Deep Learning: A tutorial Overview

By Sylvain Dindy-Bolongo

1.0 Introduction

Machine-learning technology powers many aspects of modern society: from web searches to content filtering on social networks to recommendations on e-commerce websites, and it is increasingly present in consumer products such as cameras and smartphones
Conventional machine-learning techniques were limited in their ability to process natural data in their raw form[4]. These machine-learning systems required careful engineering and considerable domain expertise to design a feature extractor that transform the raw data into a suitable internal representation or feature vector from which the learning subsystem, often a classifier, could detect or classify patterns in the input.
Representation learning is a set of methods that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification[3].
Deep-learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. With the composition of enough such transformations, very complex functions can be learned[4].

2.0 Supervised and Unsupervised Learning

The objective of supervise machine learning is to build a model to make prediction based on evidence in the presence of uncertainties. Supervised data has labels which are some kind of teaching signals that goes with the data feature vectors. If the data vectors are unlabeled the machine learning is unsupervised. The algorithm for such unsupervised learning are called clustering algorithms. In this situation, the goal is to groups unlabeled data vector that are closed.

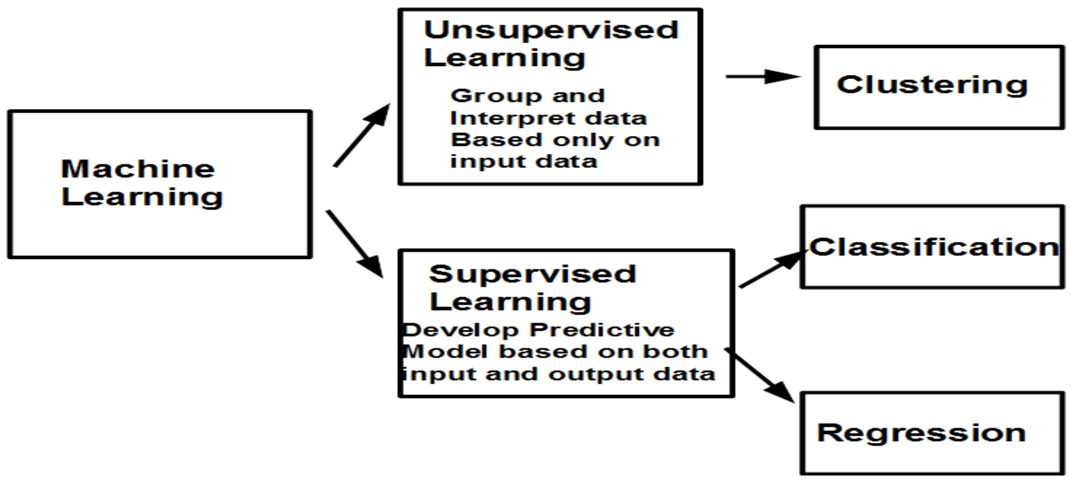
Machine Learning
Unsupervised Learning
Group and Interpret data Based only on input data
Clustering
Supervised Learning
Develop Predictive Model based on both input and output data
Classification
Regression

Figure 2.1 Supervised and unsupervised learning

Supervised learning can be categorical or the data can be numeric or ordered labels. When the data is numeric we say that we are doing regression. Supervised learning can involved one-to-one pairing of labels with data vector or it may consist of deferred learning (sometimes called reinforcement learning). In reinforcement learning, the data label (also called reward or punishment) can come long after the individual data vector is observed.

In this tutorial after a brief introduction to neural network and multi-layer perceptron, some of of the most commonly used deep neural network architectures will be presented.

3.0 The Neural Network

3.1 The Neuron

In an artificial neural network, the unit analogous to the biological neuron is referred to as a "processing element" A processing element has many input paths (dendrites) and combines, usually by a simple summation, the values of these input paths[11]. The result is an internal activity level for the processing element. The combine input is then modified by a transfer function. This transfer function can be a threshold function which only passes information if the combined activity reach a certain level, or it can be a continuous function of the combine input.

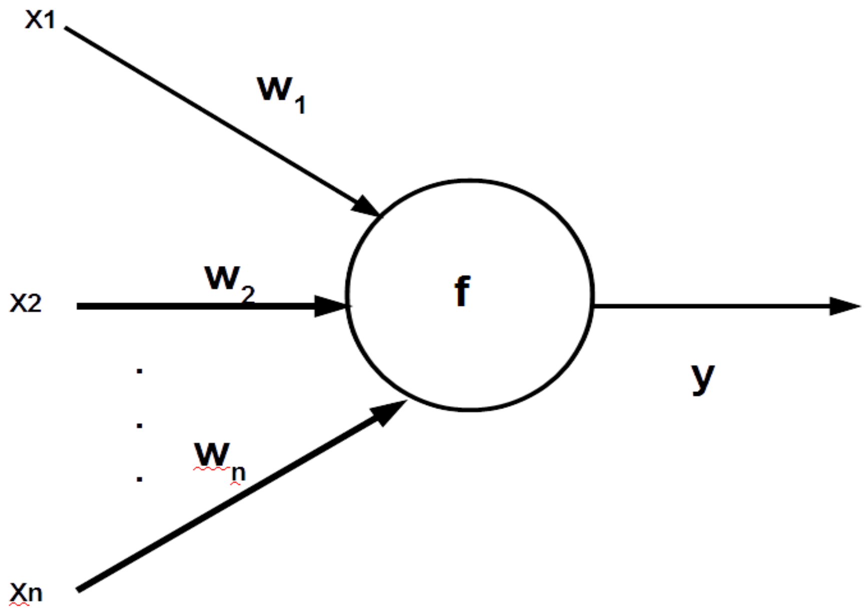

Figure 3.1 Schematic of an artificial neuron

The output path of a processing element can be connected to input paths of other processing elements through connection weights which correspond to the synaptic strength of neural connections. Since each connection has a corresponding weights, the signals on the input lines to a processing element are modified by these weights prior to being summed. The summation function is a weighted summation.
Processing elements are usually organized into groups called layers. A typical network consists of a sequence of layers with full or random connection between successive layers. There are

typically two layers with connections to the outside world: An input buffer where data is presented to the network, and output buffer which holds the response of the network to a given input. Layers distinct from the input and output buffers are called hidden layers.
Most neuron types are defined by the function f they apply to their logit z (summation of weighted input). Let's first consider layers of neurons that use a linear function in the form of $f(z) = az + b$

Linear neurons are easy to compute, but they run into serious limitations. In fact, it can be shown that any feed-forward neural network consisting of only linear neurons can be expressed as a network with no hidden layers. This is problematic because hidden layers are what enable the net to learn important features from the input data. In other words, in order to learn complex relationships, we need to use neurons that employ some sort of nonlinearity. The most common used nonlinearity are described in the following section.

Sigmoid, Tanh, and ReLU Activation function
There are three major types of activation function that are used in practice that introduce nonlinearities in their computations:

Sigmoid. The first of these is the sigmoid neuron, which uses the function

$$f(z) = \frac{1}{1+e^{-z}} \tag{3.1}$$

Intuitively, this means that when the logit is very small, the output of a logistic neuron is very close to 0. When the logit is very large, the output of the logistic neuron is close to 1
Tanh. The tanh use a similar kind of s-shaped nonlinearity, but instead of ranging from 0 to 1, the output of tanh neurons range from -1 to 1. As one would expect, they use

$$f(z) = \tanh(z) \tag{3.2}$$

Restrict Linear Unit (ReLU). A different kind of nonlinearity is used by the restricted linear unit (ReLU) neuron. It uses the function

$$f(z) = \max(0, z) \tag{3.3}$$

That results in a characteristic hockey stick shaped.

