PG Diploma in Data Analytics

Course: Predictive Analytics - II

Instructor: Prof. G. Srinivasaraghavan

PGDDA-Lecture Notes/PA-II/5

Neural Networks & Deep Learning

Contents

List of Figures

List of Tables

1 Learning Objectives

2 Introduction

Neural Networks are one of the many bio-inspired models used in computer science and machine learning in particular. The broad idea is to have a network of *neurons* that exhibits characteristics that we associate with the 'intelligence' that is manifested in the human brain. A hypothesis such as to build 'intelligent machines' rests on three 'take aways' from the way the human brain functions and we as humans perceive, learn and react to the world around us.

1. A large complex network of almost identical, but simple devices (neurons) can is adequate 'hardware' on which to build an 'intelligent system'.
2. It requires extensive training on lots of data for the network to 'learn'.
3. The system must embody a mechanism to represent objects at multiple levels of abstraction. Each representation is a 'feature vector' for the object that captures certain characteristics of the object and ignores some others.

Modern deep learning incorporates all of these aspects to a significant extent. See Figure 1 for an illustration of learning happening at multiple layers of abstraction in a deep neural network.

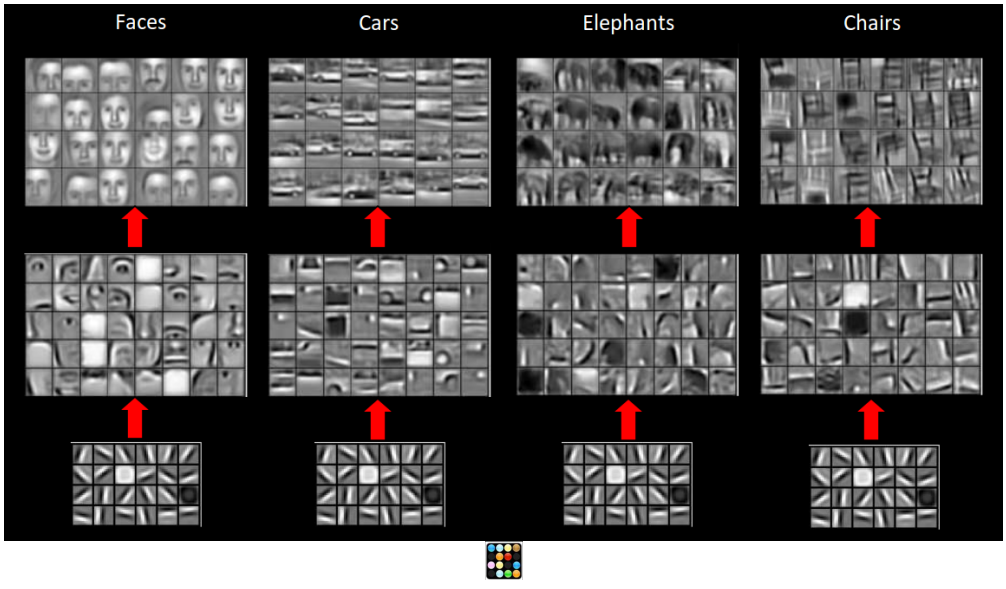


Figure 1: Learning at Different Levels of Abstraction. **Source**: talk delivered by Andrew Ng at IPAM 2012

The neurons used in Artificial Neural Networks (ANN) are (conceptual) devices that takes inputs from various other neurons, computes their *weighted* sum, adds its own *bias* and finally

generates a transformed version (using an *activation function*) of the sum as its output. The output of each neuron can in turn feed into several other neurons. The basic artificial neuron is illustrated in Figure 2.

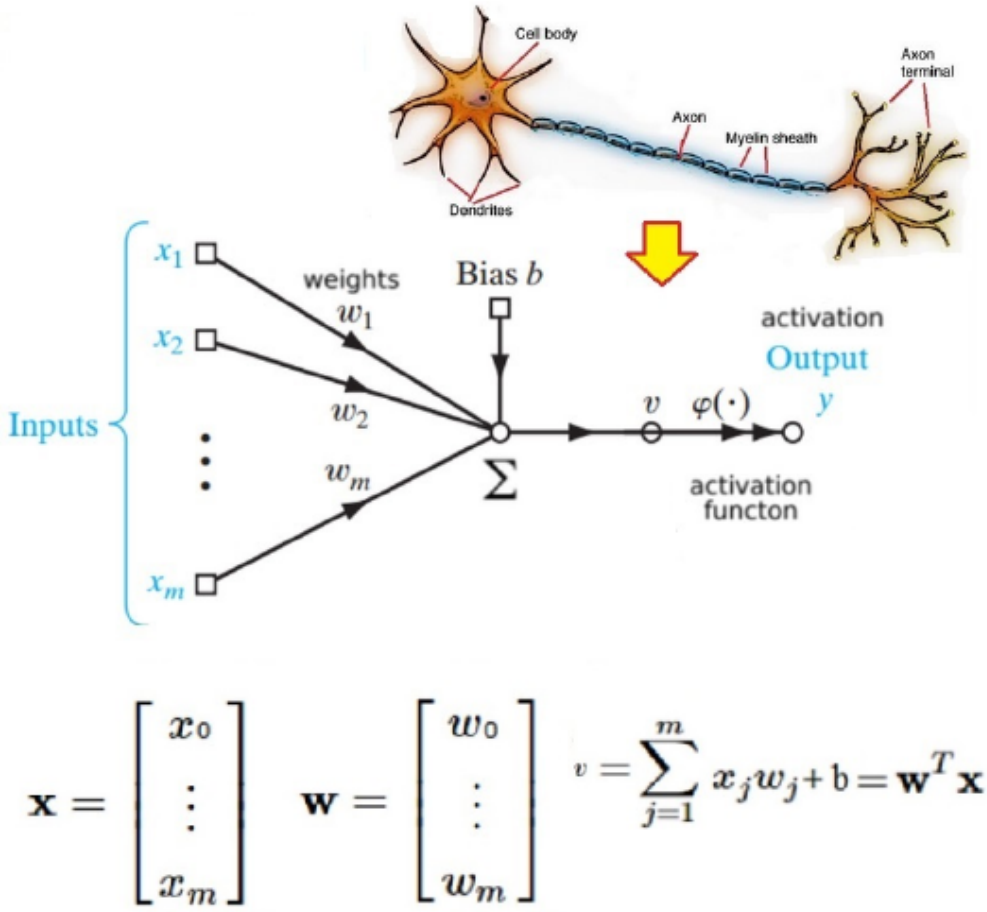


Figure 2: Basic Artificial Neuron

2.1 Simplifying Assumptions

There are a number of assumptions that are typically made about the structure of the neural network to make our analysis (training them and inferring from them) of the networks tractable, both mathematically and computationally. Each of these assumptions are typically at work in most networks developed in practise, by and large. In any given specific network architecture however one or more of these assumptions may not hold and the deviations from these assumptions are largely governed by ease of training and of course requirements of the particular problem at hand.

