# Lecture Notes

# Introduction to Neural Networks

In this module, you were introduced to one of the hottest topics in the Analytics Industry – Neural Networks. These notes will cover the important aspects of structuring and building a Neural Network.

## Perceptrons

Inspired by the structure of the human brain, neural networks have established a reputation for successfully learning complex tasks such as object recognition in images, automatic speech recognition (ASR), machine translation, image captioning, video classification etc.

As the name suggests, the design of Artificial Neural Networks (ANNs) is inspired by the animal brain. Although not as powerful as the brain (yet), artificial neural networks are the most powerful learning models in the field of machine learning. In the past few years, deep artificial neural networks have proven to perform surprisingly well on complex tasks such as speech recognition (converting speech to text), machine translation, image and video classification, etc. Such models are also commonly called deep learning models.

Artificial neural networks are said to be inspired by the structure of the brain. The biological neuron works as follows - it receives signals through its dendrites which are either amplified or inhibited as they pass through the axons to the dendrites of other neurons. Similarly, Artificial Neural Networks, are a collection of a large number of simple devices called artificial neurons. The network 'learns' to conduct certain tasks, such as recognising a cat, by training the neurons to 'fire' in a certain way when given a particular input, such as a cat. In other words, the network learns to inhibit or amplify the input signals in order to perform a certain task, such as recognising a cat, speaking a word, identifying a tree etc.

Let's now understand one of the earliest proposed models for learning simple classification tasks, the Perceptron, which later became the fundamental building block of artificial neural networks.

Perceptrons take some signals as inputs and perform a set of simple calculations to arrive at a decision.

The perceptron takes a **weighted sum** of multiple inputs (along with a bias) as the cumulative input and applies a **step function** on the cumulative input, i.e. it returns 1 if the input is positive, else -1. In other words, the perceptron "fires" (returns 1) if the cumulative input is positive and "stays dormant" (returns 0) if the input is negative.

Note that there are different ways to define the step function. One can use 1 and -1 as well instead of 1 and 0:

$$y=1 \text{ if } x>0$$

$$y=0 \text{ if } x<=0$$

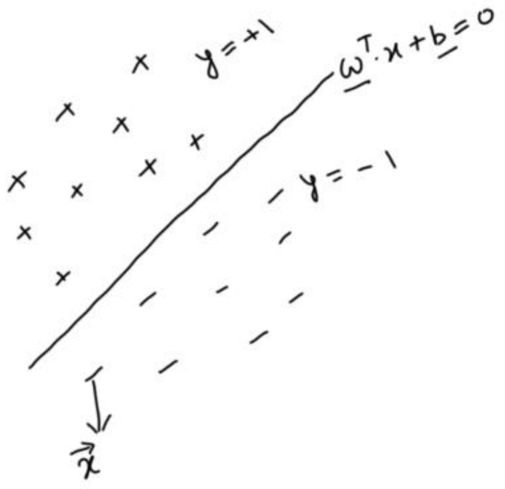
The expression for the cumulative input which is the weighted sum is as follows, which can also be written using vectors.

$$\text{Cumulative Input} = w^T.x + b = w_1x_1 + w_2x_2 + \ldots + w_kx_k + b$$

where,

$$w = \begin{bmatrix} w_1 \\ w_2 \\ . \\ . \\ w_k \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_k \end{bmatrix}$$

A perceptron uses a step function to perform binary classification which can be understood using the following diagram.



Here, the step function is such that

y = 1 if x>0

y = -1 if x<=0

To find a valid separator we need to find a certain set (w, b) such that y($w^T$.x+b) > 0 **for all the data points** and not a valid separator if y($w^T$.x+b) < 0 for **any one** of the data points.

This can be further simplified using homogeneous coordinates. In a d-dimensional space with a bias ‘b’, under homogeneous coordinates,

$$x \text{ earlier represented as this } \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_d \end{bmatrix} \text{ transforms to this } \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_d \\ 1 \end{bmatrix}.$$

$$w \text{ earlier represented as this } \begin{bmatrix} w_1 \\ w_2 \\ . \\ . \\ w_d \end{bmatrix} \text{ transforms to this } \begin{bmatrix} w_1 \\ w_2 \\ . \\ . \\ w_d \\ b \end{bmatrix}$$

Hence, the criteria for a valid separator becomes $y(w^T.x) > 0$.

The training of a perceptron happens using a simple iterative solution suggested by Rosenblatt:

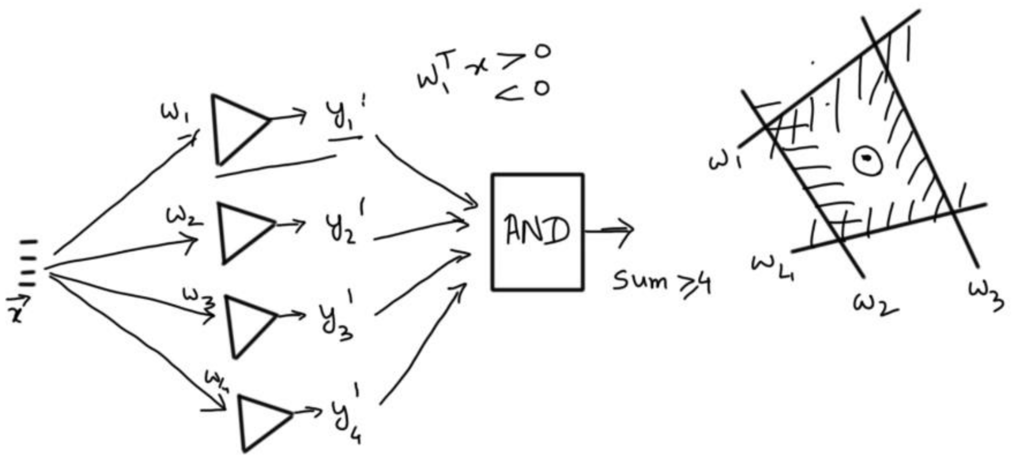
$$\omega_{t+1} \leftarrow \omega_t + \underbrace{y_{i_t} \cdot \vec{x}_{i_t}}_{\text{error}}$$

where $y_{it}.x_{it}$ is the error term. It is important to note here that $x_{it}$ in this iterative procedure is a **misclassified data point** and $y_{it}$ is the corresponding true label. Also, note that the dot in $y_{it}.x_{it}$ is not a dot product.

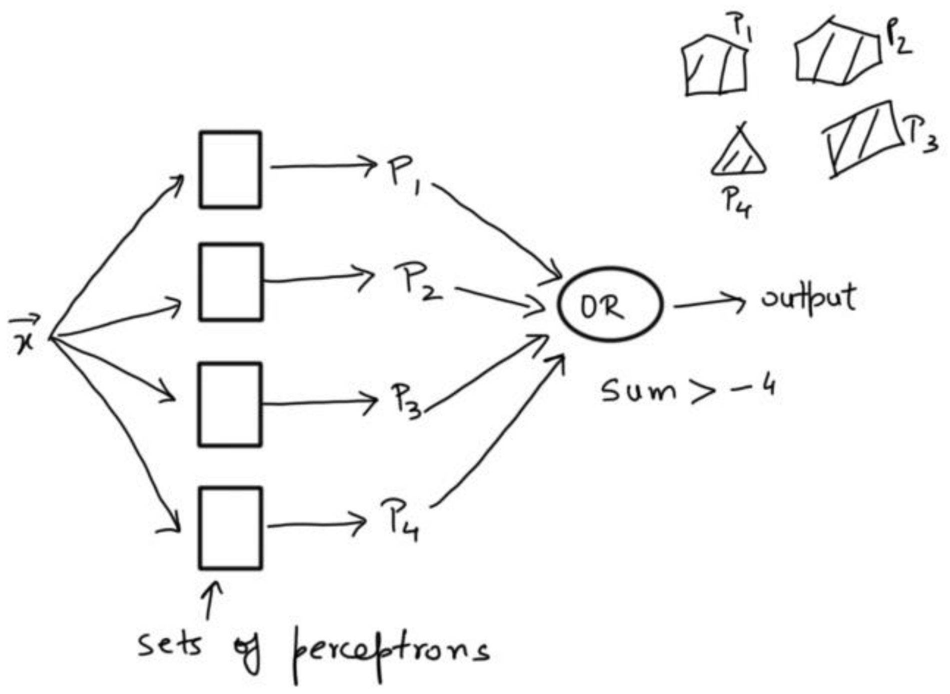
A sample demonstration showing the training is given on the platform as a comprehension.

You also saw how a group of perceptrons can act as a universal function approximators. This also helps in understanding why Neural Network is considered a universal function approximator.