3.2 Softmax Output Layer

There are times when we want our output vector to be a probability distribution over a set of mutually exclusive labels. Using a probability distribution gives us a better idea of how confident we are in our predictions. The output vector in this case has the following form $[p_0, p_1, \ldots\ldots p_{n-1}]$ with n the number of the processing element in the output layer.

And $$\sum_{i=0}^{n-1} p_i = 1 \tag{3.4}$$

This is achieved by using a special output layer called a softmax layer[11]. Unlike in other kinds of layers, the output of a neuron in a softmax layer depends on the outputs of all of the other neurons in its layer. This is because we require the sum of all the outputs to be equal to 1. Letting z_i be the logit of the ith softmax neuron, we can achieve this normalization by setting its output to:

$$y_j = \frac{e^{z_j}}{\sum_{i=0}^{n-1} e^i} \tag{3.5}$$

A strong prediction would have a single entry in the vector close to 1 while the remaining entries were close to 0.

4.0 Multi Layer Perceptron

4.1 Gradient Descent

Let's consider the following problem. We have a certain number of training examples. We want to train the neuron so that we pick the optimal weights possible - the weights that minimize the errors we make on the training examples. In this case, let's say we want to minimize the square error over all of the training examples that we encounter.
Let d_i be the desired output of the ith neuron in the output layer and o_i the value computed by the ith neuron, we want to find the value of the weights that minimize the value of the error function E which is given as follows:

$$E = \frac{1}{2}\sum_{i=1}^{n}(d_i - o_i)^2 \tag{4.1}$$

The squared error is zero when our model makes a perfectly correct prediction on every training example. Moreover, the closer E is to 0, the better our model is. As a result, our goal will be to select our parameter vector w (the values for all the weights in our model) such that E is as close to 0 as possible. The gradient descent method is an algorithm to find the minimum point that can be described as follows:
From a given starting point, i.e a trial solution, the direction of the steepest descent is determined. A point lying a small distance along this direction is then taken as the new trial solution. The process is repeated until it is no longer possible to descent, at which point it is assumed that the minimum is reached.

4.2 Backpropagation Algorithm

The multilayer perceptron is a neural network that is comprised of an input layer , an output layer and at least on hidden layer. There is no theoretical limit on the number of hidden layers. These network are also called feedforward networks. They operate by feeding data forwards along the interconnections from input layer, through the hidden layer to the output layer.
The multilayer perceptron is a neural network that still ranks among top-performing classifiers, especially for text recognition. It can be rather slow in training because it uses gradient descent to minimize error by adjusting weighted connections between the numerical classification nodes within the layers. In the test mode it is quite fast.
The following notation are used in order to described the learning rule.

$x_j^{[s]}$: the current output state of the jth neuron in layer s

$w_{ji}^{[s]}$: weight on connection joining ith neuron in layer $s-1$ to jth in layer s

$I_j^{[s]}$: weighted summation of inputs to jth neuron in layer s

A back-propagation element therefore transfers its input as follows:

$$\begin{aligned} x_j^{[s]} &= f\left(\sum_i \left(w_{ji}^{[s]}.x_i^{[s-1]}\right)\right) \\ &= f\left(I_j^{[s]}\right) \end{aligned} \qquad (4.2)$$

Where f is traditionally the sigmoid function but can be any differentiable function. The sigmoid function is defined as

$$f(z) = \frac{1}{1+e^{-z}}$$

Backpropagating the local error

Suppose that the network has some global error function E associated with it which is a differentiable function of all the connection weights in the network. The critical parameter that is passed back through the layers is defined by

$$e_j^{[s]} = \partial E / \partial I_j^{[s]} \qquad (4.3)$$

Using the chain rule twice in succession gives the relationship between the local error at a particular processing element at level s and all the local errors at the level $s+1$

$$e_j^{[s]} = f'(I_j^{[s]}) \sum_k \left(e_k^{[s+1]}.w_{kj}^{[s+1]}\right) \qquad (4.4)$$

If f is the sigmoid function as defined, then its derivative can be expressed as a simple function of itself as follows:

$$f'(z) = f(z)(1-f(z)) \qquad (4.5)$$

Therefore after combining the local error can be written as follows:

$$e_j^{[s]} = x_j^{[s]}(1-x_j^{[s]}) \sum_k \left(e_k^{[s+1]} w_{kj}^{[s+1]}\right) \qquad (4.6)$$

Minimizing the Global error

Given the current set of weights $w_{ij}^{[s]}$, we need to determine how to increment or decrement them in order to decrease the global error[11].

$$\Delta w_{ji}^{[s]} = -\eta \, \partial E / \partial w_{ji}^{[s]} \qquad (4.7)$$

Where η is the learning rate. The partial derivative can be calculated from the local error.

$$\begin{aligned} \partial E / \partial w_{ji}^{[s]} &= \left(\partial E / \partial I_j^{[s]}\right)\left(\partial I_j^{[s]} / \partial w_{ji}^{[s]}\right) \\ &= -e_j^{[s]} x_i^{[s-1]} \end{aligned} \qquad (4.8)$$

After combining the two previous equation this gives

$$\Delta w_{ji}^{[s]} = -\eta e_j^{[s]} x_i^{[s-1]} \qquad (4.9)$$

The Global Error function

Suppose that a vector I is presented at the input edge layer of the network and suppose the desired output d is specified. Let denote the actual output produced by the network with its current set of weights. Then a measure of the error in achieving that desired output is given by

$$E = \tfrac{1}{2}\sum_{k}\left((d_k - o_k)^2\right) \tag{4.10}$$

The scale local error for each element of the output layer is given as follows[11]:

$$e_k^0 = -\partial E / \partial I_k^0 = -\partial E / \partial o_k . \partial o_k / \partial I_k = (d_k - o_k)\, f^{'}(I_k) \tag{4.11}$$

Summary of the Standard Back-Propagation Algorithm

Given an input vector I and desired output vector d do the following:

1. Present I to the input layer of the network and propagate it through to the output to obtain an output vector o.
2. As this information propagates through the network, it will also set all the summed input I_j and output states x_j for each processing element in the network.
3. For each processing element in the output layer, calculate the scaled local error as given in [4.11] and then calculate the delta weight using [4.9]
4. For each layer s, starting at the layer below the output layer and ending with the layer above the input, and for each processing element in the layer s, calculate the scaled local error as given in [4.6] then calculate the delta rule using [4.9]
5. Update all weights in the network by adding the delta weights to the corresponding previous weights.

4.3 Some advice for implementing neural networks

• Make sure to check the correctness of your gradient computed by backpropagation by comparing it with the numerical approximation[6],[7].

• It's important to "break the symmetry" of the neurons in the networks or, in other words, force the neurons to be different at the beginning. This means that it's important to initialize the parameters w randomly.

• Also make sure that the random initialization does not "saturate" the networks. This means that most of the time, for your data, the values of the neurons should be between 0.2 and 0.8.