1. The neurons are arranged in *layers*. Neurons within a layer are not connected to each other directly.

2. Layers are arranged in an ordered sequence — connections exist only between pairs of adjacent layers in the ordering. Usually the first layer is treated as an *input* layer and is typically just a pass-through of the input data to the rest of the network. The last layer is considered th *output* layer. The others are the *hidden* layers.

3. All neurons (except possibly the output layer neurons) use the same activation function.

4. Every neuron in one layer is connected to every neuron in the next layer in the ordering. The connections are 'directed' — we assume that the input to the network gets transformed and passed on from layer to layer from the input layer to the output layer. The 'movement' of information is one-way through the network.

5. Each connection between two neurons has a weight associated with it and the weights for all the connections are independent of each other.

There are many variations of this broad theme that have been explored in the recent past. However the analysis for networks in which all the above assumptions hold will more-or-less carry through to all the other variations, with minor modifications. Unless otherwise specified we therefore assume in the rest of this module that all the above assumptions hold for the networks we consider.

2.2 Specifying an Artificial Neural Network — Model Parameters

Given the above assumptions, specifying a neural network model involves specifying the following:

1. The network *topology* — the number of layers and the number of neurons in each layer. A concise way to specifying this would be as a list of numbers $(n_0, n_1, \ldots, n_h, n_{h+1})$ for a network that consists of h hidden layers, the 0^{th} layer being the input layer and the $(h+1)^{th}$ layer the output the layer, in which the i^{th} layer contains n_i neurons for $0 \leq i \leq (h+1)$.

2. The activation function σ for each neuron in the network.

3. The weights for all the connections between neurons. We need one set of weights for the incoming connections for the neurons in each layer. Let's denote the weights for the incoming connections to the neurons layer l as the matrix $W^{(l)}, 1 \leq l \leq (h+1)$. The weights associated with the inputs to the i^{th} neuron in the l^{th} layer would be $\left(w_{i1}^{(l)}, \ldots, w_{in_{l-1}}^{(l)}\right), 1 \leq i \leq n_l$ — these form one row of $W^{(l)}$. For each $1 \leq l \leq (h+1)$, the weight matrix $W^{(l)}$ is of order $(n_l \times n_{l-1})$.

4. The biases of the neurons in the l^{th} layer are represented by the vector $\boldsymbol{b}^{(l)}$ of length n_l.

The first two (network topology and activation function) are usually fixed before the network is trained. The network training is usually to find the best possible values for the weights and biases. We will explore the training mechanism for neural networks in lot more detail later in this

document. Note that the behavior of a neural network (given its topology and the activation functions) is determined entirely by its weights and biases. Neural networks being 'universal approximators' one could in principle train any network for any task, by designing an appropriate cost function and finding a set of weights and biases that would minimize that cost function. The same 'network' would do different things based on the values of the weights and biases. Of course each network architecture would be 'good' at certain tasks (it is possible to train them with a high accuracy in carrying out that task) and possibly not so good at others. A schematic of a neural network is shown in Figure 3 illustrating all the concepts introduced above.

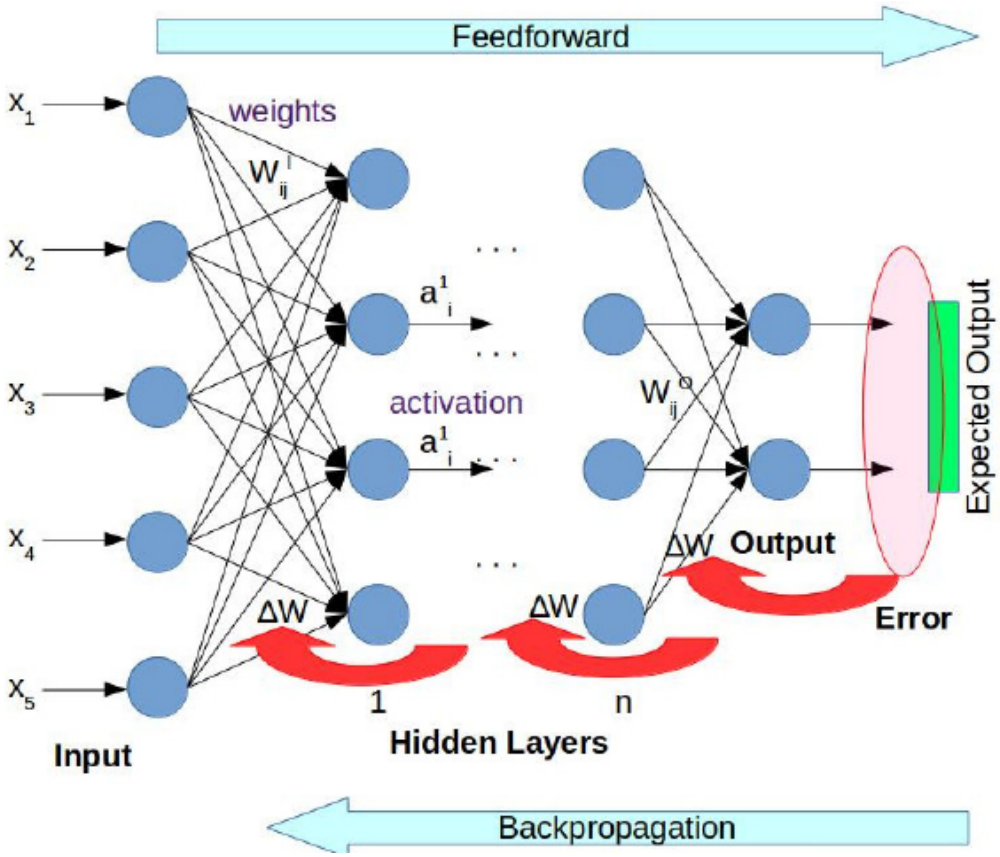


Figure 3: Standard Neural Network

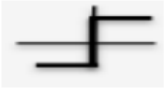
2.2.1 Commonly Used Activation Functions

Some of the commonly used activation functions are shown in Figure 4.

