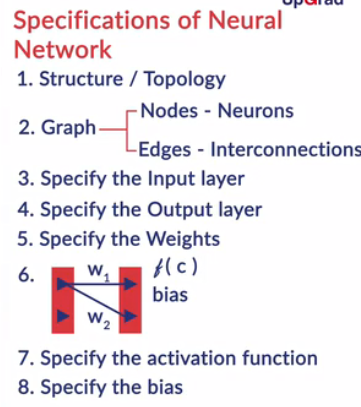
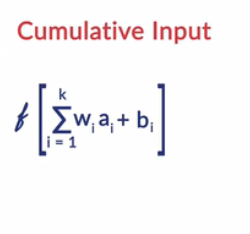
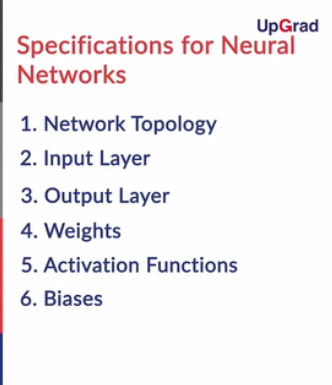


Neural Networks, or Artificial Neural Networks, are a collection of a large number of simple devices called artificial neurons. The most important takeaway is that the brain learns by **training the neurons** to behave in a certain way when given an input, such as a cat. The behaviour is basically how the **inhibitors and amplifiers**of the neurons adjust themselves.

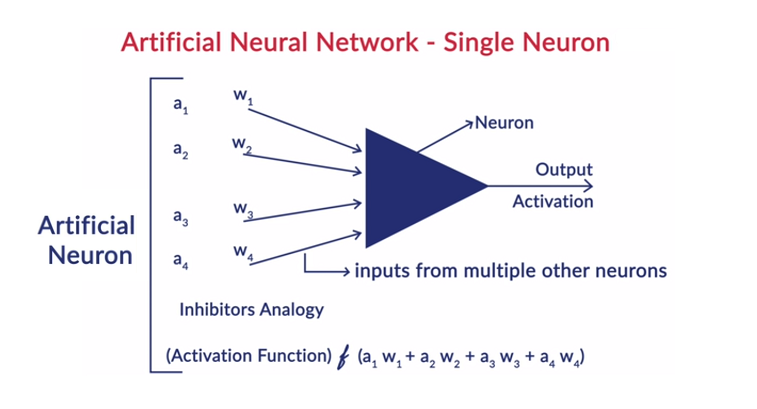
Neural networks are a collection of artificial neurons arranged in a particular topology or structure. In this segment, you will understand how an artificial neuron works i.e. how it converts inputs into outputs. You will also understand the topology or the structure of neural networks.





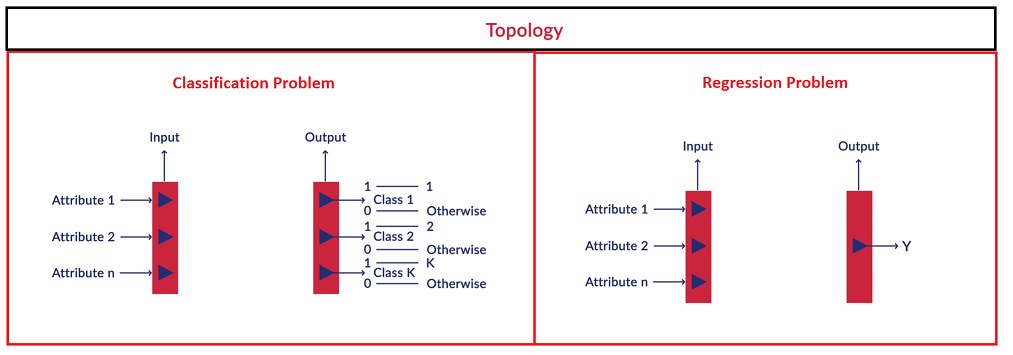


You learnt that an artificial neuron can take multiple inputs (which are outputs of other neurons). On the cumulative weighted sum of the inputs, the activation function is applied to get the output from that neuron. This output can then be fed as input to multiple other neurons. The **weights and the biases** act as **inhibitors or amplifiers** for the inputs.



**Fig:1- Artificial Neuron**

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 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.

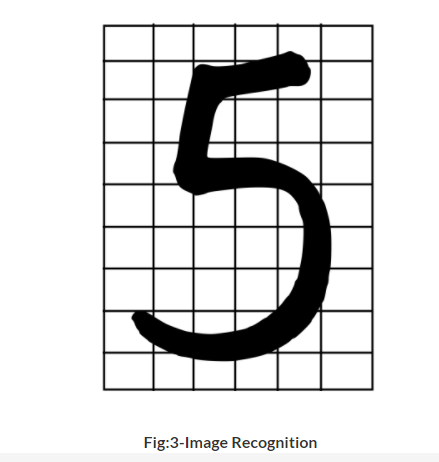


**Fig:2- Topology**

Before you can start working with neural networks you need to specify a few things:

1. **Structure** of the network
2. **Activation** Function
3. The number of neurons in the **input and output layers**

How do you decide the number of neurons in the input layer? For image recognition the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit.



**Fig:3-Image Recognition**

When you are given the image to identify the digit on it (as shown in the image above), how would you get the digit right? Let's first understand the meaning of a pixel. It is the short form for a **picture element**. When we see graphic images on digital devices, the display divides the screen into a large number of pixels arranged in rows and columns (called an array of pixel elements). Each pixel has its own address on this grid and is represented by dots or squares. Pixels build up a sample of an original image and are the smallest component of a digital image. The more pixels used to represent an image, the closer it will resemble the original.

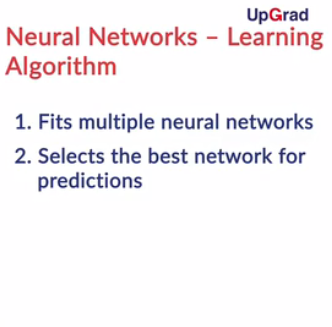
In a neural network, **each pixel**is considered as **one observation**, for example, if you have a 28 X28 pixels grid, the input layer should contain 784 neurons which will be passed to the next layer.

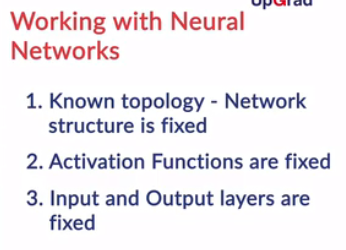
The topology or structure of a neural network defines the overall, zoomed out view of the network. There’s an input layer which takes in the input data and the output layer which generates the network’s output.