• Have a way to monitor the progress of your training. Perhaps the best method is to compute the error function E on the current example or on a subset of the training data or on a held-out set.

• Picking a good learning rate η can be tricky. A large learning rate can change the parameters too aggressively or a small learning rate can change the parameters too conservatively. Both should be avoided, a good learning rate is one that leads to good overall improvements in the objective function E. To select good η, it's also best to monitor the progress of your training. In many cases, a learning rate of 0.1 or 0.01 is a very good start[7].

• Picking good hyperparameters (architectural parameters such as number of layers, number of neurons on each layers) for your neural networks can be difficult. A standard way to pick architectural parameters is via cross-validation: Keep a hold-out validation set that the training never touches. If the method performs well on the training data but not the validation set, then the model overfits: it has too many degrees of freedom and remembers the training cases but does not generalize to new cases. If the model overfits, we need to reduce the number of hidden layers or number of neurons on each hidden layer. If the method performs badly on the training set then the model underfits: it does not have enough degrees of freedom and we should increase the number of hidden layers or number of neurons[7].

4.3 Stochastic and Mini-Batch Gradient

In the algorithms described previously, we've been using a version of gradient descent known as batch gradient descent. The idea behind batch gradient descent is that we use our entire dataset to compute the error surface and then follow the gradient to take the path of steepest descent.

In stochastic training or online training, after the gradients are computed for a single training item, weights and biases are updated immediately.

Mini-batch training combine batch and stochastic training: instead of using all training data items to compute gradients (as in batch training) or using a single training item to compute gradients (as in stochastic training), mini-batch training uses a user-specified number of training items.

4.4 How to Diagnose Machine Learning Problem

Getting machine learning to work well can be more of an art than a science. Here we have some rules of thumb regarding machine learning: more data beats less data, and better features beat better algorithms[1]. There are two common problems see Figure 4.1:

Under fit model: That mean model assumptions are too strong for the data, so the model won't fit well. The possible solutions are as follows:

- More features can help make a better fit
- Use a more powerful algorithm

Over fit model: That means the algorithm used has memorized the data including noise so that it can't generalized. Some solutions are given as follows:

- More training data can help smooth the model
- Fewer features can reduce overfitting
- Use less powerful algorithm

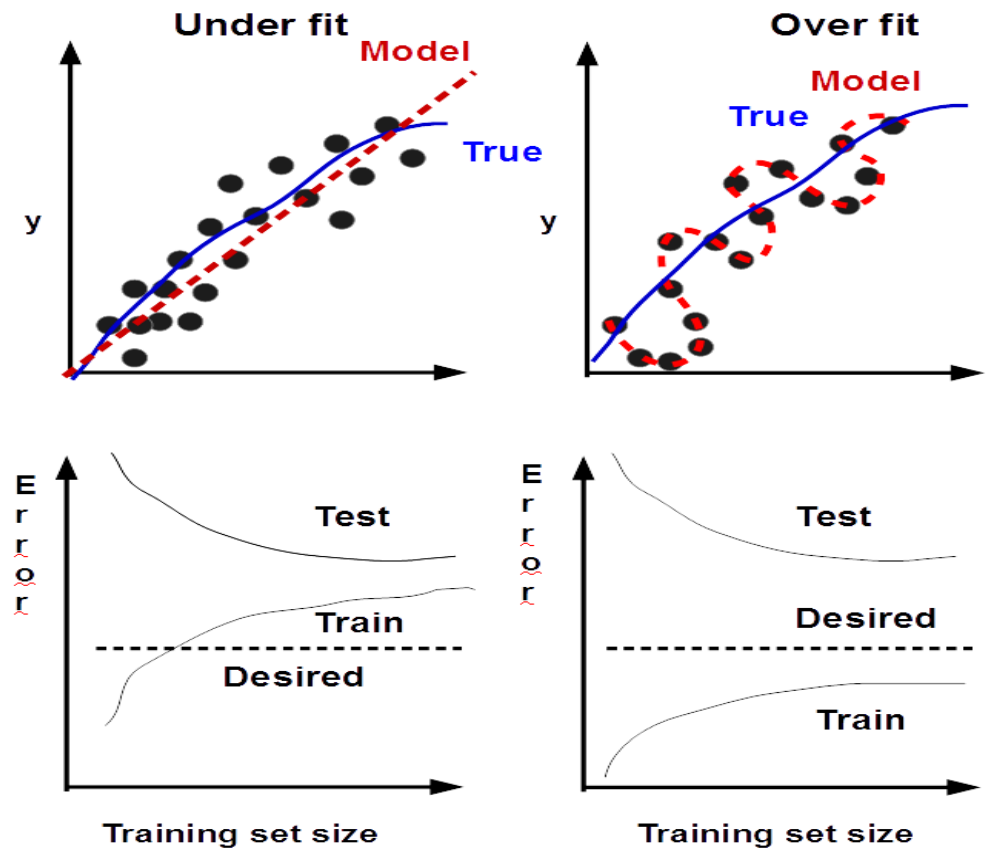

Figure 4.1 Pour model fitting in machine learning and its effect on training and test prediction performance.

Figure 4.1 shows under and overfitting of data on top and the corresponding error in term of the training set size. If we use the model that is to restrictive, we can never fit the true parabola in blue in the top left. The fit to both the training and the test data is poor. On the right side we fit the training data set exactly. It memorizes the training data as well as well as the noise in the data, the resulting fit to the test data is poor.

4.5 Cross-Validation, Bootstrapping, ROC, and Confusion Matrix

There are some basic tools that are used in machine learning to measure results. In supervised learning, one of the most basic problems is simply knowing how well your algorithm has performed: how accurate is it at classifying or fitting data? In order to yield more accurate measures of actual performance of the machine learning algorithms, techniques such as cross-validation and/or bootstrapping are used [1].

Cross-validation involves dividing the data into K different sub-sets of data. The algorithm is trained on $K-1$ subset and test on the final subset of the data that wasn't trained on. This is done K times, where each of the K subsets gets a "turn" at being the validation set, and the average the result.

Bootstrapping is similar to cross-validation, but the validation set is selected at random from the training data.

Two other useful ways of assessing, characterizing, and tuning classifiers are plotting the receiver operating characteristic (ROC) and filling in a confusion matrix see Figure 4.2

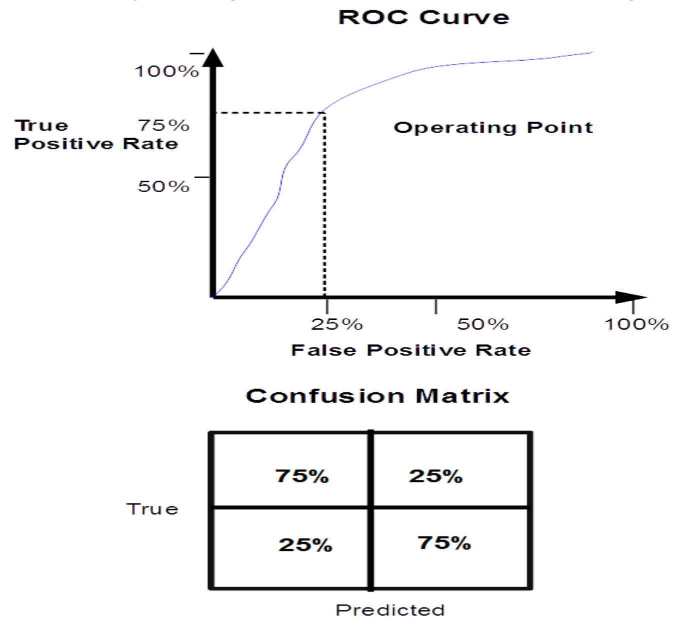

Figure 4.2 ROC curve and associated confusion matrix.
The ROC curve measures the response over the performance parameter of the classifier over the full range of settings of the parameters. The figure also shows a confusion matrix. This is just a chart of true and false positives along with true and false negatives.