2.3 Some Schematic Examples of ANNs

2.3.1 Image Classification

Input is an image that needs to be classified. The size of the input layer is the number of pixels in the image and the pixel values are the inputs to the network. The output layer has as many

	Unit Step	$\sigma(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$
	Linear	$\sigma(z) = z$
	RelU	$\sigma(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$
	Sigmoid	$\sigma(z) = \frac{1}{1+e^{-z}}$
	Tanh	$\sigma(z) = tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$

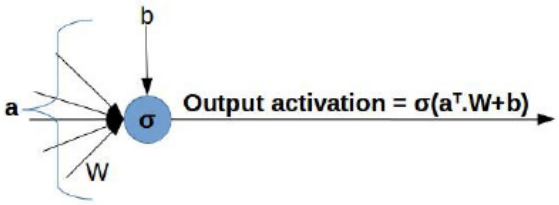


Figure 4: Commonly Used Activation Functions

neurons as the number of classes. For binary (0/1) classification the output layer would be just a single neuron whose output gives the probability that the input image belongs to class 1. In this case the activation function would be just the logistic function that we used for logistic regression. Assuming the weights of the inputs to the last layer neuron are $w_1, \ldots, w_m$ and the outputs of the neurons in the previous layer (with m neurons) are $a_1, \ldots, a_m$ then the output of the network is

$$p(y = 1 \mid \text{image}) = \frac{1}{1 + e^{-\sum_{i=1}^{m} w_i . a_i}}$$

For multiclass classification the last layer would typically be a *softmax* layer — this is the multiclass extension of the logistic function. Suppose there are k classes, the output layer would have k neurons, and the i^{th} neuron gives the probability that the input image belongs to class i. Suppose the weights of the inputs to the i^{th} neuron in the last layer are $w_{i1}, \ldots, w_{im}$ then the output of the i^{th} neuron in the last layer would be

$$p(y = i \mid \text{image}) = \frac{e^{\sum_{j=1}^{m} w_{ij} . a_j}}{\sum_{l=1}^{k} e^{\sum_{j=1}^{m} w_{lj} . a_j}}$$

See Figure 5 for an illustration.

2.3.2 Standard Regression

The explanatory variables form the inputs to the network — size of the input layer is the number of explanatory features. The network produces a single output with the value of the response variable which is compared against the expected value.

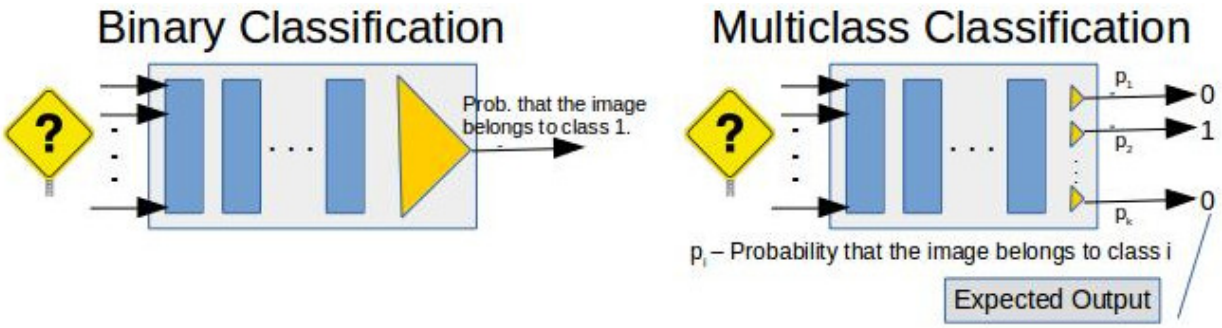


Figure 5: ANN for Classification — Illustration

3 Inference and Training in Neural Networks

Inference is the (simple) problem is computing the network output given an input. This is what we do when we 'use' a network after it has been trained i.e., the weights and biases have been determined, and have been fixed. We first discuss how inference is carried out in a neural network and then examine how a given network (with the topology and activation functions fixed) can be trained for a particular task. The basic inference mechanism in neural networks is called *feedforward* and the most basic training construct used is called *backpropagation*. We describe feedforward and backpropagation in the subsections below.

3.1 Feedforward

Suppose the network consists of h hidden layers with the nunbers of neurons in the layers given by the list $(n_0, n_1, \ldots, n_{h+1})$. Let's assume the activation function of each neuron in the network is σ. Let $W^{(l)}$ be the weight matrix for the connections between layers $(l-1)$ and l. $W^{(l)}$ will be of order $(n_l \times n_{l-1})$ — the i^{th} row of this matrix consists of the weights associated with the inputs to each of the neurons in the l^{th} layer. Each of the n_l neurons in the layer has n_{l-1} inputs coming into it (these are the outputs / activations from the neurons in the previous layer(s)). Finally let the (constant) bias at the i^{th} neuron in the l^{th} layer be $b_i^{(l)}$. The cumulative input $z_i^{(l)}$ (including the bias) at the i^{th} neuron in the layer is therefore $z_i^{(l)} = \sum_{j=1}^{m} w_{ij}^{(l)}.x_j + b_i^{(l)}$. It is written more concisely in the vector form. We denote the cumulative inputs to the neurons in the layer as the vector $\boldsymbol{z}^{(l)}$, the vector of biases of the neurons in the layer as $\boldsymbol{b}^{(l)}$ and the output of the $(l-1)^{th}$ layer as $\boldsymbol{a}^{(l-1)}$. Note that $\boldsymbol{z}^{(l)}$ and $\boldsymbol{b}^{(l)}$ are vectors of dimension n_l, $\boldsymbol{a}^{(l-1)}$ is of dimension n_{l-1}. We can now write the expression for $\boldsymbol{z}^{(l)}$ as

$$\boldsymbol{z}^{(l)} = W^{(l)}.\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}$$

The output of the l^{th} layer is obtained by applying the activation function σ on the cumulative inputs into each of the neurons in the layer. Therefore

$$\boldsymbol{a}^{(l)} = \sigma\left(\boldsymbol{z}^{(l)}\right) = \sigma\left(W^{(l)}.\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)}\right)$$

The network output $\boldsymbol{a}^{(h+1)}$ given an input vector $\boldsymbol{x}$ (of length n_0) can be computed as follows.

$$\begin{aligned}\boldsymbol{a}^{(1)} &= \sigma\left(W^{(1)}.\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)\\ \boldsymbol{a}^{(2)} &= \sigma\left(W^{(2)}.\boldsymbol{a}^{(1)} + \boldsymbol{b}^{(2)}\right)\\ \ldots &= \ldots\\ \boldsymbol{a}^{(h+1)} &= \sigma\left(W^{(h+1)}.\boldsymbol{a}^{(h)} + \boldsymbol{b}^{(h+1)}\right)\end{aligned}$$

This sequential computation scheme to compute the output of the network (inference) given an input is called the feedforward mechanism. Notice that we can think of the network as computing a (very complex) function $\mathscr{G}(\boldsymbol{x} \mid W^{(1)},\ldots,W^{(h+1)},\boldsymbol{b}^{(1)},\ldots,\boldsymbol{b}^{(h+1)})$ that gives a value for every input $\boldsymbol{x}$, given the set of weights and biases. Notice that feedforward boils down just a series of multiplications and additions over matrices and vectors along with of course applications of the activation function over vectors.

