Neural Networks

Perceptron

A perceptron (node) is a computational unit, based on a biological neuron, which accepts inputs, applies a transfer function (sum of inputs and weighted values), then passed the sum of those into an activation function which defines the output.

A single perceptron is often called a single layer neural network. The main issue with single layer neural networks is that they can only solve problems which are linearly separable. (add example)

Multilayer Perceptron

In multilayer perceptrons (MLP), nodes are organised into layers to create a network. MLPs have an input layer which channel the input variables into a hidden layer comprimised of mutliple nodes. The nodes process the variables then output to the output layer which is also built up of nodes directly outputs to the the output variables (classification nodes).

MLPs have the ability to represent convex regions, being able to draw shapes around examples in high-dimensional space that can then separate and classify them.

# Network Architecture

Topology

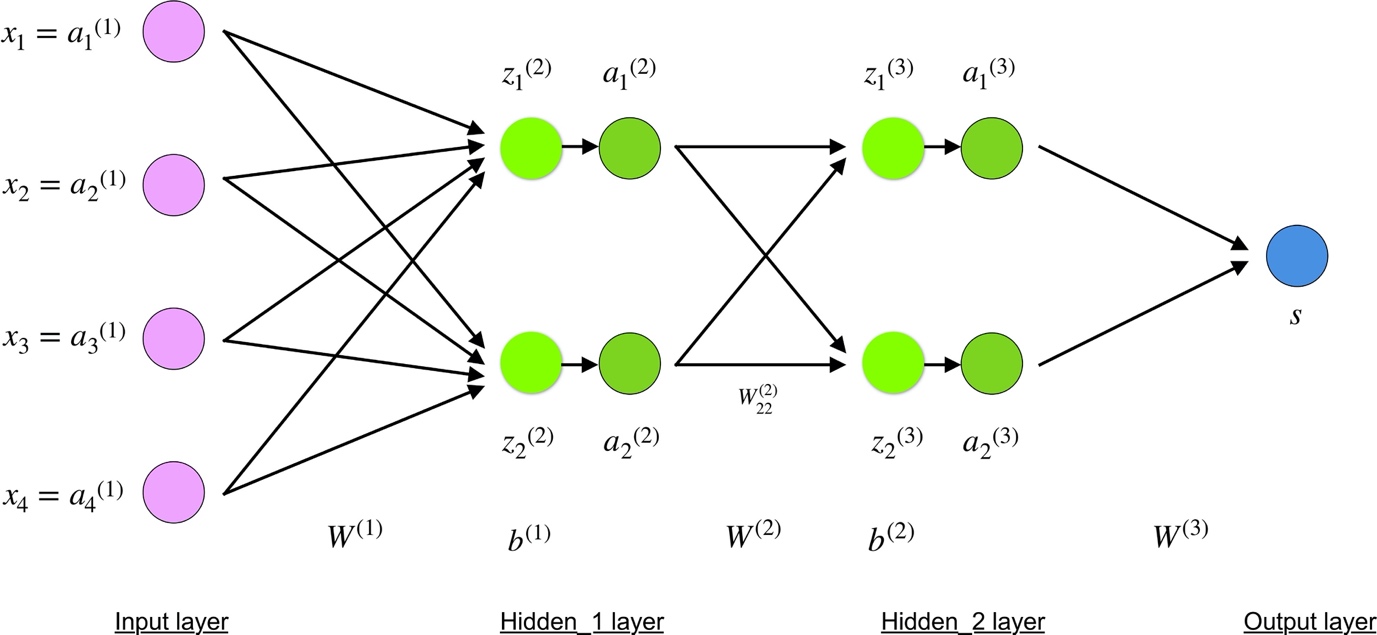
Size - Number of nodes in the model

Width - Number of nodes in a specific layer

Depth - Number of layers in a neural network

Capacity - Type of functions that can be learned by a network configuration (representational capacity)

Architecture - Specific arrangement of layer and nodes in network



Number of Nodes and Layers

Neural networks have two main hyperparameters which control the topology of the network, number of nodes and number of hidden layers. Since both of these require specifying when the network is created, first they must be configured. Unfortunatley, currently there is no analytical way to create the optimum parameters, therefore the parameters must be calculated through systematic experimentation.

However, deeper networks tend to model problems better than wider ones.