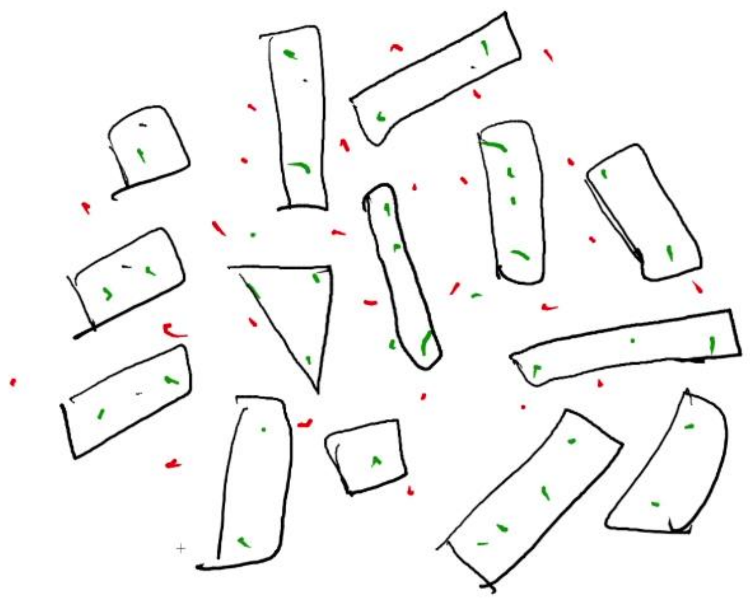
You have seen how a group of perceptrons using an AND gate can enclose a certain class:



A bunch of such perceptron networks can be used to perform multiclass classification:



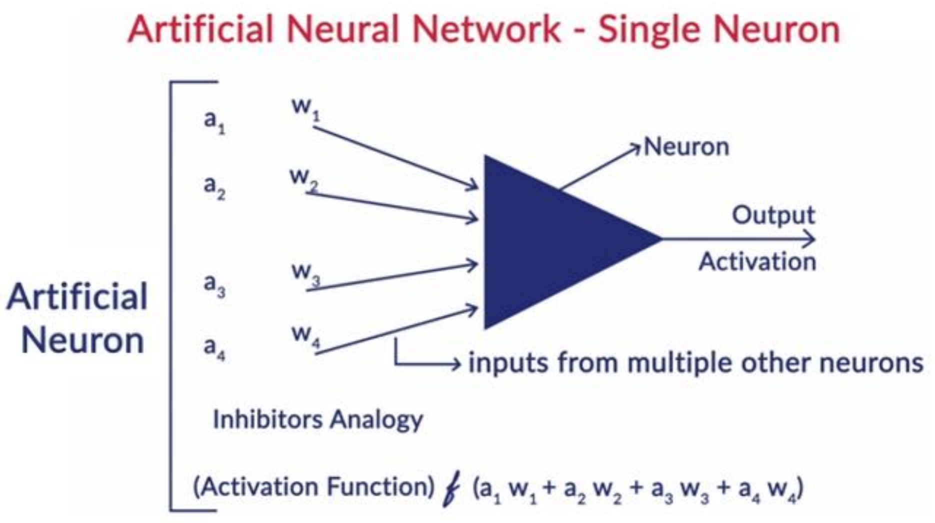
Here $P_1$, $P_2$, $P_3$, $P_4$ refer to the different classes. You have seen how this can be extended to solving a problem as complex as this:



We next move on to understanding how an artificial neuron is defined and the various details entailing an artificial neural network.

## Artificial Neuron

An artificial neuron is very similar to a perceptron, except that the activation function is not a step function.

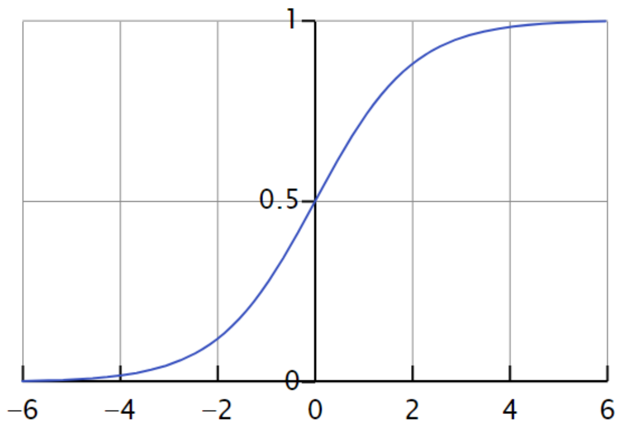

Artificial Neural Network - Single Neuron
Artificial Neuron
$a_1$
$a_2$
$a_3$
$a_4$
$w_1$
$w_2$
$w_3$
$w_4$
Neuron
Output
Activation
inputs from multiple other neurons
Inhibitors Analogy
(Activation Function) $f$ $(a_1 w_1 + a_2 w_2 + a_3 w_3 + a_4 w_4)$

Like a perceptron, the input is the weighted sum of inputs. The output is the activation function applied on the input. The activation function could be any function, though it should have some important properties such as:

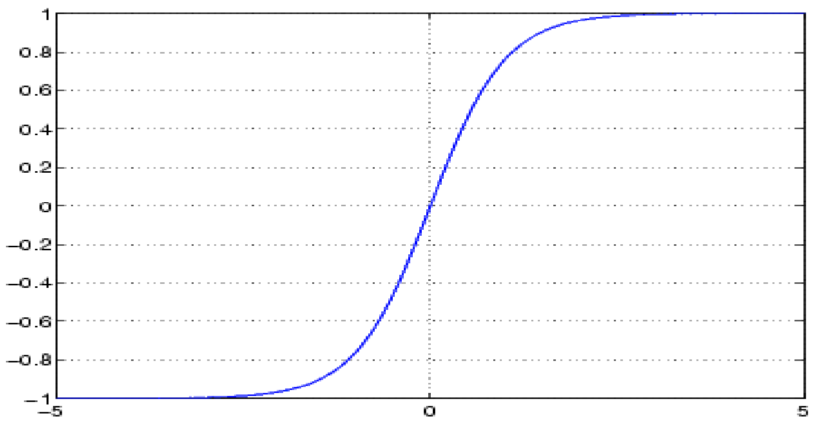
- Activation functions should be **smooth** i.e. they should have no abrupt changes when plotted.
- They should also make the inputs and outputs **non-linear** with respect to each other to some extent. This is because non-linearity helps in making neural networks more compact.

Let's look at the commonly used activation function that can be applied in case of an artificial neuron.

1. Logistic function - $output = f(x) = \frac{1}{1+e^{-x}}$



2. Hyperbolic tangent function - $output = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$



3. Rectilinear Unit - $output = x$ for x > = 0 and 0 otherwise.
4. Leaking Relu - $output = x$ for x > = 0 and $output = \alpha x$ otherwise.

An artificial neural network is a network of such neurons. Neurons in a neural network are arranged in **layers**. The first and the last layer are called the **input and output** layers. Input layers have as many neurons as the number of attributes in the data set and the output layer has as many neurons as the number of classes of the target variable (for a classification problem). For a regression problem, the number of neurons in the output layer would be 1 (a numeric variable).

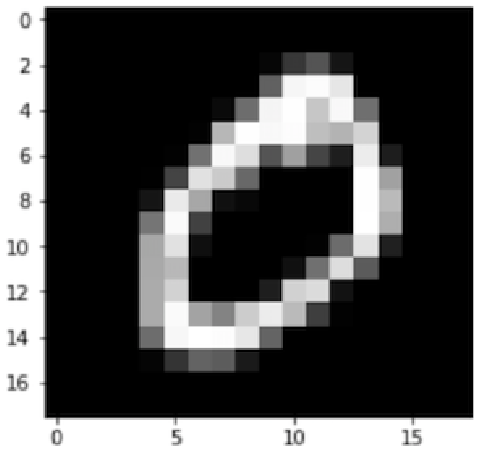
There are six main things that need to be specified for specifying a neural network completely:

1. Network Topology
2. Input Layer
3. Output Layer
4. Weights
5. Activation functions
6. Biases

An important thing to note is that the inputs can only be numeric. For different types of input data, we use different ways to convert the inputs to a numeric form.

In case of **text data**, we either use **a one-hot vector** or **word embeddings** corresponding to a certain word. For example, if the vocabulary size is |V|, then you can represent the word $w_n$ as a one-hot vector of size |V| with a '1' at the nth element while all other elements being zero. The problem with one-hot representation is that usually the vocabulary size |V| is huge, in tens of thousands at least, and hence it is often better to use word embeddings which are a lower dimensional representation of each word.