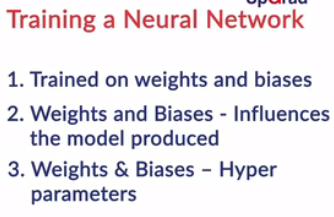
Apart from the input and output layers, a neural network becomes complete when we specify the activation function and all the weights and biases. A network thus contains:

* Input and output layers (number of neurons)
* The weights of all connections
* The bias of each neuron
* The activation function of each neuron

Each neuron i**n the input layer** takes in**one attribute as the input** and each neuron in the output layer generates **an output** of the network. For example, a binary classification task (which generates probabilities of two classes as outputs) may have 2 neurons in the output layer, each representing the probability of a data point belonging to a class.







The neural network learning algorithm, when it’s being trained on data, fits various models to the training data and selects the best model for prediction. The learning algorithm is trained with a fixed network structure, activation functions and input and output layers. It is **trained on the weights and the biases.**

This implies that the best model is the **optimal set of the weights and the biases.**The structure, activation functions and input and output layers remain the same, and thus these are the hyperparameters.

The process of finding the best combination of weights and biases is computationally very expensive. Therefore, we have to make some simplifications and make a few assumptions.

Since neural networks can have a very complex structure, and the tuning of weights and biases can be computationally very expensive, there are certain assumptions which are made to make the computation easy. . Let’s have Prof. Raghavan talk more about those assumptions.

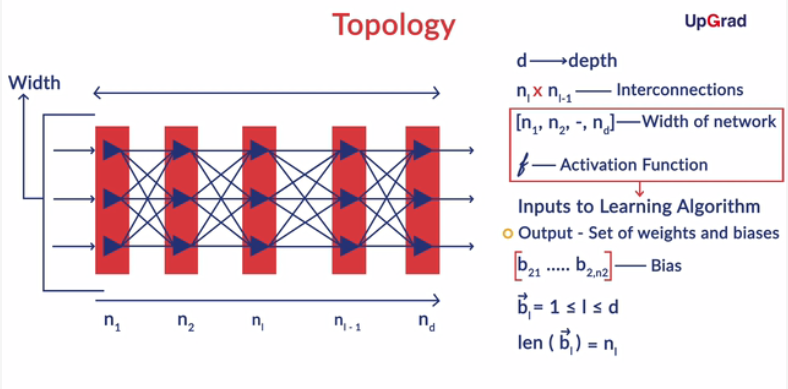
The assumptions for the most common neural network architecture are as follows:

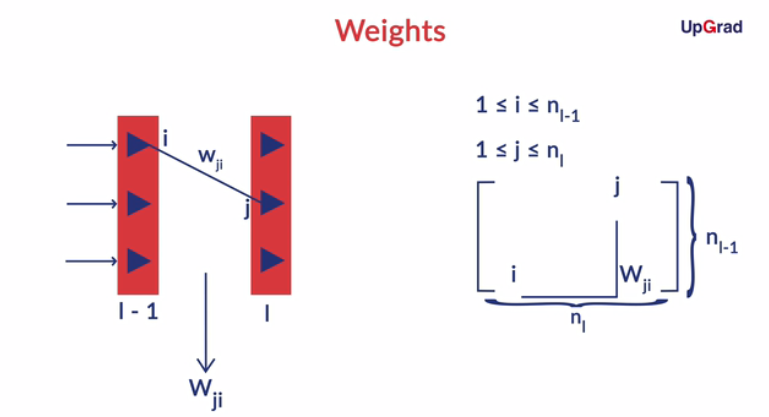
1. The neurons are **arranged in layers**and the layers are arranged **sequentially.**
2. The neurons **within a layer do not interact** with each other.
3. All the inputs enter the network through the input layer and all the outputs go out of the network through the output layer.
4. Neurons in consecutive layers are **densely connected**. This means that all the neurons in layer l_{i}are connected to all the neurons in layer l_{i+1}  and layer l_{i-1}.
5. For **every interconnection** in the neural network there is a **weight**associated with it and for **every neuron**there is a **bias** associated with it.
6. All the layers between the input and the output layers are known as the hidden layers.
7. There can only be an interconnection between neurons of adjacent/consecutive layers.
8. All neurons use same activation function

All of these assumptions have exceptions and neural networks can be made which violate these assumptions, but still perform very well on the data.

**Specifying the Hyperparameters**

To build a neural network, the hyperparameters such as the number of neurons in the input and the output layers, activation functions, the number of layers etc. are to be specified. In this segment, you will learn about hyperparamters.





To specify a network completely means to specify all of these elements:

1. The number of neurons in each layer and the number of layers
2. The activation function
3. A list of biases for each layer
4. A weight matrix for each pair of layers.

The first two elements are used by the learning algorithm to generate the neural network, i.e. they are specified by the network designer beforehand as hyperparameters. The learning algorithm then **tunes the weights and biases** to produce the best model.

Remember (the simplifying assumption) that the entire network has the same activation function.  Activation function is fixed by the network creator and then the learning algorithm uses this activation function to generate output from neurons, eventually tuning the weights and the biases.

In the next segment, you will learn more about activation functions and the most commonly used ones .

We know that a completely defined neural network contains all the following:

* The number of layers and neurons
* The weights of all connections between neurons
* The bias of each neuron
* The activation function

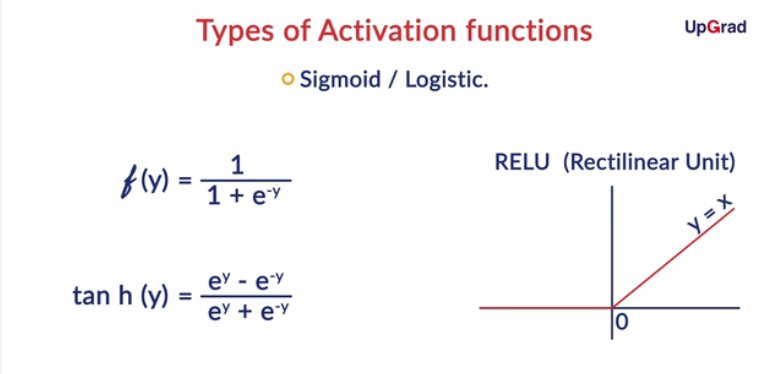
We specify the number of layers, number of neurons in the layers and the activation function. The network is then trained on the weights W and biases b, i.e. the weights and biases are found by optimising some cost function. An analogy with logistic regression is that we specify the number of inputs, outputs and the sigmoid function and train the model to find the optimal values of the coefficients.

