Feed Forward Neural Networks

Erdi KARA

1 Introduction

Deep learning can be considered as a subclass of machine learning that utilize multiple-layer architecture to extract information from a given data. Each layer consist of units, called *neurons*, which is a numerical representation of the processed information. Each neuron in the architecture has two typical parameters (possibly other parameters as well), *a weight* and *a bias*. A neural network architecture is composed of an *input layer, hidden layer(s)* an *output layer*. Each hidden layer has a certain level of connection with the previous and the next layer. We define the *depth* of a network as the number of layers except the first one while the *width* is the the maximum number of neurons in a layer. A network with depth 4 and width 5 can be seen in Fig-1.

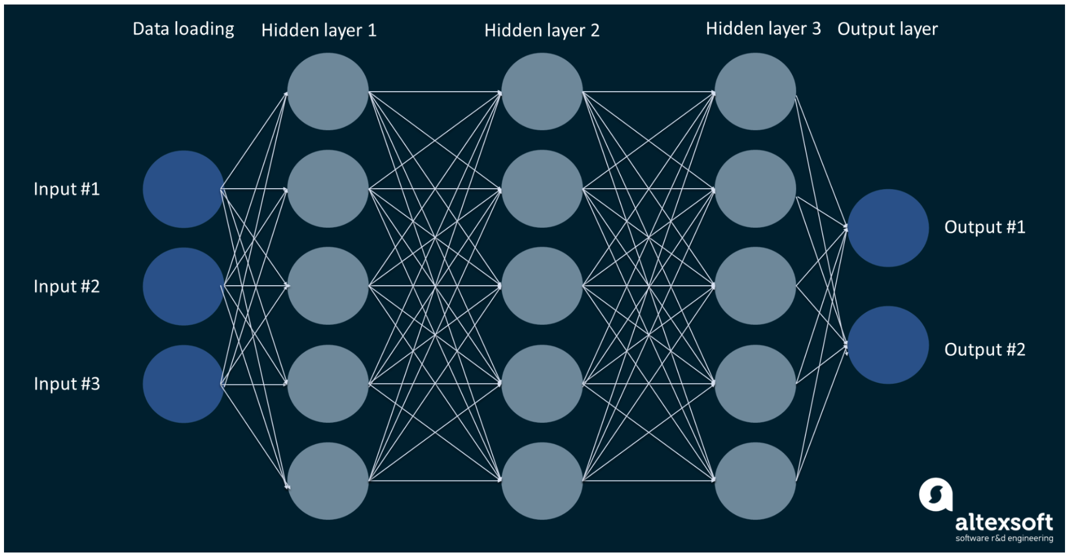

Figure 1: A generic neural network architecture with 2 hidden layers.

Input layer is used to hold the row input data. We then pass the data to the hidden layers which process this data using weights and biases of the neurons belonging to the layer. Passing the information from one layer to another is called a *forward pass*. By this way, information flows through the hidden layers until the output layer. After the forward pass is completed, we compute the loss of the algorithm against a known true value using a *cost function.* In other words, we measure the quality of the prediction. We then use this information to update the weights and biases of the layers starting from the last one to the first. This backward-updating process is called *back propagation.* We process the entire data by forward-passing and back propagating. One such complete cycle is called an *epoch.* We repeat the same process for several epochs to improve the quality of the prediction. The main goal is to update the weights and biases (possibly other parameters as well) to minimize the cost function in each epoch so that we can obtain more accurate predictions.

Deep neural network (DNN) can be regarded as a collective name of all similar methods. The network above is called the feed forward (sometimes multilayer perceptrons) because the output is not fed back to the system. Neural networks having this property are called recurrent neural networks(RNN). Another type of neural network is the convolution neural networks (CNN) which we will discuss in next chapter. In this chapter, we primarily focus on feed forward and convolution networks. However, fundamental principles we will discuss in the next section is applicable in a broad range of neural network models.

2 Multiple Linear Regression

In mathematical terms, a feed forward neural network defines an approximation scheme $\boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x};\boldsymbol{\theta})$ to approximate a function $y = f(x)$. The network optimizes or *learns* the parameter family $\boldsymbol{\theta}$ to find the best approximation to the function $f(x)$. We can start with an illustrative example from multiple linear regression. Consider the sample data in table below.

