A Gentle Introduction to Neural Networks

Erdi KARA

1 Multiple Linear Regression

Include linear regression stuff here.

2 Building a Fully Connected Neural Network(FCNN)

2.1 Forward Pass

In the previous section, we introduced multiple linear regression as a stepping stone to understanding neural networks (also known as deep learning). In this section, we will provide a brief overview of the mathematical foundations of neural networks. We recommend [1] as a more in-depth resource. As we delve into the topic, we will use the following conventions to describe the connections between the various components of a neural network.

$w^l_{jk} \rightarrow$ 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 generic fully connected network with L-layer can be seen in the Figure 1. Carefully note that every *neuron* is connected to all neurons in the previous and the next layer giving the name *fully connected neural network*. However, for simplicity, we display just a few connections that are sufficient to understand the construction.

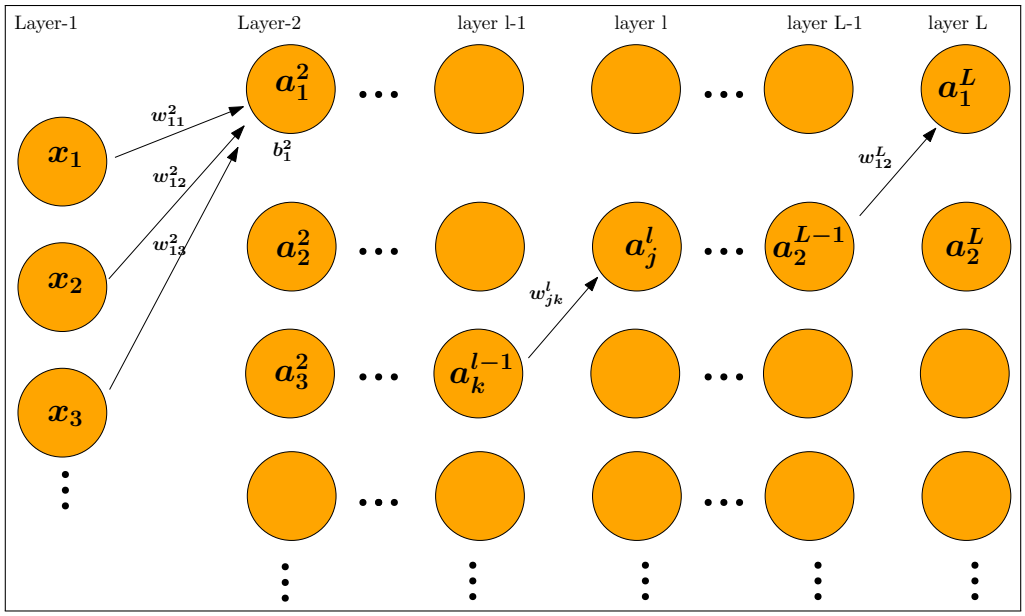

Figure 1: Schematic illustration of a fully connected neural network with L-layers

Let $\sigma(x)$ be a generic non-linear function. We will call it the *activation function*. In the next section, we will discuss the details of what type of functions are used in neural network theory. Starting from the input later, the information will flow through the network with the following equations. This process is called **forward pass**.

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

$$a_j^l = \sigma(z_j^l) \tag{2}$$

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:

$$\begin{bmatrix} z_1^l \\ z_2^l \\ .. \\ z_j^l \\ .. \end{bmatrix} = \begin{bmatrix} w_{11}^l & w_{12}^l & w_{13}^l & ... \\ w_{21}^l & w_{22}^l & w_{23}^l & ... \\ .. & .. & .. & .. \\ .. & .. & w_{jk}^l & .. \\ .. & .. & .. & .. \end{bmatrix} \begin{bmatrix} a_1^{l-1} \\ a_2^{l-1} \\ .. \\ a_k^{l-1} \\ .. \end{bmatrix} + \begin{bmatrix} b_1^l \\ b_2^l \\ .. \\ b_j^l \\ .. \end{bmatrix} \tag{3a}$$

$$\begin{bmatrix} a_1^l \\ a_2^l \\ .. \\ a_j^l \\ .. \end{bmatrix} = \sigma \begin{bmatrix} z_1^l \\ z_2^l \\ .. \\ z_j^l \\ .. \end{bmatrix} \tag{3b}$$

In matrix-vector notation, we can write

$$z^l = W^l a^{l-1} + b^l \tag{4}$$

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

2.2 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

$$C = C(a^L; y) \tag{5}$$

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. **The core idea of neural networks is to find the optimal value of all weights and biases in the network to minimize the cost function $\boldsymbol{C}$**. The smaller the cost function, the better the approximation. To achieve this, we will use *gradient decent method* as we discussed in the previous section.Let us look at the structure of the last layer in Fig-2.

