**Multilayer feed-forward neural networks (MFNN)**

**Overviews**

MFNN is a classic machine learning algorithm that simulates the way human brain works. MFNN is very useful for studying large datasets due to its ability to tolerate noise data as well as to classify patterns on which they have not been trained. It is especially efficient for real world data, where we have little knowledge of the relationships between attributes and classes. Generally, MFNN requires a large amount of data in order to well perform.

Each MFNN consists of an input layer, one more hidden layers and an output layer. Layers are connected in acyclic graph. Each layer is made up of neurons. Neurons between two adjacent layers are pairwise connected, but neurons in one layer share no connection. No direct connection exits between input and output layer. Cycles are not allowed.

Inputs are fed into the neurons making up the input layer. The outputs produced by this layer are weighted and passed simultaneously to the first hidden layer. This hidden layer outputs are again weighted and input to an another hidden layer and so on. It is arbitrary how many hidden layers there should be, but normally we only use one. The weighted outputs of the last hidden layer are input to the output layer, where the prediction for the given tuples will be produced.

Neurons in the input layer are called input units. Neurons in the hidden layers and the output layers are called ‘neurodes’ or sometimes referred to as ‘output units’. The number of input units are not necessarily equal number of input units. There can be more or less number of hidden units than number of input or output units. Each output unit applies a nonlinear (activation) function to its input. The activation function will be described in section 4.7.1. The output is suggested as in the following function:

(11)

Where   
 f is the neural activation function   
 is output from each hidden node j, where node j precedes node k  
 is the weight of the connection between node j and k  
 is the bias of node k  
 is the output computed by node k

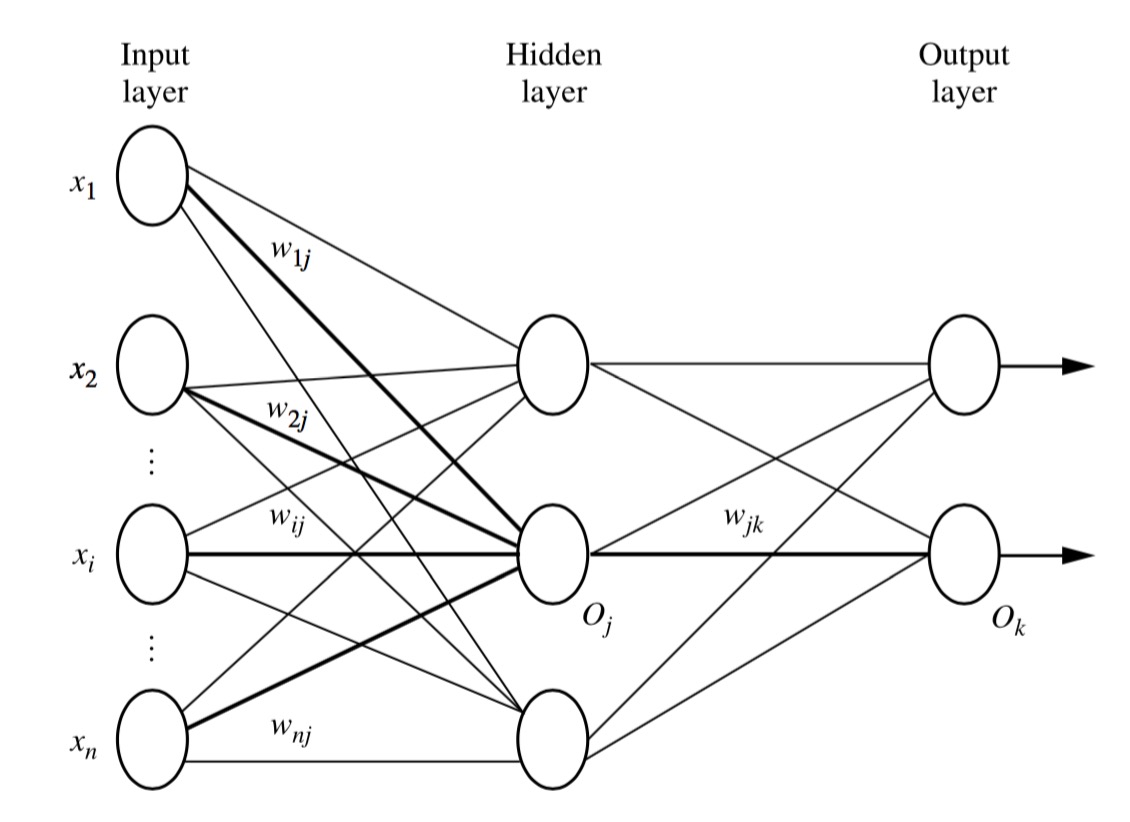
****

Figure 4.1.6.1. fully connected layers. Photo from Data Mining Concepts and Techniques, 3rd edition, Jiawei Han & Micheline Kamber