3.2 Backpropagation

Given a network (topology and the activation functions) how do we train the network for a particular task — for what values of the weights and biases will the network carry out the required task most accurately? This is the question that we seek to answer during training of the neural network. This is done by first defining a cost function that quantifies the difference between the output of the network (for some given values of the weights and biases) and the expected output. Notice that this is just a supervised learning scheme — we assume we have a training dataset for which the expected output (the value of the attribute, predicting which is the task that we are training the network for) is given for every one of the datapoints. The idea is to minimize the average cost incurred across all the datapoints. Suppose the dataset consists of datapoints $\{(\boldsymbol{x}_1,\boldsymbol{y}_1),\ldots,(\boldsymbol{x}_n,\boldsymbol{y}_n)\}$, where $\boldsymbol{x}_i$ is the input vector and $\boldsymbol{y}_i$ is the expected output. Let the cost function be $C(.,.)$ that takes two arguments — the predicted and the expected output — and returns a number that grows with the 'difference' between the two arguments. In our case the two arguments to the cost function would be $\mathscr{G}(\boldsymbol{x}_i \mid W^{(1)},\ldots,W^{(h+1)},\boldsymbol{b}^{(1)},\ldots,\boldsymbol{b}^{(h+1)})$ and $\boldsymbol{y}_i$ for the data points $1 \leq i \leq n$. The training problem in a neural network is therefore the minimization problem

$$\min_{W,b} \frac{1}{n}\sum_{i=1}^{n} C\left(\mathscr{G}\left(\boldsymbol{x}_i \mid W^{(1)},\ldots,W^{(h+1)},\boldsymbol{b}^{(1)},\ldots,\boldsymbol{b}^{(h+1)}\right),\boldsymbol{y}_i\right) = \min_{W,b}\mathscr{C}(W,b)$$

This minimization is for an extremely complex function with a very large number of parameters (see the box below). For brevity we refer to all the weight matrices together as W and the bias vectors together as b.

Parameters in a Neural Network

Consider a neural network with the topology $(n_0,n_1,\ldots,n_h,n_{h+1})$. For this network the weight matrix $W^{(l)}$ is of order $(n_l \times n_{l-1})$ and the bias vector $\boldsymbol{b}^{(l)}$ is of length n_l. The to-

tal number of parameters is therefore

$$\sum_{l=1}^{h+1} n_l(1+n_{l-1})$$

.

The optimization is therefore done using gradient descent. The general gradient descent scheme is shown in Algorithm 1. The gradient gives the 'direction' in which the function value increases

Algorithm 1 Gradient Descent in Neural Networks

1: Initialize starting values for all the parameters — $W, b = W_0, b_0$.
2: ▷ A common initialization strategy is to sample the W and b values from a normal distribution
3: **while** No appreciable change in W, b **or** $\mathscr{C}(W,b)$ **do**
4: Compute $\nabla W = \frac{\partial \mathscr{C}(W,b)}{\partial W}$ and $\nabla b = \frac{\partial \mathscr{C}(W,b)}{\partial b}$ ▷ Gradients of the cost function w.r.t W and b
5: *# Update weights and biases*
6: $W \leftarrow W - \eta.\nabla W$ ▷ η is the learning rate
7: $b \leftarrow b - \eta.\nabla b$
8: **end while**

most rapidly. We therefore take a step in the negative gradient direction (ours is a minimization problem). The learning rate η determines how big a step we take — too large a step could result in our overstepping the optimum, too small a step will mean too many iterations to reach the bottom. There is no universal prescription for how the learning rate is to be chosen (it can even be changed over iterations) is still an active area of research in neural network optimization. So the key issue here is to compute the gradients of the error/cost with respect to the weights and biases. Backpropagation is the scheme used to compute the gradients. An intuitive explanation of the entire neural network training scheme using backpropagation is given in the box below.

Training Neural Networks Using Backpropagation

Intuitively we start with some values for W, b. We carry out a feedforward for each given training data point and compare the output in each case to the expected output. The 'errors' incurred in each case are recorded and finally 'backpropagated' to determine by how much each of the weights and biases need to be adjusted to reduce the errors. This process is repeated till convergence.

The gradients $\nabla W, \nabla b$ are computed in a cascading manner — we first compute $\nabla W^{(h+1)} = \frac{\partial \mathscr{C}(W,b)}{\partial W^{(h+1)}}$ and $\nabla \boldsymbol{b}^{(h+1)} = \frac{\partial \mathscr{C}(W,b)}{\partial \boldsymbol{b}^{(h+1)}}$ (the gradients w.r.t the weights and biases for the output layer), and then work our way *back* through the network to compute $\nabla W^{(l)}, \nabla \boldsymbol{b}^{(l)}$ for $h \geq l \geq 1$, from the last hidden layer to the first. Refer Figure 3. To compute the gradient with respect to the weights and biases of the last (output) layer, let us first get some idea of the kind of cost functions that we could use for training neural networks.

3.2.1 Cost Functions

We look at two commonly used cost functions in neural network training — *square error* and *cross entropy*. Recall that a cost function needs to increase as the 'dissimilarity' between the network output and the expected value increases and if the two are identical then the cost must be zero. Clearly cost must be non-negative.

1. **Square Error**: Square error cost function is simply the square of the difference between the expected output and the network output. Consider any of the data points $(\boldsymbol{x}_i, \boldsymbol{y}_i)$. Network output on $\boldsymbol{x}_i$ is the vector $\boldsymbol{a}_i^{(h+1)}$ and the expected output is $\boldsymbol{y}_i$, both of length n_{h+1}. The square error on $\boldsymbol{x}_i$ is just the square distance between the two vectors

$$C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right) = \sum_{j=1}^{n_{h+1}} \left(a_{ij}^{(h+1)} - y_{ij}\right)^2$$

   Note that the square cost would be the same as the one we used for linear regression, if the network is trained for a regression task. With this the average cost function (across all data points) would be

$$\frac{1}{n}\sum_{i=1}^{n} C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right) = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{n_{h+1}} \left(a_{ij}^{(h+1)} - y_{ij}\right)^2$$

2. **Cross Entropy**: Cross entropy is typically used for classification problems. Suppose the output of the network for the input $\boldsymbol{x}_i$ is $a_{ij}^{(h+1)} = p_{ij}, 1 \leq j \leq k$ for a k-class classification problem. See Figure 5. The expected output is just the class of the given input converted into a k-dimensional vector (called one-hot encoding) — if the input belongs to class c then the one-hot vector representation is a k-d vector that has 1 in the c^{th} position and zero everywhere else. Let $\boldsymbol{y}_i$ be the one-hot encoding for the output on $\boldsymbol{x}_i$. Then cross entropy cost on input $\boldsymbol{x}_i$ is defined as

$$C_{ce}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right) = -\sum_{j=1}^{k} \left(y_{ij} \log p_{ij} + (1 - y_{ij}) \log(1 - p_{ij})\right)$$

   Note that we expect the network output p_{ij} to be close to 1 when $y_{ij} = 1$ and that $p_{ij} \approx 0$ when $y_{ij} = 0$. Cross entropy becomes very large if for some j, $p_{ij} << 1$ and $y_{ij} = 1$ or $p_{ij} \approx 1$ and $y_{ij} = 0$. Also if $p_{ij} \approx y_{ij}$ always then the cross entropy is close to zero. The total cost function would now be:

$$\frac{1}{n}\sum_{i=1}^{n} C_{ce}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right) = \frac{-1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} \left(y_{ij} \log a_{ij}^{(h+1)} + (1 - y_{ij}) \log(1 - a_{ij}^{(h+1)})\right)$$

We are now ready to compute the gradient $\nabla W^{(h+1)}, \nabla \boldsymbol{b}^{(h+1)}$ for the last layer. You may find the next two sections too heavy and technical. Please skip over to Algorithm 2 in case you want to skip the details of how these gradients are computed. The key point though is that it is possible to compute these gradients using backpropagation and that this forms the basis for most of the neural network training algorithms in use today.