5.0 Deep learning Concept

Conventional machine-learning techniques were limited in their ability to process natural data in their raw form. Traditional machine-learning system required careful engineering and considerable domain expertise to design a feature extractor that transformed the raw data (such as the pixel values of an image) into a suitable internal representation or feature vector from which the learning subsystem, often a classifier, could detect or classify patterns in the input[4].
Representation learning is a set of methods that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification. Deep-learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. With the composition of enough such transformations, very complex functions can be learned [4].

6.0 Autoencoder Network

6.1 Autoencoder

Autoencoders belong to a class of learning algorithms known as unsupervised learning. Unlike supervised algorithms, unsupervised learning algorithms do not need labeled information for the data[9],[12].

An autoencoder is typically a feedforward neural network which aims to learn a compressed, distributed representation (encoding) of a dataset.
Conceptually, the network is trained to “recreate” the input, i.e., the input and the target data are the same. In other words: you’re trying to output the same thing you were input, but compressed in some way.

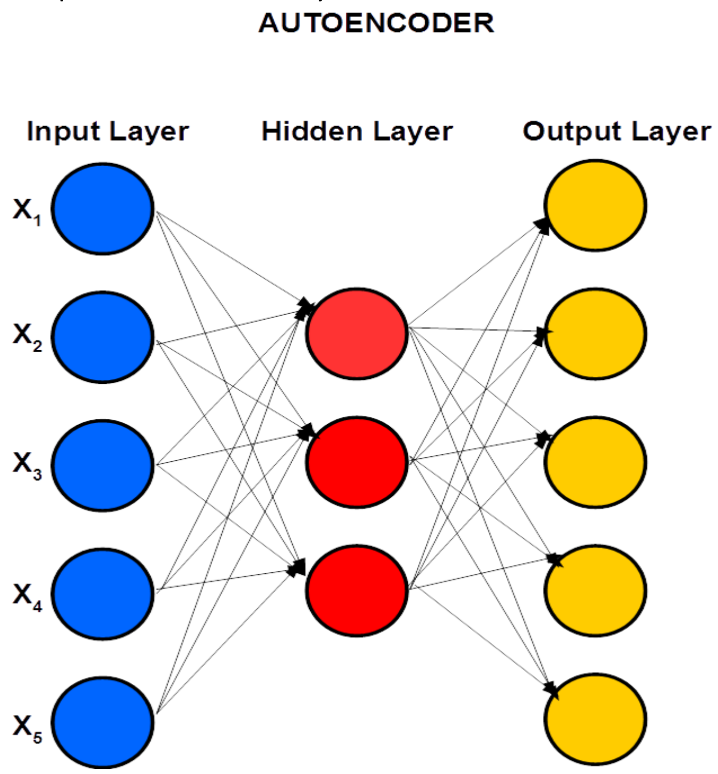

Figure 6.1: Example of autoencoder

6.2 Deep Autoencoder

The deep autoencoder is composed of the stack of autoencoder[9] as given in Figure[6.2]

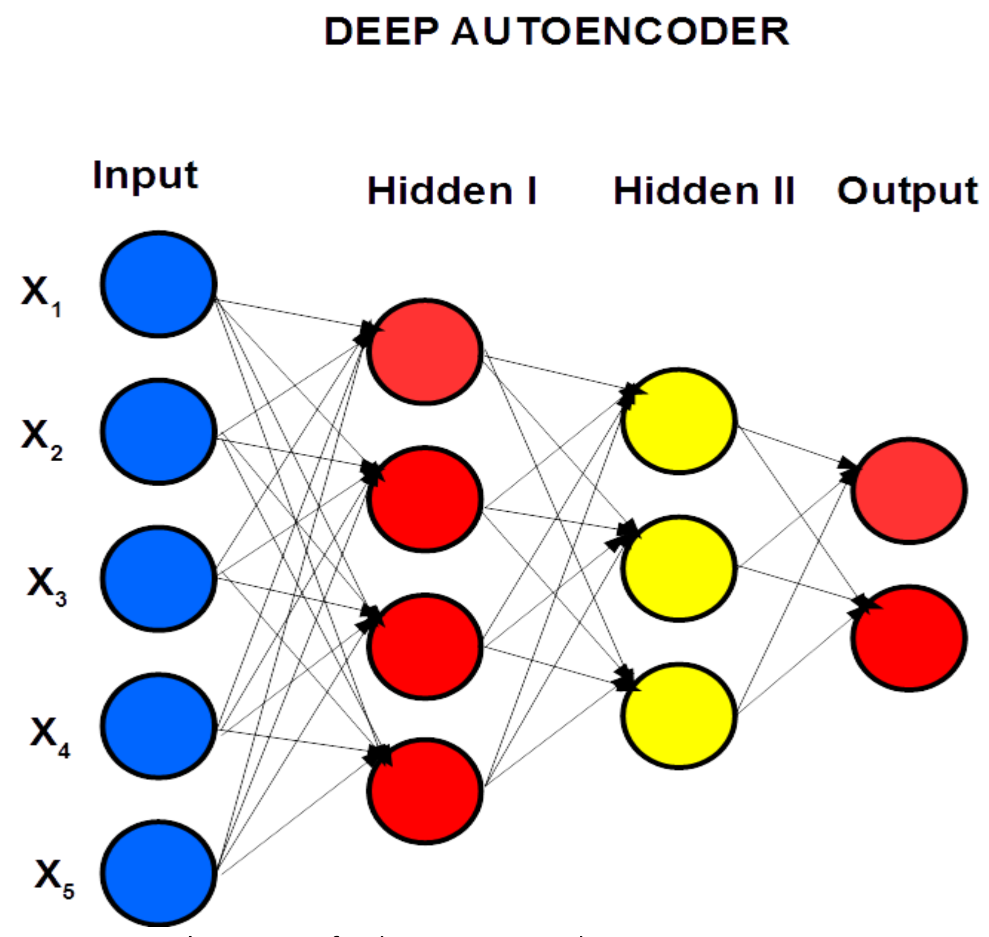

Figure 6.2 Architecture of a deep Autoencoder

6.3 Deep Autoencoder Training

Autoencoders have many interesting applications, such as data compression, visualization, etc. But later it was observed that autoencoders could be used as a way to “pretrain” neural networks[7], for the reason that training very deep neural networks is difficult:

- The magnitudes of gradients in the lower layers and in higher layers are different,
- It is difficult for stochastic gradient descent to find a good local optimum, on the error surface
- Deep networks have many parameters, which can remember training data and do not generalize well.

The goal of pretraining is to address the above problems. With pretraining, the process of training a deep network is divided in a sequence of steps:

- Pretraining step:
    - train a sequence of shallow autoencoders, greedily one layer at a time, using unsupervised data,
- Fine-tuning
    - step 1: train the last layer using supervised data,

- step 2: use backpropagation to fine-tune the entire network using supervised data.