**Back propagation**

Back propagation is the learning algorithm of neural networks. Total error is the summation of error at each output unit, which can be calculated using the squared error function:

Where

The error rate at each output unit can thus be calculated from this total error:

This error is then propagated back to the networks:

With learning rate , this error will affect the corresponding weight by an amount of:

where

**Convolutional neural networks**

Back propagation is a process to adjust the learnt weight and bias by comparing the network’s prediction for each tuple with the known target value. The aim is to minimize the error between the network’s prediction and the known target value. The target value can either be a known class label or a continuous value. The weight and bias modification process is done in a backward direction, from softmax layer through fully connected, pooling and ReLU layers to the convolutional layer. The forward and backward process is repeated until all the weights in the network converges. The steps are as follows:

1. Initialize all weights and bias in the network with some small random values and choose a learning rate.
2. Propagate the inputs forward the networks using our initialized weights and bias until we reach the output layer.
3. Calculate error rate and consequently update the weight at each layer.
4. Repeat the process until when one of the following conditions is reached:

* All in the previous epoch are bellow some pre-specified threshold
* A pre-specified number of epochs has reached
* The percentage of tuples misclassified in the previous epoch is below some thresh- old

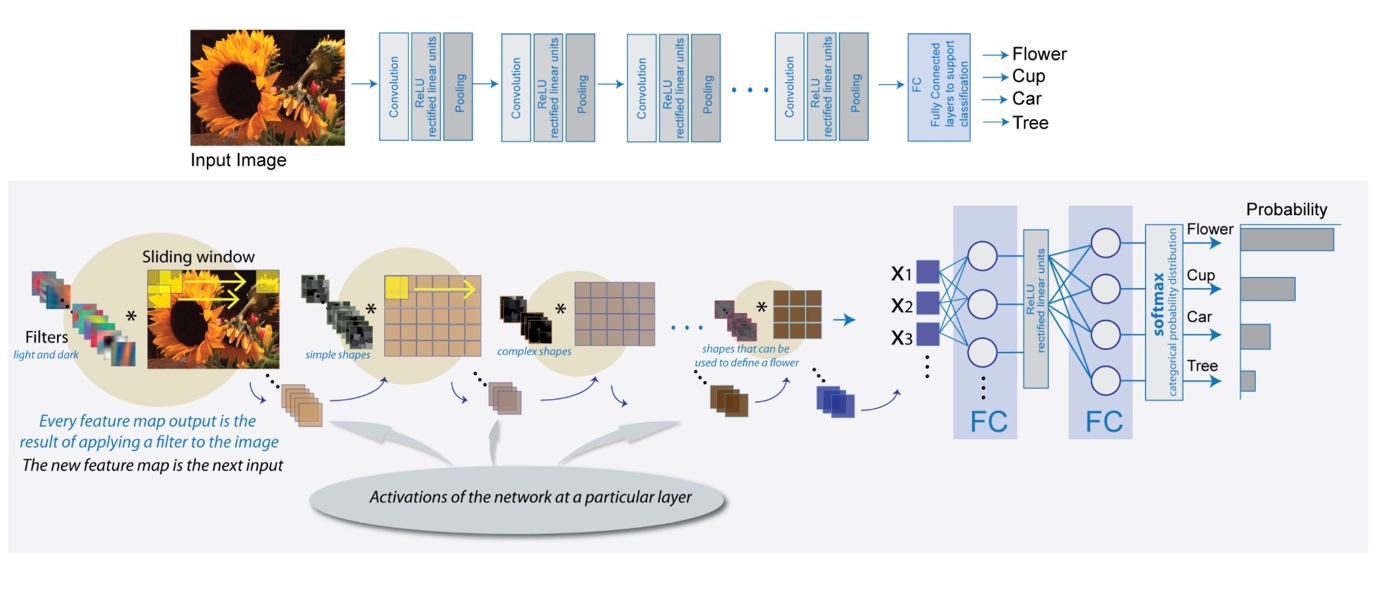


Figure layers of convolutional neural networks. Image by Mathworks, Convolutional neural networks documentation

**Softmax layer**

The total error(will be refer to later on as E) is the sum of errors for each output neuron and can be calculated using log-loss (sometimes called cross entropy) function:

where N is the number of output neurons, is the expected output of neuron k and is the real output of neuron k.

