Deep Learning

Lecture # 18

Introduction

A deep neural network (DNN) is an Artificial Neural Network (ANN) with multiple layers between the input and output layers. The DNN finds the correct mathematical manipulation to turn the input into the output, whether it be a linear relationship or a non-linear relationship. The network moves through the layers calculating the probability of each output. For example, a DNN that is trained to recognize dog breeds will go over the given image and calculate the probability that the dog in the image is a certain breed. The user can review the results and select which probabilities the network should display (above a certain threshold, etc.) and return the proposed label.

Each mathematical manipulation as such is considered a layer, and complex DNN have many layers, hence the name "deep" networks.

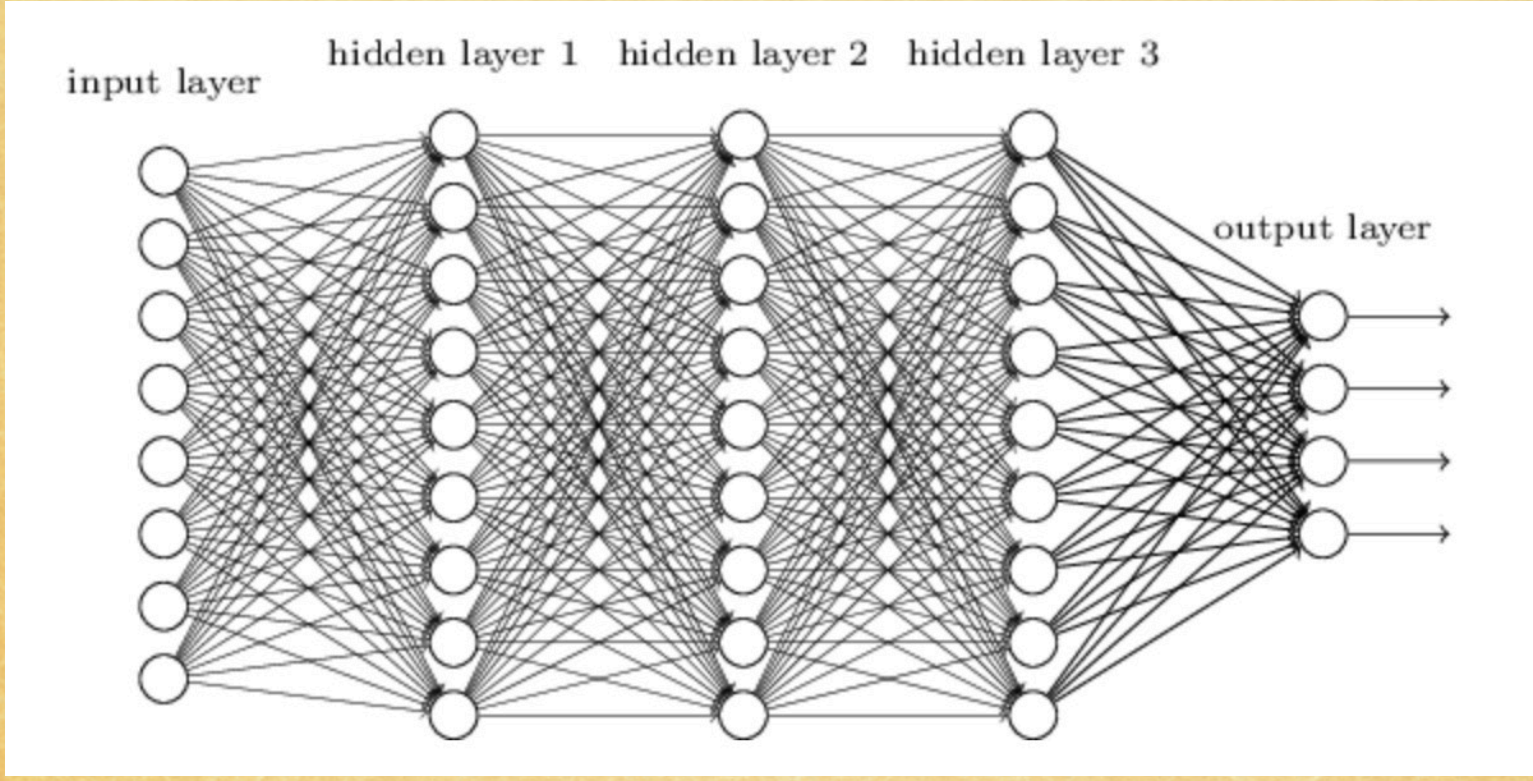
DNNs can model complex non-linear relationships. DNN architectures generate compositional models where the object is expressed as a layered composition of primitives. The extra layers enable composition of features from lower layers, potentially modeling complex data with fewer units than a similarly performing shallow network.

DNNs are typically feedforward networks in which data flows from the input layer to the output layer without looping back. At first, the DNN creates a map of virtual neurons and assigns random numerical values, or "weights", to connections between them. The weights and inputs are multiplied and return an output between 0 and 1. If the network didn't accurately recognize a particular pattern, an algorithm would adjust the weights. That way the algorithm can make certain parameters more influential, until it determines the correct mathematical manipulation to fully process the data.

Recurrent Neural Network (RNNs), in which data can flow in any direction,

Convolutional deep neural networks (CNNs) are used in computer vision. CNNs also have been applied to acoustic modeling for automatic speech recognition (ASR). In ANN adjacent network layers are fully connected to one another (Figure 1). That is, every neuron in the network is connected to every neuron in adjacent layers:

Figure 1: Neurons are connected to one another via a complex networks



Bias

Lets take a simple example of one input and one output. The input is $x_1.w_1$. what does changing w_1 do in the network? This is as in Figures 2a and 2b. Changing weight changes the slope of the output of the sigmoid activation function. This is useful if we want to study the strength of the relation between the input and output functions (Fig 3a). What if we only want the output to change when x is greater than 1. This is where the bias comes in. Now consider the same network with also a bias input.