X_1	X_2	Y
3	3	27
8	6	68
8	1	58
3	4	29
1	6	19
4	9	46

Our task is to build a *network(model,arhitecture,..)* to predict the output Y given X_1 and X_2. Consider the following network with 2 input neurons and a single output. Two neurons represent two real input values x_1, x_2 and $\hat{y}$ is the predicted output in each row. As a convention, weights and bias belong to the next layer.

Let's start with a *linear model.* In other words,

$$\hat{y} = w_1x_1 + w_2x_2 + b \tag{1}$$

for the weights w_1, w_2 and the bias b. In machine learning terminology, we call them *learnable parameters.* To measure the error between the predicted value $\hat{y}$ and the real ouput y, we need a *cost function.* The easiest way is to consider the mean-square cost function

$$C(\hat{y}) = \frac{1}{N}\sum_{i=1}^{N}\Big(\hat{y}_i - y_i\Big)^2 = \frac{1}{N}\sum_{i=1}^{N}\Big(w_1x_1^{(i)} + w_2x_2^{(i)} + b - y_i\Big)^2 \tag{2}$$

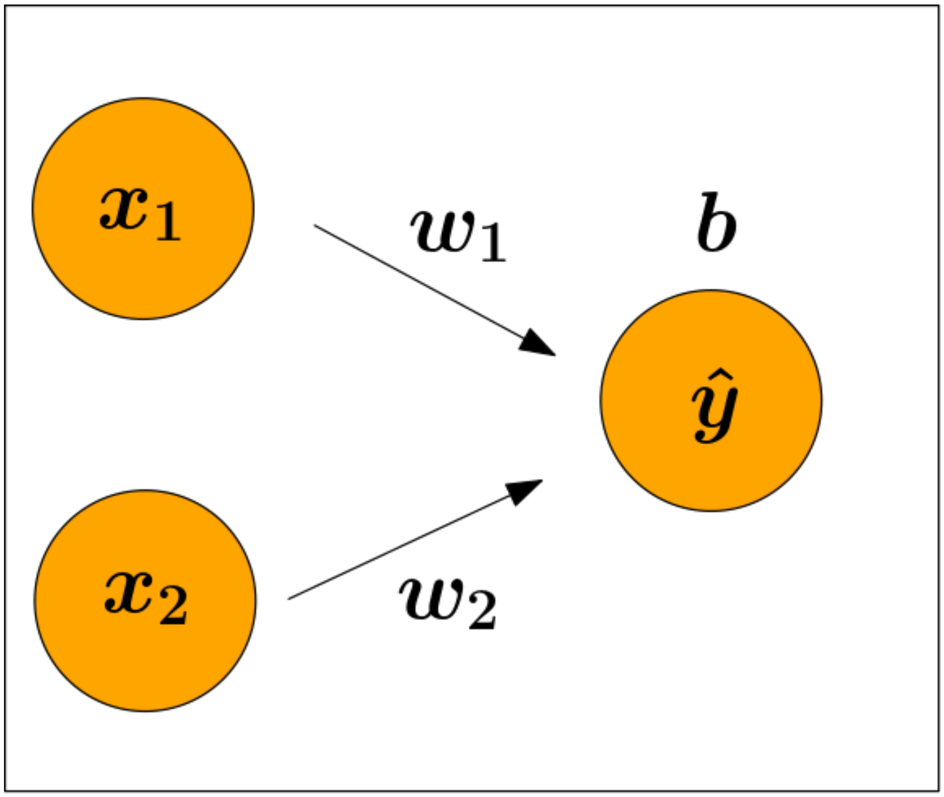

Figure 2: A neural network with two layers; two inpus in the first layer no hidden layer and one output in the output layer

where, $\left(x_1^{(i)}, x_2^{(i)}\right)$ are the input values in the i^{th} row. Notice that the cost function is over all predictions. With these ingredients, the problem statement reads

Find w_1, w_2, b such that $C(\hat{y})$ is minimized.

Thus, the core idea in a neural network is to find the optimal values of learnable parameters to minimize the cost function so that $\hat{y}_i \approx y_i$ for each data points. The most common method to tackle this problem is the *gradient decent* algorithm.

From Calculus-3, we know that the minimum rate of change of a differentiable function occurs in the direction of the negative gradient. Roughly speaking, if you walk in the opposite direction of the derivative of a function, you reach to a local minimum, or global one if you are lucky.The cost function we define is an *implicit function* of learnable parameters.

