.

What it does is propagating the "error" backwards in the neural network. On the way back it is finding how much each weight is contributing in the overall "error". The weights that contribute more to the overall "error" will have larger derivation values, which means that they will change more (when computing Gradient descent).

Now that we have sense of what Backpropagation algorithm is doing we can dive deeper in the concepts and math behind.

Why derivatives ?

The derivative of a function (in our case ***J(θ)***) on each variable (in our case weight ***θ***) tells us the **sensitivity of the function with respect to that variable** or **how changing the variable impacts the function value**.

Let's look at a simple example neural network

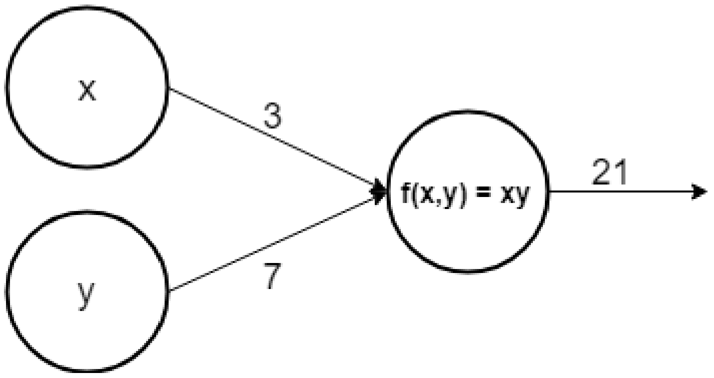


Image 24: Simple Neural Network

There are two input nodes x and y. The output function is calculating the product x and y. We can now compute the partial derivatives for both nodes

$$\frac{\mathrm{d}f}{\mathrm{d}y} = x \qquad \frac{\mathrm{d}f}{\mathrm{d}x} = y$$

$$\frac{\mathrm{d}f}{\mathrm{d}x} = 7 \qquad \frac{\mathrm{d}f}{\mathrm{d}y} = 3$$

Image 25: Derivatives to respect to y and x of f(x,y) = xy function

The partial derivative with respect to x is saying that if x value increase for some value ϵ then it would increase the function (product xy) by 7ϵ and the partial derivative with respect to y is saying that if y value increase for some value ϵ then it would increase the function by 3ϵ.

As we defined, Backpropagation algorithm is calculating the derivative of cost function with respect to each θ weight parameter. By doing this we determine how sensitive is the cost function $J(\theta)$ to each of these θ weight parameters. It also help us determine how much we should change each θ weight parameter when computing the Gradient descent. So at the end we get model that best fits our data.

Math behind Backpropagation

We will by using the neural network model below as starting point to derive the equations.

