Assignment-12 : Neural Networks

Harshvardhan Patidar
Department of Artificial Intelligence
Indian Institute of Technology Hyderabad
ai24btech11015@iith.ac.in

1 Deep Learning

Deep learning is a family of techniques of machine learning in which the hypotheses are represented by complex algebraic circuits, with configurable connection strength of each connection. The word "deep" is to refer to the fact that the circuits are organized into many layers, or simply the computation path of input to output consists of a lot of steps. One of the very first implementations of deep learning was to make an artificial model network of neurons like in the brain, in the form of computational circuits by McCulloch and Pitts in 1943. This is in fact the reason why the networks trained by the technique of deep learning are called neural networks.

Now the question arises, why Deep learning has advantages over the earlier discussed linear techniques. The reason is that even though the linear and logistic regression can take in a large number of inputs, the computation path is very short, and each input contributes individually to the output. This causes their expressive power to become limited. Which causes it to fail to represent the real-word complex scenarios.

Decision lists and decision trees also don't fulfill the criteria as even though they allow long computation paths, the number of input vectors is relatively small. Deep learning takes advantage here by making all the input variables interact with each other in a complex manner. The models developed by this method are sufficiently expressive to represent the real-word complex data.

1.1 Simple Feedforward Networks

The feedforward networks have connections only in one direction. Each node takes the input, computes the corresponding output and then passes that output to the next node in the network. No closed loops are formed in such networks. Whereas, in a recurrent network, the intermediate or final outputs are feeded back into its inputs. This arranges the signal values in a dynamical system maintaining its own internal state and memory.

Boolean circuits are a good example of feedforward networks, in which the inputs and the outputs are limited to 0 and 1. In neural networks, typically the input values are continuous and produce continuous outputs. In such models, some of the inputs play the roles of parameters. The model learns by configuring these parameters, so that the network best fits the training data.

1.1.1 Networks as complex functions

Each node in the network is termed as an unit. It follows the traditional design given by McCulloch and Pitts, in which a unit calculates the weighted sum of the inputs from its previous nodes, then applies some non-linear function to produce the output. Let a_j denote the output of unit j and let $w_{i,j}$ be the weight corresponding to the link from unit i to unit j, then

$$a_j = g_j(in_j) \equiv g_j(\sum_i w_{i,j} a_j)$$

where g_j is the nonlinear activation function, for unit j an in_j denotes the weighted sum of the inputs to the unit. As we did earlier, we will add an extra dummy unit 0 with a fixed value of 1 and a weight $w_{0,j}$ for that input. This makes our total weighted input remain non-zero. The previous equation can be rewritten as

$$a_j = g_j(\mathbf{w}^\top \mathbf{x})$$

It is important to note that the activation function g_j is nonlinear because if it were not, then any composition of the units and functions would only represent the linear functions and hence have limited expressive power. The universal approximation, states that a network with as low as two layers of computational units can approximate any continuous function to an arbitrary degree of accuracy.

A great variety of activation functions are used depending on the type of problem. Some of them are :

- The logistic or the **sigmoid** function which we discussed in the previous assignment on logistic regression
$$\sigma(x) = 1/(1 + e^{-x})$$
- The **ReLU** function. (abbreviated form of rectified linear unit):
$$\text{ReLU}(x) = \max(0, x)$$
- The **softplus** function, a smoother version of the ReLU function:
$$\text{softplus}(x) = \log 1 + e^x$$
It is important to note that the sigmoid function is the derivative of the softplus function.
- The **tanh** function
$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

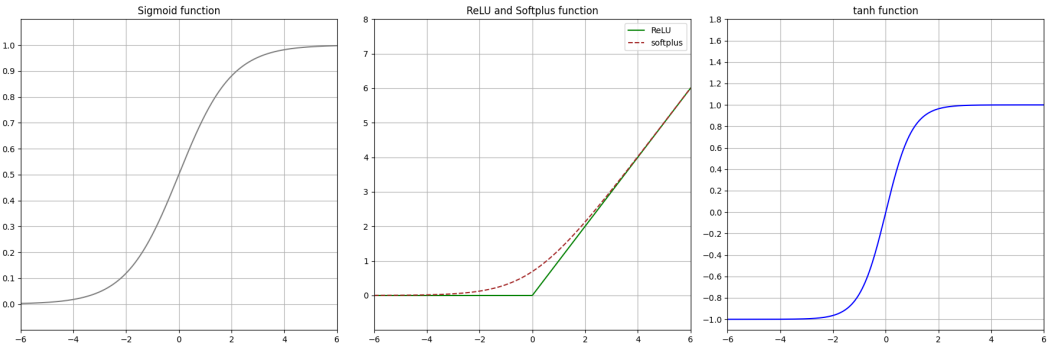

Figure 1: Common Activation Functions

An expression for the output $\hat{y}$ can be written as

$$\begin{aligned}\hat{y} &= g_5(\text{in}_5) = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\ &= g_5(w_{0,5} + w_{3,5}g_3(\text{in}_3) + w_{4,5}g_4(\text{in}_4)) \\ &= g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2))\end{aligned}$$

