SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

01.112 Machine Learning, Fall 2019
Lecture Notes for Week 6

11. Neural Networks and Deep Learning

Last update: Saturday 19th October, 2019 19:10

1 From Logistic Regression to Neural Network

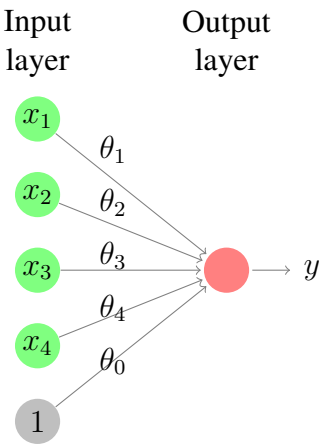
As we have discussed in the last class, logistic regression model can be viewed as a simple neural network. In fact, not only logistic regression, but also many other models that we have learned can be regarded as a neural network, such as the perceptron algorithm that we have learned at the early stage of this course. So, what is really a neural network? And what is deep learning?

Recall in logistic regression, the output is given by the formula

$$h(x) = \frac{\exp(\theta \cdot x + \theta_0)}{1 + \exp(\theta \cdot x + \theta_0)} = \sigma(\theta \cdot x + \theta_0)$$

where *logistic function* $\sigma(s) = \frac{\exp(s)}{1+\exp(s)}$ varies continuously between 0 and 1.

Let us assume x is a d-dimensional vector where $d = 4$. We can use the following plot to visualize the input and output of the function h:

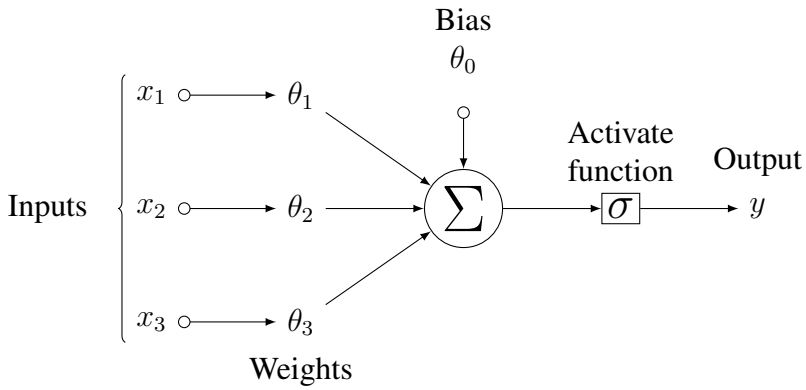


The resulting graph is a neural network. Specifically it is a single-layer feedforward neural network. As we can see from this graph, the neural network consists of an input layer and an output layer, where there are 5 inputs to the neural network, and there is only one output from the neural network. Each input corresponds to one dimension from the x vector, and there is one

more input node that always takes in the value 1, which is regarded as the *offset* or the *bias* term. Along the edges, there are weights. The sum z is calculated based on the multiplications of the x values and their corresponding weights θ. The output y is then calculated as $\sigma(z)$ where σ is the *activation function*.

1.1 A Neuron in a Neural Network

Now let us take a closer look at the above procedure, which defines how a neuron in a artificial neural network works. We use the following picture to illustrate the process again. As we can see, each neuron requires the following components: 1) inputs, 2) weights (including the weight associated with the bias term), 3) an activation function. Roughly speaking, we can regard the activation function as a "gate" which controls when the neuron is fired. The weighted combination of the inputs (as specified by the weights) serves as the input to the activation function. The output y is the value returned by the activation function.



Note that the activation function is a user-specified function. There are several popular choices for this function. Some widely used choices can be found in Figure 1. The first is the sigmoid function that was used in our logistic regression model. The second is the hyperbolic tangent function $\tanh$, which is defined as follows:

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

This activation has a very similar shape as the sigmoid function. The difference here is in this function the output ranges from $(-1, 1)$ rather than $(0, 1)$.

The next two activation functions are called ReLU (rectifier linear unit) and its leaky version, which are defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{2}$$

