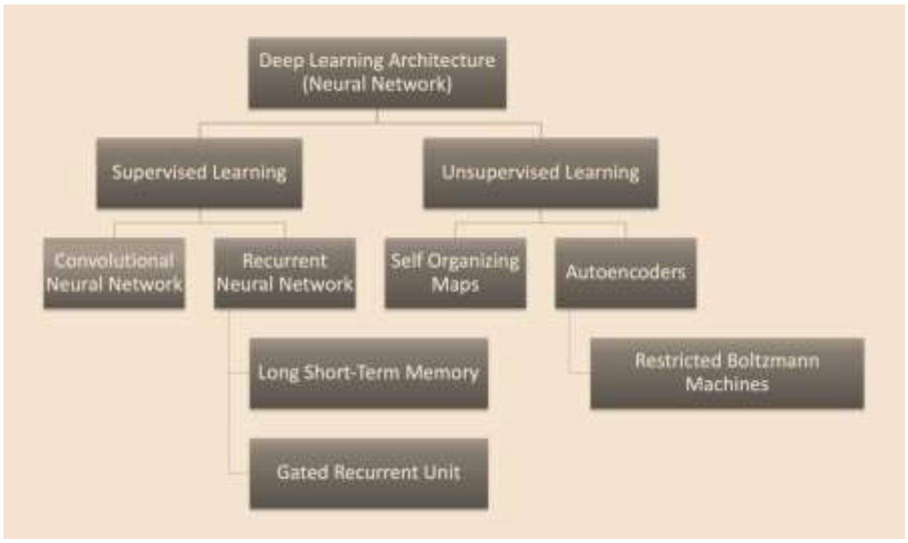
1) What are Deep learning architectures? And give brief note on CNN and its operation?

A) The number of architectures and algorithms that are used in deep learning is wide and varied. This section explores six of the deep learning architectures spanning the past 20 years. Notably, long short-term memory (LSTM) and convolutional neural networks (CNNs) are two of the oldest approaches in this list but also two of the most used in various applications.

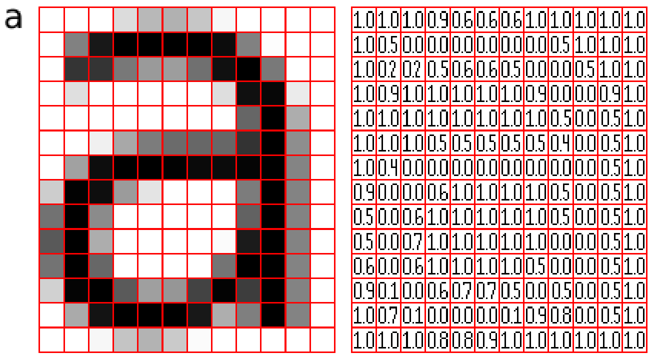


<u>Supervised deep learning:</u> Supervised learning refers to the problem space wherein the target to be predicted is clearly labelled within the data that is used for training. The most popular supervised deep learning architectures are

- Convolutional neural networks
- Recurrent neural networks.

<u>Convolutional neural networks</u>: Convolutional networks (LeCun, 1989), also known as convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. The first CNN was created by Yann LeCun; The architecture is particularly useful in image-processing applications. at the time, the architecture focused on handwritten character recognition, such as postal code interpretation.

A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be.