Feeding **images** (or videos) is straightforward since images are naturally represented as **arrays of numbers**. These numbers are the raw **pixels** of the image. Pixel is short for picture element. In images, pixels are arranged in rows and columns (an array of pixel elements). The figure below shows an image of a handwritten 'zero' in the MNIST dataset (black and white) and its corresponding representation in Numpy as an array of numbers. The pixel values are high where the **intensity** is high, i.e. the color is white-ish, while they are low in the black regions.



```
[[   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   6  55  84  22   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0  96 244 250 228   0   0   0   0   0]
 [   0   0   0   0   0   0   0   9 108 243 252 196 247 110   0   0   0   0]
 [   0   0   0   0   0   0   2 181 252 247 251 189 178 210   0   0   0   0]
 [   0   0   0   0   0   2 112 247 220  84 159  69  30 234  29   0   0   0]
 [   0   0   0   0   1  68 223 201 103   0   0   0   0 252 160   0   0   0]
 [   0   0   0   0  21 232 166  17   7   0   0   0   0 252 184   0   0   0]
 [   0   0   0   0 116 248  65   0   0   0   0   0   0 253 172   0   0   0]
 [   0   0   0   0 167 223  15   0   0   0   0   2 107 225  33   0   0   0]
 [   0   0   0   0 168 182   0   0   0   0  16 111 219  90   0   0   0   0]
 [   0   0   0   0 169 208   0   0   0  30 207 217  18   0   0   0   0   0]
 [   0   0   0   0 169 248 162 130 202 234 184  62   2   0   0   0   0   0]
 [   0   0   0   0 108 245 253 251 229  99   0   0   0   0   0   0   0   0]
 [   0   0   0   0   5  52  98  91  26   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]]
```

In a neural network, each **pixel** of the input image is a **feature.** For example, the image above is an 18 x 18 array. Hence, it will be fed as a **vector of size 324** to the network.

Note that the image above is a **black and white** image (also called **greyscale image**), and thus, each pixel has only one 'channel'. If it were a **color image** (called an RGB image - Red, Green, Blue), each pixel would have **three channels** - one each for red, blue and green as shown below. Hence, the number of neurons in the input layer would be 18 x 18 x 3 = 972. You'll learn about this in detail in the next module on Convolution Neural Networks.

The output layer used in case of multiclass classification problem is the softmax layer. A softmax output is a **multiclass logistic function** commonly used to compute the 'probability' of an input belonging to one of the multiple classes. It is defined as follows:

$$p_i = \frac{e^{w_i \cdot x'}}{\sum_{t=0}^{c-1} e^{w_t \cdot x'}}$$

where c is the number of neurons in the output layer.

In the neural network, the input x' is a large vector and the $w_i$s are rows of a weight matrix which is present between the layers. The softmax function translates to a **sigmoid function** in the special case of **binary classification**.

Let's look at the proof. Assume that the softmax function has two neurons with the following outputs:

$$p_0 = \frac{e^{w_0 \cdot x'}}{e^{w_0 \cdot x'} + e^{w_1 \cdot x'}} \text{ and } p_1 = \frac{e^{w_1 \cdot x'}}{e^{w_0 \cdot x'} + e^{w_1 \cdot x'}}$$

Dividing by the numerator, we can rewrite **p1** as:

$$p_1 = \frac{1}{1+\frac{e^{w_0 . x'}}{e^{w_1 . x'}}} = \frac{1}{1+e^{(w_0 - w1).x'}}$$

And if we replace $w_1 - w_0$ = some w, we get the sigmoid function.

Since large neural networks can potentially have extremely complex structures, certain assumptions are made to simplify the way information flows in them:

1. Neurons are **arranged in layers** and the layers are arranged **sequentially.**
2. Neurons **within the same layer do not interact** with each other.
3. All the inputs enter the network through the **input layer** and all the outputs go out of the network through the **output layer**.
4. Neurons in consecutive layers are **densely connected,** i.e. all neurons in layer l are connected to all neurons in layer l+1.
5. **Every interconnection** in the neural network has a **weight** associated with it, and **every neuron has a bias** associated with it.
6. All neurons in a particular layer use the **same activation function**.

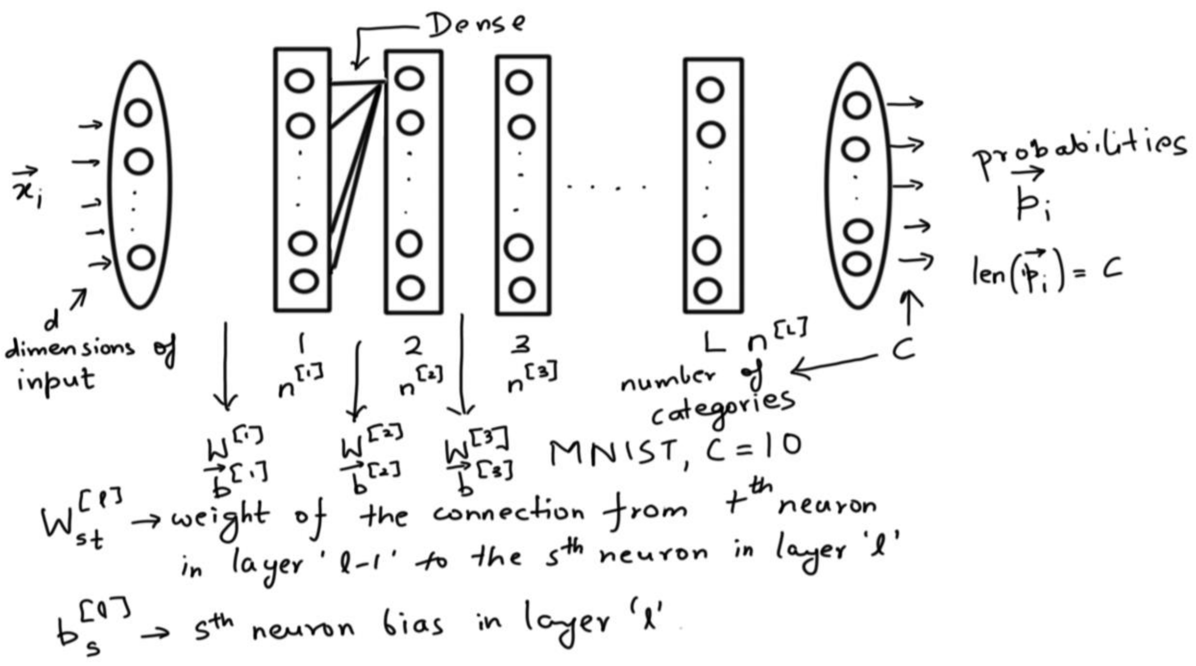
Neural networks require rigorous training.

Recall that models such as linear regression, logistic regression, SVMs etc. are trained on their coefficients, i.e. the training task is to find the optimal values of the coefficients to minimize some cost function.

Neural networks are no different - they are **trained on weights and biases**.

During training, the neural network learning algorithm fits various models to the training data and selects the best model for prediction. The learning algorithm is trained with a **fixed set of hyperparameters** - the network structure (number of layers, number of neurons in the input, hidden and output layers etc.). It is trained on the **weights and the biases,** which are the **parameters of the network.**

Let us revise the notations that have been used in the lectures. The following image depicts the general structure and notations used for a neural network.


Dense
$\vec{x}_i$
d dimensions of input
1 $n^{[1]}$
2 $n^{[2]}$
3 $n^{[3]}$
L $n^{[L]}$
number of categories
C
probabilities $\vec{p}_i$
$\text{len}(\vec{p}_i) = C$
$W^{[1]}$ $\vec{b}^{[1]}$
$W^{[2]}$ $\vec{b}^{[2]}$
$W^{[3]}$ $\vec{b}^{[3]}$
MNIST, $C = 10$

$W^{[l]}_{st} \rightarrow$ weight of the connection from $t^{th}$ neuron in layer '$l-1$' to the $s^{th}$ neuron in layer '$l$'

$b^{[l]}_{s} \rightarrow$ $s^{th}$ neuron bias in layer '$l$'.

The notations that we shall be going forward are as follows:

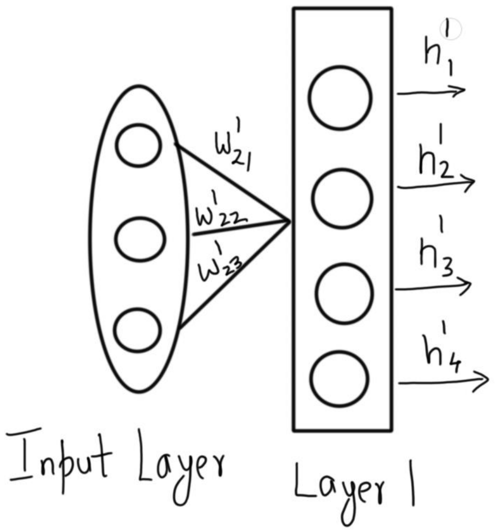
1. W is for weight matrix
2. b shall stand for the bias
3. x stands for input
4. y is the ground truth label
5. p is the probability vector of the predicted output
6. h is the output of the hidden layers
7. superscript stands for layer number
8. subscript stands for the index of the individual neuron

For an MNIST dataset, the input is a 28x28 = 784 sized vectors for each image. The output layer has 10 neurons, one for each of the digits. Note that a neural network with 'L' hidden layers is referred to as a 'L+1' layered neural network.

## Feedforward in Neural Networks

Having learnt about the structure, topology, hyperparameters and the simplifying assumptions of neural networks, we shall now see how information flows from one layer to the adjacent one in a neural network.