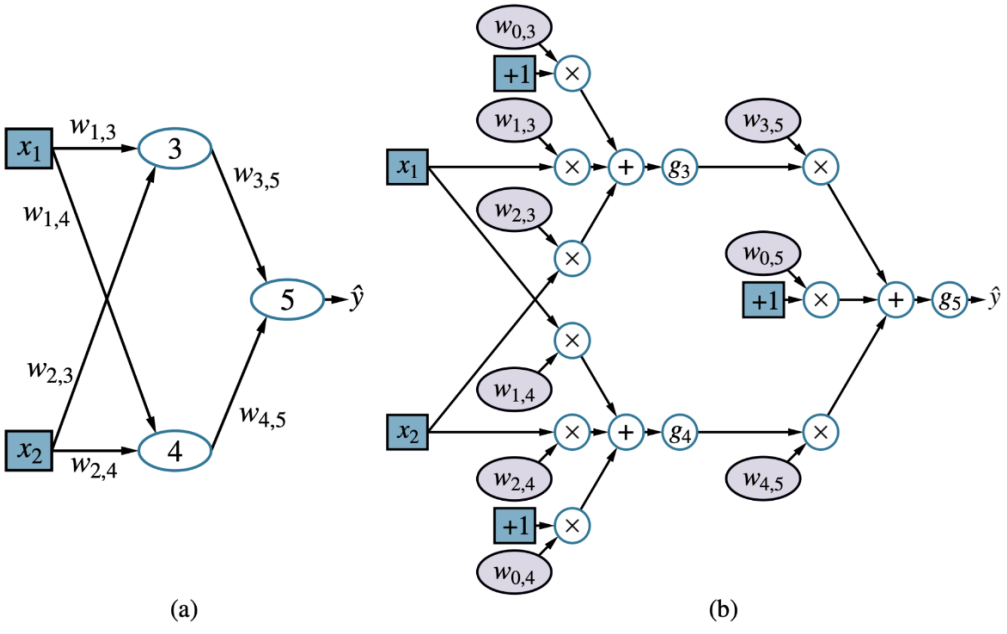

Figure 2: Graph of neural network

The general way (b) to represent the network is called the computation graph or the dataflow graph. These are circuits in which each node represents an elementary computation. To represent the network in matrix form to make it easy to work on in computers, we denote **W** to denote the weight matrix. For the network in figure, $\mathbf{W}^{(1)}$ denotes the weight of the first layer $(w_{1,3}, w_{1,4}, \ldots)$ and $\mathbf{W}^{(2)}$ denotes the weights in the second layer. Let $\mathbf{g}^{(1)}$ and $\mathbf{g}^{(2)}$ denote the activation function in the first and second layers, then the network can be written as

$$\mathbf{h}(x) = \mathbf{g}^{(2)}\left(\mathbf{W}^{(2)}\mathbf{g}^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x}\right)\right).$$

This expression corresponds to a computation graph, much simpler than in Figure 2(b). The graph in Figure 2(b) is relatively very small and shallow, but the same concept is applied in large and more complex models. The figure is also fully connected, which means that every node in each layer is connected to every node in the next layer.

1.1.2 Gradients and Learning

In the linear regression, we used the gradient descent algorithm to find the best hypothesis. We can apply the same algorithm here to find the best model. For the weights leading into the output layer, the process is identical to what we did earlier, but for the weights leading to the hidden layer, we need to use the chain rule to get to the gradient. We will consider the squared loss function (L_2)

$$Loss(h_{\mathbf{w}}) = L_2(y, h_{\mathbf{w}}(\mathbf{x})) = \|y - h_{\mathbf{w}}(\mathbf{x})\|^2 = (y - \hat{y})^2$$