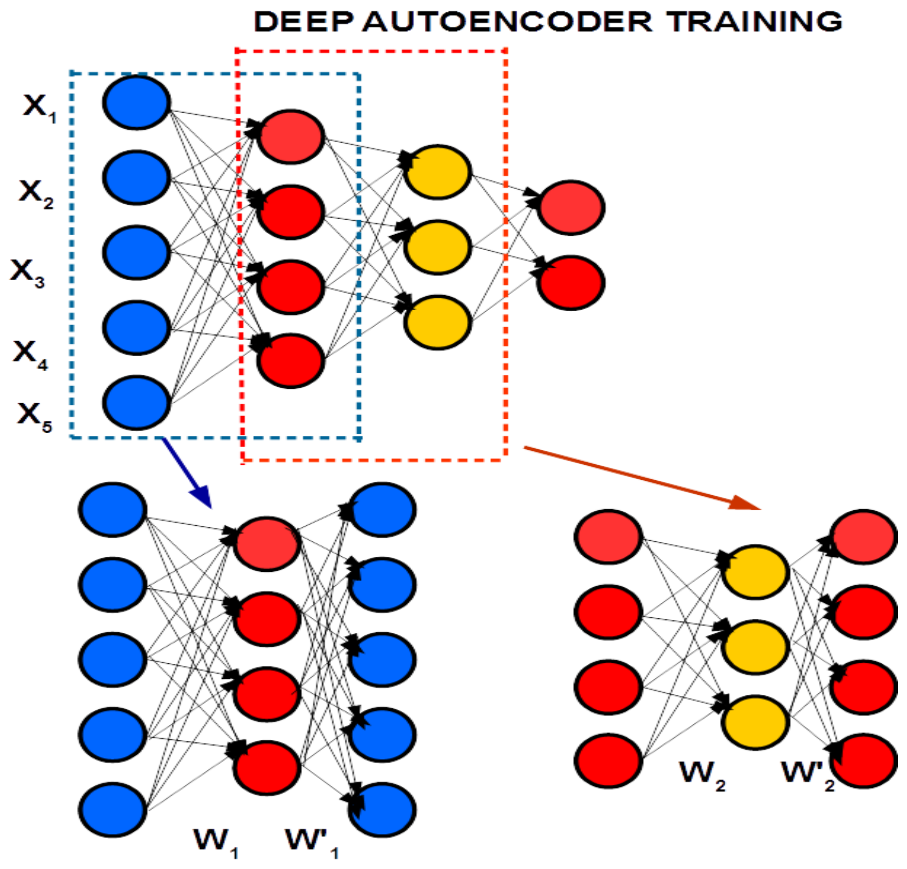

Figure 6.3 Deep autoencoder training

Let's consider the problem of training a relatively deep network of two hidden layers to classify some data. The parameters of the first two hidden layers are W_1 and W_2 respectively. Such network can be pretrained by a sequence of two autoencoders, in the following manner: The autoencoder with parameters W_1 and $W_1^{'}$ will be trained. After this, W_1 is used in order to compute the values for the red neurons for all of the data, which will then be used as input data to the subsequent autoencoder. The parameters of the decoding process $W_1^{'}$ will be discarded. The subsequent autoencoder uses the values for the red neurons as inputs, and trains an autoencoder to predict those values by adding a decoding layer with parameters W_2 Researchers have shown that this pretraining idea improves deep neural networks [7]; perhaps because pretraining is done one layer at a time which means it does not suffer from the difficulty of full supervised learning.

7.0 Deep Belief Network

7.1 Restricted Boltzmann Machine

RBMs are composed of a hidden, visible, and bias layer. Unlike the feedforward networks, the connections between the visible and hidden layers are undirected (the values can be

propagated in both the visible-to-hidden and hidden-to-visible directions) and fully connected (each unit from a given layer is connected to each unit in the next)[9].
The standard RBM has binary hidden and visible units: that is, the unit activation is 0 or 1 under a Bernoulli distribution, but there are variants with other non-linearities.

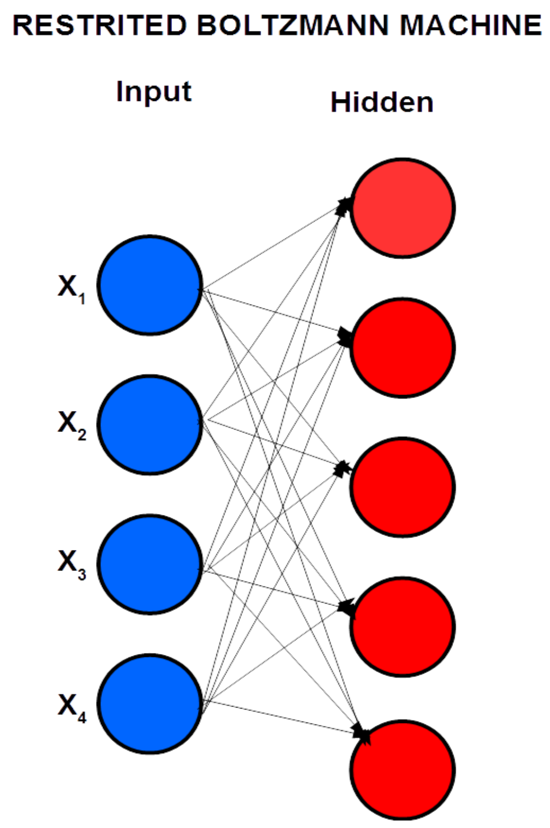

Figure 6.4 Restricted Boltzmann Machine

While researchers have known about RBMs for some time now, the recent introduction of the contrastive divergence unsupervised training algorithm has renewed interest.

Contrastive Divergence
Here is the description of a single-step contrastive divergence algorithm:

Positive phase:
An input sample v is fed to the input layer.
v is propagated to the hidden layer in a similar manner to the feedforward networks. The result of the hidden layer activations is h

Negative phase:
Propagate h back to the visible layer with result v' (the connections between the visible and hidden layers are undirected and thus allow movement in both directions).
Propagate the new v' back to the hidden layer with activations result h'.
Weight update:

$$w(k+1) = w(k) + \eta(vh^T - v'h'^T) \quad (6.1)$$

Where η is the learning rate and v, v', h, h', and w are vectors.

7.2 Deep Belief Network

As with autoencoders, we can also stack Boltzmann machines to create a class known as deep belief networks (DBNs)[12].