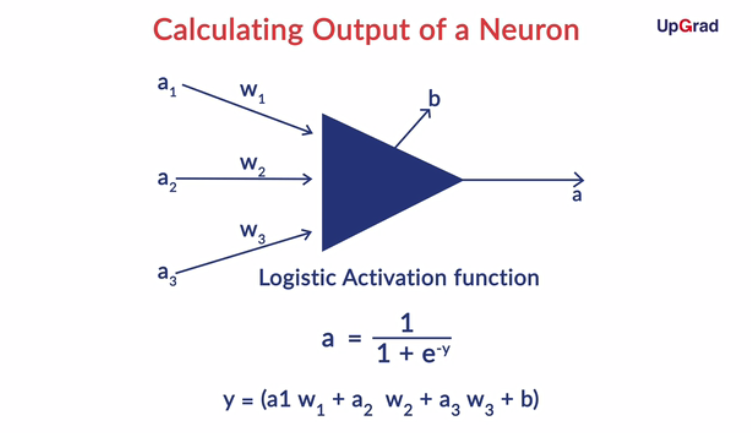
We know that a completely defined neural network contains all the following:

* The number of layers and neurons
* The weights of all connections between neurons
* The bias of each neuron
* The activation function

We specify the number of layers, number of neurons in the layers and the activation function. The network is then trained on the weights W and biases b, i.e. the weights and biases are found by optimising some cost function. An analogy with logistic regression is that we specify the number of inputs, outputs and the sigmoid function and train the model to find the optimal values of the coefficients.

In the previous segments, you learnt about the topology of neural networks, the underlying simplifying assumptions and the specifications related to the hyper-parameters. In this segment, you will understand how the output is calculated from a neuron using an activation function and the types and properties of an activation function.





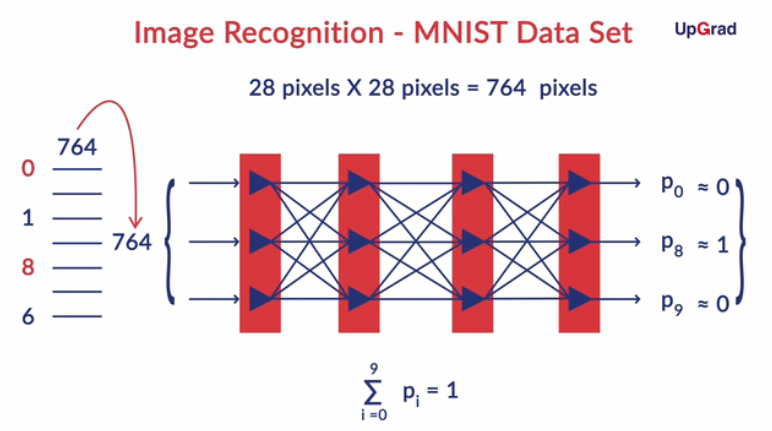
To strike the correct architecture, you need to find the best combination of the number of layers in the network and the number of neurons in each layer. Also, you must choose an activation function.

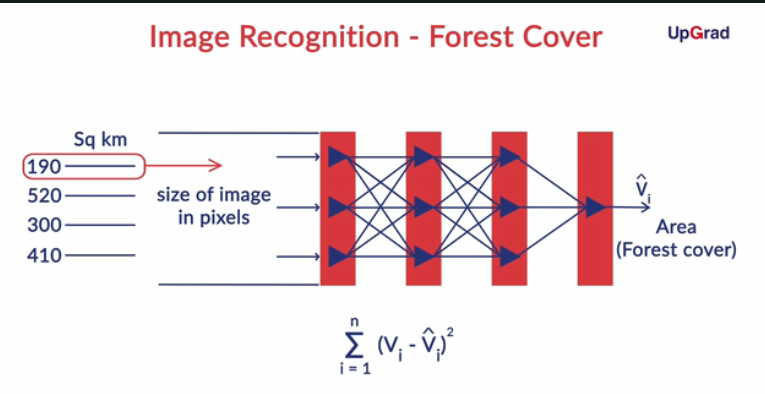
Essentially, activation function could be any function but it should have some properties. They are:

* 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 should be ensured because non-linearity helps in making neural networks more compact.

The most popular activation functions used for neural networks are:

* Logistic function - f(y) = \frac{1}{1 + e^{-y}}
* Hyperbolic tangent function - tan h(y) =\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}
* Rectilinear Unit - y = x \forall x \geq 0 \& 0 otherwise.
* The output of a neuron is basically the activation function applied to the cumulative input to that neuron. If the cumulative input to the neuron is  y = a_{1}w_{1} +a_{2}w_{2} + a_{3}w_{3} + b then using the logistic activation function the output out of that neuron will be a = \frac{1}{1 + e^{-a_{1}w_{1} -a_{2}w_{2} -a_{3}w_{3} -b}} .  Neural networks are also known as universal approximators, because with non-linear activations they can approximate any measurable function to any desired degree of accuracy.
* Interestingly, note that **each hidden layer**in a network acts as a **mini logistic regression model**if you use the sigmoid activation function. If you treat the cumulative input of the previous layer as a feature vector, the **current layer applies a sigmoid function** on it and converts it to the output the output vector.





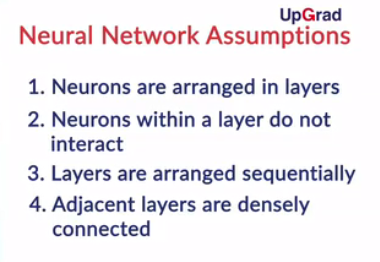
 There are various problems you face while trying to recognise handwritten text using an algorithm such as:

* Noise in the image
* Orientation of the text
* Non-uniformity in the spacing of text
* Non-uniformity in handwritings.

MNIST data set takes care of a few of problems listed above because in the images it has, the digits are written in a box and they do not touch the box. Now the only problem the network needs to take care of the non-uniformity in handwritings.  Since the images in the MNIST data set are 28 X 28 pixels, the input layer has 764 neurons (each neuron takes 1 pixel as input) and the output layer has 10 neurons each giving the probability of the input image belonging to any of the 10 classes.  The image is classified to the class represented by the neuron with the highest probability.

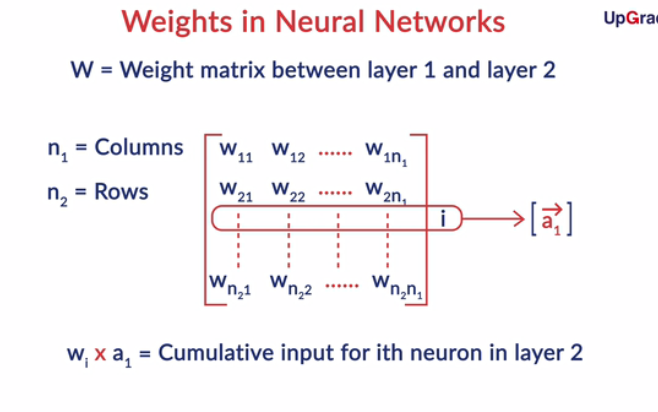
# Flow of Information in Neural Networks - Between 2 Layers