In artificial neural networks, the output from one layer is used as input to the next layer. Such networks are called **feedforward neural networks**. This means there are no loops in the network - information is always fed forward, never fed back. Let's understand this by using the input layer and the first hidden layer.



We have the input, weight matrix, bias and the output of the first layer as follows:

$$x_i = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \\ w^1_{41} & w^1_{42} & w^1_{43} \end{bmatrix}, b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \\ b^1_4 \end{bmatrix}, h^1 = \begin{bmatrix} h^1_1 \\ h^1_2 \\ h^1_3 \\ h^1_4 \end{bmatrix}$$

The equation showing the relation between the above is as follows:

$$h^1 = \begin{bmatrix} h^1_1 \\ h^1_2 \\ h^1_3 \\ h^1_4 \end{bmatrix} = \sigma(W^1 . x_i + b) = \begin{bmatrix} \sigma(w^1_{11}x_1 + w^1_{12}x_2 + w^1_{13}x_3 + b^1_1) \\ \sigma(w^1_{21}x_1 + w^1_{22}x_2 + w^1_{23}x_3 + b^1_2) \\ \sigma(w^1_{31}x_1 + w^1_{32}x_2 + w^1_{33}x_3 + b^1_3) \\ \sigma(w^1_{41}x_1 + w^1_{42}x_2 + w^1_{43}x_3 + b^1_4) \end{bmatrix}$$

To summarize, the procedure to compute the output of the ith neuron in the layer l is:

- Multiply the ith row of the weight matrix with the output of layer l-1 to get the weighted sum of inputs
- Convert the weighted sum to cumulative sum by adding the ith bias term of the bias vector
- Apply the activation function, σ(x) to the cumulative input to get the output of the ith neuron in the layer l

Hence, the feed forward algorithm for a neural network with L hidden layer and a softmax output becomes:

```
1. $h^0 = x_i$
2. for $l$ in [1,2,.......,L]:
    1. $h^l = \sigma(W^l . h^{l-1} + b^l)$
3. $p_i$ = $e^{W^o . h^L}$
4. $p_i$ = normalize($p_i$)
```

Here, the final $p_i$ we get is as follows:

$$p_i = \begin{bmatrix} p_{i1} \\ p_{i1} \\ . \\ . \\ p_{ic} \end{bmatrix} \text{ where } p_{ij} = \frac{e^{w_j . h^L}}{\sum_{t=1}^{c} e^{w_t . h^L}} \text{ for } j = [1, 2, \ldots, c] \text{ \& } c = \text{number of classes}$$

The above algorithm performs feed forward for a single data point through the neural network. In real life, the data is processed in batches. The above algorithm translates to a batch as follows:

```
1. $H^0 = B$
2. for $l$ in $[1, 2, \ldots\ldots, L]$ :
    1. $H^l = \sigma(W^l . H^{l-1} + b^l)$
3. $P$ = normalize(exp($W^o . H^L + b^o$))
```

where B is the input which is a batch of m data points stacked side by side as shown below:

$$B = \begin{bmatrix} | & | & | & | & | \\ x_i & x_{i+1} & . & . & x_{i+m-1} \\ | & | & | & | & | \end{bmatrix}$$

Similarly,

$$H^l = \begin{bmatrix} | & | & | & | & | \\ h_i^l & h_{i+1}^l & . & . & h_{i+m-1}^l \\ | & | & | & | & | \end{bmatrix} \text{ and } P = \begin{bmatrix} | & | & | & | & | \\ p_i & p_{i+1} & . & . & p_{i+m-1} \\ | & | & | & | & | \end{bmatrix}$$

Notice that this is very similar to the algorithm for a single data point with some notational changes. Specifically, we have used the uppercase notations $H^l$ and P instead of $h^l$ and p respectively to denote an entire batch of data points. In other words, $H^l$ and P are matrices whose *ith* column represents the $h^l$ and p vectors respectively of the *ith* data point. The number of columns in these 'batch matrices' is equal to the number of data points in the batch: m.

You have also seen how parallelisation happens in the above vectorised implementation. This can be understood as follows:

$$W^l.H^{l-1} + b^l = [\,W^l.h_i^{l-1} \quad W^l.h_{i+1}^{l-1} \quad . \quad . \quad W^l.h_{i+m-1}^{l-1}\,] + [\,b^l \quad b^l \quad . \quad . \quad b^l\,]$$

$$W^l.H^{l-1} + b^l = [\,W^l.h_i^{l-1} + b^l \quad W^l.h_{i+1}^{l-1} + b^l \quad . \quad . \quad W^l.h_{i+m-1}^{l-1} + b^l\,]$$

The addition of $b^l$ is referred to as broadcasting. $h_i^{l-1}$ refers to the output of the $i^{th}$ datapoint from the l-1 layer.

The next section will cover the next steps in training a neural network which is the calculation of the loss function and subsequently backpropagation.

## Backpropagation in Neural Networks

Until now, you learnt how the information passes through neural networks.

Let's recap how neural networks are **trained.** Recall that the training task is to compute the optimal weights and biases by **minimizing some cost function**.

The task of training neural networks is exactly the same as that of other ML models such as linear regression, SVMs etc. The desired output (output from the last layer) minus the actual output is the **cost** (or the **loss**), and we to tune the parameters w and b such that **the total cost is minimized.**

An important point to note is that if the data is large (which is often the case), loss calculation itself can get pretty messy. For example, if you have a million data points, they will be fed into the network (in batch), the output will be calculated using feedforward and the loss/cost $L_i$ (for ith data point) will be calculated. The total loss is the sum of losses of all the individual data points. Hence,

$$\text{Total Loss} = L = L_1 + L_2 + L_3 + \ldots\ldots\ldots + L_{1000000}$$

There is one important thing you should note here. We minimize the average of the total loss and the not the total loss. Minimizing the average loss implies that the total loss is getting minimized.

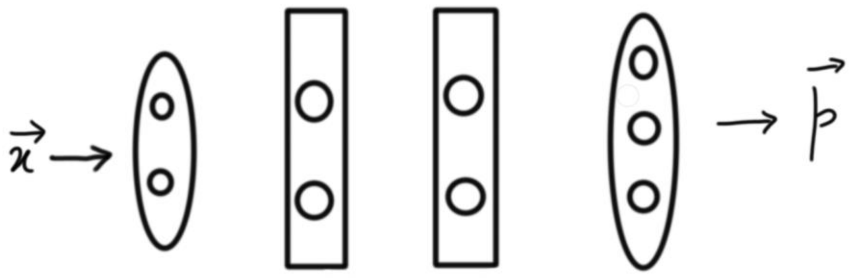
The loss function is defined as follows:

$$G(w,b) = \frac{1}{n}\sum_{i=1}^{n} L(F(x_i), y_i)$$

The loss function is defined in terms of the network output $F(x_i)$ and the ground truth $y_i$. Since $F(x_i)$ depends on the weights and biases, the loss, in turn, is a function of (w, b) . The average loss across all data points is denoted by G(w, b) which we want to minimize.

For a large neural network, the number of weight elements and biases becomes so large and minimizing the loss with so many parameters is a difficult task. This complex task is achieved using **gradient descent.** An interesting property of gradient descent which is widely used is the chain rule.

This has been understood in the lectures using the following 3-layer neural network having 2 hidden layers.



The last layer which is the output layer has 3 neurons for the purpose of understanding the backpropagation of a softmax layer. For a classification task, the output is hence a softmax output.

The feedforward can be summarized as follows:

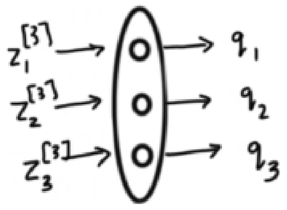
1. $h^1 = \sigma(W^1.x + b^1)$
2. $h^2 = \sigma(W^2.h^1 + b^2)$
3. $p_i$ = normalized$(e^{W^o.h^L})$

The loss used for a multiclass classification is the Cross-Entropy loss which can written as follows:

$$-y^T.log(p) = -\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix}.\begin{bmatrix} log(p_1) \\ log(p_2) \\ log(p_3) \end{bmatrix} = -(y_1log(p_1) + y_2log(p_2) + y_3log(p_3))$$

where y is the true label and p is the predicted label as a one hot matrix.

To perform backpropagation of the last layer which is the softmax layer, let's isolate the last layer.



We have the feed forward through the above layer as follows:

$$q_1 = \frac{e^{z_1^3}}{e^{z_1^3}+e^{z_2^3}+e^{z_3^3}} \qquad q_2 = \frac{e^{z_2^3}}{e^{z_1^3}+e^{z_2^3}+e^{z_3^3}} \qquad q_3 = \frac{e^{z_3^3}}{e^{z_1^3}+e^{z_2^3}+e^{z_3^3}}$$

Please note that p and q have been used interchangeably to denote the network output.