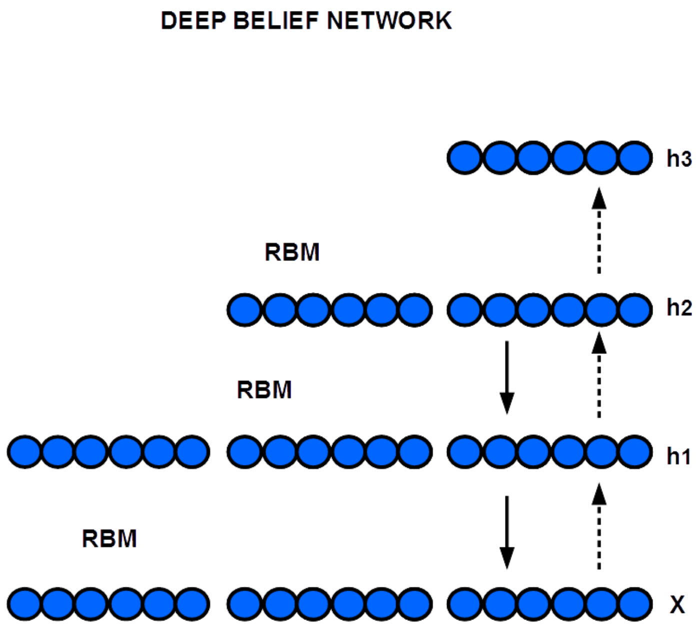

Figure 6.5 Deep Belief Network

7.3 Deep Belief Network Training

In this case, the hidden layer of RBM k acts as a visible layer for RBM k+1[9]. The input layer of the first RBM is the input layer for the whole network, and the pre-training works like this:

- Train the first RBM k=1 using contrastive divergence with all the training samples.
- Train the second RBM k=2. Since the visible layer for k=2 is the hidden layer of k=1, training begins by feeding the input sample to the visible layer of k=1, which is propagated forward to the hidden layer of k=1. This data then serves to initiate contrastive divergence training for k=2.
- Repeat the previous procedure for all the layers.

After pre-training, the network can be extended by connecting one or more fully connected layers to the final RBM hidden layer. This forms a multi-layer perceptron which can then be fine tuned using backpropagation.

8.0 Convolutional Neural Network

8.1 Introduction

A convolutional neural network, or CNN, is a network architecture for deep learning. A CNN is made up of several layers that process and transform an input to produce an output.

In a typical neural network, each neuron in the input layer is connected to a neuron in the hidden layer. However, in a CNN, only a small region of input layer neurons connect to neurons in the hidden layer. These regions are referred to as local receptive fields. The local receptive field is translated across an input to create a feature map from the input layer to the hidden layer neurons. You can use convolution to implement this process efficiently. That's why it is called a Convolutional Neural Network.

Like a typical neural network, a CNN has neurons with weights and biases. The model learns these values during the training process, and it continually updates them with each new training example. However, in the case of CNNs, the weight and bias values are the same for all the hidden neurons in a given layer. This means that all the hidden neurons are detecting the same feature.

The activation step applies a transformation to the output of each neuron by using activation functions. Rectified Linear Unit, or ReLU, is an example of a commonly used activation function. It takes the output of a neuron and maps it to the highest positive value. Or, if the output is negative, the function maps it to zero.

You can further transform the output of the activation step by applying a pooling step. Pooling reduces the dimensionality of the feature map by condensing the output of small regions of neurons into a single output. This helps simplify the following layers and reduces the number of parameters that the model needs to learn.

A CNN can have tens or hundreds of hidden layers that each learn to detect different features of an image resulting in a Deep Convolution Neural Network DCNN

8.2 Architecture of a Deep Convolution Neural Network DCNN

DCNN contains three kinds of layers, which are the convolutional layer, pooling layer, and fully-connected layer[4][5]. As shown in Figure 6.6, the first several layers of a typical DCNN usually consist of a combination of two types of layers—convolutional layers, followed by pooling layers—and the last layer is a fully-connected layer. In the following part, we will describe these three kinds of layers in more detail.

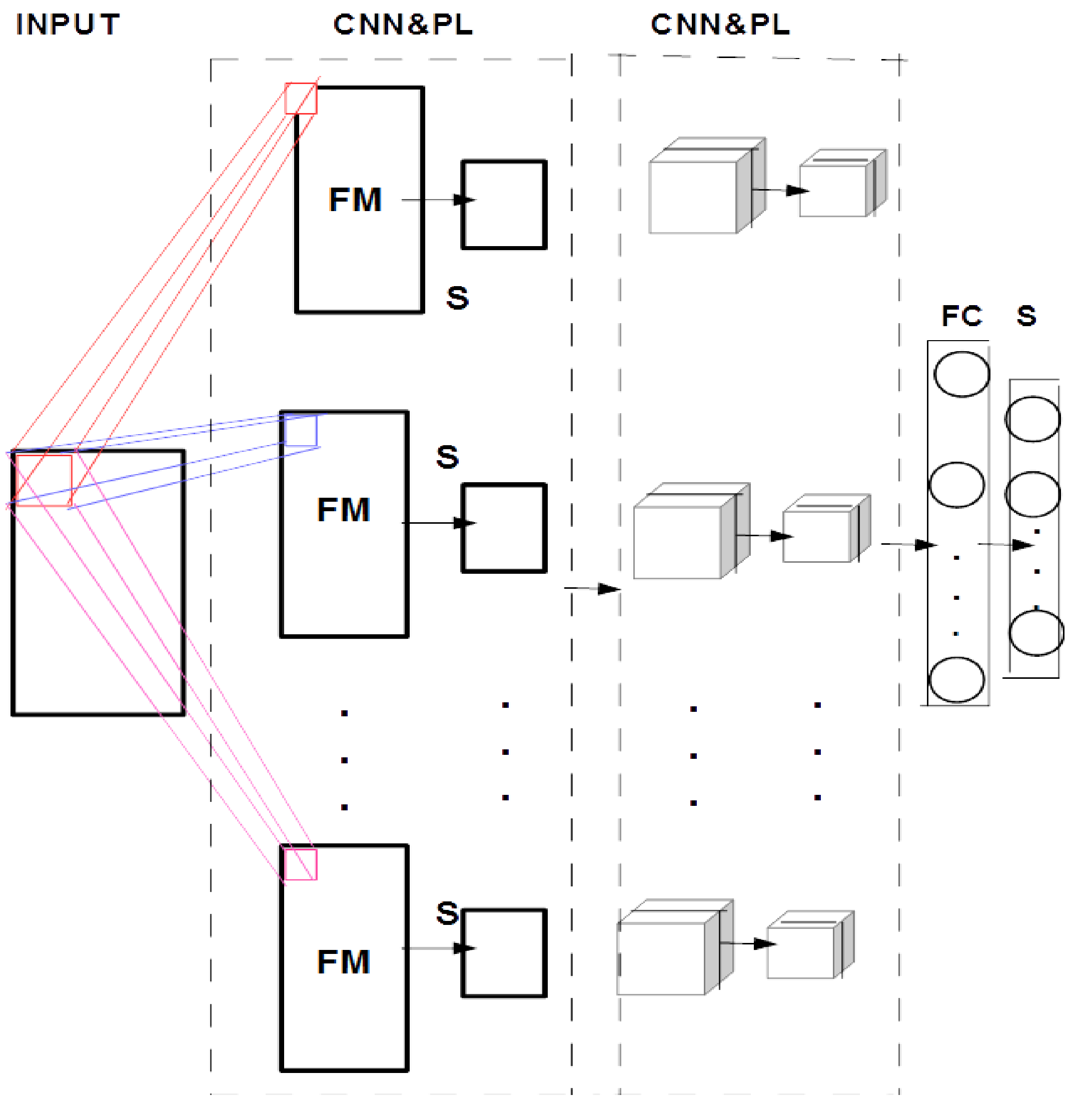

Figure 8.1 Deep Convolution Network: CNN&PL: Convolution Network and Pooling Layer; FC: Fully Connected Layer; S: Softmax Layer. FM: Feature Map; S: Subsampling.