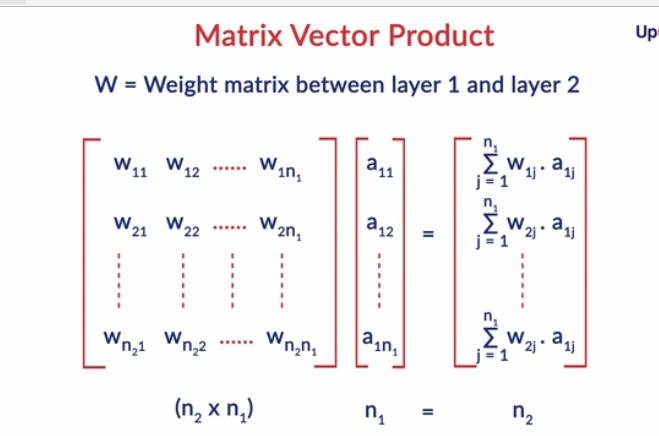
In the previous session. you learnt about the structure, topology, hyperparameters and the simplifying assumptions of neural networks. In this segment, you will understand through an example of two layers how the information transefers from one layer to the adjacent one in a neural network.  The flow of information from one layer to the next one, i.e. from left to right is called **feedforward.**

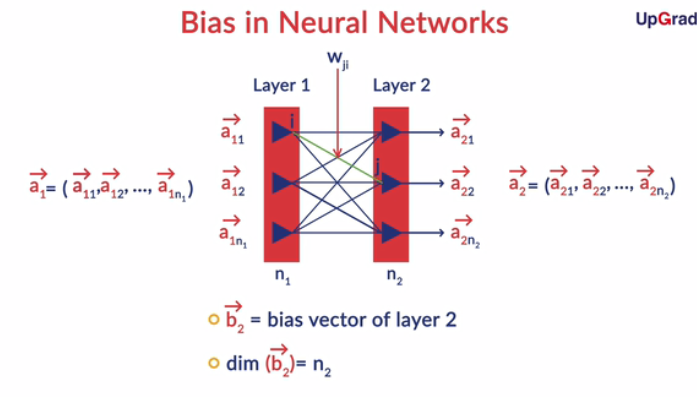


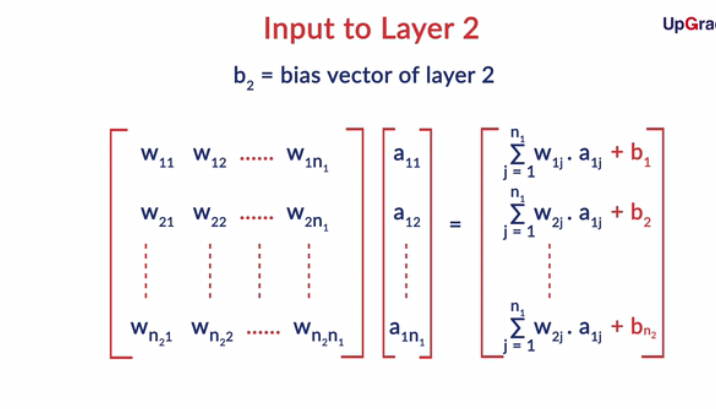
The output of a layer is also called its **activation**(since the activation function produces the output). If layer 1 has three neurons, the output / activation of  layer 1 is a: **Vector of length 3**

Each neuron has exactly one output (a number). Thus, the output of layer 1 is a vector of 3 individual outputs

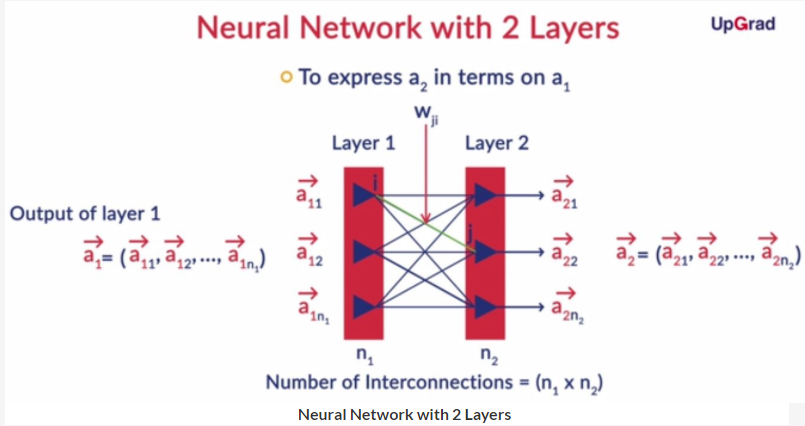








In this segment, you learnt how information flows from one layer to the other in neural networks using a simple example of 2 layers. The number of neurons in layer 1 and layer 2 are assumed to be n_{1} and n_{2} respectively. Therefore the output coming out of the first layer is the vector  a_{1} =(a_{11}, a_{12}, ..... ,a_{1n_{1}})  and the output from the second layer is the vector a_{2} = (a_{21}, a_{22}, a_{23}......a_{2n_{2}}). Also, there would be weights attached to each of the interconnection and for each pair of layers, there is a weight matrix associated which is of the order n_{2}Xn_{1}  .



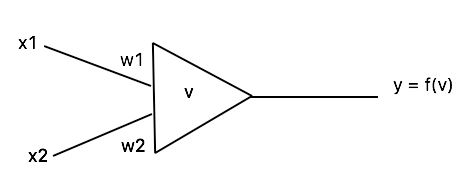
**Neural Network with 2 Layers**

Now, to compute the output of the i^{th} neuron in the second layer:

* Multiply the i^{th}  row of the weight matrix with the output of layer 1 to get the weighted sum of inputs.
* Convert the weighted sum to cumulative sum by adding the bias term associated with the i^{th} neuron to the weighted sum.
* Apply the activation function to the cumulative input to get the output of the i^{th} neuron in the second layer.

# Comprehension - Calculating Output of a Neuron

The following figure shows a single neuron which takes two logical inputs x_{1} and x_{2} each of which can take the values 0 or 1 only. The weights corresponding to them are w_{1} and w_{2} and the bias of the neuron is 0. The weighted average of the inputs is called v, which is then converted by the activation function f to the output y.



**Single Neuron**

This simple ‘network’ can act as a logical gate. For example, the AND gate returns 1 as the output only when both inputs are 1, else it returns 0.

|  |  |
| --- | --- |
| Table 1: The AND Gate | |
| Input (x_{1}, x_{2}) | Output(y) |
| (0, 0) | 0 |
| (0, 1) | 0 |
| (1, 0) | 0 |
| (1, 1) | 1 |

For an AND gate, the function f and the weights are defined as:

f(v) = 1 if v\geq 2

             0 otherwise

The weight matrix for an AND gate is w = \begin{bmatrix} 1 &1 \end{bmatrix}, which is a 1 x 2 matrix. The input vector x = \begin{bmatrix} x1 & x2 \end{bmatrix}^{T}, which is a 2 x 1 matrix.  The value of v is thus ​​​​​​wx.

Consider that the inputs x_{1} and x_{2} are two attributes and the output y is a logical value - 0 or 1. The task of the network is to implement a logical gate.

# Information Flow in Neural Networks

In the previous segment, you learnt how information is passed between two layers using an example of 2 layers. You will now understand how it is passed in a complete neural network.