The idea is to evaluate the gradients of the cost function with respect to the learnable parameters and take repeated steps in the direction of the negative gradient, which will lead to the local minimum(ideally the global minimum) of the cost function.

The step size taken is called the *learning rate* which measures how fast we update the learnable parameters. One step gradient decent for the first data point can be summarized as follows. Let's set the learning rate $r = 0.01$

1. Initilize the weights and biases(only once) $w_1 = 5, w_2 = 2, b = 3$
2. Forward Pass
   (a) Feed the data

   inputs: $(x_1^{(1)}, x_2^{(1)})$ and output: y_1

   (b) Get the predictions $\hat{y}_i$

   $\hat{y}_i = w_1 x_1^{(1)} + w_2 x_2^{(1)} + b = 5 \cdot 3 + 2 \cdot 3 + 3 = 24$

   (c) Evaluate the loss $C(\hat{y})$

   $C(\hat{y}) = (\hat{y} - y)^2 = (24 - 27)^2 = 9$
3. Back Propogation
   (a) Evaluate the gradients

   $$\frac{\partial C}{\partial w_1} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_1} = 2(\hat{y} - y)x_1 = 2(24 - 27)3 = -18$$

   $$\frac{\partial C}{\partial w_2} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2} = 2(\hat{y} - y)x_2 = 2(24 - 27)3 = -18$$

   $$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = 2(\hat{y} - y)1 = 2(24 - 27)1 = -6$$

   (b) Update the weights(gradient decent)

   $$w_1 = w_1 - r\frac{\partial C}{\partial w_1} = 5 - 0.01 \cdot (-18) = 5.18$$

   $$w_2 = w_2 - r\frac{\partial C}{\partial w_2} = 2 - 0.01 \cdot (-18) = 2.18$$

   $$b = b - r\frac{\partial C}{\partial b} = 3 - 0.01 \cdot (-6) = 3.06$$

To obtain the optimal parameters for this model, we should scan all the data points until a reasonable approximation is attained. Following simple Python code evaluates the optimal parameters. Since this is a very simple demonstration, we don't need to vectorize the parameters.

```python
import numpy as np
X1 =  np.array([3,8,8,3,1,4])
X2 =  np.array([3,6,1,4,6,9])
Y = np.array([27,68,58,29,19,46])
N = X1.shape[0]
w1 = 5.; #initial guess for w1
w2 = 2.; #initial guess for w2
b = 3.   #initial guess for b
num_epochs = 100;
lr = 0.01;
for i in range(num_epochs):
    #FORWARD PASS
    y_pred = w1 * X1 + w2 * X2 +b          #get the prediction
    loss = 1/N * np.sum((y_pred - Y)**2)   #compute the loss

    #BACK-PROPOGATION
    w1_der = (1/N)*np.sum(2*(y_pred-Y)*X1) #compute the gradient w.r.t w1
    w2_der = (1/N)*np.sum(2*(y_pred-Y)*X2) #compute the gradient w.r.t w2
    b_der = (1/N)*np.sum(2*(y_pred-Y))*1   #compute the gradient w.r.t b
    w1 = w1 - lr * w1_der  # update w1
    w2 = w2 - lr * w2_der  # update w2
    b = b - lr * b_der     # update b

print(f'w1 = {w1:0.5f} w2 = {w2:0.5f}, b = {b:0.5f} loss: {loss:0.6f}')
```

Output of this code is

$$w1 = 6.944, w2 = 2.012, b = 0.372, \text{loss} : 0.076$$

which is indeed close to the desired values. Because, the output values in the sample data is generated such that

$$7 \cdot X1 + 2 \cdot X2 + 0 \cdot B = Y$$

In fact, the predicted vector is as follows;

$$\hat{Y} = [27.244, 68.003, 57.939, 29.257, 19.394, 46.265]$$

Remark. *At this point, we should note the following points;*

- *Learning rate defines the behavior of the gradient decent. If we choose $\eta = 0.1$, the iteration simply diverges. Thus choosing an optimal learning rate is an essential task in any deep learning model.*
- *Number of epochs is also another factor which effects the approximation. Although it is tempting to consider a large value to get a better approximation, execution time can be a huge hurdle if the model is relatively large.*
- *This network is essentially a linear model. We will modify it in a nonlinear fashion using an activation function to make it learn more complex tasks. In fact, introducing an activation function has huge implications on learning task that we will discuss in the next sections.*

3 Feed Forward Neural Networks

3.1 Forward Pass

In this section, we will generalize the ideas above for any feed forward network We will closely follow the conventions in [1]. Lets consider the following notations:

$w^l_{jk} \rightarrow$ weight from the k^{th} neuron in layer $l-1$ to the j^{th} neuron in layer l.
$b^l_j \rightarrow$ bias of the j^{th} neuron in layer l
$a^l_j \rightarrow$ activation of the j^{th} neuron in layer l

A typical fully connected network with L-layer can be seen in the Figure 3. For simplicity, we did not draw all connections.