The convolutional layer is composed of a number of two-dimensional (2D) filters with weighted parameters. These filters convolute with input data and obtain an output, named as feature maps. Each filter shares the same weighted parameters for all the patches of the input data to reduce the training time and complication of the model, which is different from a traditional neural network with different weighted parameters for different patches of the input data. Suppose the input of the convolutional layer is X , which belongs to $\Re^{A\times B}$,and A and B are the dimensions of the input data. Then the output of the convolutional layer can be calculated as follows [5]:

$$Y_n = f(\sum_{i=1}^{C} X_i^{l-1} * W_n^l + B_n^l) \tag{8.1}$$

where Y_n is the nth output of the convolutional layer, and N is the total number of output, which is also equal to the total number of filter; $*$ is an operator of convolution; X_i^{l-1}

represents the input data of ith channel of previous layer $l-1$, and the total number of channel is C; W_n^l is the weight of the nth filter of the current layer l; the width and height of the filter are W and H, respectively; the nth bias is denoted with B_n^l; f is an activation function, typically hyperbolic tangent or sigmoid function.

The pooling layer is a sub-sampling layer, whose purpose is to merge semantically similar features into one and also improves the robustness of learned features. A pooling layer generally follows a convolutional layer with a max pooling method and it outputs only the maximum of each sub-sampling patch of the feature maps to subsample the feature maps from the previous convolutional layer. The output can be described as follows [5]:

$$P_n = \max_{n \in S} Y_n \qquad (8.2)$$

where P_n is the nthoutput of the pooling layer, and N the total number of output; S is the pooling block size. This function will sum over each distinct S pooling block in the input data so that the output will become S times smaller along both spatial dimensions.

The fully-connected layer is the last layer of the DCNN model. It follows several combinations of the convolutional layers and the pooling layers, and classifies the higher-level information from the previous layers. A fully-connected layer is similar to a traditional multilayer neural network with a hidden layer and a classification layer, typically using a softmax regression. Assuming that the task is a K-label problem, the output of the softmax regression can be calculated as follows:

$$o_j = \begin{bmatrix} p(y=1 \mid x;\theta \\ p(y=2 \mid x;\theta \\ \cdots \\ p(y=k \mid x;\theta \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} \exp(\theta_j x)} \begin{bmatrix} \exp(\theta_1 x) \\ \exp(\theta_2 x) \\ \cdots \\ \exp(\theta_k x) \end{bmatrix} \qquad (8.3)$$

Where $\theta_1, \theta_2, \theta_3 .... \theta_\mathrm{k}$ are the parameters of the model, and o_j is the final result of the DCNN.

9.0 Long Short Term Memory Network

9.1 Recurrent Network (RNN)

For tasks that involve sequential inputs, it is often better to use RNNs Figure 9.1. RNNs process an input sequence one element at a time, maintaining in their hidden units a ‘state vector’ that implicitly contains information about the history of all the past elements of the sequence. When we consider the outputs of the hidden units at different discrete time steps as if they were the outputs of different neurons in a deep multilayer network , it becomes clear how we can apply backpropagation to train RNNs.

RNNs are very powerful dynamic systems, but training them has proved to be problematic because the backpropagated gradients either grow or shrink at each time step, so over many time steps they typically explode or vanish[5]

The artificial neurons (for example, hidden units grouped under node s with values s_k at time k) get inputs from other neurons at previous time steps. In this way, a recurrent neural network can map an input sequence with elements x_k into an output sequence with elements o_k, with each o_k depending on all the previous $x_{k'}$ (for $k' \le k$). The same parameters (matrices U, V, W) are used at each time step. Many other architectures are possible. The backpropagation algorithm can be directly applied to the computational graph of the unfolded network, to compute the derivative of a total error with respect to all the states s_k and all the parameters.

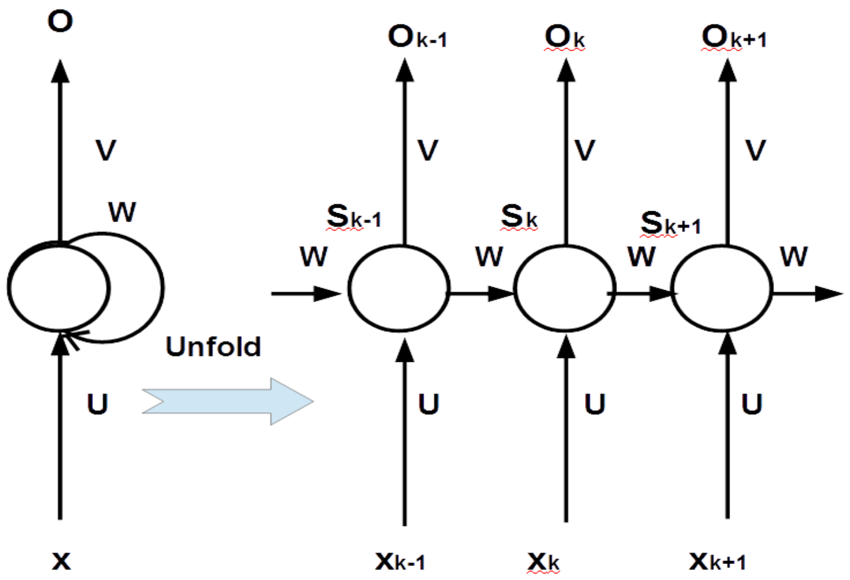

Figure 9.1 A recurrent neural network and the unfolding in time of the computation involved in its forward computation.

9.2 Deep Long Short Term Memory Network

The LSTM is a network that contains special units called memory blocks in the recurrent hidden layer. The memory blocks contain memory cells with self-connections storing the temporal state of the network in addition to special multiplicative units called gates to control the flow of information.

Each memory block contained an input gate an output gate and a forget gate. The input gate controls the flow of input activations into the memory cell. The output gate controls the output flow of cell activations into the rest of the network. The forget gate scales the internal state of the cell before adding it as input to the cell through the self-recurrent connection of the cell, therefore adaptively forgetting or resetting the cell's memory. In addition, the modern LSTM architecture contains peephole connections from its internal cells to the gates in the same cell to learn precise timing of the outputs [10].

An LSTM network computes a mapping from an input sequence $x = (x_1, x_2, \ldots x_k)$ to an output sequence $y = (y_1, y_2, \ldots y_k)$ by calculating the network unit activations using the following equations iteratively from $k \in [1, K]$:

$$i_k = \sigma(W_{ix} x_k + W_{im} m_{k-1} + W_{ic} c_{k-1} + b_i)$$

$$f_k = \sigma(W_{fx} x_k + W_{fm} m_{k-1} + W_{fc} c_{k-1} + b_i)$$

$$c_k = f_k \otimes c_{k-1} + i_k \otimes g(W_{ix}x_k + W_{im}m_{k-1} + b_i)$$
$$o_k = \sigma(W_{ox}x_k + W_{om}m_{k-1} + W_{oc}c_{k-1} + b_o)$$
$$m_k = o_k \otimes h(c_k)$$
$$y = \phi(W_{ym}m_k + b_y)$$

Where the W terms denote weight matrices (e.g. W_{ix} is the matrix of weights from the input gate to the input), W_{ic}, W_{fc}, W_{oc} are diagonal weight matrices for peephole connections, the b terms denote bias vectors (b_i is the input gate bias vector), σ is the logistic sigmoid function, and i, f, o and c are respectively the input gate, forget gate, output gate and cell activation vectors, all of which are the same size as the cell output activation vector m,