By changing the bias, one can change where the function activates. Without introducing a bias term, the activation function will not shift along (Fig 3b). This is useful when introducing an "if statement"

Fig 2a: Input-Output node

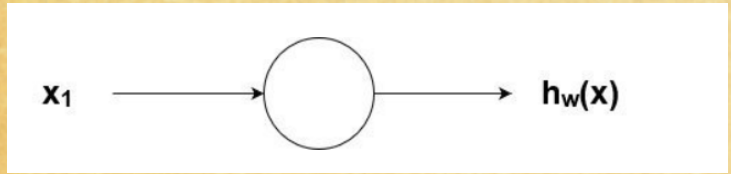


Fig 2b: Input +bias node ($w_1x_1+b_1$)

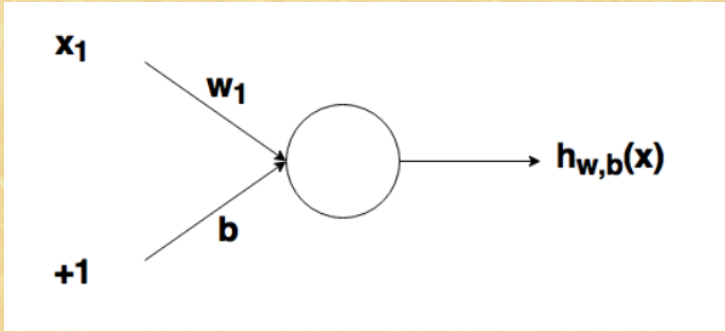


Fig 3a: Weight (w) changes

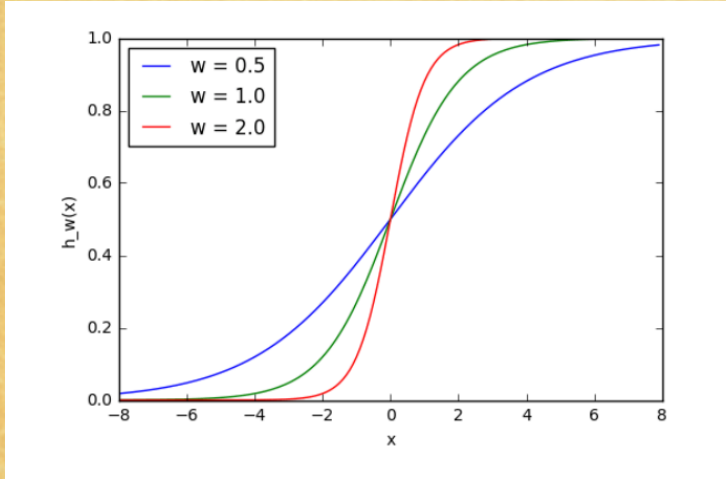
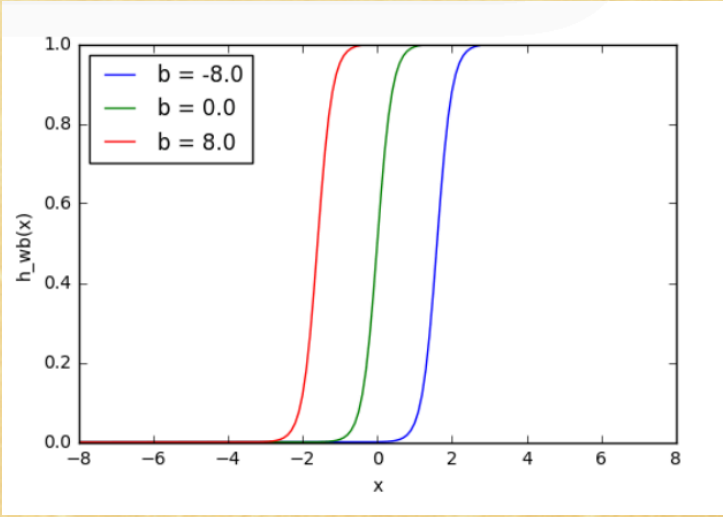


Fig 3b: Bias changes



Cost Function

As described in the previous lecture, one would minimize the error between the expected and the actual outputs and find the weights corresponding to the minimum error. This could be generalized by introducing cost function. The cost function for a single training pair is defined as

$$J(w,b,x,y) = \frac{1}{2} \| y^z - h^{(n_l)}(x^z) \|^2$$
$$= \frac{1}{2} \| y^z - y_{pred}(x^z) \|^2$$

Where h(x) is the output. This is the cost function for the z^{th} training sample. The vertical lines are sum of the squares of errors. The ½ is for the ease of taking derivative of the cost function.

The formulation for the cost function is for a single pair of nodes (x,y). We want to define the cost function over all the m training nodes. The mean square error is:

$$J(w,b) = \frac{1}{m}\sum_{z=0}^{m}\frac{1}{2} \parallel y^z - h^{(n_l)}(x^z)\|^2$$

$$= \frac{1}{m}\sum_{z=0}^{m} J(W,b,x^{(z)},y^{(z)})$$

The source function is used in gradient descent and backpropagation to estimate the weight and the bias.

Convolution

Convolution is a simple mathematical operation between two matrices. Consider a 6x6 matrix A and a 3x3 matrix B. Convolution between A and B is denoted by (A * B) and results in a new matrix C whose elements are obtained by sum of the produces of elements between matrix of A and B.

The convolution is shown visually in Figure 4. Matrix B is the lighter shaded one which slides over the darker matrix A from left-top to right-bottom. At each overlapping position, the corresponding elements of A and B are multiplied with all the products added to obtain the corresponding elements of C. The resulting matrix C will be smaller in dimension than matrix A and larger than matrix B.

What is the property of the convolution operation? Matrix A is typically a raw image where each cell in the matrix is a pixel value and matrix B is called a filter (or kernel) which when convolved with the raw image, results in a new image that highlights only certain features of the raw image. All the three images are pixel maps. In this case the filter will work to change the shape of the initial pixel image (Figures 4a and 4b).

Convolutions are useful in order to identify and detect basic features in images such as edges. The challenge is to find the right filter for a given image. ML can be used to find optimal filter values.

Determining the filter was a matter of finding the 3x3=9 values of matrix B in the above example. Matrix A has n(width) x n(height) pixels for a greyscale image. Standard color images have three channels: red, green, blue. Color images are easily handled by a 3D matrix n(width) x n(height) x n(channels) which are convolved with as many different filters as there are color channels.

Matrix C is smaller than the raw image A. If the dimensions of A and B are known, one could estimate dimension of C. Lets assume that n(width)=n(height)=n. If the filter is also a square matrix of size f, then the output C is square of dimension $n - f + 1$. In the above case n=6, f=3 and therefore C has dimension 4 x 4 (Figure 4a).