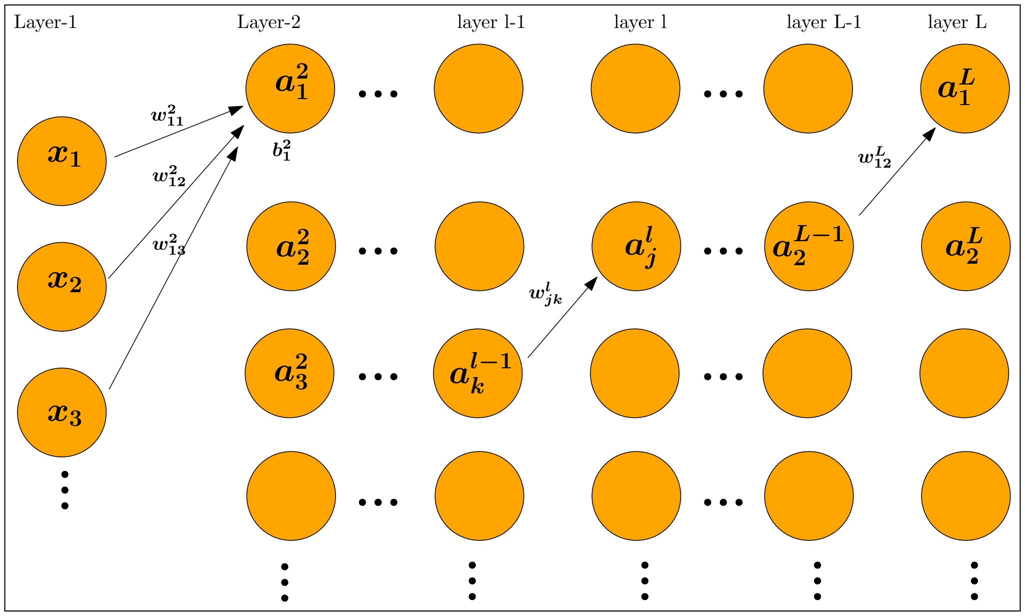

Figure 3: Schematic illustration of a fully connected neural network architecture with L-layers

Next we define an activation function. The most commons ones are the followings. The choice of activation function makes up an important component of a neural network model design.

$$\sigma(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \quad \rightarrow \text{Rectified linear unit(ReLU)} \tag{3}$$

$$\sigma(x) = \frac{1}{1+e^{-x}} \quad \rightarrow \text{Sigmoid} \tag{4}$$

$$\sigma(x) = \frac{1}{1+e^{-2x}} \quad \rightarrow \text{tangent hyperbolic} \tag{5}$$