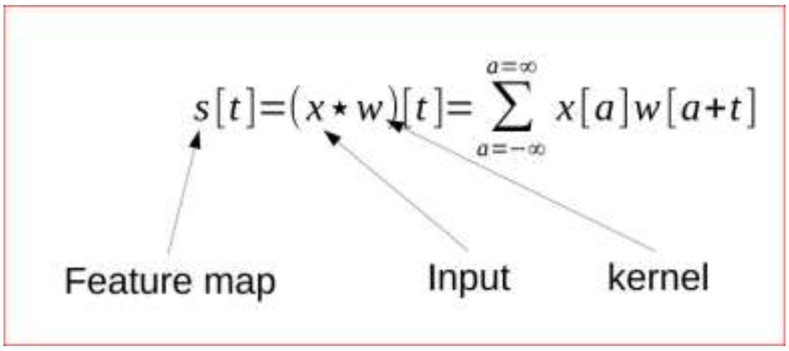


1.0	1.0	1.0	0.9	0.6	0.6	0.6	1.0	1.0	1.0	1.0	1.0
1.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	1.0	1.0
1.0	0.2	0.2	0.5	0.6	0.6	0.5	0.0	0.0	0.5	1.0	1.0
1.0	0.9	1.0	1.0	1.0	1.0	1.0	0.9	0.0	0.0	0.9	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.5	0.4	0.0	0.5	1.0
1.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0
0.9	0.0	0.0	0.6	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
0.5	0.0	0.6	1.0	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
0.5	0.0	0.7	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.5	1.0
0.6	0.0	0.6	1.0	1.0	1.0	1.0	0.5	0.0	0.0	0.5	1.0
0.9	0.1	0.0	0.6	0.7	0.7	0.5	0.0	0.5	0.0	0.5	1.0
1.0	0.7	0.1	0.0	0.0	0.0	0.1	0.9	0.8	0.0	0.5	1.0
1.0	1.0	1.0	0.8	0.8	0.9	1.0	1.0	1.0	1.0	1.0	1.0

Figure 1: Representation of image as a grid of pixels

The human brain processes a huge amount of information the second we see an image. Each neuron works in its own receptive field and is connected to other neurons in a way that they cover the entire visual field. Just as each neuron responds to stimuli only in the restricted region of the visual field called the receptive field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well. The layers are arranged in such a way so that they detect simpler patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along. By using a CNN, one can enable sight to computers.

<u>Convolutional operation:</u> Convolution is a mathematical operation where we have an input I, and an argument, kernel K to produce an output that expresses how the shape of one is modified by another. For example, We have an image "x", which is a 2D array of pixels with different color channels (Red,Green and Blue-RGB) and we have a **feature detector or kernel "w"** then the output we get after applying a mathematical operation is called a **feature map**



Convolution function

The mathematical operation helps compute similarity of two signals. we may have a feature detector or filter for identifying edges in the image, so convolution operation will help us identify the edges in the image when we use such a filter on the image. we usually assume that convolution functions are zero everywhere but the finite set of points for which we store the values. This means that in practice we can implement the infinite summation as a summation over a finite number of array elements.

$$S(i,j)=(I*K)(i,j)=\sum_m\sum_n I(m,n)K(i-m,j-n)$$

I is 2D array and K is kernel-Convolution function

Since convolution is commutative, we can rewrite the equation pictured above as shown below. We do this for ease of implementation in Machine Learning, as there is less variation in range of valid values for m and n. This is **cross correlation** function which most neural networks use.

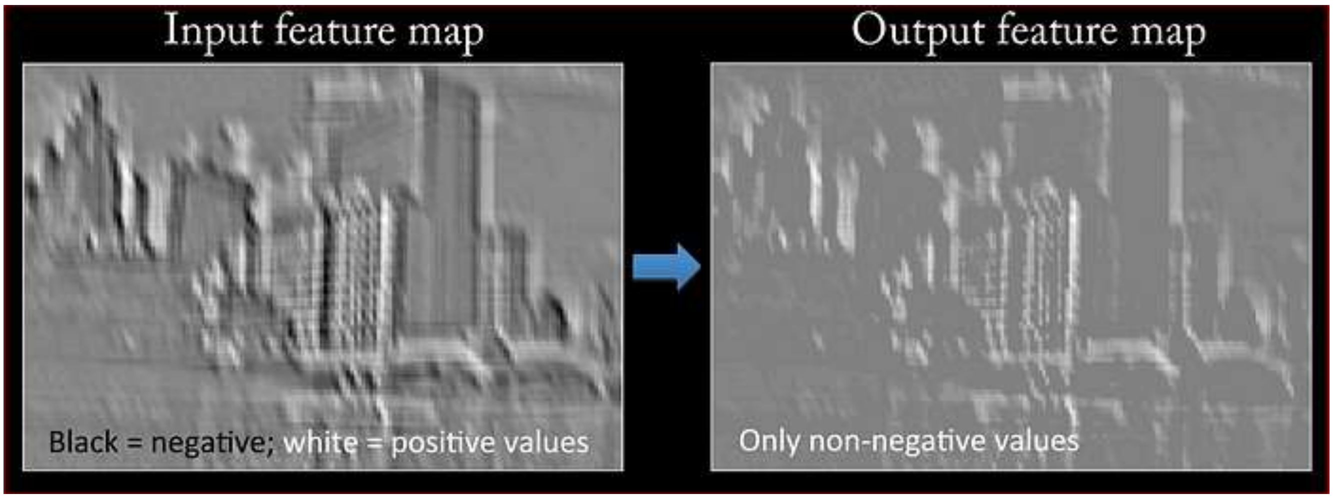
$$S(i,j)=(K*I)(i,j)=\sum_m\sum_n I(i-m,j-n)K(m,n)$$