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases} \tag{3}$$

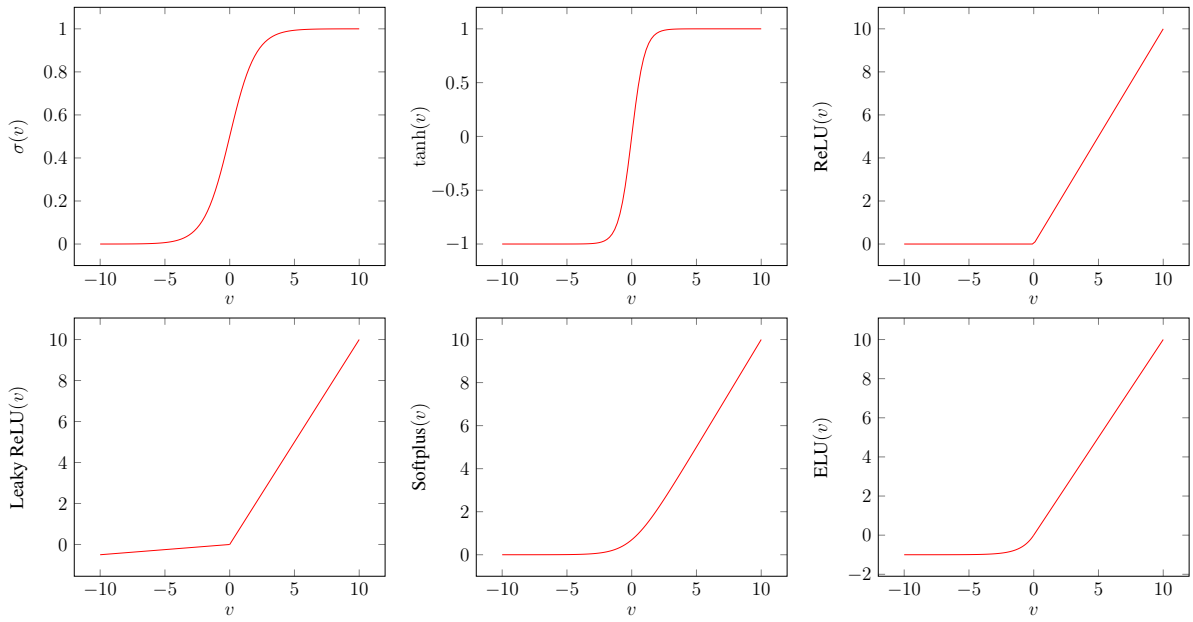


Figure 1: Example Activation Functions

As we can see, the leaky ReLU gives a small non-zero gradient when the unit is inactive, which may be helpful in learning. A smooth version of ReLU is the softplus function, which is defined as:

$$\text{softplus}(x) = \log(1 + \exp(x)) \tag{4}$$

Finally, we may also consider the exponential linear unit (ELU) activation function, which is defined as:

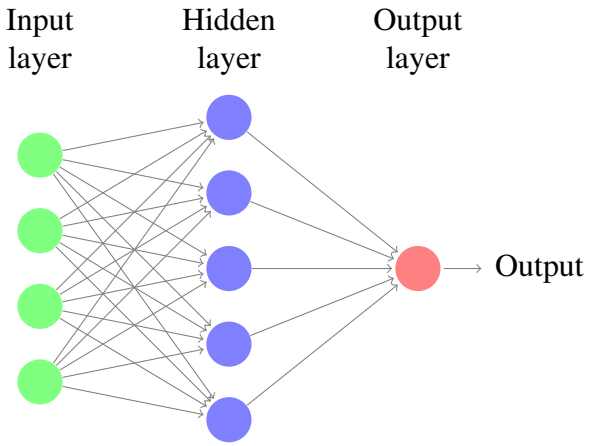
$$\text{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ a(\exp(x) - 1) & \text{otherwise} \end{cases} \tag{5}$$

1.2 Deep Neural Network

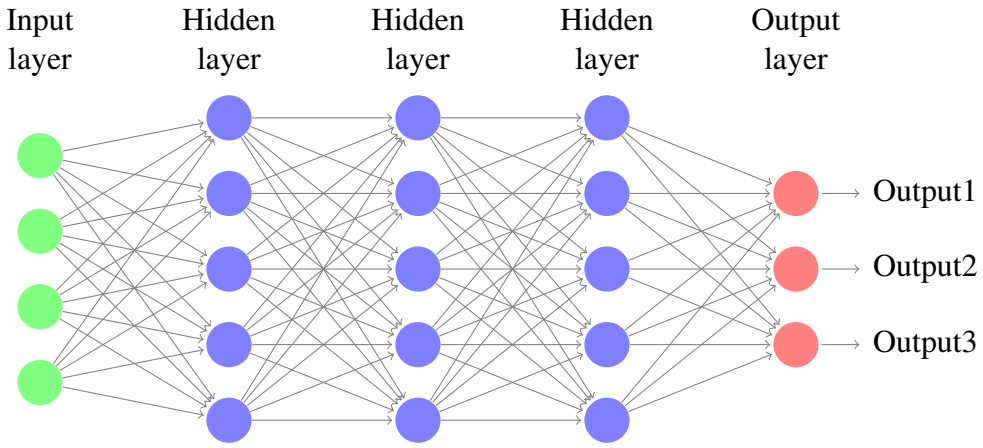
We can also consider multi-layer neural networks, or sometimes called *deep neural networks* if the number of layers is large.

Let us first look at the following example that involves one single hidden layer (for the following discussions, again, we ignore the bias node, without loss of generality. It's easy to add it back). Now, instead of directly connecting the input nodes to the output nodes, we ask them to go through some additional hidden nodes that appear between the input and output layers. This results in an additional hidden layer. Such a hidden layer will allow us to capture some interdependencies between the input variables. Hopefully, when the data is sufficient, we will be able to train a good model to learn this layer well. In that case, this hidden layer essentially gives us a mapping from the

original feature space into another new space before we applying the classification model. Does this remind you something? Yes, it is somewhat like what we did for learning non-linear SVM (using the kernel trick), where we map the data from the original feature space to another feature space. Here in neural networks, it allows us to learn such a projection explicitly from the data!