$$\sigma(x) = \arctan x \quad \rightarrow \text{inverse tangent} \tag{6}$$

The graphs of these activation functions can be seen below

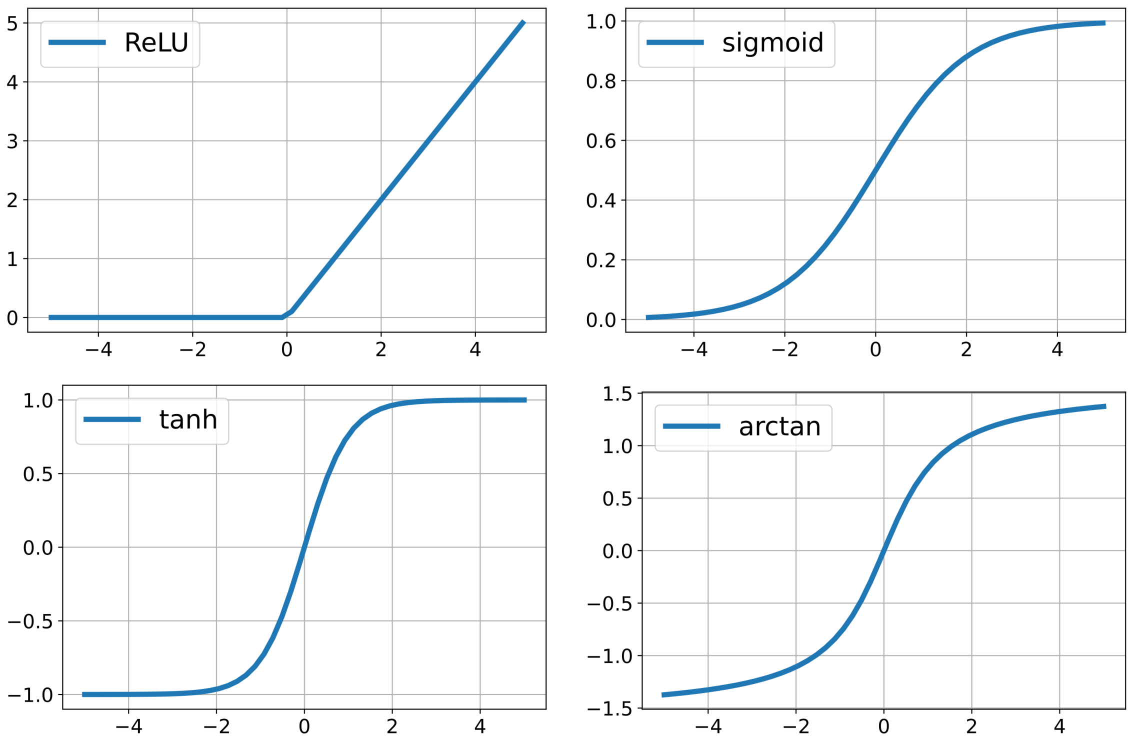

Figure 4: Some of the most common activation functions

We define a model by taking a generic activation function $\sigma(x)$ as follows:

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \tag{7a}$$

$$a_j^l = \sigma(z_j^l) \tag{7b}$$

where the sum is over the neurons in the $(l-1)^{th}$ layer .

We will call z_j^l as an output and a_j^l as an activation. Relationship between two consecutive layers can be expressed as a matrix multiplication as follows:

$$\begin{bmatrix} z_1^l \\ z_2^l \\ .. \\ z_j^l \\ .. \end{bmatrix} = \begin{bmatrix} w_{11}^l & w_{12}^l & w_{13}^l & ... \\ w_{21}^l & w_{22}^l & w_{23}^l & ... \\ .. & .. & .. & .. \\ .. & .. & w_{jk}^l & .. \\ .. & .. & .. & .. \end{bmatrix} \begin{bmatrix} a_1^{l-1} \\ a_2^{l-1} \\ .. \\ a_k^{l-1} \\ .. \end{bmatrix} + \begin{bmatrix} b_1^l \\ b_2^l \\ .. \\ b_j^l \\ .. \end{bmatrix} \tag{8a}$$

$$\begin{bmatrix} a_1^l \\ a_2^l \\ .. \\ a_j^l \\ .. \end{bmatrix} = \sigma \begin{bmatrix} z_1^l \\ z_2^l \\ .. \\ z_j^l \\ .. \end{bmatrix} \tag{8b}$$

In matrix-vector notation, we can write

$$z^l = W^l a^{l-1} + b^l \tag{9}$$

Note that if there are $\boldsymbol{M}$ neurons in l^{th} layer and $\boldsymbol{N}$ neurons in $(l-1)^{th}$ layer, W becomes **MxN** matrix.

3.2 Backpropagation

Let m and n be the dimensions of the input and output layers. By following a similar path, we will derive the fundamental equations which enable us to approximate a given function $\vec{y}_n = f(\vec{x}_m)$. We should note that this function has a discrete form in practical applications. In other words, we try to optimize the parameters of the architecture to find the best approximate for the data points $(\vec{x}, \vec{y})$. Let

$$C = C(a^L; y) \tag{10}$$

be a generic cost function which we will specify later on. Although C is defined over the last output layer, it is an implicit function of all weights and biases in the architecture. We can re-look at the structure of the last layer in Fig-5.