Figure 4a: Visual representation of convolution

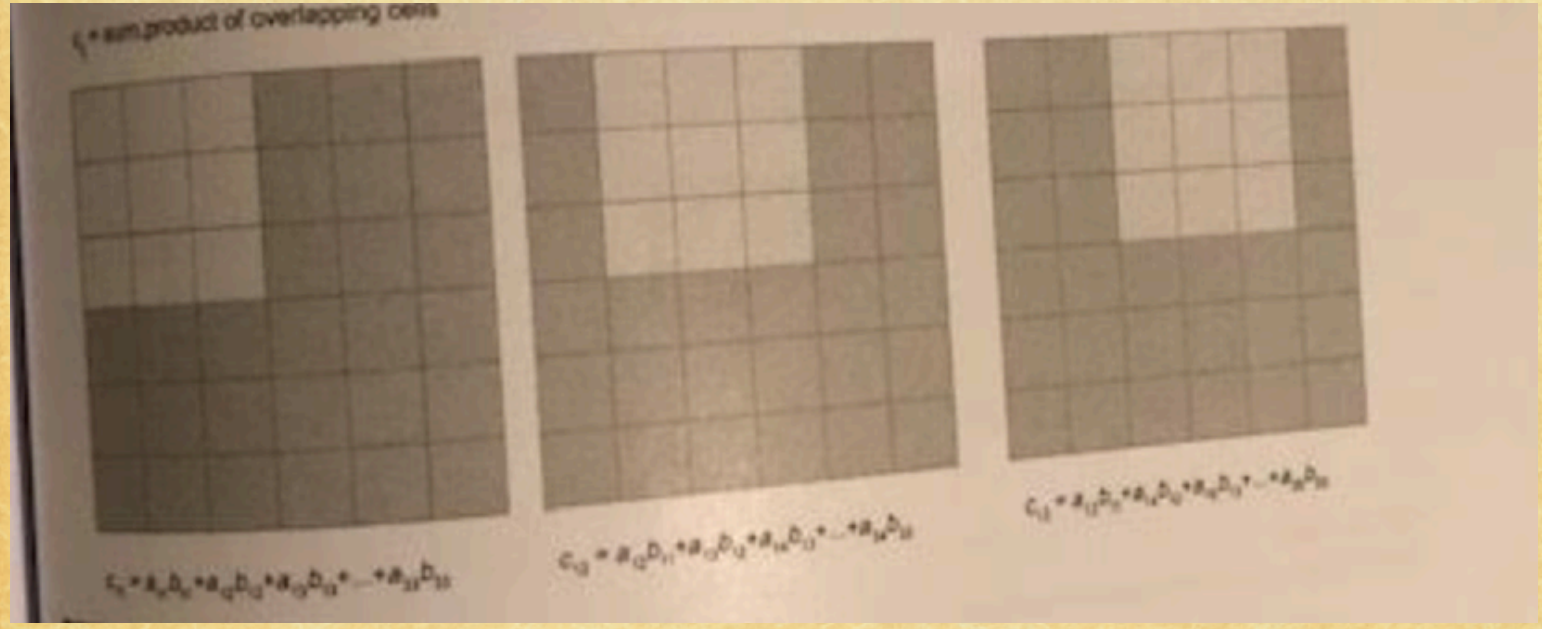
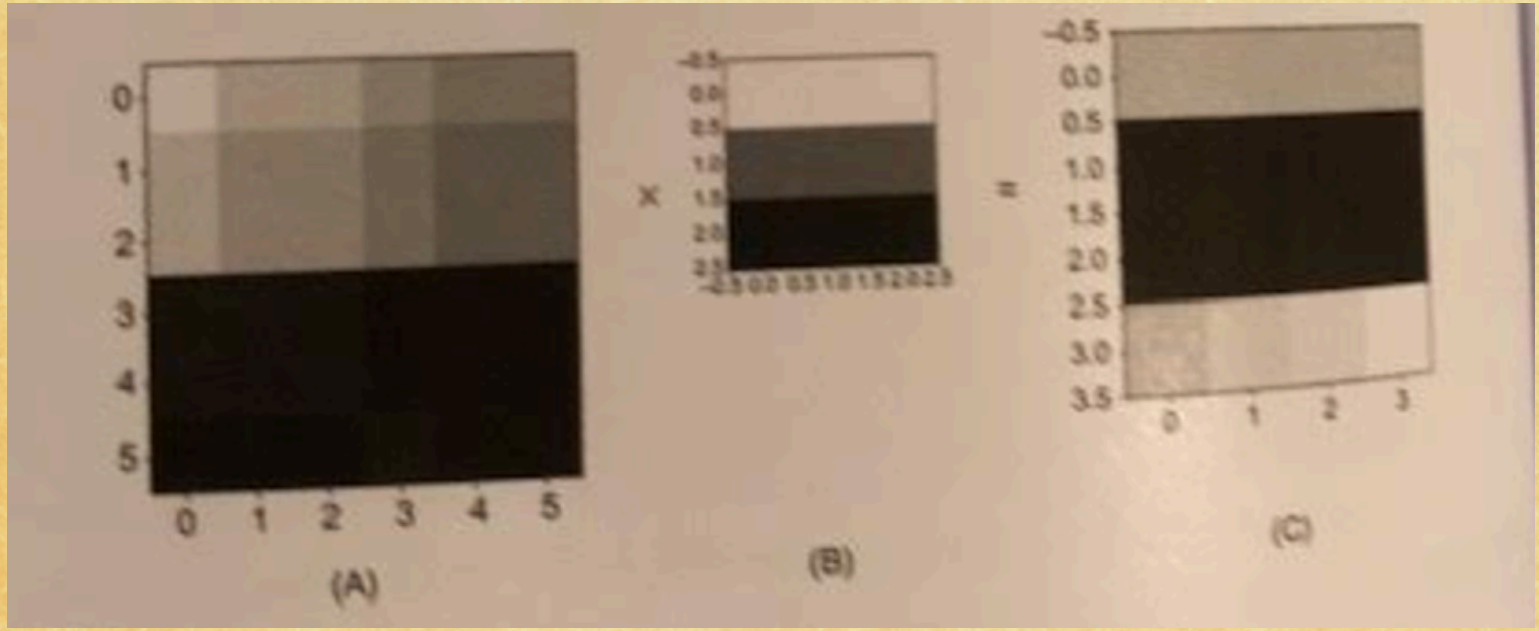