As we can see, the hidden layers are responsible for learning good representations for the inputs. To increase the expressiveness of the function that can be possibly learned from the data, we can also consider more hidden layers, resulting in the following multi-layer neural network. As we discussed in class, theoretical analysis shows that a neural network with multiple layers will be able to potentially help us learn from the data better projections from the original input space to another space:

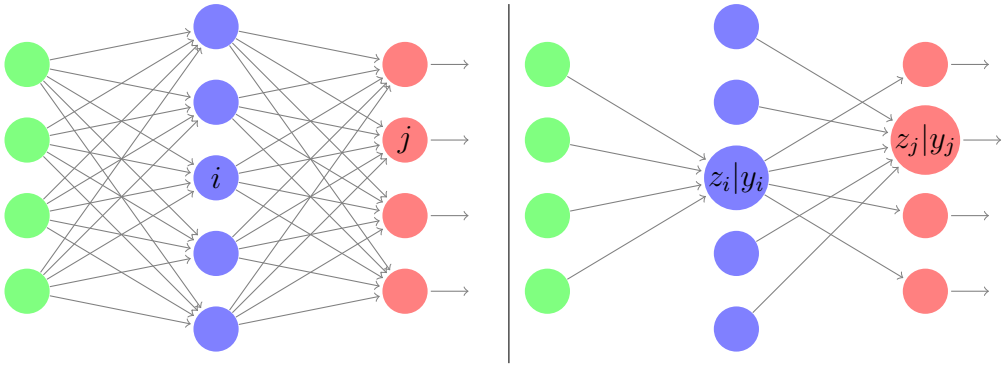


2 Learning with Backpropagation

In logistic regression, we have used stochastic gradient descent to learn the model parameters (or the weights). In a general neural network that involves multiple layers, how do we learn the model parameters?

Turns out there is a general procedure for us to do so. The procedure was invented many years ago, and earlier it was not shown effective in learning weights for a multilayer neural network. Only recently, after 2006, it was shown that the procedure is effective. This is partially because of the advancement of machines and the availability of data we have, and is also because we found better ways to initialize (and control) the weights and we have designed good activation functions. The general procedure for updating the weights is called *backpropagation*.

The underlying idea for backpropagation is simple: applying the chain rule for calculating the gradients for the weights. Let us take a look at the following example, which is part of a (potentially large) neural network.



Let us assume there is an error function E, and we would like to minimize the error associated with the neural network. Let us consider the weight defined between two nodes i and j in the neural network, which is w_{ij}. We are interested in $\frac{\partial E}{\partial w_{ij}}$. Let us assume the input-output pairs for i and j nodes are (z_i, y_i) and (z_j, y_j) respectively. In other words, we have $y_i = \sigma(z_i)$ and $y_j = \sigma(z_j)$. As an example we assume the activation functions are the same and are both sigmoid functions.

Based on the above, we will be able to calculate the following:

$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j} \tag{6}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j} \tag{7}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j} \tag{8}$$

Now, we have the following recursion:

$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j} = y_j(1-y_j)\sum_k w_{jk}\frac{\partial E}{\partial z_k} \tag{9}$$

where $w_{j,k}$ is the weight associated with the edge between the nodes j and k, with k being any node from the layer after j.

We can introduce the following notation:

$$\delta_j = \frac{\partial E}{\partial z_j} \tag{10}$$

, which leads to:

$$\delta_j = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j} = y_j(1-y_j)\sum_k w_{jk}\delta_k \tag{11}$$

From here we will be able to figure out how to get the gradients for all the weights using stochastic gradient descent.

Let us assume the final error function associated with the neural network is given as:

$$E = \frac{1}{2}\sum_t ||y_t - y_t^*||^2 \tag{12}$$

where y_t^* is the desired (true) output for the t-th output node, and y_t is the output predicted by the neural network for the t-th output node.

Backpropagation

1. Randomly initialize the weights (with small non-zero values).
2. Perform a forward pass to calculate the outputs (y values) from each neuron layer by layer.
3. For each output neuron t, calculate δ_t as follows:

$$\delta_t \leftarrow -y_t(1-y_t)(y_t^* - y_t) \tag{13}$$

4. For each hidden neuron j, calculate δ_j as follows:

$$\delta_j \leftarrow y_j(1-y_j)\sum_{k:j\to k} w_{jk}\delta_k \tag{14}$$

   where $j \to k$ means there is a direct edges from the node j to the node k.

5. Update the weights:

$$w_{ij} \leftarrow w_{ij} - \eta\delta_j y_i \tag{15}$$

   where η is the learning rate.

6. Return to 2, unless you are satisfied.

Learning Objectives

You need to know:

1. What is a single layer and multi-layer neural network.
2. How is each neuron defined in a neural network.
3. What is backpropagation and the idea behind backpropagation for learning the weights for a neural network.