MulitLayer Perceptrons
AI/ML Guild Lab #2b
November 3, 2021

1 Multilayer Perceptrons

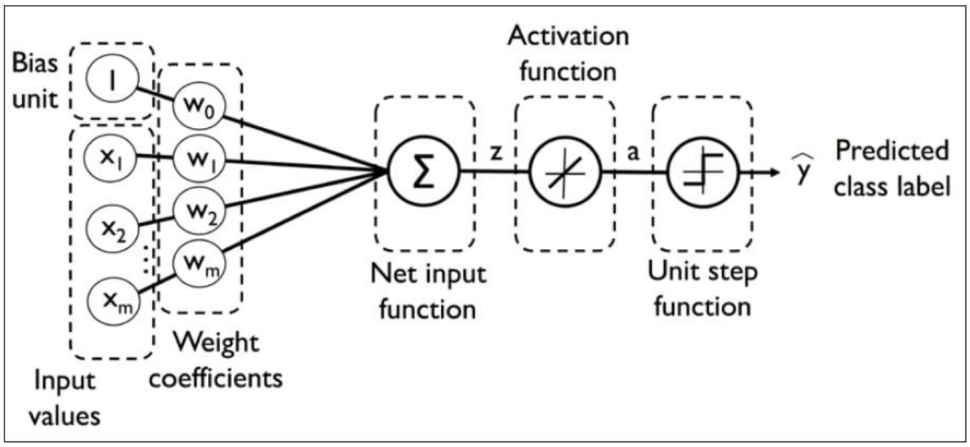
It's really a misnomer to call Multilayer Perceptrons *multi-layer*. It's not a multiple layers of perceptrons but instead it is a single network of multiple "neuron-like" processing units but not every layer in the network is a perceptron.

Recall that the perceptron equation:

$$Z = \sum_{i=0}^{N} W_i x_i + b$$

What does it mean to extend this equation with additional layers?

A single layer perceptron has 2 layers, the input an d the output layer. A Multilayer perceptron has at least 3 layers; the same input and output but also a layer in between known as a "hidden layer".



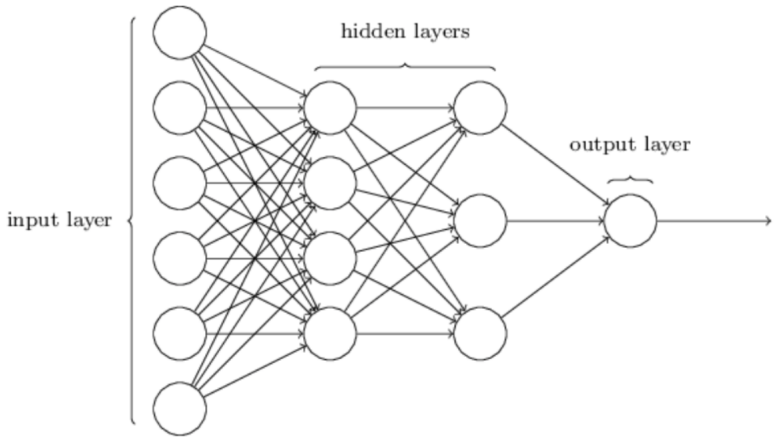
2 History

After Rosenblatt's Perceptron was discredited in the 60s by a book by Marvin Minsky, neural networks weren't really studied again until the 1980s. Dudes

named Rumelhart, Hinton, and Williams introduced/discovered a new training algorithm for what they termed 'Multi-Layer Perceptrons'. Rumelhart was a mathematical pychologist working at UC San Diego. He led a research group on neural networks, picking up where Rosenblatt left off after the Minksy book pretty much destroyed public funding for this type of research.

Rumelhart wanted to train a neural network with multiple layers. Rumelhart came up with an idea called "Back Propagation" but it was Hinton that used it in conjunction with the training model Rumelhart developed for sigmoidal model training.

This is what is typically referred to as a multilayered perceptron today.



3 The Definition of a MLP

Formally, a multilayered perceptron is one where these conditions are present:

- a linear function that aggregates the input values
- a sigmoid function (the activation function)
- an output function (i.e. a threshold function for classification)
- a loss function that computes the overall error of the network
- a learning procedure to adjust the weights of the network, i.e., the "back propagation" algorithm

4 The Math

4.1 Linear Function

The linear function is very similar to the single layer perceptron. It is a weight sum of the inputs plus a bias. The difference is that the weights are matrix and no longer a vector.

$$z_m = \sum_{i=0}^{N} W_{m,n} X_m + b$$

4.2 Sigmoid Function

Every element of the Z_m becomes an input to the sigmoid function.

$$a_m = \sigma(z_m) = \frac{1}{1 + e^{-z_m}}$$

The output of the sigmoid function is a vector of size m where every element is a unit in the hidden layer.

a stands for activation! Which is typically how the output of the hidden layer is described. Using the sigmoid funciton is somewhat arbitrary, you can use other non-linear functions here like tanh or $ReLU$.

4.3 Output Function

Similar to single layer but adjusted for sigmoid.

$$f(a) = \begin{cases} 1, & a \geq 0.5 \\ 0, & \text{else} \end{cases}$$

4.4 Cost function

We need a measure of how well the network is performing. We did this with the single layer too. This is used in the training step. This often goes by different names in the literature such *objective function*, *loss function*, or *error function*.

Typically this cost function is tuned for your problem at hand. The one that Rumelhart came up with was

$$E = \frac{1}{2} \sum_k (a_k - y_k)^2$$

This is commonly known as the sum of squared errors function. It's pretty basic but often gets the job done.

4.5 Training Algorithm: Forward & Back Propagation

Multilayer Perceptron training is composed of two steps. Forward propagation where information flows forward in the network to compute predictions and the error. And then the back propagation stage where the error derivatives is calculated and used to update the weight matrix.

The forward propagation stage is just chaining the linear, sigmoid and output functions (just like the single layer!).

The back propagation part is more complicated but it's basically asking "how does the error change if we tweak the weights"? This question has additional questions that need answers first. How does the error change when we change the activation a_m a bit? How does the activation change if we fiddle with linear function a bit?

It takes a fair amount of calculus to derive all of this. I'll leave it as an excercise to the reader. The outcome of that derivation can be solved for using the gradient descent algorithm.