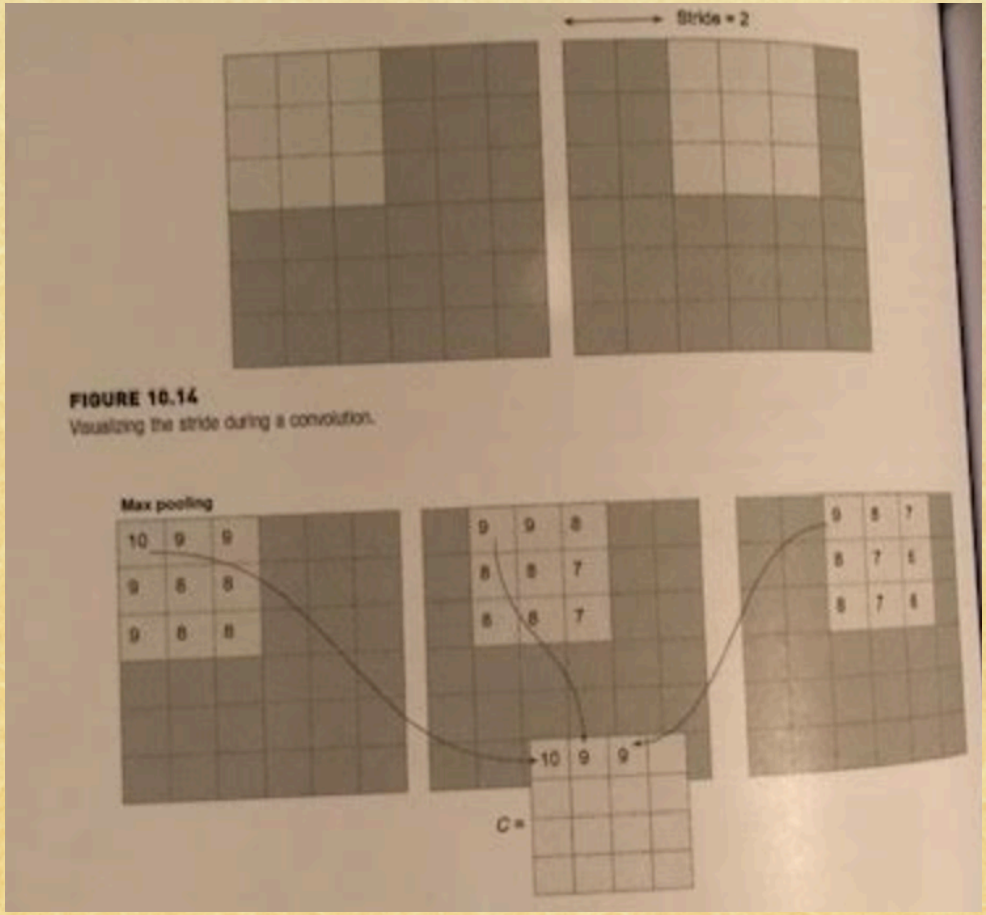
Figure 4b: Shows the effect of filter colors (depth)



Max Pooling

As we discussed, the convolution of two is calculated by the sum-product of the elements of the two matrices- the raw and filter matrix. However, instead of performing a sum-product, if one simply used the highest pixel value in the overlapping cells at each computed location, a process called max pooling would be obtained (Fig 5). Max pooling has the property that identifies the most dominant feature in an image. This is another feature detection tactic widely used in image processing. Similar to max pooling, an average pooling could also be to the raw image where the values are averages in the overlapping cells, instead of doing a sum-product.

Fig 5: Max Pooling



Sometimes the elements of matrix C are passed through a RELU non-linearity. This forms one convolutional layer. The output of this convolutional layer is sent to the next layer which can of another convolutional layer (with a different filter)or flattened and sent to a regular layer of nodes- Fully Connected.

Suppose A[0] is the raw image and C is the result of its convolution with filter B. A[1] is the result of adding a bias term to each element of C and passing them through a ReLU activation function. B is analogous to a weight matrix b. Convolution could therefore be used as a part of a deep learning network. In an NN backpropagation can be used to compute the elements of the weight matrix b with a similar process used to find the elements of a filter matrix B.

ReLU stands for rectified linear unit, and is a type of activation function. Mathematically, it is defined as $y = max(0, x)$.

Multiple filters can be applied to the same layer. Let B1 be a horizontal edge detector and B2 a vertical edge detector. Both filters could be applied to A[0]. The output C can be represented as a volume with dimension 4x4x2 – stacking two 4x4 matrices, each the result of the convolution between A and B.

To determine the filter function, backpropagation needs to be used. Therefore, a cost function needs to be constructed and minimized/maximized using gradient descent.

Convolutional Neaural Network- A Review

For each pixel in the input image, we encoded the pixel's intensity as the value for a corresponding neuron in the input layer. For the 28×28 pixel images we have been using, this means our network has 28x28 = 784 input neurons (Figure 2). We then trained the network's weights and biases so that the network's output would correctly identify the input images.

However, using networks with fully-connected layers to classify images have problems. The reason is that such a network architecture does not take into account the spatial structure of the images. For instance, it treats input pixels which are far apart and close together on exactly the same footing. Such concepts of spatial structure must instead be inferred from the training data. But what if, instead of starting with a network architecture, we used an architecture which tries to take advantage of the spatial structure? This brings us to the convolutional neural networks.

Convolutional neural networks use three basic ideas: local receptive fields, shared weights, and pooling. Let's look at each of these ideas in turn.

Local receptive fields: In the fully-connected layers shown earlier, the inputs were depicted as a vertical line of neurons. In a convolutional net, it'll help to think instead of the inputs as a 28x28 pixel intensities we are using as inputs. We connect the input pixels to a layer of hidden neurons. But we won't connect every input pixel to every hidden neuron. Instead, we only make connections in small, localized regions of the input image. For example, each neuron in the first hidden layer will be connected to a small region of the input neurons- a sample of 5x5 region corresponding to 25 pixels (Figure 6a, b). For a particular hidden neuron, we might have connections that look like this:

Figure 6a: The network treats the image for spatial features. A sample of 5 x 5 pixels are selected across the image to allow identification of the spatial features.

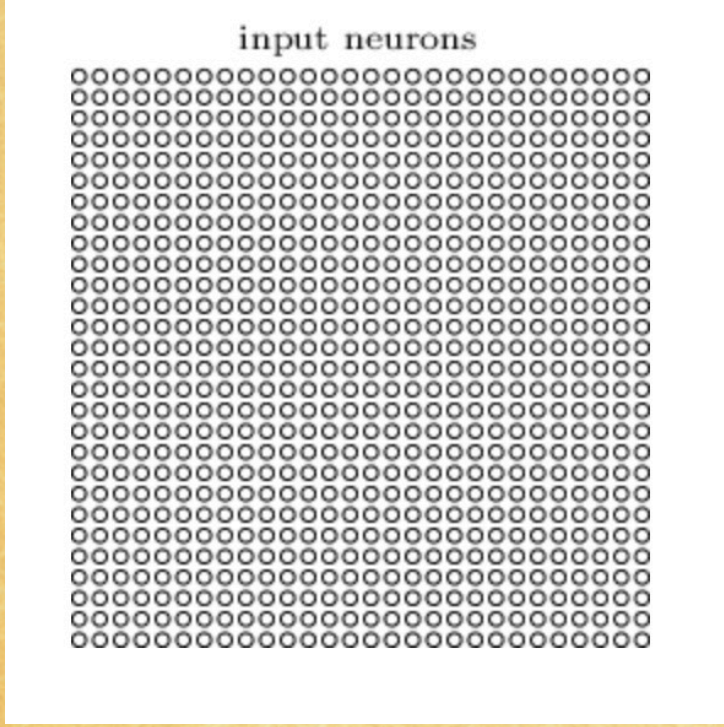
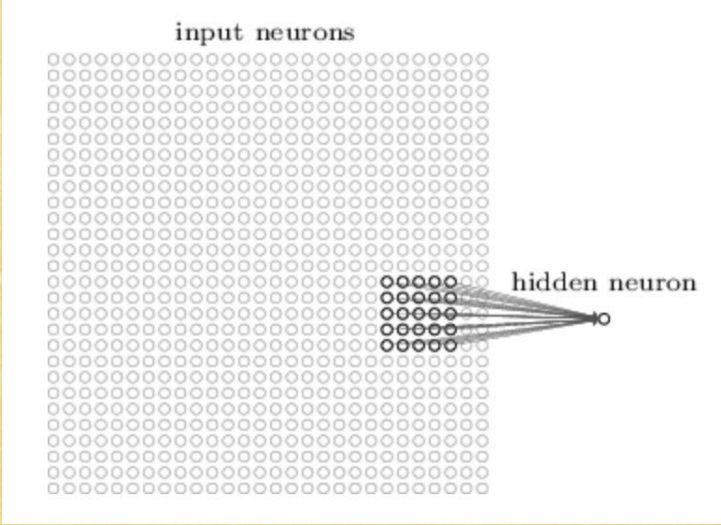