The process of computing output from the input is called **feedforward**or **inference.**To summarize the feedforward process, we use the following to compute the output of a network (obviously assuming that we have the values of weights and biases which are calculated using **backpropagation**which you'll study later):

* a_{0} is the input to the network
* The network has h hidden layers and the h+1^{th} layer is the output layer.
* The weight matrix for layer l is of the order n_{l}Xn_{l-1}
* The bias vector for layer l has n_{l} elements.

The output from the network will be calculated in a sequential manner. Given the input a_{0} to the network, you multiply the input vector with the matrix w_{1} and add the bias vector b_{1} to get the cumulative input z_{1}. Activation function is then applied to z_{1} to give \sigma(z_{1}) which is the output of layer 1 (or the input to layer 2).

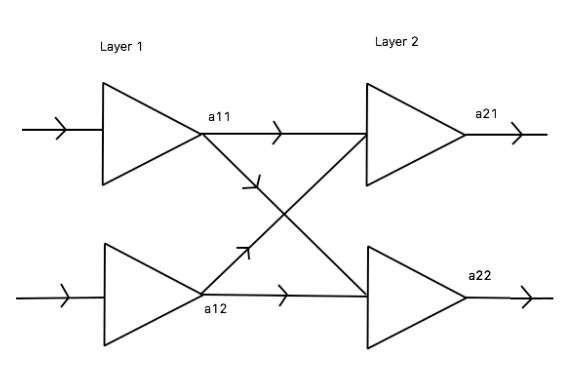
The input to layer 2 is now a_{1}. You multiply a_{1} with w_{2}, add the bias vector b_{2}. This becomes z_{2} and upon applying activation to this you get the output of layer 2. You can continue this process till you get the output of the last layer.

**Comprehension - Inference in a Neural Network**

Consider two adjacent hidden layers 1 and 2 inside a neural network. Both layers contain two neurons and the layer 1feeds into the layer 2, i.e. output of layer 1 is the input of layer 2.

Given the output from the previous layer 1, we can calculate the output of the current layer 2 using the weights, biases and the activation function.

* Assume that the output vector of layer 1,  [a_{11}, a_{12}] is given as  a = [1, -1]
* The activation function of all 4 neurons is f(x) = x + 1
* The weight matrix is defined below with wji representing the input from ith neuron in the previous layer to the jth neuron in the current (second) layer
  + w_{11} = 1
  + w_{12} = 2
  + w_{21}= -1
  + w_{22}= 0



**Inference in Neural Networks**

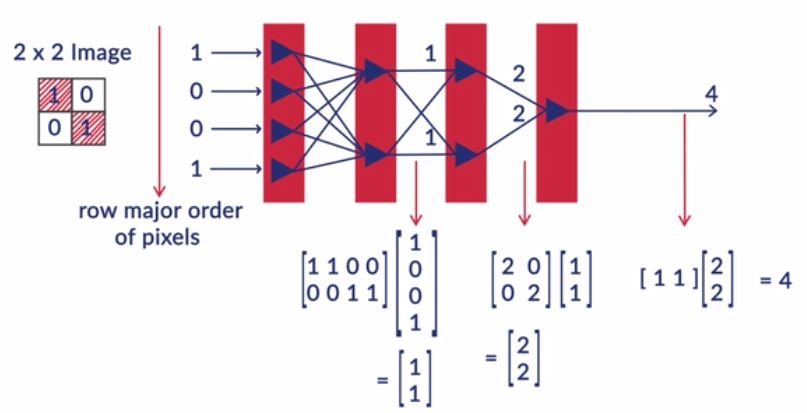
The output of layer 2 is then given by f(Wa +b) where W is the weight matrix.

# Information Flow - Image Recognition

In this segment, you will understand how **feedforward** or **inference** happens using an example of a 2 pixel X 2 pixel grey scale image. We will discuss a simple network whose aim is to find an amplified count of the number of grey (or 'on') pixels in the image.

In the example discussed, the objective of the network is to calculate the amplified count (or number) of 'on' pixels in the 2 x 2 image.

The first layer in the network counts the number of grey pixels in each layer of the image. Since, its a 2 X 2 image, the weight matrix associated with it is   \begin{bmatrix} 1 & 1 &0 &0 \\ 0&0 &1 & 1 \end{bmatrix} . The second layer in the network simply amplifies the count by 2 by multiplying the output with the weight matrix \begin{bmatrix} 2& 0\\ 0 & 2 \end{bmatrix}. And the output layer finally gives the final amplified count.

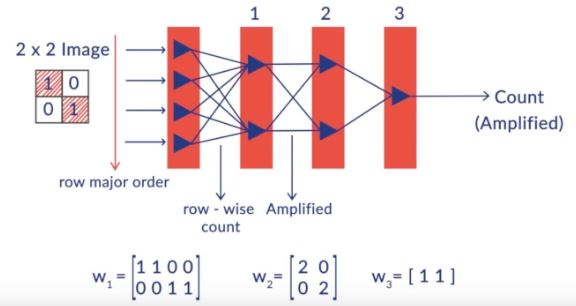


**Counting Dark Pixels in 2X2 Image**

# Comprehension - Count of Pixels

The figure below shows an artificial neural network which calculates the count of the number of pixels which are ‘on’, i.e. have a value of 1. It further amplifies the output by a factor of 2; so if 2 pixels are on, the output is 4, if 3 pixels are on it is 6 and so on.

We’ll call the input layer as layer 0 or simply the input layer. The other three layers are named 1, 2 and 3 (3 is the output layer).



**Pixel Count**

The weight matrices of the 2 hidden layers and the third output layer are shown above. The first and the second neurons of (hidden) layer 1 represent the number of ‘on’ pixels in row 1 and 2 respectively. The second (hidden) layer amplifies the output of layer 1 by a factor of 2 and the third (output) layer sums up the amplified counts.

The biases are all 0 and the activation function is f(x) = x.

Let’s represent the input with 4 pixels as a vector with pixels **counted clockwise**. Thus the **input shown in the figure is [1, 0, 1, 0]'**(and not [1, 0, 0, 1] which is called the row-major order).

Training a Neural Network

# Introduction

In the previous session, you learnt how the information flows in a neural network and how can a simple image can be recognised using neural networks. In this session, you will understand what exactly does it mean to train a neural network and how are neural networks trained.

## In this Session :

* Training a Neural Network
* Cost Function and its Complexity
* Gradient Descent in Neural Networks
* Backpropagation

# What does training a Network mean?

In the previous session, you learnt how the information/messages pass through neural networks.  Basically, you learnt how to reach to the final output given an input to the network (the name for which is feedforward or inference).

In this segment, you will understand how neural networks are **trained.**Recall that the training task is to calculate the weights and biases by **minimizing some cost function**. What do you think the cost should be? And to minimize it, can we use a technique we have already used before?

The idea of training ANNs is exactly the same as that in other models like linear regression, SVMs etc. The desired output (output from the last layer) - the actual output equals the cost, and we try to find the parameters w and b such that **the total cost is minimized.**

A crucial point to note is that if the data is large, cost calculation itself can get pretty messy in ANNs. If you have a million data points, then each point will be fed into the network, the output will be calculated using feedforward and the cost Ci (for ith data point) will be calculated. To get the total cost, this will be repeated a million times (Total Cost C = C_{1} + C_{2} + C_{3} ...... C_{1000000}).