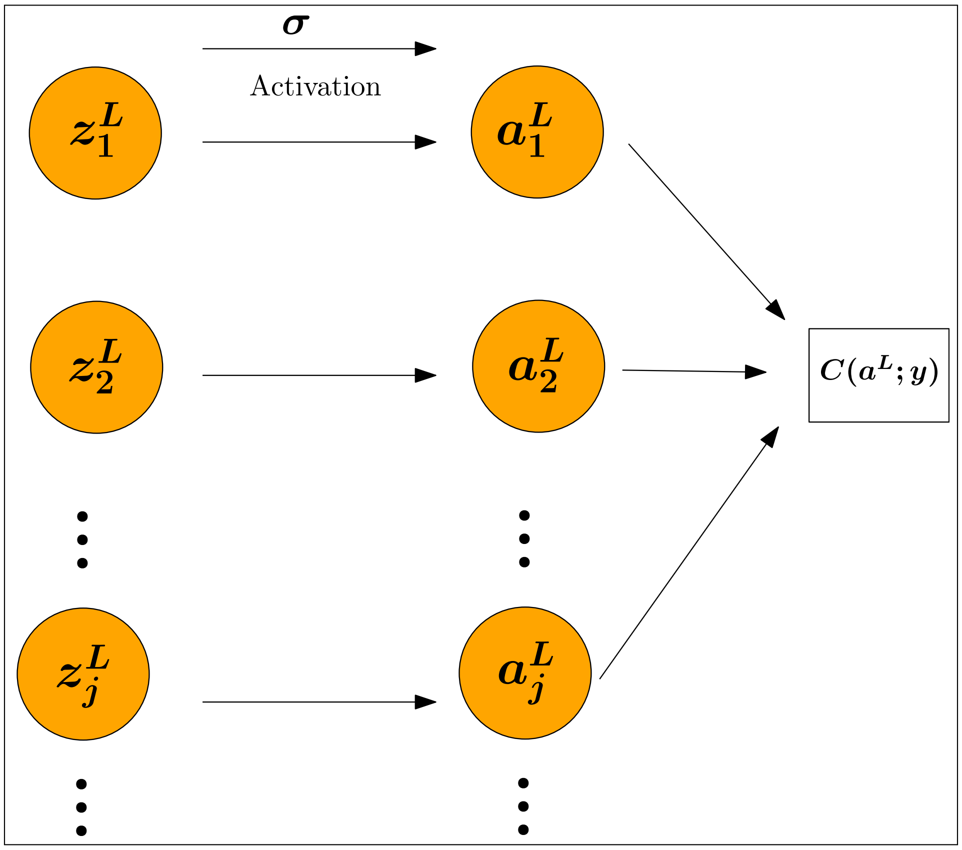

Figure 5: Generic structure of the output layer of a fully-connected network.

We want to find an expression of the rate of change of the cost function with respect to the weights and biases of the architecture. Lets define a generic error expression:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} \tag{11}$$

Using the chain rule with 7b, we can write

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{12}$$

In vectorial form

$$\boxed{\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{13}}$$

where $\odot$ represents component-wise (Hadamard) product with the gradient $\nabla_a C = \left[\frac{\partial C}{\partial a_1^L}, \frac{\partial C}{\partial a_2^L}, .., \frac{\partial C}{\partial a_n^L}\right]^T$ Once z^L is computed through forward pass, (13) can be evaluated with a prescribed cost function. In other words, this information is sufficient to update the weights and biases in the last layer. Now, we will propagate this error all the way back to the second layer. Lets look at the rate of change of the cost function with respect to the output z_j^l. Carefully note that all neurons z^{l+1} in the $(l+1)^{th}$ layer is dependent on z_j^l. Thus the following relations holds:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \frac{\partial C}{\partial z_k^{l+1}} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}$$

If we rewrite the relation (7a), $z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}$, we obtain $\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l)$. This implies

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \tag{14}$$

In vectorial form, we have

$$\delta^l = (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l) \tag{15}$$

Our main goal is to propagate the error on the previous layers to update the weights and biases. From (7a), we can write

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial z_j^l}{\partial b_j^l} \frac{\partial C}{\partial z_j^l} = 1 \cdot \delta_j^l \tag{16}$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial z_j^l}{\partial w_{jk}^l} \frac{\partial C}{\partial z_j^l} = a_k^{l-1} \delta_j^l \tag{17}$$

Using these last two equations in the light of (13) and (15), we can describe one cycle forward pass and back prorogation as follows:

Algorithm 1 Gradient Decent for a Feed Forward Network

```
Set input data $\vec{x}_m$ and learning rate $\eta$
Assign initial values to all weights and biases.
**Feed Forward**
**for** $l = 2, 3...L$ **do**
    $a^l = \sigma(W^l a^{l-1} + b^l)$
**end for**
Compute the output loss and update the last layer parameters
$\delta^L = \nabla_a C \odot \sigma'(z^L)$
$W^L \leftarrow W^L - \eta \delta^L (a^{L-1})^T$
$b^L \leftarrow b^L - \eta \delta^L$
**Back Propagation**
**for** l = L-1, L-2, ...2 **do**
    $\delta^l = (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$
    $W^l \leftarrow W^l - \eta \delta^l (a^{l-1})^T$
    $b^l \leftarrow b^l - \eta \delta^l$
**end for**
```

In the back propagation step, $\delta^l(a^{l-1})^T$ is the matrix form of the component-wise expression (17). Above algorithm is the description for a single epoch and a single data point.

3.3 Final Remarks

Note that we use a generic learning rate η for all layers but in practice one can use a parameter-based learning rate and adjust the learning rate layer-wise or parameter-wise. For example, one of the most popular optimization methods in deep learning ADAM(Adaptive Moment Estimation)[2] computes adaptive learning rates for each parameter but trade off in this scheme is that it requires more time to train comparing to other methods.

Let's consider the equation (13) with the sigmoid function. As we can see from the Fig-(4), the range of sigmoid function is $[0, 1]$. Thus if $\sigma(z_j^L) \approx 0$ or 1, its derivative at this point is near flat, i.e, $\sigma'(z_j^L) \approx 0$. Thus the weight associated with this node most likely changes very slowly in gradient decent or we say learns slowly. This type of neurons are called saturated or near saturation.

In the light of the same discussion, let's take the equation (15) which we we utilize to update the weights and biases at the corresponding layers. Notice that the saturation effect in the last layers may propagate back to the early layers and cause neurons to learn very slowly or even completely stop the parameter updates. The effect can be observed more clearly especially in the first layers of very deep architectures. This problem is known as the *vanishing gradient problem*. That being said, we should note that the others terms in (13) can balance the saturation problem and the model may not suffer from vanishing gradient. Thus this phenomena requires careful mathematical analysis.

One way to avoid saturation problem would be to define an activation function whose derivative does not vanish. Probably the most common activation function used in this sense is the rectified linear unit (Relu) $\sigma(x) = max(x, 0)$ which is always positive with a positive derivative. In practice, Relu is almost the best choice. See figure (6).

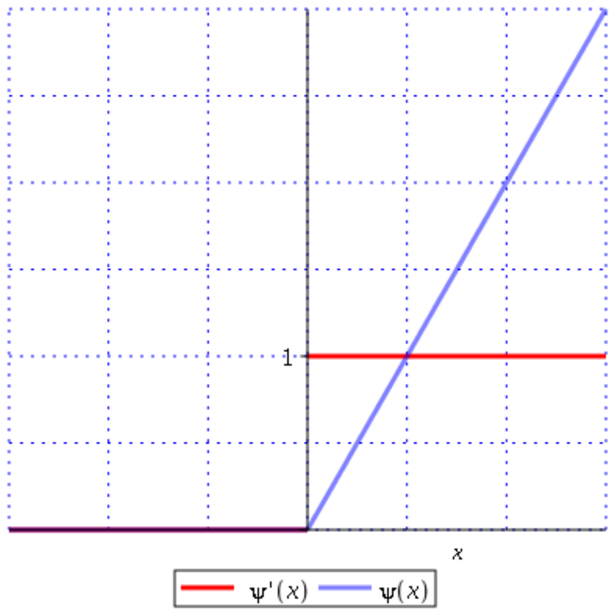

Figure 6: Relu activation has a constant derivative

We have not discussed a specific form of the cost function to this point. Assume that we have data set containing N_p data points of the forms

$$\mathcal{D} = \{(\vec{x^i}, \vec{y^i})\}_{i=1}^{N_p} \tag{18}$$

These are predefined input and output values. We prefer to use vector notation because we usually deal with multidimensional input and output. We will use this data set to update the learnable parameters of the architecture. In other words, we train the model using this data set. Thus we will call $\mathcal{D}$ as a *training set.* A proper cost function for this data set must be of the following form

$$C = \sum_{i=1}^{N_p} C^{(i)}(\vec{y^i}; a^{(i)L}) \tag{19}$$

where $a^{(i)L}$ is the output activation or predicted value for $\vec{x^i}$ and the form of $C^{(i)}$ is to be specified. One common cost function is the mean-square error function defined as