Figure 6b: Shows local respective field shifted to the first pixel



Building the Hidden Layers

That region in the input image is called the local receptive field for the hidden neuron. It's a little window on the input pixels. Each connection learns a weight. And the hidden neuron learns an overall bias as well. You can think of that particular hidden neuron as learning to analyze its particular local receptive field. We then slide the local receptive field across the entire input image (Figure 6c, d). For each local receptive field, there is a different hidden neuron in the first hidden layer.

Then we slide the local receptive field over by one pixel to the right (i.e., by one neuron), to connect to a second hidden neuron And so on, building up the first hidden layer. If we have a 28x28 input image and 5x5 local representative field, then there will be 24x24 nureons in the hidden layer (Figure 6d).

Figure 6c: Shows local respective field shifted to the first pixel.

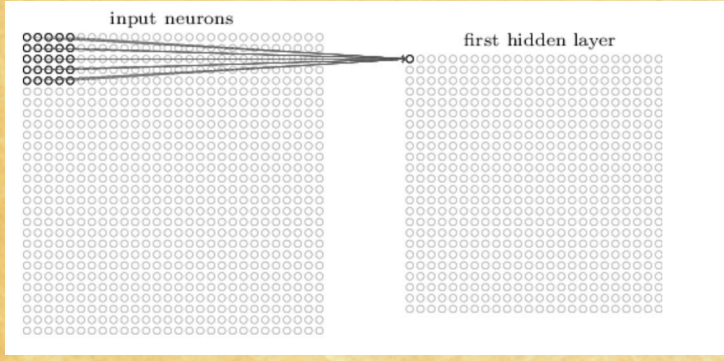
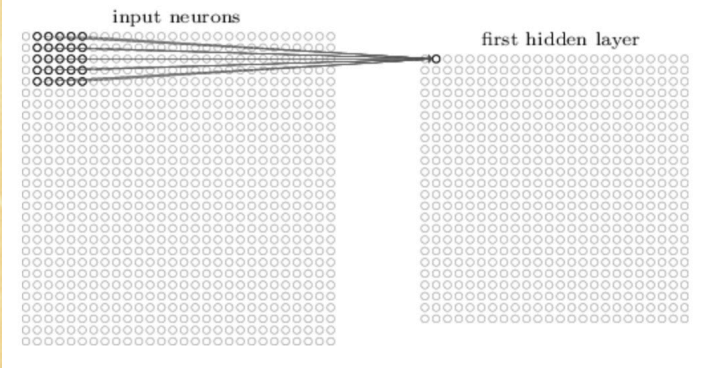


Figure 6d: The local respective field is shifted by one pixel.



Shared weights and biases: Each hidden neuron has a bias and 5x5 weights connected to its local receptive field. We use the same weights and bias for each of the 24x24 hidden neurons. For example for J-th and k-th hidden nureon the output is:

$$\sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{lm}\, a_{j+l,k+m}\right)$$

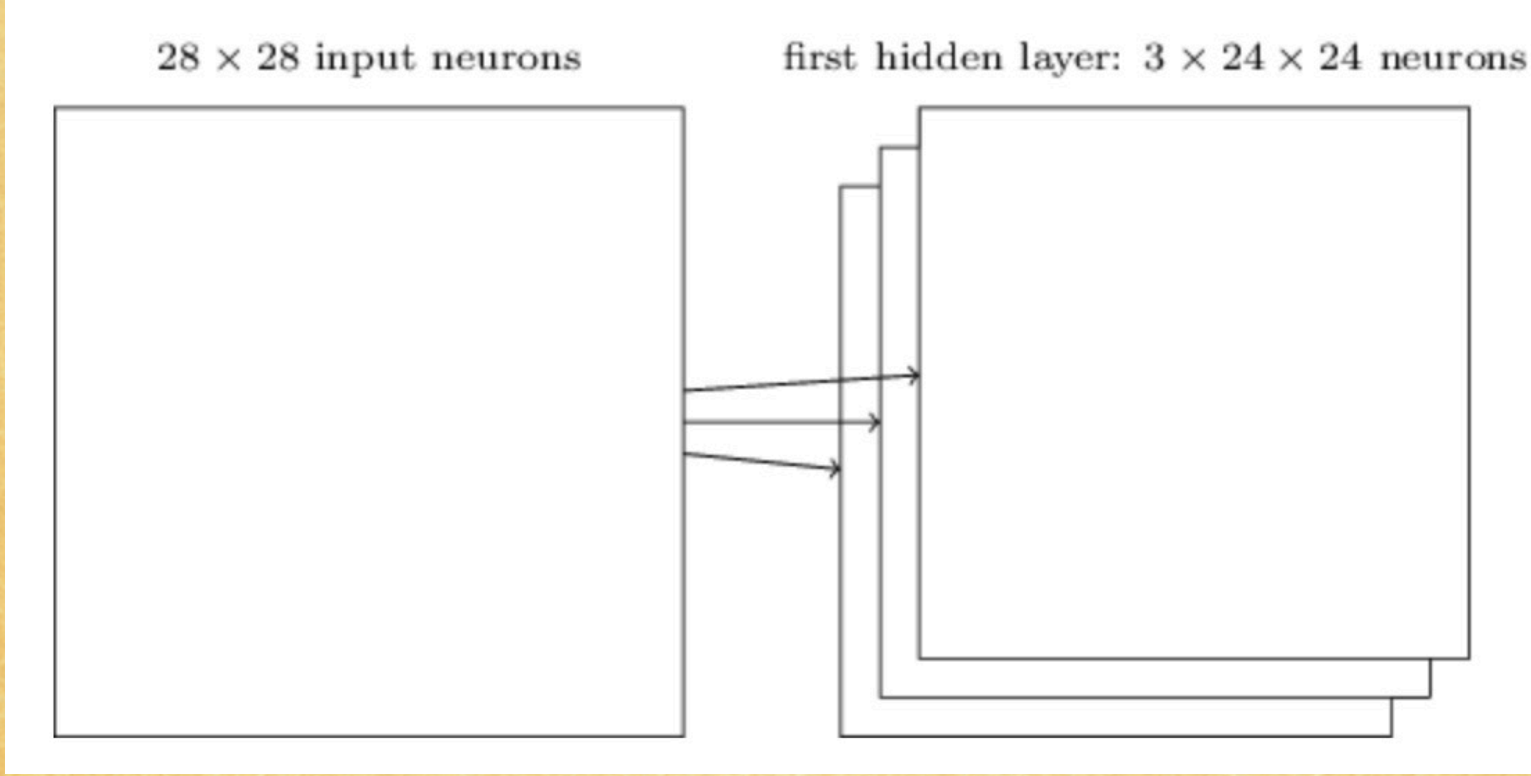
where σ is the neural activation function (eg. sigmoid function). b is the shared value for the bias. w_{lm} is a 5x5 array of shared weights. We use a_{xy} to denote the input activation at position x,y. This means that all the neurons in the first hidden layer detect exactly the same feature just at different locations in the input image. A property of convolutional networks is that they are well adapted to the translation invariance of images (Figure 7).