Once the total cost C is represented as a function of w's and b's, we can minimize it to find the optimal w's and b's. In fact, you have already studied the technique to do this - **gradient descent.**We will discuss that in the next few segments.

# Complexity of the Cost Function

In the last segment, you understood that the training of neural networks is simply finding the combination of w and b which minimises the overall cost keeping all the other hyperparameters such as structure specified. In this segment, you will understand the complexity of the cost function and how can it be minimised.

The idea is the same old **gradient descent** algorithm to minimize cost with respect to w's and b's. Recall that in gradient descent, the parameter being optimised (w or b) is iterated in the direction of reducing cost as W(new) = W(old) - d(Cost)/d(W). The same can be written for the biases.

The only challenge in ANNs is that since W is a huge matrix and b is a set of vectors, the total cost C as a function of W and b is a complex function.

In this segment, you learnt that the cost function for a very small and simple neural network can be very complex. The best way to minimise this complex cost function is by using gradient descent. The process of applying gradient descent is as follows:

* Start with arbitrary w and b values.
* Take a small step towards the steepest direction i.e. towards the most negative gradient
* The magnitude of the step is decided by the learning rate lambda

So training a NN is basically finding the best combination of w and b using gradient descent such that the cost is minimised. The update is carried out until the algorithm converges. Convergence happens when there is no change in the w and b values with further iterations of gradient descent.

# Updating the Weights and Biases - I

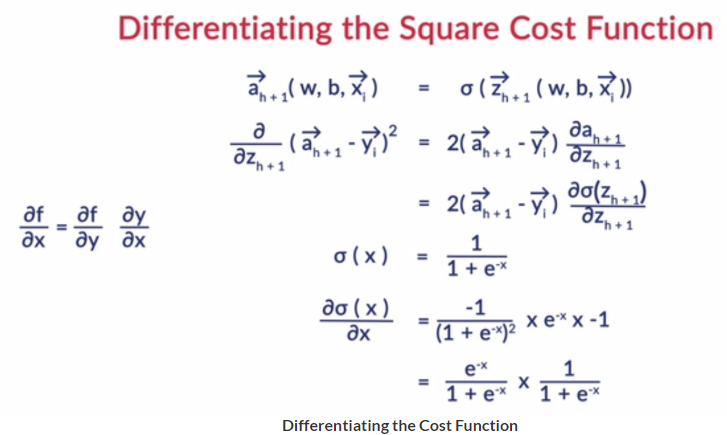
In the previous segment, you learnt that the training process is the updating of weights and biases using gradient descent to find the best combination such that the cost is minimised. In this segment, you will understand how exactly the update process is carried out.

Going forward, we will use the variable z to simplify the notations. Recall that the cumulative input into a layer l is represented as z_{l} = W_{l}a_{l-1} + b_{l}, where W_{l} is the weight matrix between layers l-1 and l and a_{l-1} is the output of layer l-1. The activation function applied on this gives f(W_{l}a_{l-1} + b_{l}) = f(z_{l}) = a_{l}  which is the output of the layer l.

Assuming a square cost function, C_{i} for the ith data point can be written as C_{i} = (a_{h+1} - y_{i})^{2} . But the choice of cost function does not really matter as the methodology of updates remains the same.

The aim is to find  \frac{\partial C_{i}}{\partial w_{z+1}}  and \frac{\partial C_{i}}{\partial b_{z+1}} where the cost function is C_{i} = (a_{h+1} - y_{i})^{2}

* Instead of differentiating the cost function C_{i} = (a_{h+1} - y_{i})^{2}  w.r.t w_{h+1} and b_{h+1} directly, we first differentiate the cost function w.r.t z_{h+1} as a_{h+1} can be written as \sigma(z_{h+1}) .
* In the first session, you learnt that an activation function should be smooth. This is because a smooth function is easy to differentiate. S0 for a logistic function - f(x) = \frac{1}{1 + e^{-x}}, \frac{\partial f(x)}{\partial x}= f(-x)f(x).



* Using a logistic activation function, the differential of C_{i} w.r.t z_{h+1} comes out to be 2(a_{h+1} - y_{1}) \sigma(-z_{h+1})\sigma(z_{h+1}).
* To get the final differential - \frac{\partial C_{i}}{\partial w_{h+1}} and \frac{\partial C_{i}}{\partial b_{h+1}}, we will first differentiate z_{h+1} = w_{h+1}a_{h} + b_{h+1} w.r.t to w and b and then use the rule    and  . And the term  \frac{\partial C_{i}}{\partial z_{h+1}} has already been calculated before and the terms \frac{\partial z_{h+1}}{\partial w_{h+1}} and \frac{\partial z_{h+1}}{\partial b_{h+1}} are the vectors a_{h} and 1 respectively.
* Differentiating a vector with a vector, you get a matrix and differentiating a matrix with a vector you get a tensor.

This calculation was done assuming the weights and biases are specified i.e. it's a feed forward network where the input traverses through all the hidden layers with specified weights and biases.

# Updating the Weights and Biases - II

In the last segment, you learnt how to calculate the gradients of the cost function C_{i} w.r.t w and b  for the last layer. Since the cost is equal to the output of the last layer - actual output, we started with calculating da_{h+1} / dw_{h+1}, i.e. the derivative of output of last layer with respect to the weights of last layer. To optimize all the w's and b's, we need to calculate the **derivative with respect to w's and b's of all layers.**

Now you will learn how to calculate the gradients of the cost function for all the layers. It turns out that we can calculate the gradients of the previous layers using the gradient of the last layer, which we have already calculated.

 We can break the gradient computation process into the following steps:

1. We want to calculate \frac{\partial C}{\partial w_{h+1}} = \frac{\partial C}{\partial z_{h+1}} \frac{\partial z_{h+1}}{\partial w_{h+1}} which is done in two steps
2. The second term \frac{\partial z_{h+1}}{\partial w_{h+1}} is easy to calculate and is equal to a_{h} since the a_{h} does not depend on the weights of the layer h+1 and can thus be treated as a constant (remember these are partial differentials).
3. Calculation of the**first term \frac{\partial C}{\partial z_{h+1}} is the key step in backpropagation.**Once you express \frac{\partial C}{\partial z_{h}} in terms of \frac{\partial C}{\partial z_{h+1}}, it means that you can now calculate the **gradient at every layer working backwards**.
   * The following relationship is the mathematical representation of the above statement:dC/dz_{l} = dC/dz_{l+1} W_{l+1} f(z_{l}) f(z_{-l})

So the entire training happens in the following steps:

1. Initialize the w's and b's to arbitrary (guessed) values
2. For each data point, do a **feedforward using the current values of w and b**and calculate the output
3. Using **backpropagation,**calculate the gradient dC/dw_{l} for all layers and find the total gradient dC/dW
4. Do the above for all data points and calculate **the total cost C = \sum C_{i}**
5. Update the weights and biases to complete one iteration; perform step 5 till the total  cost converges
6. **Comprehension - Training a Neural network**
8. Training a network essentially means to find the optimal set of weights and biases to minimise the cost function. The cost function is the difference between the actual output of a training point and the output predicted by the network.
10. From the example of logical OR gate taken before, we know that the input data (i.e. the definition of an OR gate) is as given in the following table. In an OR gate, the output is 1 if at least one of the inputs is 1, else it is 0.

|  |  |
| --- | --- |
| Table 1 - OR Gate | |
| Input(x_{1}, x_{2}) | Output(y) |
| (0, 0) | 0 |
| (0, 1) | 1 |
| (1, 0) | 1 |
| (1, 1) | 1 |

2. The table above is the training dataset. We have fixed the number of neurons to 1 (in the only hidden layer). The activation function f for an OR gate is defined as follows:
4. f(v) = 1 if v\geq 1
5. 0 otherwise

# Comprehension - How Weights and Biases are Updated

We have established that the most difficult challenge in training a network is to use the gradient descent algorithm to minimise the cost. The task is to find the weights and biases such that the total cost, i.e. cost corresponding to all data points is minimised. The idea of gradient descent is this -

* Start with an initial guess of w and b
* Calculate the total cost using w and b and
* Change w and b in the direction of reducing cost.

We start with an initial guess of w and b as usual. We already know how cost C can be represented as a function of the weights w and biases b. Given an input, and the guessed weights and biases, we can calculate the output of the network using feedforward  and hence also calculate the cost C.

dC/dW measures how cost changes as the weights change. The cost C depends on the output of the last layer, which depends on weights of last layer W, thus \frac{\partial C}{\partial W_{l}} is easily calculated.Using  \frac{\partial C}{\partial W_{l}}, \frac{\partial C}{\partial W_{l-1}}  is calculated and the chain continues in the backwards direction.

The process of calculating \frac{\partial C}{\partial W}and \frac{\partial C}{\partial b} is called **backpropagation,**since each layer's gradient is calculated using the next layer's and the process continues starting from the output layer till the first layer. Both the gradients depend on the current values of W and b.

The loop of steps 1, 2 and 3 continues till the total cost is sufficiently minimised.

Training in Batches

# Introduction

In the previous session, you learnt how using gradient descent optimal weights and biases are found at each iteration of weights and biases. Then using the new set of weights and biases the second iteration of an update is carried out. This is known as backpropagation, and you also understood that backpropagation is computationally very expensive. In this session, you will learn how the problem of computational complexity is handled in neural networks.

## In this Session:

* Stochastic Gradient Descent
* Learning in Epochs and Batches
* Exploration and Exploitation - Finding Global Optimum

# Stochastic Gradient Descent

In the previous session, you learnt how exactly can the weights and biases be updated using backpropagation. For updating weights and biases using plain backpropagation,  to make a single update you have to scan through the entire data set once. This is computationally very expensive and unrealistic for large data sets. In this segment, you will learn stochastic gradient descent which helps you reduce the computational complexity of the algorithm significantly.

To counter the problem of computational complexity while training the neural networks, batch learning is performed.

* Divide the data in epochs and in each epoch further divide it into small batches.
* At the start of each epoch, shuffle the data set and find the gradient values for each batch.
* The average of the gradient values for each batch are then used to make a weight bias update for that batch. So, instead of making 1 update after scanning through the entire data set, in batch learning, you make one update per batch.

# Exploration and Exploitation

In this segment, you will understand the problem of getting stuck at a local optimum while optimising a complex nonlinear function and will also learn how to avoid this problem.

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 minimise the cost function with various different starting points of w  and b i.e. you initialise w and b to various different values to find the gradients to minimise the cost function. And exploitation means that you try to achieve the best optimum given a particular starting point of w and b  and do not explore for better options. That optimum might not be the global optimum but locally it will be the best optimum.

# Comprehension - Stochastic Gradient Descent

Training neural networks is a horrendously complex task especially when the number of data points and the size of the network is large. If you have a million data points (say images), and each image has 1000 attributes (pixels), then the network would have to scan all the million data points to make a single update to the w’s and b’s. It may take several such iterations to get the right set of w’s and b’s.

To solve this problem, we use stochastic gradient descent. We divide the million data points in batches of size b (say b = 100). In each update of w’s and b’s, we use only b = 100 data points. To scan the whole training data once, we need 1 million / b = 10,000 iterations.

An epoch is one pass of all the training examples.

Representation Learning

# Introduction

In the previous session, you learnt how the problem of computational complexity is handled in neural networks using stochastic gradient descent and how learning is carried out in epochs and batches. In this session, you will understand the various problems when using backpropagation and would learn about representation learning and convolutional neural networks.

## In this Session:

* Problems with Backpropagation
* Representation Learning
* Convolutional Neural Networks

# Problems with Backpropagation

In the previous sessions, you understood the entire process of weight-bias update which is carried out from the **last layer to the first layer** and is thus referred to as **backpropagation**. The same old gradient descent algorithm makes updates to the weights and biases till the cost reduces to an acceptable level.

Since the weights and biases get updated from the last to the first layer, i.e. in the backward direction, the weights and biases in the initial layers face a problem called **vanishing gradient.**

You learnt that as the network gets deeper, the gradients get smaller towards the initial layers. Due to this the initial layers stop learning because the small gradients when propagated back to the early layers do not make any change to the weights and biases. The vanishing gradient problem and it occurs in two ways:

* Gradients become 0 and learning stops.
* Learning becomes highly inconsistent as the gradients become very large.

To tackle the vanishing gradient problem you can -

* Train **single layers or pairs of layers** one by one and at the end do an end to end learning pass.
* Instead of plain vanilla backpropagation (where you tune the weights and biases of the entire network for 1 data point and then tune them again using the next data point and so on), you tune the weights and biases of each layer using the entire dataset and then make an end to end learning pass.

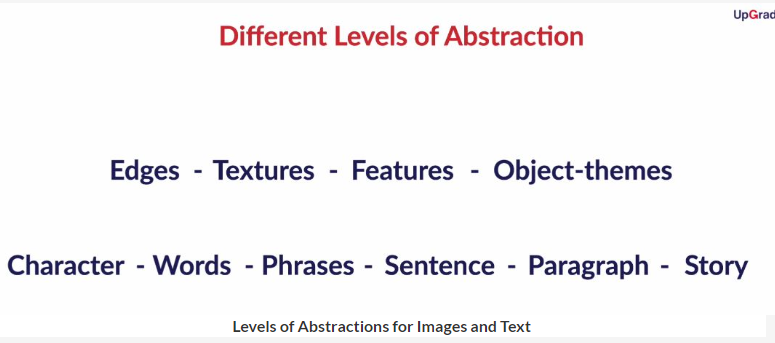
# Representation Learning

When you look at an image, do you notice each pixel separately to understand it? Not really. You probably break a cat's image into broader chunks like ears, eyes, feet etc. and immediately see a cat. But until now, we have been talking about training neurons to recognise things at a pixel level. Similarly, in case of audio, we do not learn to hear at the level of bytes of data, but at a broader level of pitch, accent etc.