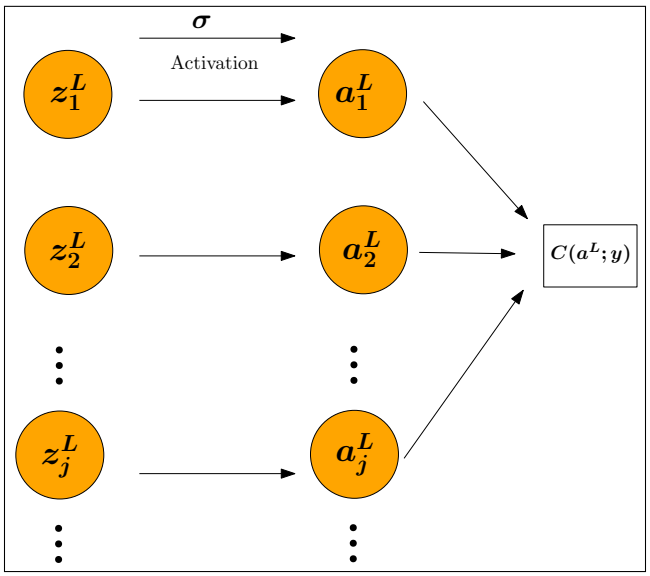

Figure 2: Structure of the output layer of a fully-connected network.

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:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} \tag{6}$$

Using the chain rule with (2), we can write

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{7}$$

In vectorial form

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{8}$$

where $\odot$ represents component-wise (Hadamard) product with the gradient

$$\nabla_a C = \left[\frac{\partial C}{\partial a_1^L} \quad , \frac{\partial C}{\partial a_2^L} \quad , .. \quad , \frac{\partial C}{\partial a_n^L}\right]^T$$

Once z^L is computed through forward pass, (8) 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:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\frac{\partial C}{\partial z_k^{l+1}} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l}\delta_k^{l+1} \tag{9}$$

If we rewrite the relation (1), we obtain

$$z_k^{l+1} = \sum_j w_{kj}^{l+1}a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1}\psi(z_j^l) + b_k^{l+1} \implies \frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1}\sigma'(z_j^l)$$

Thus

$$\delta_j^l = \sum_k w_{kj}^{l+1}\delta_k^{l+1}\sigma'(z_j^l) \tag{10}$$

In vectorial form, we have

$$\delta^l = (W^{l+1})^T\delta^{l+1} \odot \sigma'(z^l) \tag{11}$$

Our main goal is to propagate the error on the previous layers to update the weights and biases. From (1)

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial z_j^l}{\partial b_j^l}\frac{\partial C}{\partial z_j^l} = 1 \cdot \delta_j^l \tag{12}$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial z_j^l}{\partial w_{jk}^l}\frac{\partial C}{\partial z_j^l} = a_k^{l-1}\delta_j^l \tag{13}$$

These last two equations in the light of (8) and (11) allow us to compute the gradient of the cost function with respect to model weights and biases. This process is called **backpropogation**.

2.3 Gradient Decent for Neural Networks

With forward pass and backpropogation, we can establish the gradient decent algorithm for neural networks as follows.

Algorithm 1 Gradient Decent for a Feed Forward Network

Set input data $\vec{x}_m$ and learning rate η
Assign initial values to all weights and biases.
Feed Forward
for $l = 2, 3...L$ **do**
$a^l = \psi(W^l a^{l-1} + b^l)$
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 (13). Above algorithm is the description for a single epoch and a single data point.

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.

2.4 Some Remarks on Activation Function and Cost Function

In the previous section,we worked with a generic activation function and cost function. Let's have a closer look at some common forms.

The choice of activation function makes up an important component of a neural network model design. Some of the most common ones are defined below.

$$\sigma(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \quad \rightarrow \text{Rectified linear unit(ReLU)} \tag{14a}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad \rightarrow \text{Sigmoid} \tag{14b}$$

$$\sigma(x) = \frac{1}{1 + e^{-2x}} \quad \rightarrow \text{tangent hyperbolic} \tag{14c}$$