We will look at how the networks adjust its weight using by looking at effect on each layer.

**Fully connected layer (Neural networks)**

The real output at fully connected layer can be expressed in terms of net output by function () in section

Therefore, we can calculate

Each neuron at the output layer has a net value defined by the following function

where is the net output of neuron k at layer l, is the real output of neuron j at layer (l-1) where j = {1,2,3,…,m}, is the weight of the connection between neuron j and k, is the bias value of neuron k.

The amount of error that we want to adjust at each neuron is equal to:

(as a result of chain rule)

where N is the number of output neurons, o stands for output layer, is the error rate at neuron N in the output layer, is the net output of neuron N in the input layer, , is the real output of neuron N in the input layer.

We will propagate this error forward. Accordingly, this rate will affect the weight at neuron N by an amount of

(as a result of chain rule)

where is the amount of weight to adjust between neuron j and neuron , is the real output of neuron at output layer, is the net output of neuron at output layer, is the current weight between neuron j and neuron .

From function () and (), function () can be rewritten as:

where is the amount of weight to adjust between neuron j and neuron , is the current weight between neuron j and neuron is is the error rate at neuron N in the output layer, is the real output of neuron at hidden layer h1 where h1 precedes o.

To decrease the error, we then subtract this value from the current weight

where denotes the learning rate, which is normally set to 0.5

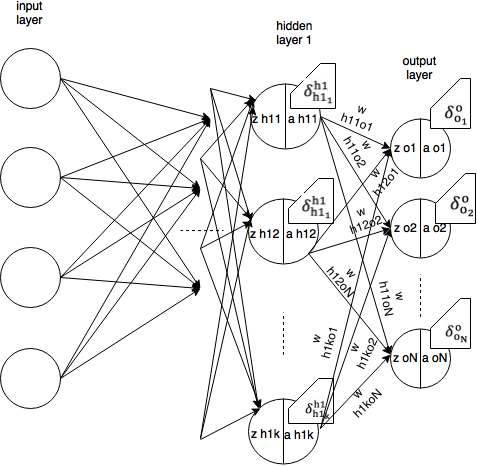


Figure fully connected layer

For every neuron at each hidden and input layer of the neural networks, we can calculate and by applying the same technique. Thus our calculation can be generalized as follows:

where is the error rate at neuron j in layer l, is the net output of neuron j in layer l, is the real output of neuron j in layer l , represents the total net output of all neurons in layer (l+1) that are pairwise connected with neuron j, m is the total number of neurons in layer (l+1), is the error rate at neuron i’ in layer (l+1) and is the weight between neuron j and neuron i’ where i’ = {0,1,…,m}.

We will propagate this error forward

where is the amount of weight to adjust between neuron i and neuron j, is the error rate at neuron j in layer l, is the real output of neuron i in layer (l-1) .

**ReLU layer**

The output for each neuron can be written as

where is the real output value of neuron i of ReLU layer (l-1). Thus for each output at layer ReLU layer, we want to adjust an error amount equivalent to

We need not to adjust weight at this layer because it has no weight.

**Max pooling layer**

The output for each neuron can be represented by

where is the the real output value of neuron at position (x,y) of max pooling layer l, is the real output value of neuron at position (x+p,y+q) of convolutional layer (l-1) where (p,q) is the position in a kernel of size kxk.

For each output pixel at max pooling layer, we want to adjust an amount of

We need not to adjust weight at this layer because it also has no weight.

**Convolutional layer**



Figure feedforward in CNN is identical with convolution operation. Image by Grzegorzgwardys

The net output of each neuron at convolutional layer can be represented by

as discussed in section

The amount of error that we want to adjust at each neuron is equal to:

(as discussed in fully connected layer)

function () is thus equivalent to

where is net output of pixel at position (x,y) at layer l, is real output of pixel at position (x,y) at layer l, represents the total net output of all pixels at layer (l+1) that are pairwise connected with pixel at position (x,y), is the error rate of pixel at position (x,y) at layer l, is the error rate of pixel at position (x,y) at layer (l+1), is the weight between (i’,j’) and (x,y), is the net output of pixel (i’,j’), is the real output of pixel (i’,j’).

This error is contributed into the current weight as follows:

where is the weight between pixel (a,b) and pixel (x,y), is net output of pixel (x,y) at layer l, is real output of pixel (x,y), is the error rate of pixel (x,y) at layer l.