The gradient calculations are below

$$\begin{aligned}
\frac{\partial}{\partial w_{3,5}} Loss(h_{\mathbf{w}}) &= \frac{\partial}{\partial w_{3,5}}(y-\hat{y})^2 \\
&= -2(y-\hat{y})\frac{\partial \hat{y}}{\partial w_{3,5}} \\
&= -2(y-\hat{y})\frac{\partial}{\partial w_{3,5}} g_5(in_5) \\
&= -2(y-\hat{y})g_5'(in_5)\frac{\partial in_5}{\partial w_{3,5}} \\
&= -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{3,5}}(w_{0,5}+w_{3,5}a_3+w_{4,5}a_4) \\
&= -2(y-\hat{y})g_5'(in_5)a_3.
\end{aligned}$$

This calculation was easy as $w_{0,5}$ and $w_{4,5}a_4$ do not depend on $w_{3,5}$. The more complex case might involve weights such as $w_{1,3}$, which are not directly to the output layer, so we need to apply chain rule for this one more time.

$$\begin{aligned}
\frac{\partial}{\partial w_{1,3}} Loss(h_{\mathbf{w}}) &= -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{1,3}}(w_{0,5}+w_{3,5}a_3+w_{4,5}a_4) \\
&= -2(y-\hat{y})g_5'(in_5)w_{3,5}\frac{\partial a_3}{\partial w_{1,3}} \\
&= -2(y-\hat{y})g_5'(in_5)w_{3,5}g_3'(in_3)\frac{\partial in_3}{\partial w_{1,3}} \\
&= -2(y-\hat{y})g_5'(in_5)w_{3,5}g_3'(in_3)\frac{\partial}{\partial w_{1,3}}(w_{0,3}+w_{1,3}x_1+w_{2,3}x_2) \\
&= -2(y-\hat{y})g_5'(in_5)w_{3,5}g_3'(in_3)x_1.
\end{aligned}$$

Now, we will try to understand the term back-propagation.

- **For unit 5**: The error is given as $\Delta_5 = 2(\hat{y}-y)g_5'(in_5)$ and the gradient w.r.t $w_{3,5}$ is $\frac{\partial}{\partial w_{3,5}} = \Delta_5 a_3$. This means the error is scaled by the input a_3. If both Δ_5 and a_3 have the same sign, increasing $w_{3,5}$ increases the error, while if it has opposite signs then it is decreased.
- **For unit 3**: The error at unit 5 is propagating backward, contributing to error at unit 3. This is given by $\Delta_3 = \Delta_5 w_{3,5} g_3'(in_3)$. The gradient with respect to $w_{1,3}$ is $\frac{\partial}{\partial w_{1,3}} = \Delta_3 x_1$, evident of the fact that the error at unit 5 is scaled by $w_{3,5}$, the activation gradient at unit 3, and the input x_1.

This phenomenon gives rise to the term back-propagation as the error at output is being passed back through the network

The derivations we did above are just for illustration purposes and we don't need to do them ourselves while training a model. All such gradients can be calculated by the technique of automatic differentiation. All the major packages provide automatic differentiation. So the user does not need to do such complex computations, and can be more focused on training the model better.

2 Computation Graphs for Deep Learning

We will look at how to put together computation graphs and tune the weights by computing the gradient to make our model best-fit the data.

2.1 Input Encoding

The input and output nodes are those which are connected directly to the input and output. For factored data, the encoding is pretty straightforward as each training example contains values of the n input attributes. But image data doesn't fit well in the category of factored data. If we have a fully connected model for these, then the importance of adjacency of pixels will be lost. For this reason, in the hidden layers, they aren't completely interconnected, but connected to only a few adjacent pixels. For categorical attributes, where multiple possible attributes are possible, we represent it by d separate input bits, with the bit of chosen one being 1 and the others being 0. We can't use $1, 2, 3 \ldots$ to represent the options as the difference between 1& 2 and that between 1& 3 are different but this doesn't make sense.

2.2 Output Encoding

After output encoding, now we need to represent the output into actual meaningful categories. One example could be again using reverse of one-hot encoding. More importantly, for the loss, we will treat the output values as probabilities instead of values (like in the negative log likelihood loss function). In Deep learning we commonly use the term cross-entropy and try to minimize this. It is denoted as *H(P, Q)* is a measure of dissimilarity between *P* and *Q*. Here *P* is the true distribution over the training examples, whereas *Q* is for the predictive hypotheses.

Sometimes, to denote the data in the form of probabilities, we use the softmax layer, which gives the probability according to the given inputs. It outputs a vector of d values for a given input vector $\mathbf{in} = \langle in_1, \ldots, in_d \rangle$, where the k^{th} element is

$$\text{softmax}(\mathbf{in})_k = \frac{e^{in_k}}{\sum_{k'=1}^{d} e^{in_{k'}}}$$

The softmax outputs a vector of non-negative values which sum up to 1. These probabilities are then used to minimize the cross entropy and hence to train the model.