We use the chain rule to calculate

$$\frac{\partial L}{\partial z_1^3} = -\left(\frac{\partial(y_1log(q_1))}{\partial q_1}\frac{\partial q_1}{\partial z_1^3} + \frac{\partial(y_2log(q_2))}{\partial q_2}\frac{\partial q_2}{\partial z_1^3} + \frac{\partial(y_3log(q_3))}{\partial q_3}\frac{\partial q_3}{\partial z_1^3}\right) = -\left(\frac{y_1}{q_1}\frac{\partial q_1}{\partial z_1^3} + \frac{y_2}{q_2}\frac{\partial q_2}{\partial z_1^3} + \frac{y_3}{q_3}\frac{\partial q_3}{\partial z_1^3}\right)$$

We also have the expressions for

$$\frac{\partial q_1}{\partial z_1^3} = q_1(1-q_1) \qquad \frac{\partial q_2}{\partial z_1^3} = -q_1q_2 \qquad \frac{\partial q_3}{\partial z_1^3} = -q_1q_3$$

Substituting the above expressions, we get

$$\frac{\partial L}{\partial z_1^3} = q_1 - y_1, \ \frac{\partial L}{\partial z_2^3} = q_2 - y_2 \text{ and } \frac{\partial L}{\partial z_3^3} = q_3 - y_3.$$

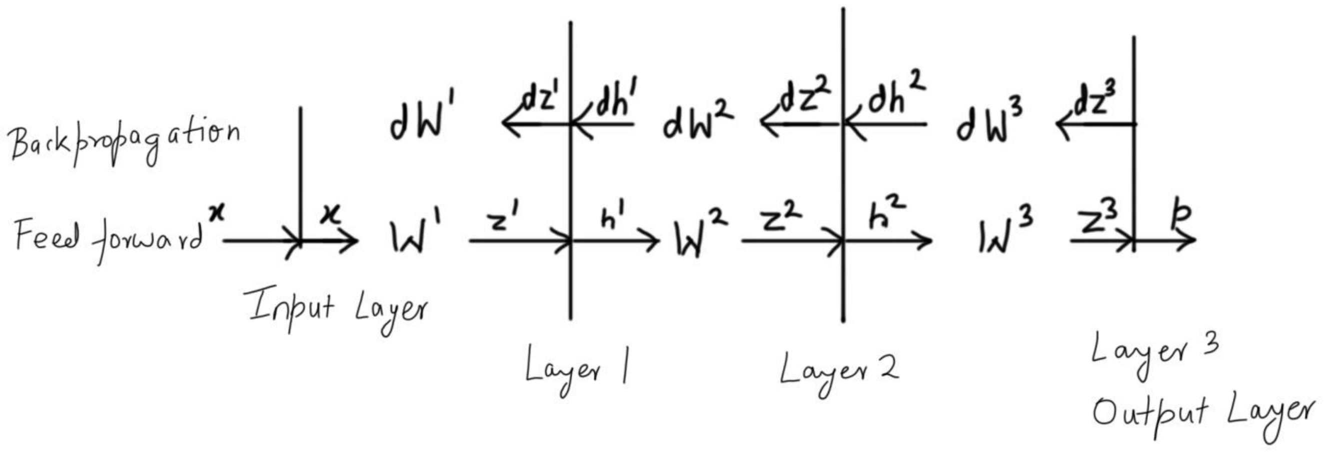
This can be written in a consolidated vector form as:

$$dz^3 = \frac{\partial L}{\partial z^3} = q - y$$

Having calculated dz$^3$, using the backpropagation logic, we can calculate

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial z^3} \frac{\partial z^3}{\partial W^3}$$

The backpropagation logic will be clearer after looking at the following image:

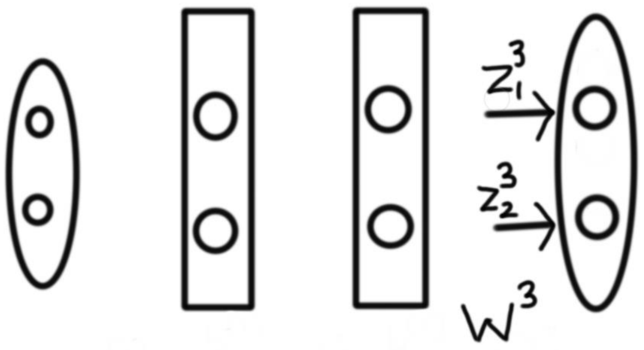


You can see that the gradients are calculated in a backward direction starting from dz$^3$. Hence, we'll calculate the gradients in the following sequence:

1. $dz^3$
2. $dW^3$
3. $dh^2$
4. $dz^2$
5. $dW^2$
6. $dh^1$
7. $dz^1$
8. $dW^1$

Therefore, the process is known as **backpropagation -** we propagate the gradients in a backward direction starting from the output layer.

Having understood the backpropagation through the softmax layer, we have simplified the network to reduce the number of equations. Hence, the new network now has 2 neurons in the output layer as opposed to 3 as shown for the softmax backpropagation derivation.



The derivative of the weight matrix $W^3$ can be expressed as follows:

$$\frac{\partial L}{\partial W^3} = \begin{bmatrix} \frac{\partial L}{\partial w_{11}^3} & \frac{\partial L}{\partial w_{12}^3} \\ \frac{\partial L}{\partial w_{21}^3} & \frac{\partial L}{\partial w_{22}^3} \end{bmatrix}$$

We have used the chain rule of differentiation as stated earlier to calculate the derivatives,

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial z^3}\frac{\partial z^3}{\partial W^3}$$

In other words, we are using $z^3$ as the **intermediary variable**.

The two components of $z^3$ are as follows:

$$z_1^3 = w_{11}^3 h_1^2 + w_{12}^3 h_2^2 + b_1^3 \qquad (1),$$
$$z_2^3 = w_{21}^3 h_1^2 + w_{22}^3 h_2^2 + b_2^3 \qquad (2),$$

We also can write the derivative of loss w.r.t $w_{11}^3$ in terms of both the components of $z^3$ as

$$\frac{\partial L}{\partial w_{11}^3} = \frac{\partial L}{\partial z_1^3} . \frac{\partial z_1^3}{\partial w_{11}^3} + \frac{\partial L}{\partial z_2^3} . \frac{\partial z_2^3}{\partial w_{11}^3}$$

From equations (1) and (2), we can infer that

$$\frac{\partial z_1^3}{\partial w_{11}^3} = h_1^2 \text{ and } \frac{\partial z_2^3}{\partial w_{11}^3} = 0$$

Substituting these in the above expression, we have

$$\frac{\partial L}{\partial w^3_{11}} = \frac{\partial L}{\partial z^3_1}.h^2_1 = dz^3_1.\,h^2_1$$

Similarly, we will get the derivatives for the other weight elements of the weight matrix W$^3$:

$$\frac{\partial L}{\partial w^3_{12}} = dz^3_1.\,h^2_2 \qquad \frac{\partial L}{\partial w^3_{21}} = dz^3_2.\,h^2_1 \qquad \frac{\partial L}{\partial w^3_{22}} = dz^3_2.\,h^2_2$$

These equations can be combined and written in a matrix form as:

$$dW^3 = \frac{\partial L}{\partial W^3} = \begin{bmatrix} dz^3_1 \\ dz^3_2 \end{bmatrix} \begin{bmatrix} h^2_1 & h^2_2 \end{bmatrix} = dz^3.\,(h^2)^T$$

Having calculated dW$^3$, we can use it to calculate the weight updates using:

$$W^3 = W^3 - \alpha.\,\frac{\partial L}{\partial W^3}$$

This essentially means performing the following update steps:

$$w^3_{11} = w^3_{11} - \alpha.\,\frac{\partial L}{\partial w^3_{11}}$$
$$w^3_{12} = w^3_{12} - \alpha.\,\frac{\partial L}{\partial w^3_{12}}$$
$$w^3_{21} = w^3_{21} - \alpha.\,\frac{\partial L}{\partial w^3_{21}}$$
$$w^3_{22} = w^3_{22} - \alpha.\,\frac{\partial L}{\partial w^3_{22}}$$

The next step in the backpropagation scheme of things is to calculate dh$^2$. We have used dz$^3$ as the intermediate variable to get the following derivative equations:

$$\frac{\partial L}{\partial h^2_1} = \frac{\partial L}{\partial z^3_1}.\frac{\partial z^3_1}{\partial h^2_1} + \frac{\partial L}{\partial z^3_2}.\frac{\partial z^3_2}{\partial h^2_1} \qquad (3)$$

$$\frac{\partial L}{\partial h^2_2} = \frac{\partial L}{\partial z^3_1}.\frac{\partial z^3_1}{\partial h^2_2} + \frac{\partial L}{\partial z^3_2}.\frac{\partial z^3_2}{\partial h^2_2} \qquad (4)$$

Using equations (1) and (2) above, we have,

$$\frac{\partial z^3_1}{\partial h^2_1} = w^3_{11} \text{ and } \frac{\partial z^3_2}{\partial h^2_1} = w^3_{21}$$