We can certainly look at things at a pixel or byte level, but that would be inefficient for learning. We look at a task at  a certain **level of abstraction.**

In this segment, we will cover a very important type of learning called **representation learning.**The philosophy is inspired from how humans learn by breaking down complex problems to different levels or **layers of abstraction.**

In this segment, you learnt that learning in the human brain happens at different layers or at different levels of abstraction.



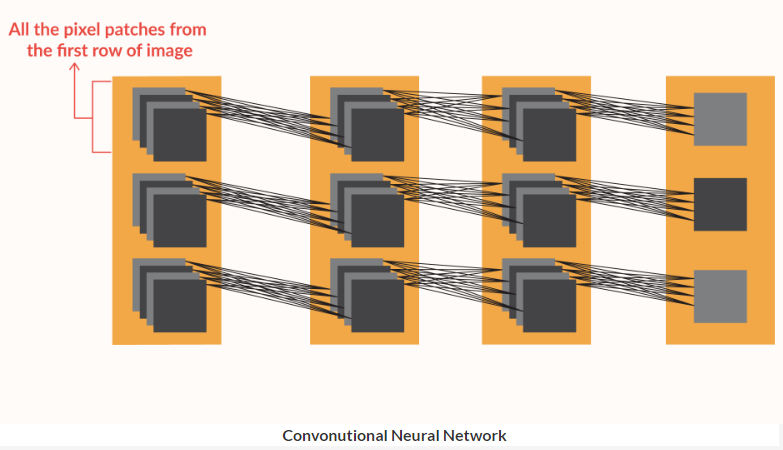
**Levels of Abstractions for Images and Text**

To translate this notion of representation learning into neural networks to solve a complex problem you can:

# Convolutional Neural Networks

In this segment, you will understand the image recognition (classification) and the forest cover prediction (regression)  examples which were discussed in the previous sessions from the point of view of representation learning. You will also understand another type of neural networks called**convolutional neural networks** which are very useful in image analysis.

Feed Forward neural networks are most popularly known as Multi-Layered Perceptrons as each neuron in the layers is analogous to a perceptron in working. Convolutional neural networks are an extension of the MLP architecture which takes as input the pixel patches of the image.



Each of the pixel patch in the input layer feeds into corresponding pixels in the next layer and each of the pixel tries to recognise a feature (could be same or different) in the original pixel patch and there could be several such layers of convolution.

Recurrent Neural Networks

# Introduction

In the previous session, you learnt about representation learning, which is learning from different layers of abstraction. You also learnt about Convolutional neural networks which are heavily used in image recognition. In this session, you will understand how sequential data i.e. text is recognised using neural networks.

## In this Session:

* Sequential Data
* Recurrent Neural Networks.
* Regularisation in neural Networks

# Dealing with Sequential Data

Until now, we have not discussed anything about the **memory**of a network. We know that memory forms a large part of learning that we undergo as humans, and there's no reason we shouldn't extend this idea to artificial networks.

When you listen to the word 'neural', your memory suggests that the next word is likely to be 'network'. Also, you have been able to learn this pattern because you have heard the words 'neural network' **in that order**. You will not be able to learn or identify this pattern if the words 'neural' and 'network' are spoken in random order every time.

This suggests that 1) we may be able to apply the **idea of memory**to artificial networks and 2) the order in which data points are fed into the networks is important.

**Recurrent Neural Networks**attempt to incorporate these ideas into artificial networks.

While implementing an ANN, the data is assumed to be IID (independent and identically distributed). But while performing text analysis, the data cannot be treated as IID because the choice of the first word has clear implications on the choice of the second word. Sequential data is very well handled by recurrent neural networks, and its more advanced version is LSTM (Long -short-term memory networks).  In **RNNs** or **LSTMs** you want the neurons and the network to have the following properties:

* The neurons must be isolated i.e. the output of a neuron should not corrupt the output of the other neurons.
* There should be some memory associated with the neurons.
* The neurons behave in such a way that the neural networks can make sense of the context of the text.

The memory helps because the network looks at its response for the earlier part of the sequence along with the current input and on that basis, it decides the output of the current input.

# Regularisation in Neural Networks

You have learnt in the previous sessions that neural networks inherently are very complex structures  with a lot of hyperparameters. For some of the hyperparameters, you need to find the best combination yourself such as the number of layers and number of neurons in each layer and the parameters (weights and the biases) are tuned by the algorithm. Since the network structure and the training process is so complex, overfitting is a major problem in neural networks. In this segment, you will learn how neural networks are regularised to avoid the problem of overfitting.

In this segment, you learnt the two regularisation strategies which can be used to regularise neural networks:

* The first strategy is similar to the regularisation in regression. To the cost function, regularisation terms \lambda w^{2}and \eta b^{2} are added.
* Another strategy of regularisation which is also the better strategy is to drop neurons at random between epochs. Between epochs, when the data is reshuffled, weights of certain neurons are made 0 at random and they are dropped from the network. Although, deciding the number of neurons to be dropped between each epoch is again a hyperparameter.

Neural Networks in R

# Introduction

In the previous sessions, you learnt artificial, convolutional and recurrent neural networks. You also learnt about the various hyperparameters of artificial neural networks (ANNs) and how is the training carried out to find the optimal combination of weights and biases. In this session, you will implement neural networks in R using the library "h2o" and will create an artificial neural network to**recognise handwritten digits** using the**MNIST dataset.**

## In this Session:

* Data Understanding and Preparation
* Modelling in R

# Data Understanding and Preparation

In this segment, you will build a neural network to classify handwritten digits between 0-9 taken from various government forms, banks, offices etc. The objective is to correctly identify digits using images as inputs.

The analysis is performed on a small sample of the MNIST dataset as the original dataset is very huge and the analysis on the full dataset would be very time-consuming.

# Modelling

In this segment, you will understand how to implement neural networks in R using "h20.deeplearning" from the package h2o. You will also learn about the hyperparameters you need to define to implement neural networks.

In this segment, you learnt that the h2o.deeplearning function automatically splits the data into training and validation sets. The distribution used is multinomial because it is a **multiclass classification.**Y

You can choose the **activation function**and can experiment with the number of **hidden layers, the number of neurons**in each layer, the **dropout ratio**and the **number of epochs,** all of which are hyperparameters. You do not need to define the batch size as the "h2o.deeplearning" function finds the optimal batch size itself. You finally check the accuracy of the model on both the training and validation data.

* Associate every layer of the neural network other than the input and the output layer with one level of abstraction. Each layer will try to capture, understand and output one feature or layer of abstraction correctly.