$$C = \frac{1}{2N_p} \sum_{i=1}^{N_p} \|\vec{y^i} - a^{(i)L}\|^2 \tag{20}$$

where the vector norm $\|\|$ can be defined as the Euclidean distance, i.e,

$$\|\vec{y^i} - a^{(i)L}\|^2 = \sum_{k=1}^{n}(y_k^i - a_k^{(i)L})^2$$

In gradient decent algorithm, we must compute the gradient of the cost function $\nabla_a C$ to evaluate the term $\delta^L = \nabla_a C \odot \sigma'(z^L)$. Using (19), we have

$$\nabla_a C = \sum_{i=1}^{N_p} \nabla_a C^{(i)} \tag{21}$$

Once we have this value, we can then enter the loop and update the parameters. However if N_p is very large, i.e. training set consists of large number of samples, finding the exact value of the gradient is not feasible from computational cost viewpoint. So the most common approach is to update the parameters after some specified value M such that $M < N_p$. This approach is called *stochastic gradient decent* .In mathematical terms,

$$\nabla_a C = \sum_{i=1}^{N_p} \nabla_a C^{(i)} \approx \sum_{i=1}^{M} \nabla_a C^{(i)} \tag{22}$$

M is called the *batch* size and along with the learning rate this is another parameter which should be taken into account in training stage.

Cross Entropy Cost Function

Let's consider the first example in (2) with an activation $\sigma(x)$. If we rewrite (12) for a generic data point $(\vec{x}, \vec{y})$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) = (a_j^L - y_j)\sigma'(z_j^L) \tag{23}$$

If we employ this result in (17) for the last layer we obtain

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}\delta_j^L = a_k^{L-1}(a_j^L - y_j)\sigma'(z_j^L) \tag{24}$$

In the discussion above, we see that the term $\sigma'(z_j^L)$ can lead to vanishing gradient problem. One idea would be to use the rectified linear unit as an activation function. Another approach is to define a cost function which can eliminate the derivative term in (24). Let's define so called *cross entropy* cost function along with sigmoid activation. Again for a generic point $(\vec{x}, \vec{y})$

$$C = -\sum_j \left[y_j \ln(a_j^L) + (1 - y_j)\ln(1 - a_j^L) \right] \tag{25}$$

where $a_j^L = \sigma(z_j^L)$. First note that sigmoid function satisfies $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. From here,

$$\frac{\partial C}{\partial a_j^L} = -\frac{y_j}{a_j^L} + \frac{1 - y_j}{1 - a_j^L} = -\frac{y_j}{\sigma(z_j^L)} + \frac{1 - y_j}{1 - \sigma(z_j^L)} = \frac{\sigma(z_j^L) - y_j}{\sigma(z_j^L)(1 - \sigma(z_j^L))} = \frac{\sigma(z_j^L) - y_j}{\sigma'(z_j^L)}$$

If we plug this result into (24), we obtain

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}\delta_j^L = a_k^{L-1}(a_j^L - y_j) \tag{26}$$

Notice that this last expression is free from the derivative term. Thus it theoretically avoids learning slow down.

Softmax Layer

Assume that instead of using an activation function ,we define another type of output layers of the form

$$a_j^L = \gamma(z_j^L) = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{27}$$

γ is called a *softmax* function. We can see that γ is monotonically increasing, positive and less than 1. Moreover $\sum_j \gamma(z_j^L) = 1$.Thus, we can consider the range of γ as a probability distribution. In other words, the output of softmax layer can be interpreted as a probability vector. For example, image classification problems, it is common to label different categories with number tags such as 'A':1, 'B':2, 'C':3 and set the correct output as $A : [1, 0, 0]$, $B : [0, 1, 0]$, etc. So if the model endowed with softmax outputs a vector such as $[0.90, 0.08, 0.02]$, we interpret that the input is most likely A and less likely C. Without softmax, the entries of this vector are real numbers and the maximum entry is taken as the predicted value. Notice that softmax provides a nice way to interpret the predictions from probability point of view.

It is very common to employ logarithmic softmax with negative log-likelihood cost function

$$C = -\ln(\gamma(z_j^L)) \tag{28}$$

and partial derivatives related to weight and biases gives a similar result to (26). One can also use softmax along with the cross entropy cost.

References

[1] Michael A. Nielsen. *Neural Networks and Deep Learning.* 2015.

[2] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.