Also we get,

$$\frac{\partial z^3_1}{\partial h^2_2} = w^3_{12} \text{ and } \frac{\partial z^3_2}{\partial h^2_2} = w^3_{22}$$

Substituting in (3) and (4), we get

$$\frac{\partial L}{\partial h_1^2} = dz_1^3 . w_{11}^3 + dz_2^3 . w_{21}^3$$
$$\frac{\partial L}{\partial h_2^2} = dz_1^3 . w_{21}^3 + dz_2^3 . w_{22}^3$$

This can be written in a sweet little matrix form as:

$$\frac{\partial L}{\partial h^2} = dh^2 = (W^3)^T . dz^3$$

where,

$$W^3 = \begin{bmatrix} w_{11}^3 & w_{12}^3 \\ w_{21}^3 & w_{22}^3 \end{bmatrix} \text{ and } dz^3 = \begin{bmatrix} dz_1^3 \\ dz_2^3 \end{bmatrix}$$

After calculating dh$^2$, the next derivative to be calculated is dz$^2$. You already know that,

$$h^2 = \sigma(z^2), \text{ i.e., } \begin{bmatrix} h_1^2 \\ h_2^2 \end{bmatrix} = \begin{bmatrix} \sigma(z_1^2) \\ \sigma(z_2^2) \end{bmatrix}$$

This can be further simplified as

$$\frac{\partial L}{\partial z_1^2} = dz_1^2 = \frac{\partial L}{\partial h_1^2} . \frac{\partial h_1^2}{\partial z_1^2} + \frac{\partial L}{\partial h_2^2} . \frac{\partial h_2^2}{\partial z_1^2} = dh_1^2 . \sigma'(z_1^2)$$
$$\text{since } \frac{\partial h_1^2}{\partial z_1^2} = \sigma'(z_1^2) \text{ and } \frac{\partial h_2^2}{\partial z_1^2} = 0$$

$$\text{Similarly, } dz_2^2 = dh_2^2 . \sigma'(z_2^2).$$

We can combine both dz$_1^2$ and dz$_2^2$ in a nice vector form as follows:

$$dz^2 = dh^2 \otimes \sigma'(z^2)$$

where the $\otimes$ represents element-wise multiplication (also called the Hadamard product).

Extending the above calculations, we can write the derivatives for the network as follows:

1. $dz^3 = q - y = p - y$
2. $dW^3 = dz^3 . (h^2)^T$
3. $dh^2 = (W^3)^T . dz^3$
4. $dz^2 = dh^2 \otimes . \sigma'(z^2)$
5. $dW^2 = dz^2 . (h^1)^T$
6. $dh^1 = (W^2)^T . dz^2$
7. $dz^1 = dh^1 \otimes . \sigma'(z^1)$
8. $dW^1 = dz^1 . (x)^T$

Let's consolidate the above feedforward and backprop:

```
1. $h^0 = x$
2. for $l$ in $[1, 2, \ldots\ldots, L]$:
    1. $h^l = \sigma(W^l . h^{l-1} + b^l)$
3. $p = normalize(exp(W^o . h^L + b^o))$
4. $L = -y^T . log(p)$
5. $dz^o = p - y$
6. $dW^o = dz^o . (h^2)^T$
7. for $l$ in $[L, L-1, \ldots\ldots .1]$ :
    1. $dh^l = (W^{l+1})^T . dz^{l+1}$
    2. $dz^l = dh^l \otimes . \sigma'(z^l)$
    3. $dW^l = dz^l . (h^{l-1})^T$
```

So, this is the consolidated algorithm for a single data point.

You then saw how this algorithm is extended to a batch. Before that, there is a need to have a brief understanding of how minibatch gradient descent works.

The pseudo code for minibatch gradient descent is as follows:

```
Batch size = m
for epoch = [1, ...  L] {
    Reshuffle the data set
    number of batch = $\frac{n}{m}$
    for batch = [1, ..., number of batches] {
        Compute gradients for each input in the batch
        [$batch_{i-1} \times m$, $batch_i \times m$]
    Average gradient $\nabla w = \frac{\text{Sum of gradient } \nabla w}{m}$
    Average gradient $\nabla b = \frac{\text{Sum of gradient } \nabla b}{m}$
    $w = w - \lambda \nabla w \quad b = b - \lambda \nabla b$
    }
}
```

For updating weights and biases using plain backpropagation, you have to scan through the entire data set to make a single update to the weights. This is computationally very expensive for large datasets. Thus, you use multiple batches (or **mini-batches**) of data points, compute the **average gradient** for a batch, and update the weights based on that gradient.

But there is a danger in doing this - you are making weight updates based only on gradients computed for small batches, not the entire training set. Thus, you make **multiple passes** through the entire training set using **epochs.** An **epoch is one pass** through the entire training set, and you use multiple epochs (typically 10, 20, 50, 100 etc.) while training. In each epoch, you **reshuffle** all the data points, divide the reshuffled set into m batches, and update weights based on gradient of each batch.

This training technique is called **stochastic gradient descent,** commonly abbreviated as **SGD**.

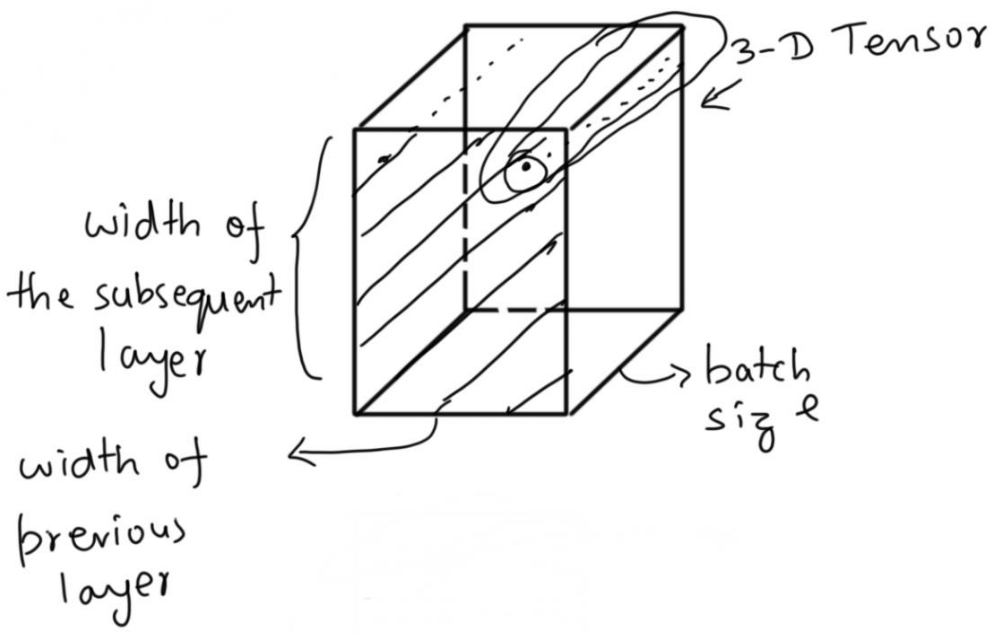
In most libraries such as Tensorflow, the **SGD training procedure** is as follows:

- You specify the number of epochs (typical values are 10, 20, 50, 100 etc.) - more epochs require more computational power
- You specify the number of batches m (typical values are 32, 64, 128, etc.)
- At the start of each epoch, the data set is **reshuffled** and divided into m batches.
- The **average gradient of each batch** is then used to make a weight update.
- The training is complete at the end of all the epochs

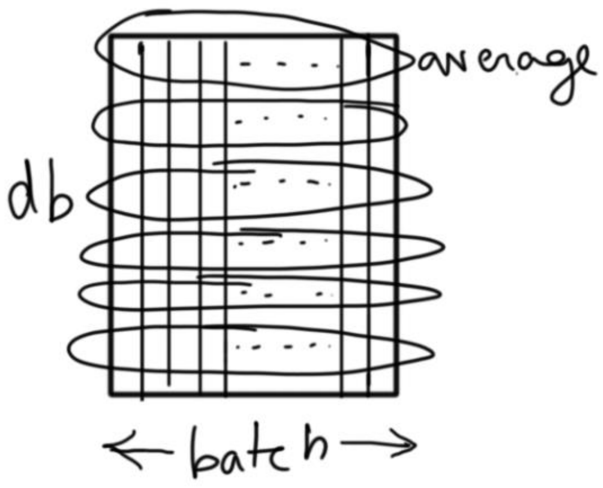
Apart from being computationally faster, the SGD training process has another big advantage - it actually helps you reach the **global minima** (instead of being stuck at a **local minima**). Also, to avoid the problem of getting stuck at a local optimum, you need to strike a balance between **exploration** and **exploitation**.

**Exploration** means that you try to minimize the loss function with different starting points of W and b, i.e., you initialize W and b with different values. On the other hand, **exploitation** means that you try to reach the global minima starting from a particular W and b and do not explore the terrain at all. That might lead you to the lowest point locally, but not the necessarily the global minimum.