3.2.2 Last Layer Gradient

Suppose we use the square error cost function. We already know the expression for the cost.

$$\sum_{i=1}^{n} C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right) = \sum_{i=1}^{n} \sum_{j=1}^{n_{h+1}} \left(a_{ij}^{(h+1)} - y_{ij}\right)^2$$

To compute the gradient we use the chain rule of differentiation. The key trick here is to use the cumulative input vector to the last layer. Recall that we defined the cumulative input as $\boldsymbol{z}^{(h+1)} = W^{(h+1)}.\boldsymbol{a}_i^{(h)} + \boldsymbol{b}^{(h+1)}$. So $\frac{\partial \boldsymbol{z}_i^{(h+1)}}{\partial W^{(h+1)}} = \boldsymbol{a}_i^{(h)}$. *Strictly speaking this statement is inaccurate — the derivative of a vector with respect to a matrix will be a 3-d tensor — the only elements of the tensor that will be non-zero in this case will be the elements of $\boldsymbol{a}_i^{(h)}$. That's the reason why all neural network implementations have extensive support for computations involving tensors. We will keep the precise details of the vector-matrix-tensor conversions and computations deliberately informal and leave the many gory details out in the rest of this discussion.* Also $\frac{\partial \boldsymbol{z}_i^{(h+1)}}{\partial \boldsymbol{b}^{(h+1)}} = \mathbf{1}$. Now using the chain rule we get

$$\begin{aligned}
\nabla W^{(h+1)} &= \sum_{i=1}^{n} \left(\frac{\partial C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial W^{(h+1)}} \right) = \sum_{i=1}^{n} \left(\frac{\partial C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(h+1)}} \frac{\partial \boldsymbol{z}_i^{(h+1)}}{\partial W^{(h+1)}} \right) \\
&= \sum_{i=1}^{n} \left(\frac{\partial C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(h+1)}} \right) \bigotimes [\![\boldsymbol{a}_i^{(h)}]\!]
\end{aligned}$$

where $\bigotimes$ denotes the generalized product between a vector and the tensor $[\![\boldsymbol{a}_i^{(h)}]\!]$ resulting in a matrix. To compute $\frac{\partial C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(h+1)}}$, we use the chain rule once more

$$\frac{\partial C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(h+1)}} = \frac{\partial C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{a}_i^{(h+1)}} . \frac{\partial \boldsymbol{a}_i^{(h+1)}}{\partial \boldsymbol{z}_i^{(h+1)}}$$

to split this derivative into the derivative with respect to $\boldsymbol{a}_i^{(h+1)}$ and then the gradient of $\boldsymbol{a}_i^{(h+1)}$ with respect to $\boldsymbol{z}_i^{(h+1)}$. Note that

$$\boldsymbol{a}_i^{(h+1)} = \sigma\left(\boldsymbol{z}_i^{(h+1)}\right) \Rightarrow \frac{\partial \boldsymbol{a}_i^{(h+1)}}{\partial \boldsymbol{z}_i^{(h+1)}} = \left[\sigma'\left(\boldsymbol{z}_i^{(h+1)}\right)\right]$$

where $\left[\sigma'\left(\boldsymbol{z}_i^{(h+1)}\right)\right]$ denotes the matrix consisting of copies of the vector $\sigma'\left(\boldsymbol{z}_i^{(h+1)}\right)$. Finally to compute $\frac{\partial C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{a}_i^{(h+1)}}$, we compute the derivative with respect to every component of $\boldsymbol{a}_i^{(h+1)}$. It is easy to see that

$$\frac{\partial C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial a_{ij}^{(h+1)}} = 2\left(a_{ij}^{(h+1)} - y_{ij}\right) \Rightarrow \frac{\partial C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{a}_i^{(h+1)}} = 2\left(\boldsymbol{a}_i^{(h+1)} - \boldsymbol{y}_i\right)$$

Putting everything together we have the expression for computing $\nabla W^{(h+1)}$:

$$\nabla W^{(h+1)} = 2\sum_{i=1}^{n}\left(\left(\left(\boldsymbol{a}_i^{(h+1)} - \boldsymbol{y}_i\right).\left[\sigma'\left(\boldsymbol{z}_i^{(h+1)}\right)\right]\right)\bigotimes[\![\boldsymbol{a}_i^{(h)}]\!]\right)$$

Similarly for b we would have

$$\nabla \boldsymbol{b}^{(h+1)} = \sum_{i=1}^{n}\left(\frac{\partial C_{Sq}\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(h+1)}}.\frac{\partial \boldsymbol{z}_i^{(h+1)}}{\partial \boldsymbol{b}^{(h+1)}}\right) = 2\sum_{i=1}^{n}\left(\boldsymbol{a}_i^{(h+1)} - \boldsymbol{y}_i\right).\left[\sigma'\left(\boldsymbol{z}_i^{(h+1)}\right)\right]$$

We could carry out these computations in a similar fashion for other cost functions like the cross entropy as well.

3.2.3 Gradients for Other Layers

Gradients for the other layers follow a generic pattern. Express the gradient with respect to the weights and biases for that layer in terms of the gradients with respect to the cumulative inputs to that layer. We also give a 'backpropagation' scheme that allows us to compute the cumulative input gradients for any layer in terms of the cumulative input gradients for the next layer. Suppose we want to compute $\nabla W^{(l)}$ and $\nabla \boldsymbol{b}^{(l)}$ for some l, $h \geq l \geq 1$. Repeated application of chain rule gives:

$$\nabla W^{(l)} = \sum_{i=1}^{n}\left(\frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial W^{(l)}}\right) = \sum_{i=1}^{n}\left(\frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(l)}}.\frac{\partial \boldsymbol{z}_i^{(l)}}{\partial W^{(l)}}\right) = \sum_{i=1}^{n}\left(\frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(l)}}\bigotimes[\![\boldsymbol{a}_i^{(l-1)}]\!]\right)$$

Similarly for $\nabla \boldsymbol{b}^{(l)}$ we have

$$\nabla \boldsymbol{b}^{(l)} = \sum_{i=1}^{n}\left(\frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{b}^{(l)}}\right) = \sum_{i=1}^{n}\left(\frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(l)}}.\frac{\partial \boldsymbol{z}_i^{(l)}}{\partial \boldsymbol{b}^{(l)}}\right) = \sum_{i=1}^{n}\left(\frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(l)}}\right)$$