$$\sigma(x) = \arctan x \quad \rightarrow \text{inverse tangent} \tag{14d}$$

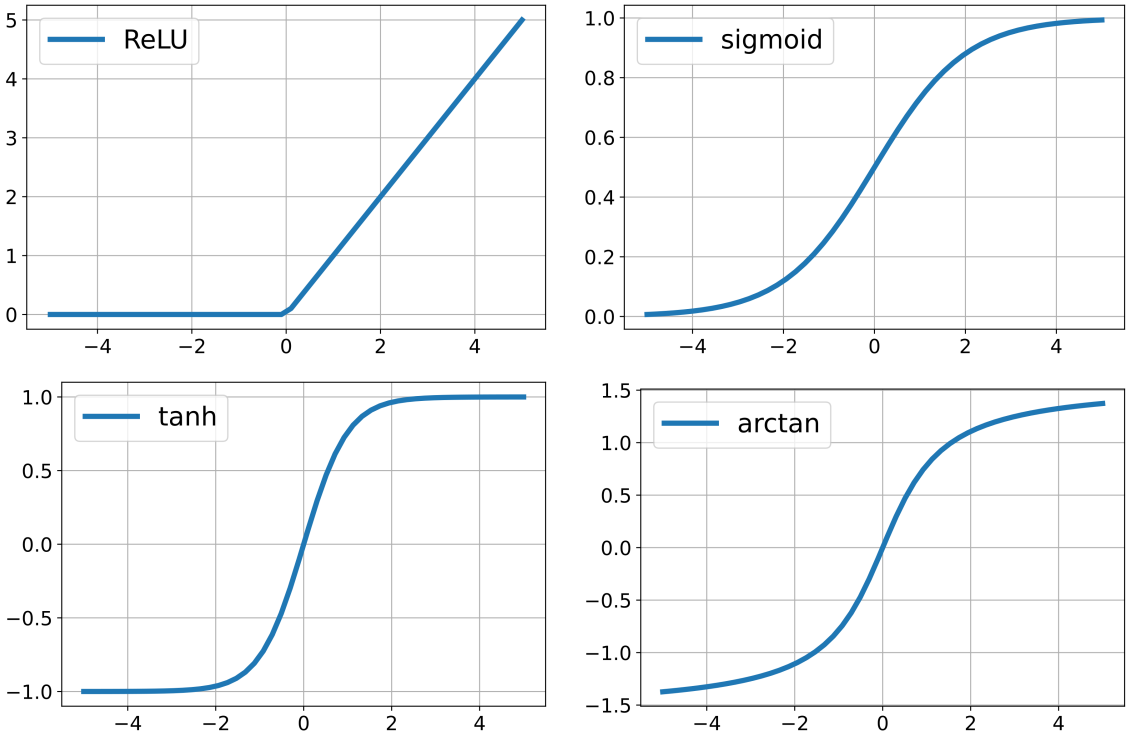

Figure 3: Some of the most common activation functions

The graphs of these activation functions are displayed in Fig-3. Although the choice of activation functions seems like a subtle detail, it can directly effect the model performance and learning efficiency. Here are few points to pay attention;

1. Let's consider the equation (8) with the sigmoid function in (14b). As we can see from the figure above, the range of σ 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.

2. In the light of the same discussion, let's take the equation (11) 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 (8) can balance the saturation problem and the model may not suffer from vanishing gradient. Thus this phenomena requires careful mathematical analysis.

3. 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) defined in (14a). It is always positive and its derivative is 1 for all positive values.

Now, let's discuss the cost function $\boldsymbol{C}$. Assume that we have data set containing N_p data points of the forms

$$\mathcal{D} = \{(\vec{x^i}, \vec{y^i})\}_{i=1}^{N_p} \tag{15}$$

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

$$C = \sum_{i=1}^{N_p} C^{(i)}(\vec{y^i}; a^{(i)L}) \tag{16}$$

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

$$C = \frac{1}{2N_p} \sum_{i=1}^{N_p} \|\vec{y^i} - a^{(i)L}\|^2 \tag{17}$$

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 (16), we have

$$\nabla_a C = \sum_{i=1}^{N_p} \nabla_a C^{(i)} \tag{18}$$

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,

$$\nabla_a C = \sum_{i=1}^{N_p} \nabla_a C^{(i)} \approx \sum_{i=1}^{M} \nabla_a C^{(i)} \tag{19}$$

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.
Let's consider sigmoid activation function defined in (14b). If we rewrite (7) for a generic data point $(\vec{x}, \vec{y})$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) = (a_j^L - y_j)\sigma'(z_j^L) \tag{20}$$

If we employ this result in (13) for the last layer we obtain

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}\delta_j^L = a_k^{L-1}(a_j^L - y_j)\sigma'(z_j^L) \tag{21}$$

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 (21). Let's define so called *cross entropy cost function* along with sigmoid activation. Again for a generic point $(\vec{x}, \vec{y})$

$$C = -\sum_j \Big[y_j \ln(a_j^L) + (1 - y_j) \ln(1 - a_j^L) \Big] \tag{22}$$

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

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

If we plug this result into (21), we obtain

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}\delta_j^L = a_k^{L-1}(a_j^L - y_j) \tag{23}$$

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

Assume that instead of using an activation function σ in the output layer, we replace it with another function defined as

$$a_j^L = \gamma(z_j^L) = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{24}$$

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,in image classification problem, we label different categories with integer 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 function 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.

One final note is about the implementation aspect of neural networks. Starting from the construction to training and evaluation, we need a robust framework to efficiently and easily carry out these tasks. For example, even small scale neural net can have millions of parameters. This means the forward pass includes the multiplication of large matrices while the back-propogation must account for derivative of extremely complex structures. Therefore, our next step will be to learn how to build and train neural networks in such a framework. We will be using **Pytorch** [3] in the rest of this course.

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.

[3] https://pytorch.org/.