You have already seen that when backpropagation happens in batches, the $DW^l = DZ^l.(H^{l-1})^T$ becomes a tensor shown below:



Also, the $Db^l$ becomes a matrix



Hence, the above tensor $DW^l$ and matrix $Db^l$ are averaged along the batch dimension to get a matrix and vector respectively.  This is also a consequence of using the loss function as average over the data points (batch size).

Once, all the gradient matrices/vectors for the weights and biases are calculated, the updates are performed on the weight matrices and the bias vectors.

Let’s now write down the algorithm for one pass of a batch of data through a neural network having L hidden layers:

```
1. $H^0 = B$
2. for $l$ in $[1, 2, \ldots\ldots, L]$ :
    1. $H^l = \sigma(W^l . H^{l-1} + b^l)$
3. $P = normalize(exp(W^o . H^L + b^o))$
4. $L = -\frac{1}{m} Y^T . log(P)$
5. $DZ^{L+1} = P - Y$
6. $DW^{L+1} = DZ^{L+1} . (H^L)^T ; Db^{L+1} = DZ^{L+1}$
7. for $l$ in $[L, L-1, \ldots\ldots .1]$ :
    1. $dH^l = (W^{l+1})^T . DZ^{l+1}$
    2. $DZ^l = dH^l \otimes \sigma'(Z^l)$
    3. $DW^l = DZ^l . (H^{l-1})^T$
    4. $Db^l = DZ^l$
    5. $\frac{\partial L}{\partial W^l} = \frac{1}{m} DW^l$
    6. $\frac{\partial L}{\partial b^l} = \frac{1}{m} Db^l$
8. for $l$ in $[1, 2, \ldots\ldots, L]$ :
    1. $W^l_{new} = W^l_{old} - \alpha . \frac{\partial L}{\partial W^l}$
    2. $b^l_{new} = b^l_{old} - \alpha . \frac{\partial L}{\partial b^l}$
```

## Modifications in Neural Networks

Neural networks are usually large, complex models with tens of thousands of parameters, and thus tend to **overfit** the training data. As with many other ML models, **regularization** is a common technique used in neural networks to address this problem.

One common regularization technique is to add a **regularization term** to the objective function. This term ensures that the model doesn't capture the 'noise' in the dataset, i.e. does not overfit the training data. This, in turn, leads to better **generalizability** of the model.

There are different kinds of regularization techniques, and the one discussed by the professor can be represented in a general form as:

Objective function = Loss Function (Error term) + Regularization term

Recall that the **bias** of a model represents the amount of error the model will commit on a given dataset, while the **variance** of the model measures how much the model changes when trained on a different dataset.

You might also remember that L1 and L2 regularisation (also called ridge and lasso regression) are commonly used to regularize regression models. The same idea can be extended to neural networks.

The objective function can be written as:

$$\text{Objective function} = L(F(x_i), \theta) + \lambda f(\theta)$$

where $L(F(x_i), \theta)$ is the loss function expressed in terms of the model output $F(x_i)$ and the model parameters $\theta$. The second term $\lambda f(\theta)$ has two components - the **regularization parameter** $\lambda$ and the **parameter norm** $f(\theta)$.

There are broadly two types of regularization techniques followed in neural networks:

1. L1 norm: $\lambda f(\theta) = ||\theta||_1$ is the sum of all the model parameters

2. L2 norm: $\lambda f(\theta) = ||\theta||_2$ is the sum of squares of all the model parameters

**Note** that you consider only the weights (and not the biases) in the norm since trying to regularize bias terms has empirically proven to be counterproductive for performance.

The 'parameter norm' regularization is similar to what you had studied in linear regression in almost every aspect. As in **lasso regression** (L1 norm), we get a **sparse weight matrix**, which is not the case with the L2 norm. Despite this fact, the L2 norm is more common because the sum of the squares term is easily **differentiable** which comes in handy during backpropagation.

Apart from using the parameter norm, there is another popular neural network regularization technique called **dropouts**.

The dropout operation is performed by multiplying the weight matrix $W^l$ with an α **mask vector** as shown below.

$$W^l . \alpha$$

For e.g. let's consider a weight matrix of the first layer

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \\ w^1_{41} & w^1_{42} & w^1_{43} \end{bmatrix}$$

Then, the shape of the vector α will be (3,1). Now if the value of q (the probability of 1) is 0.66, the α vector will have two 1s and one 0. Hence, the α vector can be any of the following three:

$$\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \text{ or } \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \text{ or } \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

One of these vectors is then chosen randomly **in each mini-batch.** Let's say that, in some mini batch, the first mask is chosen. Hence, the new (regularised) weight matrix will be:

$$\begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \\ w^1_{41} & w^1_{42} & w^1_{43} \end{bmatrix} . \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} w^1_{11} & w^1_{12} & 0 \\ w^1_{21} & w^1_{22} & 0 \\ w^1_{31} & w^1_{32} & 0 \\ w^1_{41} & w^1_{42} & 0 \end{bmatrix}$$

Some important points to note regarding dropouts are:

- Dropouts can be applied only to some layers of the network (in fact, that is a common practice - you choose some layer arbitrarily to apply dropouts to)
- The mask α is generated independently for each layer during feedforward, and the same mask is used in backpropagation
- The mask changes with each minibatch/iteration, are randomly generated in each iteration (sampled from a Bernoulli with some p(1)=q)

Why the dropout strategy works well is explained through the notion of a **manifold**. Manifold captures the observation that in high dimensional spaces, the data points often actually lie in a **lower-dimensional manifold**. This is observed experimentally and can be understood intuitively as well.

For example, in a 50-dimensional space $R^{50}$, it is likely that the data points actually lie in a much lower dimensional subspace (manifold). The dropout strategy uses this fact to find a lower-dimensional solution to the problem.

Dropouts help in symmetry breaking as well. There is every possibility of the creation of communities within neurons which restricts them from learning independently. Hence, by setting some random set of the weights to zero in every iteration, this community/symmetry is broken. Note that there, a different mini batch is processed in every iteration in an epoch, and dropouts are applied to each mini batch.

Notice that after applying the mask α, one of the columns of the weight matrix is set to zero. If the j^th column is set to zero, it is equivalent to the contribution of the j^th neuron in the previous layer is zero. In other words, you cut off one neuron from the previous layer.

There are other ways to create the mask. One is to create a matrix which has 'q' percentage of the elements set to 1 and the rest 0. You can then multiply this matrix with the weight matrix element-wise to get the final weight matrix. Hence, for a weight matrix

$$\begin{bmatrix} w_{11}^1 & w_{12}^1 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & w_{23}^1 \\ w_{31}^1 & w_{32}^1 & w_{33}^1 \\ w_{41}^1 & w_{42}^1 & w_{43}^1 \end{bmatrix}$$

the mask matrix for 'q' = 0.66 can be

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Multiplying the above matrices element-wise, we get

$$\begin{bmatrix} w_{11}^1 & 0 & w_{13}^1 \\ w_{21}^1 & w_{22}^1 & 0 \\ 0 & w_{32}^1 & w_{33}^1 \\ w_{41}^1 & w_{42}^1 & 0 \end{bmatrix}$$

You need not worry about how to implement dropouts since you just need to write one simple line of code to add dropout in Keras:

```
# dropping out 20% neurons in a layer in Keras
model.add(Dropout(0.2)
```

Please note that '0.2' here is the **probability of zeros** and not ones. This is also one of the hyperparameters. Also, note that you **do not apply** dropout to the output layer.

The mask used here is a matrix. Also, the dropout is applied only during training, not at test time.