# Layer Architecture

Each layer is built up of a number of nodes, each with:

* Weights for each input value
* Bias for entire node
* Transfer function to sum weights and inputs
* Activation function to normalise summed weights

Transfer Function

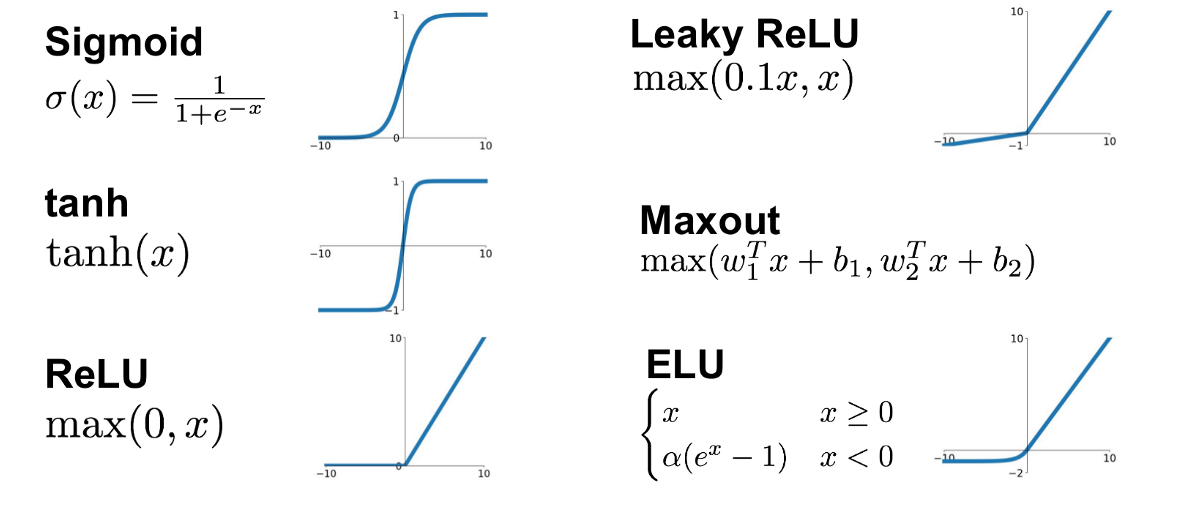
Sums all input values with weights then passes on to the activation function. There is the option to add a bias, which is a value added or subtracted from the sum total, giving certain neurons a bias on their value of activation.

The sum of the of activation in a layer is performed simply by calculating the product of the weights matrix with the input matrix, then adding the bias.

The superscript value defines what layer the variable is in, i.e. would be an input variable and would be an activation variable from outputted from layer 1.

Activation Function Selection

Take the activation value from the transfer function and weight it to a value between 0 and 1.



**Sigmoid**

* Oldschool

**ReLU**

* Easier to train than Sigmoid

Activation Function Application

Activation functions are applied to the transfer values and then outputted:

This full equation can be represented by:

# Output Layer

The output layer is very similar to the hidden layers of the network, however firstly it directly outputs to the output variables, and secondly it has an additional output function which normalises all the values of the output variables.

Output Nodes

Each output layer node is almost same as a hidden layer node, however it doesn’t apply an activation function and instead directly outputs the weighted sum to an output variable, or in other words, a classification of data.

Output Function

Once the list of output values is created from the output layer nodes, it will contain a list of weighted balues. This list still requires narrowing down to a most probable answer, however since the sum of the output variables will vary between input data, a function is required to find a weighted max of all the ouput variables.

A softmax function is generally used on ouput vaiables to add weight to each:

Each output variable then equals the exponental function of its weighted value divided by the sum of all the exponental functions of the weighted values.

# Training the Net

The most important part of creating a neural net is training it (letting it learn off a sample dataset). A net learns from data which already has correct classification values assigned to it, the input values are passed into the function, then dependent on the output of the net, the weights in the net is modified using a cost function.

Setup

* Number of nodes and layers guessed initally dependant of computational power available and complexity of problem
* Weights and biases are initalised randomly since it is hard to manually classify weights on a complex problem
* Setup can be modifed once network is trained and accuracy calculated

Datasets

* Structured dataset allow for training much quicker than random datasets

Cost Function

* Run test data through feedforward network
* Calculated the squared differences of the output values compared to the expected output values, which gives the cost vector
* Use backpropagation to change the weights and biases

Backpropagation

* Takes in values of cost function
* Calculates gradient of cost function
* Weights and biases into vector then step towards negative gradient of cost function

Optimisation

* Using simple backpropagation can create a good network, however optimisation is where good nets come from.

Step size proportional to the slope to stop from overshooting minimum of surface.

* Average cost for all training data shows the error of the function.

Generating Dataset in Python

To train and test a network, a dataset must be generated. Datasets must have well-defined properties such as linearly or non-linearity which allow you to perform specific alogirithm behaviour. Test dataset should be:

* Quickly generated

Visualising Network

The weights for each node can be visulised to see what each is looking for, this can interesting to see in image recognition, is the visualisation of the node is directly connected to the image itself.