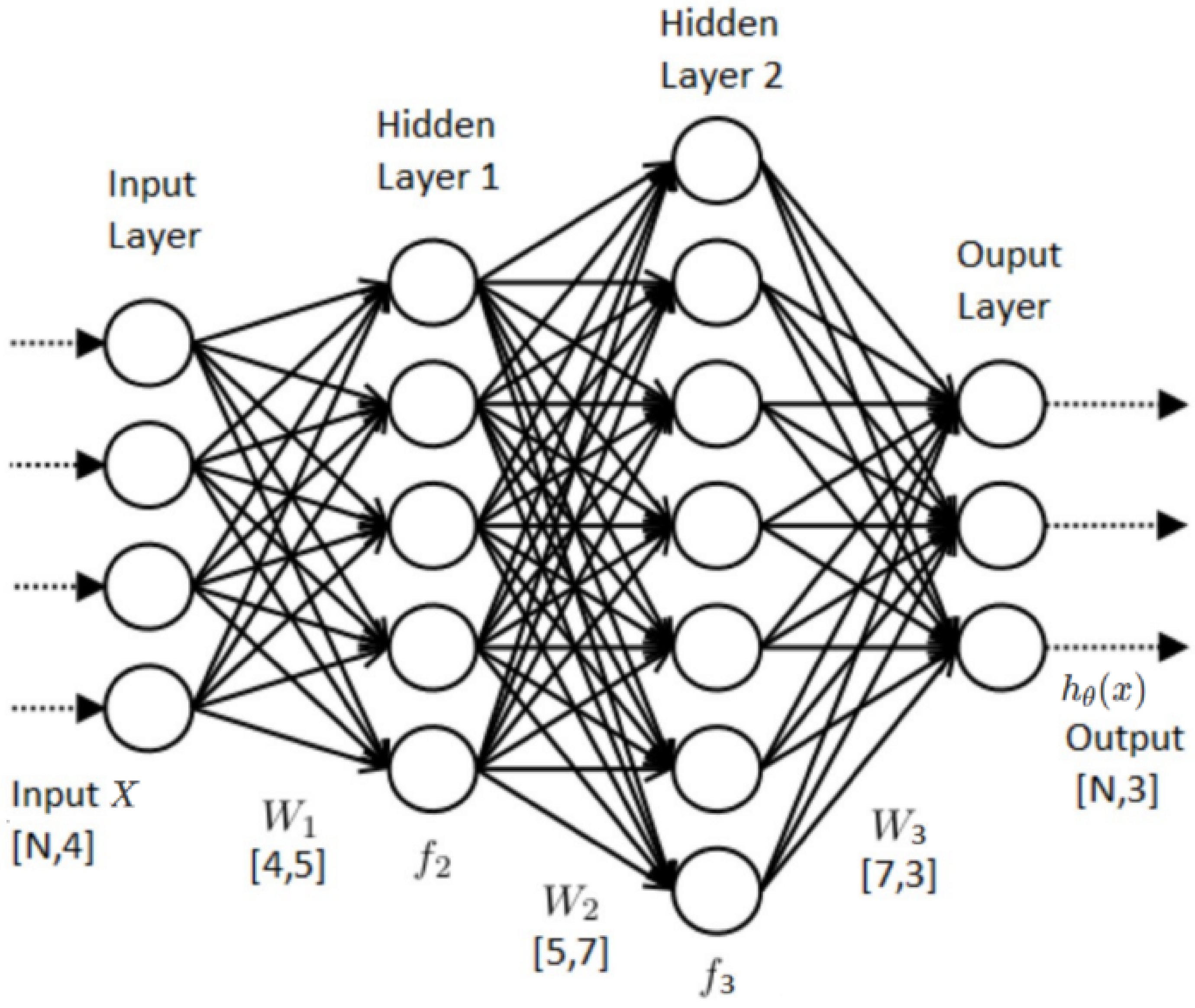


Image 26: Neural Network

In this model we got 3 output nodes (*K*) and 2 hidden layers. As previously defined, the cost function for the neural network is:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

Image 27: Generalized Cost function. source: coursera.org

What we need is to compute the partial derivative of ***J(θ)*** with respect to each ***θ*** parameters. We are going to leave out the summarization since we are using vectorized implementation (matrix multiplication). Also we can leave out the regularization (right part of the equation above) and we will compute it separately at the end. Since it is addition the derivative can be computed independently.

NOTE: Vectorized implementation will be used so we calculate for all training examples at once.

We start with defining the derivative rules that we will use.

$$(cx)' = c \tag{1}$$

$$log(u)' = \frac{1}{u}u' \tag{2}$$

$$(e^u)' = e^u u' \tag{3}$$

$$(u \pm v)' = u' \pm v' \tag{4}$$

$$\left(\frac{u}{v}\right)' = \frac{u'v - v'u}{v^2} \tag{5}$$

$$f(g(x))' = f'(g(x))g'(x) \tag{6}$$

Image 28: Derivative Rules

Now we define the basic equation for our neural network model where l is layer notation and L is for the last layer.

$$a^{(1)} = X; \tag{1}$$

$$a^{(L)} = h_\theta(x); \tag{2}$$

$$a^{(l)} = \sigma(z^{(l)}); \tag{3}$$

$$z^{(l)} = \theta^{(l-1)}a^{(l-1)}; \tag{4}$$

$$\delta^{(L)} = h_\theta(x) - y \tag{5}$$

Image 29: Initial Neural Network model equations

In our case ***L*** has value 4, since we got 4 layers in our model. So let's start by computing the partial derivative with respect to weights between 3rd and 4th layer.