Recall that $\boldsymbol{z}_i^{(l)} = W^{(l)}.\boldsymbol{a}_i^{(l-1)} + \boldsymbol{b}^{(l)}$. It remains to compute $\frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(l)}}$. This is where the 'backpropagation' comes in. The idea is again to use chain rule repeatedly.

$$\frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(l)}} = \frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(l+1)}}.\frac{\partial \boldsymbol{z}_i^{(l+1)}}{\partial \boldsymbol{a}_i^{(l)}}.\frac{\partial \boldsymbol{a}_i^{(l)}}{\partial \boldsymbol{z}_i^{(l)}} = \frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(l+1)}}.W^{(l+1)}.\left[\sigma'\left(\boldsymbol{z}_i^{(l)}\right)\right]$$

We can now summarize the full network training scheme using backpropagation as shown in Algorithm 2. Similar to W, b we use $\nabla \boldsymbol{z}_i^{(l)}$ to denote $\frac{\partial C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right)}{\partial \boldsymbol{z}_i^{(l)}}$.

Algorithm 2 Neural Network Training Using Backpropagation

Initialize starting values for all the network parameters — $W, b = W_0, b_0$.
while Convergence not reached **do**
  for $i = 1, \ldots, n$ **do**
    Feedforward - this will give us $\boldsymbol{z}_i^{(l)}, \boldsymbol{a}_i^{(l)}$ for all $1 \leq l \leq (h+1)$.
    # Compute the gradient contribution from the i^{th} data point
    $\nabla \boldsymbol{z}_i^{(h+1)} = \left(\boldsymbol{a}_i^{(h+1)} - \boldsymbol{y}_i\right).\left[\sigma'\left(\boldsymbol{z}_i^{(h+1)}\right)\right]$
    $\nabla W_i^{(h+1)} = \nabla \boldsymbol{z}_i^{(h+1)} \bigotimes [\![\boldsymbol{a}_i^{(h)}]\!]$
    $\nabla \boldsymbol{b}_i^{(h+1)} = \nabla \boldsymbol{z}_i^{(h+1)}$
    # Compute the gradient contributions for the other hidden layers (Backpropagation)
    for $l = h, \ldots, 1$ **do**
      $\nabla \boldsymbol{z}_i^{(l)} = \nabla \boldsymbol{z}_i^{(l+1)}.W^{(l+1)}.\left[\sigma'\left(\boldsymbol{z}^{(l)}\right)\right]$
      $\nabla W_i^{(l)} = \nabla \boldsymbol{z}_i^{(l)} \bigotimes [\![\boldsymbol{a}_i^{(l-1)}]\!]$
      $\nabla \boldsymbol{b}_i^{(l)} = \nabla \boldsymbol{z}_i^{(l)}$
    end for
  end for
  # Compute the Average Gradient across all the Datapoints
  for $l = 1, \ldots, h+1$ **do**
    $\nabla W^{(l)} = \frac{1}{n}\sum_{i=1}^{n} \nabla W_i^{(l)}$
    $W^{(l)} \leftarrow W^{(l)} - \eta.\nabla W^{(l)}$ ▷ Update the Weights with learning rate η
    $\nabla \boldsymbol{b}^{(l)} = \frac{1}{n}\sum_{i=1}^{n} \nabla \boldsymbol{b}_i^{(l)}$
    $\boldsymbol{b}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \eta.\nabla \boldsymbol{b}^{(l)}$ ▷ Update the biases with learning rate η
  end for
end while

3.3 Backpropagation and Deep Networks

One of the well known problems with training deep networks (networks with a large number of layers) is that the gradients for the weights for the layers far away from the output layer become negligible. Effectively therefore in deep networks the early layers soon stop 'learning' — recall that 'learning' in a neural network happens by 'adjusting' the W and b values using the gradients. Vanishing gradients imply stagnation in the learning process. Some ways to address this are:

- Training the network layer-wise before a full end-to-end training
- Using special architectures like the *highway networks* designed to address the vanishing gradient problem in deep networks

Another issue is that some particular combinations of activation function and cost function can lead to stagnation in the learning process. For instance it can be shown that square loss error function along with the logistic activation function can make learning very slow when the current values are far from the optimum. We would desire exactly the opposite - learning quickly when we are far from the optimum and slow down the learning when we are already close to the optimum. The logistic function with square loss combination however results in exactly the opposite effect. So with the logistic activation function one often uses cross entropy as the cost.

4 CNNs and RNNs

The class of sequential neural networks (called *Multilayier Perceptrons* — MLP) that we discussed above form a generic template for a large class of neural networks used in practice today. In this section we briefly explore a couple of popular variants of the MLP that have proven to be extremely useful. Convolutional Neural Networks (CNN) are a class of neural networks have that have worked phenomenally well for image processing and analysis tasks. Recurrent Neural Networks provide the basic framework for dealing with sequential (time dependent) data — data streams that consist of data points that are not necessarily independent of each other. Time series data, speech processing, natural language processing, video analysis — all of these require analysis of a sequence of data points each having an influence on the stream that is going to follow it in the sequence. The training strategy for both and CNNs and RNNs is in general no different from the backpropagation with gradient descent strategy that we discussed for MLPs.

4.1 Convolutional Neural Networks (CNNs)

A schematic illustration of CNNs is given in Figure 6. The basic construct in the CNN architecture is a *Feature Map*. Intuitively each feature map is expected to recognize occurrences of a certain primitive feature in a given image. For instance one feature map could be recognizing edges, another could be recognizing corners, yet another recognizing green patches in the

image, etc. A feature map consists of a set of neurons each of which takes inputs from a different square patch of the input image. Consider an image of size $(n \times n)$ pixels. Suppose we take a $(k \times k)$ filter and move it around the image we would get $(n-k+1) \times (n-k+1)$ different positions for the filter each corresponding to a different $(k \times k)$ patch of the image. Each of these patches feeds into one neuron of a feature map. Therefore the feature map would have $(n-k+1) \times (n-k+1)$ neurons, each with $k \times k$ inputs and hence $k \times k$ weights. The key characteristic of a CNN is that all the neurons are expected to do the same 'job' (that of recognizing some 'feature') but on a different part of the image. The weights associated with the inputs to one neuron on the feature map are 'trained' to recognize the feature given a $k \times k$ patch. Since all the neurons in the feature map are expected to play exactly the same role, it is clear that the set of weights for all the neurons in the feature map must be identical. Therefore we keep only $k \times k$ weights, with the same set of weights applied to each of the neurons in the feature map. Contrast this with the number of weights we would require if it were a normal dense connection between two layers as in a MLP. We can now extend this to create multiple feature maps from the same image, each feature map trying to recognize a different feature in the image.