Let's now look at one of the widely used tricks in training a Neural Network called **batch normalization.**

The feed forward equations for a single data point are given below:

$$h^1 = \sigma(W^1.x + b^1)$$
$$h^2 = \sigma(W^2.h^1 + b^2) = \sigma(W^2.(\sigma(W^1.x + b^1)) + b^2)$$
$$h^3 = \sigma(W^3.h^2 + b^3) = \sigma(W^3.(\sigma(W^2.(\sigma(W^1.x + b^1)) + b^2)) + b^3)$$
$$h^4 = \sigma(W^4.h^3 + b^4) = \sigma(W^4.(\ \sigma(W^3.(\sigma(W^2.(\sigma(W^1.x + b^1)) + b^2)) + b^3)) + b^4)$$

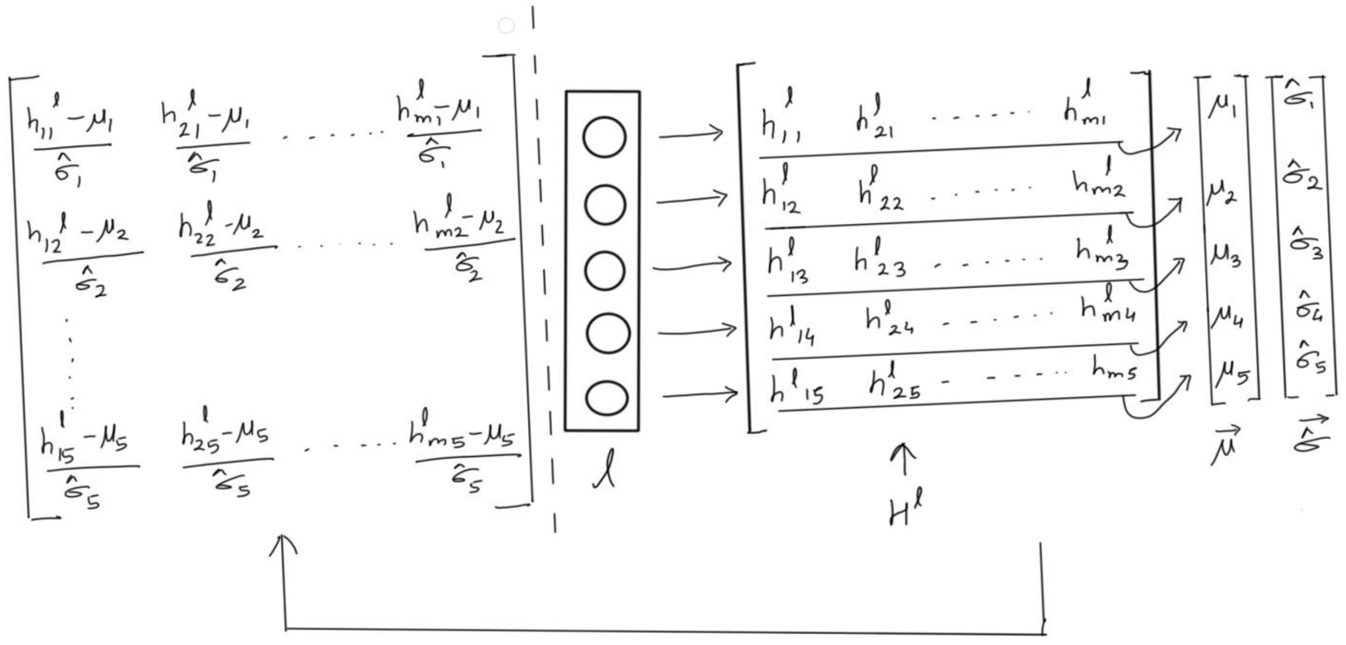
and so on, where σ is the activation function of the layer.

You can see that $h^4$ is a **composite function** of all the weights and biases from the previous layers, i.e. the output of subsequent layers depends on the previous weight matrices. Also, we can see that the interactions between the different weight matrices are non-linear. However, during backpropagation, the weights of various layers are updated independently of each other.

This is a problem because when we update the weights (say) $W^4$, it affects the output $h^4$, which in turn affects the gradient $\partial L/\partial W^5$. Thus, strictly speaking, the updates made to $W^5$ should not get affected by the updates made to $W^4$.

Batchnormalisation is performed on the output of the layers of each batch, $H^l$. It is essentially normalising the matrix $H^l$ across all data points in the batch. Each vector in $H^l$ is normalised by the mean vector μ and the standard deviation vector σ^ computed across a batch.

The following image shows the batchnormalisation process for a layer l. Each column in the matrix $H^l$ represents the output vector of layer l, $h^l$, for each of the m data points in the batch. We compute the μ and the σ^ vectors which represent the 'mean output from layer l across all points in the batch'. We then normalise each column of the matrix $H^l$ using μ and σ^.



$\frac{h_{11}^l - \mu_1}{\hat\sigma_1}$ $\frac{h_{21}^l - \mu_1}{\hat\sigma_1}$ ...... $\frac{h_{m1}^l - \mu_1}{\hat\sigma_1}$
$\frac{h_{12}^l - \mu_2}{\hat\sigma_2}$ $\frac{h_{22}^l - \mu_2}{\hat\sigma_2}$ ...... $\frac{h_{m2}^l - \mu_2}{\hat\sigma_2}$
$\frac{h_{15}^l - \mu_5}{\hat\sigma_5}$ $\frac{h_{25}^l - \mu_5}{\hat\sigma_5}$ ...... $\frac{h_{m5}^l - \mu_5}{\hat\sigma_5}$
$l$
$h_{11}^l$ $h_{21}^l$ ...... $h_{m1}^l$
$h_{12}^l$ $h_{22}^l$ ...... $h_{m2}^l$
$h_{13}^l$ $h_{23}^l$ ...... $h_{m3}^l$
$h_{14}^l$ $h_{24}^l$ ...... $h_{m4}^l$
$h_{15}^l$ $h_{25}^l$ ...... $h_{m5}$
$H^l$
$\mu_1$ $\mu_2$ $\mu_3$ $\mu_4$ $\mu_5$
$\hat\sigma_1$ $\hat\sigma_2$ $\hat\sigma_3$ $\hat\sigma_4$ $\hat\sigma_5$
$\vec\mu$ $\vec{\hat\sigma}$

Hence, if a particular layer l has 5 neurons, we will have $H^l$ of the shape (5, m) where 'm' is the batch size and μ & $\hat{\sigma}$ vectors of shape (5,1). The first element of μ ($\mu_1$) is the mean of the outputs of the first neuron for all the 'm' data points, the second element $\mu_2$ is the mean of the outputs of the second neuron for all the 'm ' data points and so on. Similarly, we get a vector $\hat{\sigma}$ as the standard deviation of the outputs of the five neurons across the m points. The normalisation step is then:

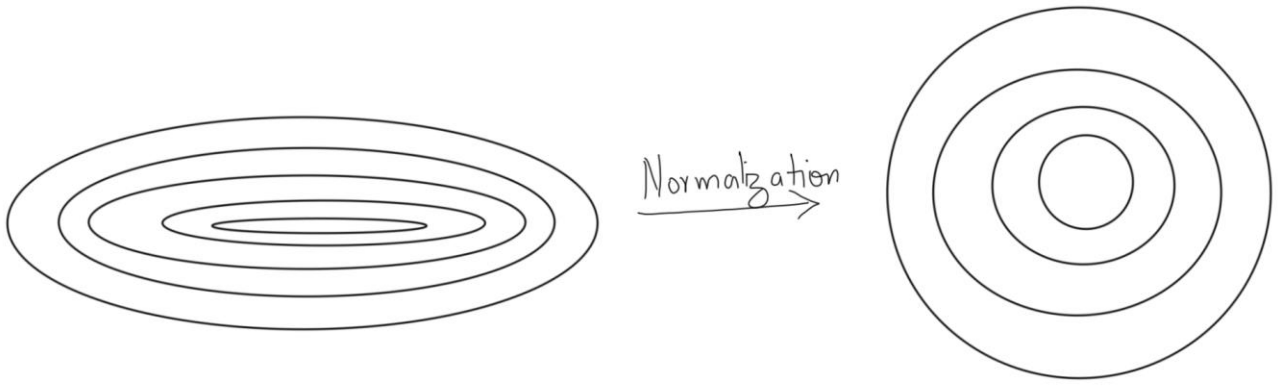
$$H^l = \frac{H^l - \mu}{\hat{\sigma}}$$

This step is performed by **broadcasting** μ and $\hat{\sigma}$. The final $H^l$ after batch normalisation is shown on the left side of the image above.

Batch normalisation is usually done for all the layer outputs except the output layer.

There is one small problem though: How to do batch normalisation during **test time**? Test data points are fedforward one at a time, hence there are no 'batches' during test time. Thus, we do not have any μ and $\hat{\sigma}$ to normalise each test data point with. Thus, we take some sort of an average, i.e. the **average of the μs and σ^s** of the different batches of the training set.

To get an intuition behind how batch normalisation solves the problem of decoupling the weight interactions and improve the training procedure, let's reiterate why normalisation of the input data works at the first place. The answer is that the loss function contours change after normalisation, as shown in the figure below, and that it is easier to find the minimum of the right contours compared to the contours on the left.



Note that the batch normalisation process can also be applied to the cumulative input vector into the layer, $Z^l$ , instead of $H^l$. These are different heuristics, though you need not worry about them since deep learning frameworks such as Keras use empirically proven techniques - you just need to write a simple line of code.

Libraries such as Keras use a slightly different form of batch normalisation. We transform the above equations as:

$$H^l = \frac{H^l - \mu}{\hat{\sigma}} = \frac{H^l - \mu}{\sqrt{\hat{\sigma}^2 + \epsilon}} = \gamma H^l + \beta$$

where the constant ϵ ensures that the denominator doesn't become zero (when the variance is 0). The constants γ, β are hyperparameters. In keras, implementation of batchnormalisation is done as follows:

```
model.add(BatchNormalization(axis=-1, epsilon=0.001, beta_initializer='zeros', gamma_initializer='ones'))
```

The 'axis= -1' specifies that the normalisation should happen across the rows.

----------------------------------------------------------**THE END**----------------------------------------------------------------------