$$\frac{dJ}{d\theta_3} = \underbrace{\left(-\frac{1}{m}\right)}_{Add\ at\ the\ end} \left(y \log(h_\theta(x)) + (1-y)\left(1 - \log(h_\theta(x))\right)\right)' \tag{1}$$

$$= \left(y\left(\log(h_\theta(x))\right)\right)' + \left((1-y)\left(1-\log(h_\theta(x))\right)\right)' \tag{2}$$

$$= \frac{y}{h_\theta(x)}(h_\theta(x))' + \frac{1-y}{1-h_\theta(x)}(1-h_\theta(x))' \tag{3}$$

$$= \frac{y}{h_\theta(x)}(\sigma(z^{(4)})z^{(4)})' + \frac{1-y}{1-h_\theta(x)}(1-(\sigma(z^{(4)})z^{(4)}))' \tag{4}$$

$$= \frac{y}{h_\theta(x)}(\sigma(z^{(4)}))'(z^{(4)})' + \frac{1-y}{1-h_\theta(x)}(-(\sigma(z^{(4)}))'(z^{(4)})') \tag{5}$$

$$= \left(\frac{y}{h_\theta(x)} - \frac{(1-y)}{1-h_\theta(x)}\right)(\sigma(z^{(4)}))'(z^{(4)})' \tag{6}$$

$$= \left(y(1-h_\theta(x)) - (1-y)(h_\theta(x))\right)(z^{(4)})' \tag{7}$$

$$= \left(y - yh_\theta(x) - h_\theta(x) + yh_\theta(x)\right)(z^{(4)})' \tag{8}$$

$$= \left(y - h_\theta(x)\right)(z^{(4)})' = \delta^{(4)}(z^{(4)})' \tag{9}$$

$$= -\left(h_\theta(x) - y\right)(z^{(4)})' = -\delta^{(4)}(z^{(4)})' \tag{10}$$

$$= -\delta^{(4)}(a^{(3)}\theta^{(3)})' = -\delta^{(4)}a^{(3)} \tag{11}$$

$$= \left(\frac{1}{m}\right)\left((a^{(3)})^T \times \delta^{(4)}\right) \tag{12}$$

Image 30: Derivative of θ parameters between 3rd and 4th layer

Step (6) — Sigmoid derivative
To explain the ***step (6)*** we need to calculate the partial derivative of sigmoid function.

$$\sigma'(z) = \left(\frac{1}{1+e^{(-z)}}\right)' = \frac{1'e^{(-z)} - 1(e^{(-z)})'}{(1+e^{(-z)})^2} = \frac{e^{(-z)}}{(1+e^{(-z)})^2} \tag{1}$$

$$= \frac{e^{(-z)}+1-1}{(1+e^{(-z)})^2} = \frac{1}{1+e^{(-z)}} - \left(\frac{1}{1+e^{(-z)}}\right)^2 = \sigma(z)(1-\sigma(z)) \tag{2}$$

Image 31: Derivative of Sigmoid function

In case of the last layer L we got,

$$\sigma(z^{(L)}) = a^{(L)} = h_\theta(x)$$

Image 32: Output layer equation

so,

$$\sigma(z)(1-\sigma(z)) = h_\theta(x)(1-h_\theta(x))$$

Image 33: Output layer equation

Step (11) — Get rid of the summarization (Σ)
Also in the last ***step (11)*** it's important to note that we need to multiply $\boldsymbol{\delta}$ by $\boldsymbol{a}$ transpose in order to get rid of the summarization (1…m for training examples).
$\boldsymbol{\delta}$ — matrix with dimensions
[number_of_training_examples, output_layer_size] so this also means that we will get rid from the second summarization (1…K for number of output nodes).
$\boldsymbol{a}$ — matrix with dimensions
[hidden_layer_size, number_of_training_examples]

Now we continue with the next derivative for the $\boldsymbol{\theta}$ parameters between 2nd and 3rd layer. For this derivation we can start from ***step (9)*** (*"Image 30"*). Since ***θ(2)*** is inside ***a(3)*** function we need to apply the *"Chain Rule"* when calculating the derivative (step(6) from derivative rules on "Image 28").