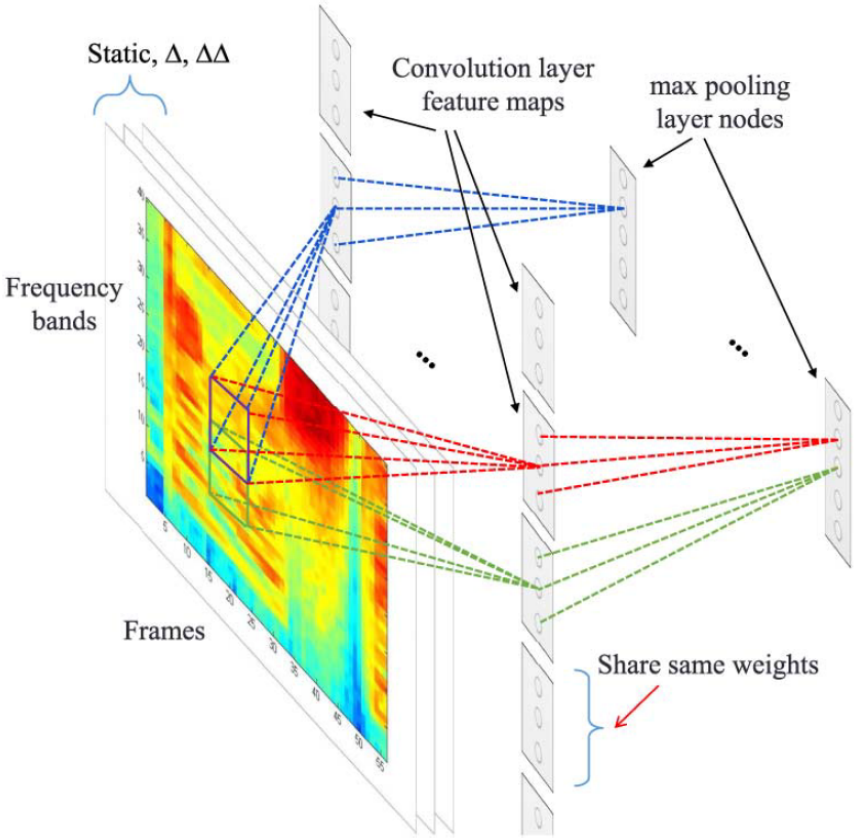


Figure 6: Convolutional Neural Network. **Source**: the CNN tutorial at `http://recognize-speech.com/acoustic-model/knn/comparing-different-architectures/convolutional-neural-networks-cnns`

Feature maps are often used along with a pooling layer which provides a 'voting' mechanism between adjacent neurons in a feature map. This provides a degree of robustness to the feature recognition task carried out by the feature map.

For sophisticated image processing tasks multiple feature map-pooling layer pairs are used in a cascading manner to provide more and more abstract layers of representation for the image.

4.2 Recurrent Neural Networks (RNNs)

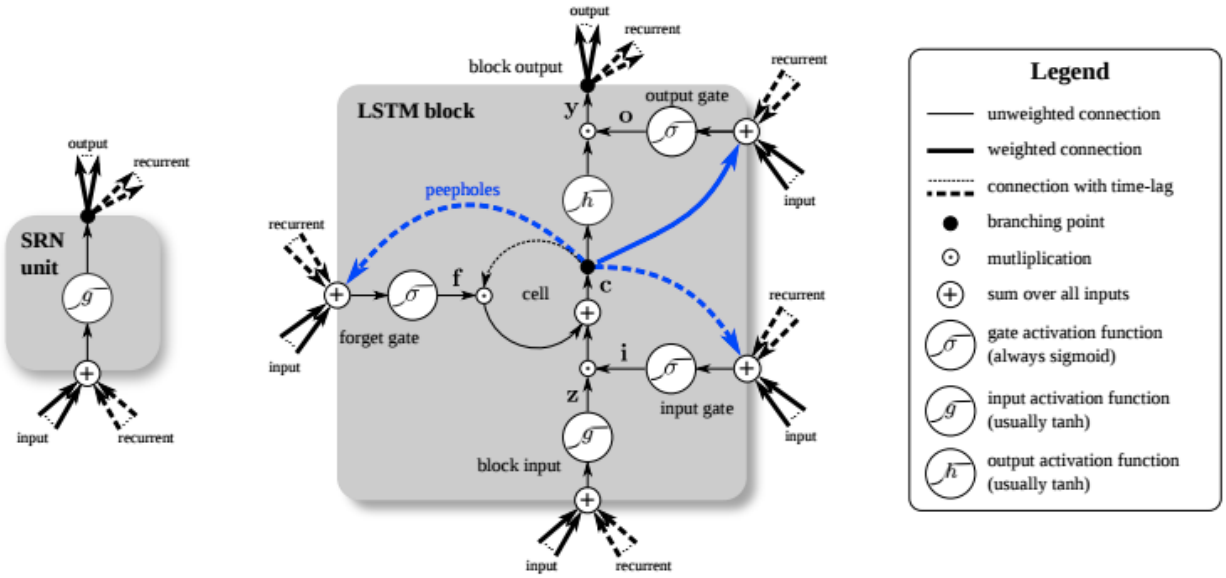


Figure 7: Recurrent Neural Network. **Source**: the DeepLearning4j page `http://deeplearning4j.org/lstm.html`

Recurrent Neural Networks illustrated in Figure 7 are neural network models that build a historical representation of a sequence of data points and uses that to make a prediction on the next data point in the sequence. A recurrent layer therefore has two kinds of inputs — (i) inputs coming from the previous layer, and (ii) its own outputs at previous times in history being treated as inputs (recurrent). The two together determine the output of the layer in the next time step. Formally let the recurrent layer (say the l^{th} layer in the network) be of width n and the sequence length (number of historical snapshots required by the network to make a decision) be T. Assume the previous layer with m neurons is densely connected to this layer. Then assuming the layer has an activation function σ, the output of the layer $\boldsymbol{a}_t^{(l)}$ at time t is computed as:

$$\boldsymbol{a}_t^{(l)} = \sigma\left(W_i^{(l)}.\boldsymbol{a}_t^{(l-1)} + W_r^{(l)}\left(\sum_{i=1}^{T}\boldsymbol{a}_{t-i}^{(l)}\right)\right)$$

where $W_i^{(l)}$ is the weight matrix associated with the inputs from the previous layer and $W_r^{(l)}$ is the weight matrix associated with the recurrent inputs (outputs of this layer during the previous time steps coming back as inputs). Note that we are considering the outputs during the last T time steps.

Training RNNs is similar to MLPs — we treat the historical snapshots as virtual layers that feed into the layer at the current time step and carry out backpropagation through time. Note

that it is as if we have created a new network that has layers corresponding to all the T previous outputs and we carry out our 'usual' backpropagation in this network.

Some of the issues we highlighted with backpropagation crop up in RNNs as well — as T increases, the 'depth' of the network for backpropagation through time increases as well. This will indeed give rise to the vanishing gradient problem.