For this reason, we sometimes call the map from the input layer to the hidden layer a feature map. We call the weights defining the feature map the shared weights.

The network structure described so far can detect just a single kind of localized feature. To do image recognition we'll need more than one feature map. And so a complete convolutional layer consists of several different feature maps (Figure 7).

In Figure 7 we show three different feature maps each defined by a set of 5x5 shared weights. The result is that the network can detect 3x3 different kinds of features, with each feature being detectable across the entire image.

Figure 7: shows three different hidden layers



Putting it all together: We can now put all these ideas together to form a complete convolutional neural network (Figures 8 and 9). It's similar to the architecture we were just looking at, but has the addition of a layer of 10x10 output neurons.

The network begins with 28×28 input neurons, which are used to encode the pixel intensities. This is then followed by a convolutional layer using a 5×5 local receptive field and 3x3 feature maps. The result is a layer of 3×24×24 hidden feature neurons (Figure 8).

Figure 8: Shows input and output from a complex network

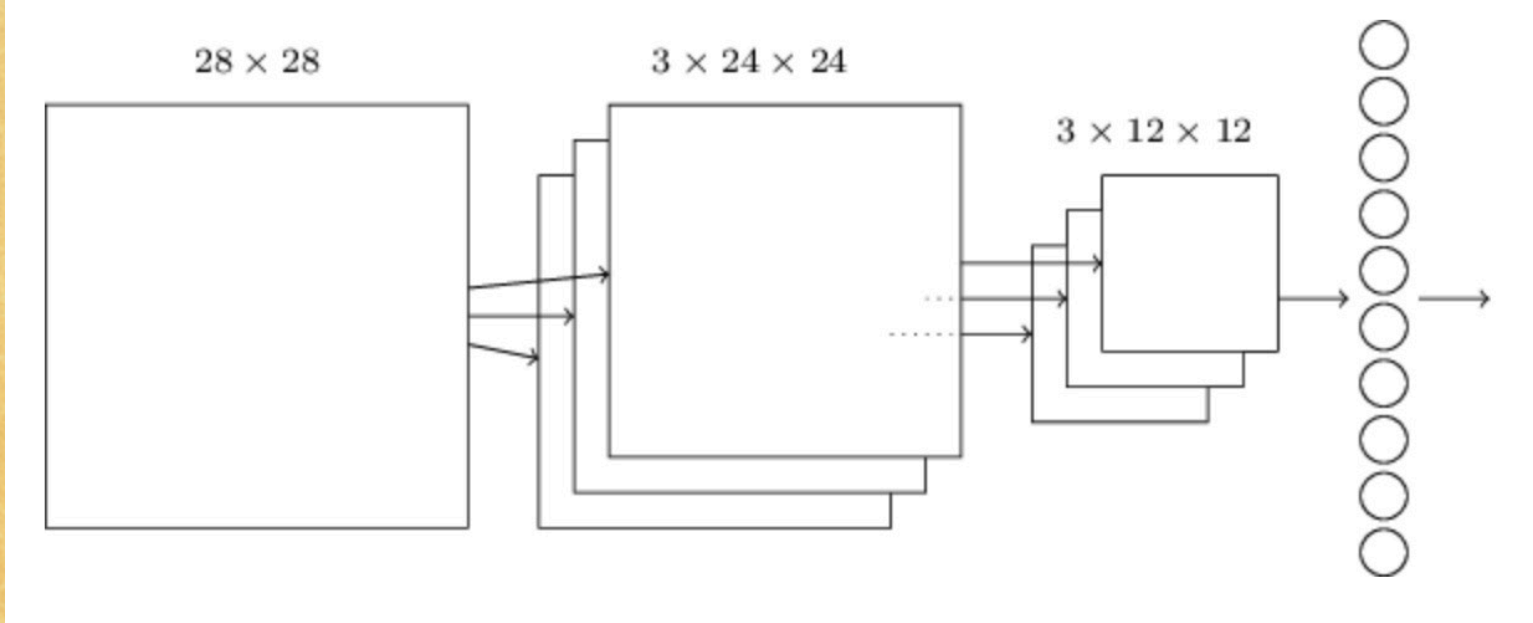
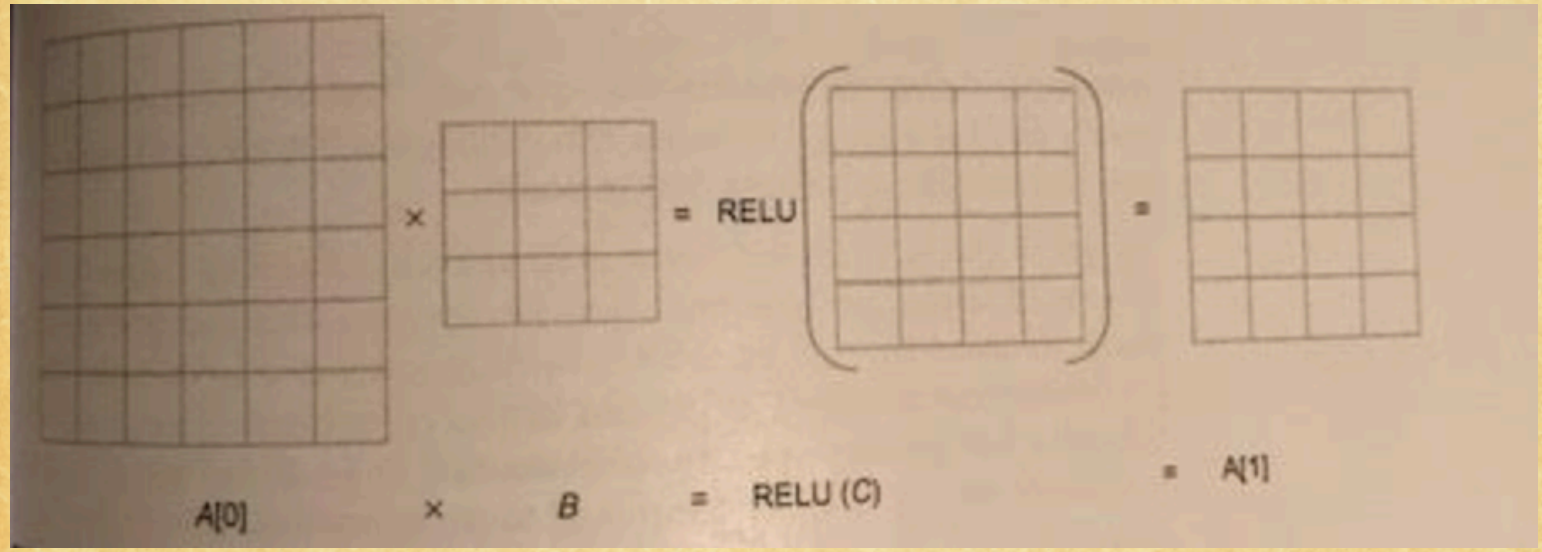