$$\frac{dJ}{d\theta_2} = \underbrace{\left(\frac{1}{m}\right)}_{Add\ at\ the\ end} \delta^{(4)}(z^{(4)})' = \delta^{(4)}(a^{(3)}\theta^{(3)})' \tag{1}$$

$$= \delta^{(4)}(\sigma(z^{(3)})\theta^{(3)})' = \delta^{(4)}(\sigma(z^{(3)})\theta^{(3)})'(z^{(3)})' \tag{2}$$

$$= (\delta^{(4)} \times (\theta^{(3)})^T)\sigma'(z^{(3)})(z^{(3)})' \tag{3}$$

$$= (\delta^{(4)} \times (\theta^{(3)})^T)\sigma'(z^{(3)})(a^{(2)}\theta^{(2)})' = (\delta^{(4)} \times (\theta^{(3)})^T)\sigma'(z^{(3)})a^{(2)} \tag{4}$$

$$= (a^{(2)})^T \times \underbrace{(\delta^{(4)} \times (\theta^{(3)})^T)\sigma'(z^{(3)})}_{\delta^{(3)}} = \left(\frac{1}{m}\right)(a^{(2)})^T \times \delta^{(3)} \tag{5}$$

Image 34: Derivative of θ parameters between 2nd and 3rd layer

Now we got the derivative for $\boldsymbol{\theta}$ parameter between 2nd and 3rd layer. What we left to do is compute the derivative for $\boldsymbol{\theta}$ parameter between input layer and 2nd layer. By doing this we will see that the same process (equations) will be repeated so we can derive general $\boldsymbol{\delta}$ and derivative equations. Again we continue from ***step (3)*** (*"Image 34"*).

$$\frac{dJ}{d\theta_1} = \underbrace{\left(\frac{1}{m}\right)}_{Add\ at\ the\ end} \underbrace{(\delta^{(4)} \times (\theta^{(3)})^T)\sigma'(z^{(3)})}_{\delta^{(3)}}(z^{(3)})' = \delta^{(3)}(z^{(3)})' \tag{1}$$

$$= \delta^{(3)}(a^{(2)}\theta^{(2)})' = \delta^{(3)}(\sigma(z^{(2)})\theta^{(2)})' \tag{2}$$

$$= \delta^{(3)}\sigma'(z^{(2)})\theta^{(2)}(z^{(2)})' = (\delta^{(3)} \times (\theta^{(2)})^T)\sigma'(z^{(2)})(z^{(2)})' \tag{3}$$

$$= (\delta^{(3)} \times (\theta^{(2)})^T)\sigma'(z^{(2)})(a^{(1)}\theta^{(1)})' \tag{4}$$

$$= (\delta^{(3)} \times (\theta^{(2)})^T)\sigma'(z^{(2)}) \underbrace{a^{(1)}}_{X^T} \tag{5}$$

$$= X^T \times \underbrace{(\delta^{(3)} \times (\theta^{(2)})^T)\sigma'(z^{(2)})}_{\delta^2} \tag{6}$$

$$= \left(\frac{1}{m}\right) X^T \times \delta^{(2)} \tag{7}$$

Image 35: Derivative of θ parameters between input and 2nd layer

From the equation above we can derive equations for $\boldsymbol{\delta}$ parameter and derivative with respect to $\boldsymbol{\theta}$ parameter.

$$\delta^{(l)} = (\delta^{(l+1)} \times (\theta^{(l)})^T)\sigma'(z^{(l)}); l = 2...L$$

Image 36: Recursive δ equation

$$\frac{dJ}{d\theta^{(l)}} = \left(\frac{1}{m}\right)(a^{(l)})^T\delta^{(l+1)}$$

Image 37: Derivative of J (cost) with respect to θ in layer l equation

At the end we get is three matrices (same as $\boldsymbol{\theta}$ weight matrices) with same dimensions as the $\boldsymbol{\theta}$ weight matrices and calculated derivatives for each $\boldsymbol{\theta}$ parameter.

Add the regularization

As already mentioned regularization is needed for preventing the model from overfitting the data. We have already defined regularization for our cost function which is the right part of the equation defined on “Image 21“.

$$+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

Image 38: Regularization equation for Cost function

In order to add the regularization for the gradient (partial derivative) we need to compute the partial derivative for the regularization above.

$$+\frac{\lambda}{m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\theta_{j,i}^{(l)}$$

Image 39: Regularization equation for gradient (partial derivative)

Which means just adding the sum of all theta values from each layer to the partial derivatives with respect to $\boldsymbol{\theta}$.