LSTMs (Long, Short-Term Memory) are the state of art in RNNs today. These are recurrent networks that also enable the network to 'remember', 'forget' or 'isolate' the memory from the rest of network. These come much closer to the way humans deal with time varying data than the plain RNNs. Incidentally LSTMs avoid the vanishing gradient problem in RNNs to a large extent.

5 Practical Strategies for Neural Network Training

Backpropagation as described above is the basis for a large part of the neural network training strategies in use today. As we have seen above it involves computing the incremental contribution to the gradient due to each of the training data points and averaging over them to get the gradients $\nabla W, \nabla b$. These gradients provide one update to the W, b parameters of the network. This makes the training extremely slow — one pass through the entire dataset update. Among the more common variants of this to make the training fast is the *Stochastic Gradient Descent* algorithm. We discuss this in more detail in the following section.

5.1 Stochastic Gradient Descent

Stochastic Gradient Descent is gradient descent where the gradient is calculated as the average gradient over a small subset (minibatch) of the training data. The minibatch size is usually of the order of a few tens. To ensure we are not ignoring the other datapoints, we shuffle the entire dataset and then sequentially pick chunks of training data, each of size equal to the chosen minibatch size. One such 'pass' through the entire training data is referred to as an *epoch*. The training and validation error of the network is monitored after every epoch. The epochs are continued till one of the following happens:

- upper limit on the (computationally) affordable number of epochs is reached
- no observable improvement in training/test accuracy
- validation error starts increasing though training accuracy is improving - sign of overfitting

The pseudocode for stochastic gradient descent is shown in Algorithm 3. Apart from being much faster than the original plain (batch) version of the gradient descent (one pass through the entire traning data for each parameter update) - it also produces much better models eventually. Stochastic gradient is able to implicitly carry out a lot more 'exploration' in the parameter space because of the stochasticity. It is therefore lot more robust against local minima — recall

Algorithm 3 Stochastic Gradient Descent

Initialize starting values for all the network parameters — $W, b = W_0, b_0$.
Select a minibatch size B and number of epochs T
for $epoch = 1, \ldots, T$ **do**
    Shuffle the dataset — randomly reorder the n data points in the training data
    for $batch = 1, \ldots, n/B$ **do**
        Form the minibatch consisting of datapoints $(batch - 1) * B + 1, \ldots, batch * B$
        Run Algorithm 2 on the minibatch
                ▷ The initialization of W, b is done only once. After each $epoch/batch$
                ▷ the final values for W, b become starting values for the next iteration
    end for
    (Optional) Check the training and validation error of the current model
end for

that a common issue with gradient descent methods is getting stuck in local minima that are far from the global minimum.

There are several other variants of gradient descent that have been explored for neural network training, the details of which are beyond the scope of this module.

5.2 Regularization in Neural Networks

Neural Networks are inherently complex. Regularization is therefore often a necessity in neural networks, to avoid overfitting. We briefly describe some of the common regularization strategies used in neural network training.

1. **Square Norm**: Just like we saw for regression, a common regularization strategy in neural networks is to add a regularization term to the cost function before minimization. Therefore the regularized cost function for neural network training is of the form:

$$\frac{1}{n}\sum_{i=1}^{n} C\left(\boldsymbol{a}_i^{(h+1)}, \boldsymbol{y}_i\right) + \lambda.\left(||W||^2 + ||b||^2\right)$$

   where $||W||^2$ ($||b||^2$) denotes the sum of the squares of all the weights (biases) in the neural network and λ is the regularization constant. Training based on this cost function is again through backpropagation — there will be an additional $2||W||, 2||b||$ term in the gradient computation for the base case that we have examined in detail earlier. A common initialization for the weights and baises is to sample them from a Gaussian with zero mean and unit variance. The idea is to strike a balance between keeping the error low and not letting the parameter complexity become too high — weights and biases becoming too large or too many non-zero weights and biases than necessary.

2. **Drop-Out**: In trying to force the network to work with as small an 'effective' network as possible, we drop some of the neurons out from the network periodically. One way to implement this is to pick a random subset of neurons from a layer and forcibly set their output to zero.

3. **Weight/Bias Clipping**: We could set a window for the weights and biases and clip their values to remain within the window.

5.3 Hyperparameter Tuning

You may have noticed that by now we have assembled quite a formidable list of hyperparameters for neural network training — the network structure itself (number of hidden layers, number of neurons in each layer, ...), learning rate for the gradient descent training (see Algorithm 1), batch size in the case of SGD, number of epochs for training, regularization constant, drop-out rate. There could be many others that are specific to the optimization algorithm being used. It is a real challenge to get just the right combination of these parameters before we get a good model after training, though in principle a neural network should do much better than most other models in general, provided we can find the right combination of hyperparameters. There is no universal recipe for how the hyperparameters need to be chosen. Some coarse thumbrules however do help in practice. Subject to these thumb rules one essentially needs to carry out a model selection among the ones obtained from various hyperparameter combinations using strategies like cross validation. The caveat though is that in practice it may not be possible to generate multiple candidate neural network models for evaluation because neural network training is invariably very costly and can take several hours (if not tens of hours) of training on moderately powerful hardware.

Some thumbrules are given below.

- if the training is becoming too slow, then reduce the number of hidden layers and/or number of neurons in the hidden layers. The other strategy is to adjust the batch size.
- as long as the validation errors are going down, it worth continuing with more epochs
- introduce more regularization when validation errors go up while training errors go down — increase drop out, increase the regularization constant, cut down the size of the network, etc.
- if both validation and training errors start going up after some epochs — try reducing the learning rate.
- if the training stagnates very quickly (first few epochs) just rerun the training with a different initialization — very likely it got stuck in a local optimum.

The architecture of course plays a very important role and much of the work in adapting neural networks to different domains is towards evolving appropriate architectures that can be trained for the given task. However choice of the right architecture for a problem along with the correct set of hyperparameters is still a very sophisticated 'art' and is far from being an objective science driven by a strong theory.

6 Appendix: Legend and Conventions, Notation

There are several parts of the test that have been highlighted (in boxes). The document uses three kinds of boxes.

Example 0: Title

One of the running examples in the document. The reader must be able to reproduce the observations in these examples with the corresponding R code provided in Section **??**.

Title

Good to remember stuff. These boxes typically highlight the key take-aways from this document.

Title

Things that are good to be aware of. This is primarily targeted at the reader who wishes to explore further and is curious to know the underlying (sometimes rather profound) connections with other fields, alternative interpretations, proofs of some statements, etc. However one could safely skip these boxes completely if you are only interested in the primary content/message of this document.

Title

`Code snippets. Implementation Tips. Do's and Dont's.`

$\boldsymbol{x}$	A *vector* $\boldsymbol{x}$
x	A scalar quantity x
term	Refer to the glossary as a quick reference for 'term'. These are typically prerequisite concepts that the reader is expected to be familiar with.
`code`	Code snippets, actual names of library functions, etc.