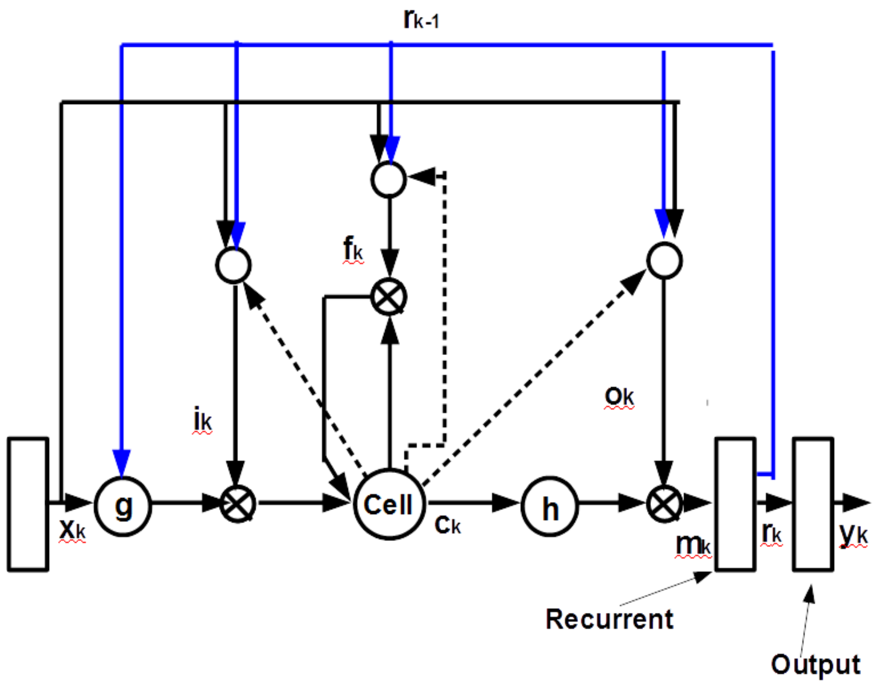

Figure 9.2 LSTM memory block

$\otimes$ is the element-wise product of the vectors, g and h are the cell input and cell output activation functions, ϕ is the network output activation function.

9.3 Deep LSTM

Deep LSTM RNNs are built by stacking multiple LSTM layers[10]. Note that LSTM RNNs are already deep architectures in the sense that they can be considered as a feed-forward neural network unrolled in time where each layer shares the same model parameters.

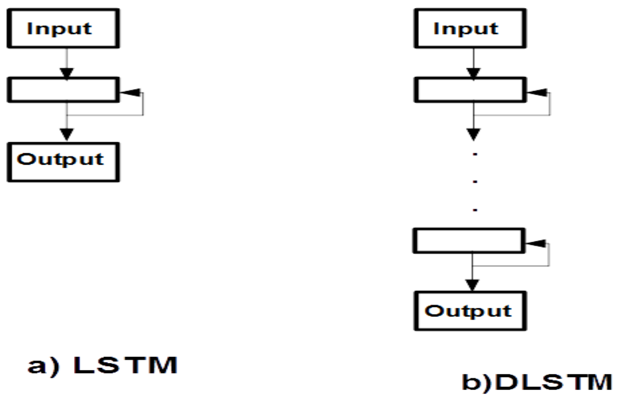

Figure 9.3 LSTM and DLSTM

10.0 Overfitting Prevention in Deep Neural Networks

The three most commonly used techniques to prevent overfitting during the training process are given as follows[8]:

Regularization. Regularization modifies the objective function to minimize by adding additional terms that penalize large weights. In other words, the objective function is modified so that it becomes $Error + \lambda f(\theta)$, where $f(\theta)$ grows larger as the components of θ grow larger and λ is the regularization strength.

Max norm constraints. Max norm constraints impose an absolute upper bound on the magnitude of the incoming weight vector for every neuron. Anytime a gradient descent step moved the incoming weight vector such that $\|w\| > c$ the norm of w is reduced toward zero. Typical values of c are 3 and 4.

Dropout. Dropout is a technique where randomly selected neurons are ignored during training. They are dropped-out randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

11.0 Conclusion

This tutorial is an overview of deep learning. We started with the definition of neural network computing followed by the presentation of back-propagation algorithm. The common deep learning neural network architecture were also presented. We covered some of the problems encountered during the training process as well as their mitigation.

12.0 Reference

[1] Ng Andrew Deep Learning - CS229
http://cs229.stanford.edu/materials/CS229-DeepLearning.pdf
[2] Schmidhuber Jurgen “Deep Learning in Neural Network: An Overview”
www.idsia.ch/~juergen/deep-learning-overview.html

[3] Hinton Geoff, Yoshua Bengio and Yann LeCun “Deep Learning: NIPS 2015 Tutorial” CIFAR (Canadian Institute For Advanced Research)
[4] LeCun Yann, Yoshua Bengio and Geoffroy Hinton “ Deep Learning”
Nature, Vol 521, May 28 2015
[5] Jing Luyang, Taiyong Wang, Ming Zhao and Peng Wang “An Adaptive multisensory Data Fusion Method Based on Deep Convolution Neural Network For Fault Diagnosis of Planetary Gearbox”
Nature, February 2017
[6] Quo V. Le “A tutorial on Deep Learning Part 1: Nonlinear Classifiers and the Backpropagation Algorithm”
https://cs.stanford.edu/~quocle/tutorial1.pdf
[7] Quo V. Le “A tutorial on Deep Learning Part 2: Autoencoder, Convolutional Neural Network and Recurrent Neural Network”
https://cs.stanford.edu/~quocle/tutorial2.pdf
[8] Buduma Nikhil “Fundamental of Deep Learning: Designing Next Generation Artificial Intelligence Algorithm” O’Reilly November 2015: First Edition
[9] Vasilev Ivan “A Deep Learning Tutorial: From Perceptrons to Deep Networks”
https://www.toptal.com/Blog
[10] Sak Hasim, Andrew Senior, Francoise Beaufays “Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling”
{hasim, andrewsenior,fsb@googe.com}
[11] Technical Publication Group, “Neural Computing”
NeuralWare, Inc., 1993
[12] Lisa-Lab , University of Montreal “Deep Learning Tutorials : Release 01”
https://github.com/lisa-lab/DeepLearningTutorials.git

About the author:

Sylvain Dindy-Bolongo is currently Principal at VizCortex, prior to that, he worked as Software Engineer, Embedded Software Engineer, Control System Engineer, Control Architect for several companies in the San Francisco Bay Area, including HP, Agilent, Schneider Electric, Tech-Mahindra, as well as a few startups. He also worked as Part-Time Instructor for the Art Institute of California at San Francisco, University of California at Bekeley Extension, Sonoma State University (Master of Engineering Program).
His interests include Robotics (SLAM Navigation, Manipulator Control), Machine Learning, Deep Learning, Computer Vision, Control Systems (Robust, Intelligent, Embedded).
He earned his BEE, MSEE from Ecole Polytechnique de Montreal (University of Montreal, Canada) and a PhD in Control Systems and Robotics from Wichita State University, Kansas.