Figure 9: Neural net with different hidden layers

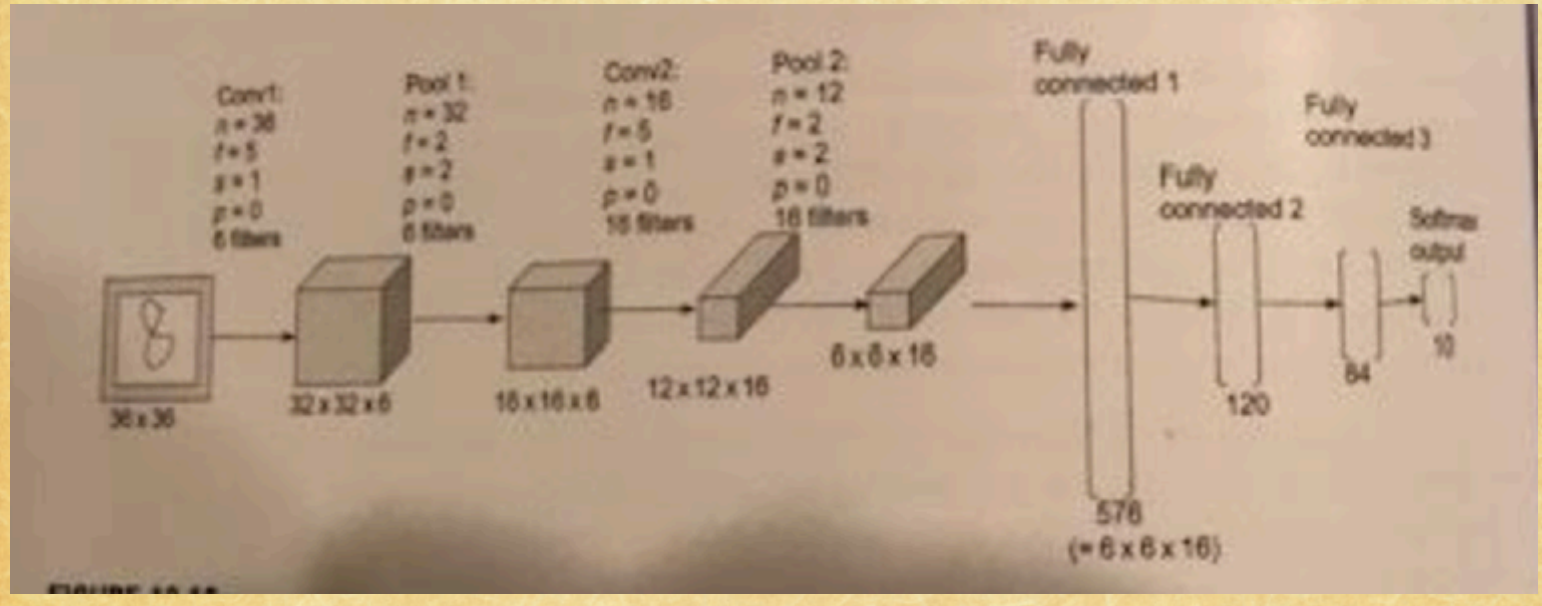
- Figure 12:



A Classic CNN Arcitecture

A classic CNN arcitecture could consist of several convolutional layers interspersed with max pool layers and followed by fully connected layers where the last convolution matrix is flattened to its constituent elements and passed through a few hidden layers before terminating at the output layer (Figure 10).

Figure 10: Example of a classical Neural Net



CNN and Deep Learning

CNN are strong deep learning networks because of the following reasons:

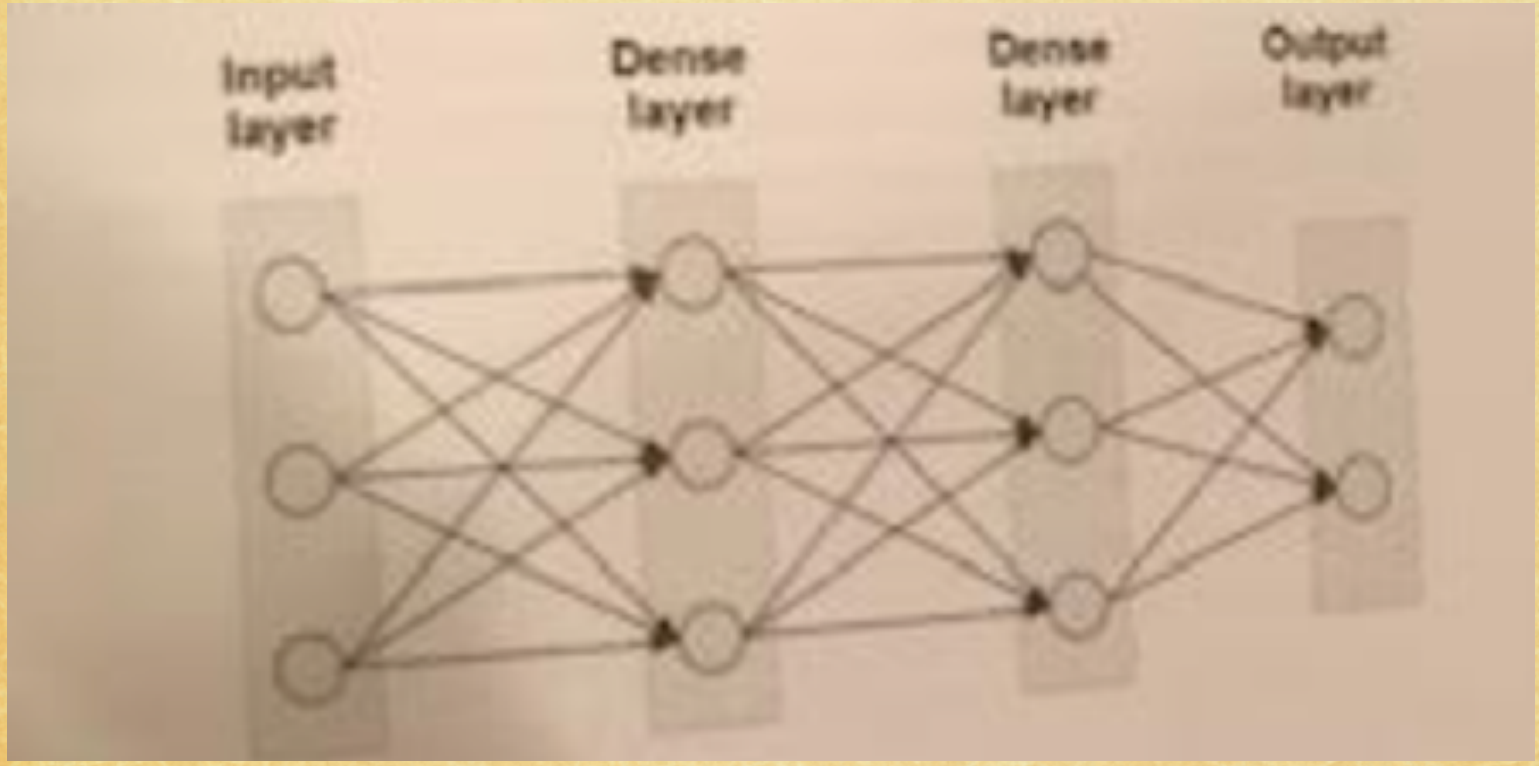
The feature detection layers (such as Conv1, Conv2, etc) are computationally fast because there are a few parameters to train (eg. each Conv1 filter is 5x5 which yields 25 + 1 (for matrix elements and bias) times 6 filters. This results in 156 parameters. Also, not every parameter in a layer is connected to all the parameters in the next layer as happens in fully connected layers- FC1 and FC2 have 576 x 120 = 69,120 parameters to train. Because of their efficiency and flexibility, CNN is a common deep learning technique.

Dense Layers, Drop out Layers

A dense or fully connected layer is one where all the nodes in the prior layer are connected to all the nodes in the next layer (Figure 11).

A dropout layer helps to prevent model overfitting by dropping the nodes randomly in the layer. The probability of dropping a node is controlled by a factor ranging between 0 and 1. A drop out factor closer to 1 drops more nodes from the layer. This is similar to regularization that reduces the complexity of the model.

Figure 11: Example of a desns neural net



Recurrent Neural Network

Apart from CNN, another widely used technique in deep learning is Recurrent Neural Network (RNN). This is used where data has a temporal component. Examples are time series, sensor data or language dependent data. The idea behind RNN is to train a network by passing the training data through it in a sequence where each example is an ordered sequence. In the example in Fig 12 , $x^{<t>}$ are the inputs where <*t*> indicates the position in the sequence. There are as many sequences as there are samples. $y^{<t>}$ are the predictions which are made for each position based on the training data. The training will determine the set of weights of this network, b_x which is a linear combination with $x_i^{<t>}$ and passed through a non-linear activation produces an activation matrix $a^{<t>}$

$$a^{<t>} = g\,(b_x\, x^{<t>})$$

RNN also uses the value of the activation from the previous time step (or previous word in a sentence since a word depends on its previous word). Therefore the value of activation matrix can be modified by adding the previous steps activation multiplied by another coefficient, b_a

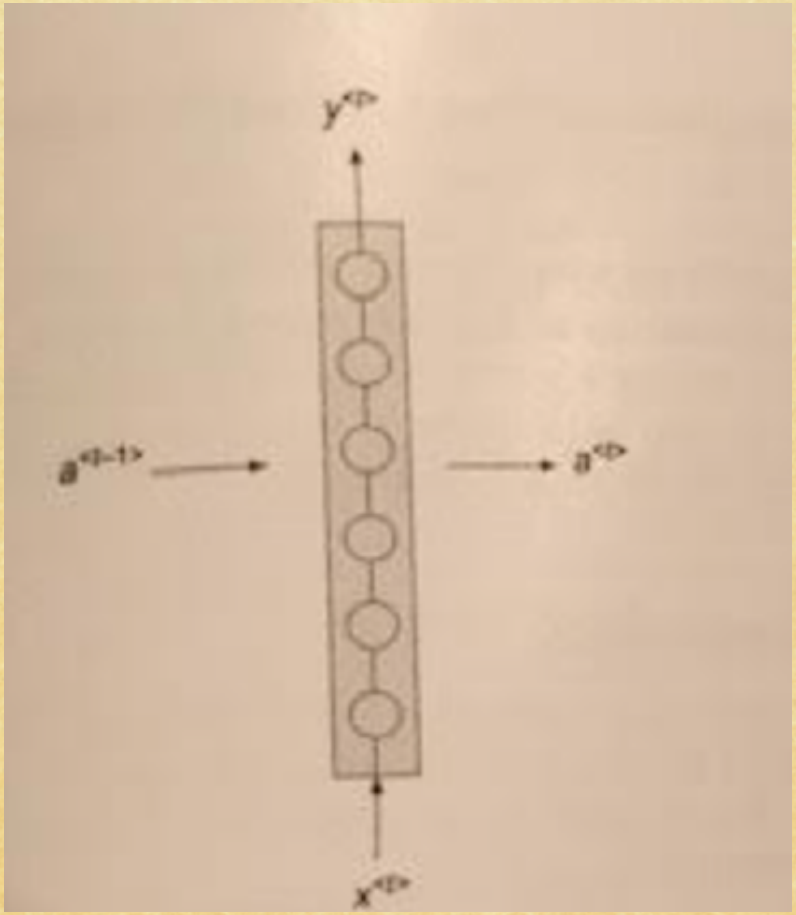
$$a^{<t>} = g\,(b_a\, a^{<t-1>} + b_x\, x^{<t>})$$

Finally the prediction for position <t> is given by

$$Y^{<t>} = g\,(b_y\, a^{<t>})$$

Where b_y is another set of coefficients. All the coefficients are found through the learning process using backpropagation (Figure 12).

Figure 12: Recurrent Neural Net



Sources used for this lecture

1. Neural Networks Tutorial – A pathway to deep learning

https://adventuresinmachinelearning.com/neural-networks-tutorial/

2. Data Science Concepts and Practice (second edition)

Vijay Kotu and Bala Deshpande

3. *"Neural Networks and Deep Learning" Chapter 6*

http://neuralnetworksanddeeplearning.com/chap6.html

by Michael Nielsen