Cross Correlation function

<u>Implementing in CNN:</u> The way we implement this is through **Convolutional Layer.** Convolutional layer is core building block of CNN, it helps with **feature detection.** Kernel K is a set of learnable filters and is small spatially compared to the image but extends through the full depth of the input image. An easy way to understand this is if you were a detective and you are came across a large image or a picture in dark, you will use you flashlight and scan across the entire image. This is exactly what we do in convolutional layer.

Kernel K, which is a feature detector is equivalent of the flashlight on image I, and we are trying to detect feature and create multiple feature maps to help us identify or classify the image. we have multiple feature detector to help with things like edge detection, identifying different shapes, bends or different colors etc.

After every convolution operation which is a linear function, we apply ReLU activation function. ReLU activation function introduces non linearity in convolutional layer. It replaces all negative pixel values with zero values in the feature map. Below figure shows the feature map transformation after applying the ReLU activation function.



2) Explain about Convolutional architecture?

A) **Convolutional Neural Network Architecture:** A CNN typically has three layers: a convolutional layer, a pooling layer, and a fully connected layer.

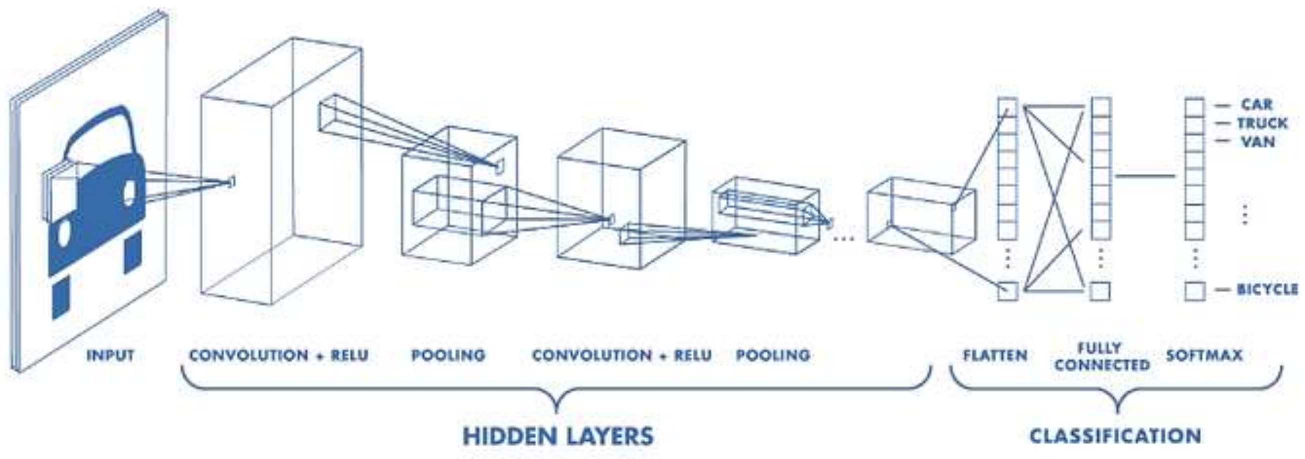


Figure : Architecture of a CNN

Convolution Layer: The convolution layer is the core building block of the CNN. It carries the main portion of the network's computational load. This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field. The kernel is spatially smaller than an image but is more in-depth. This means that, if the image is composed of three (RGB) channels, the kernel height and width will be spatially small, but the depth extends up to all three channels.

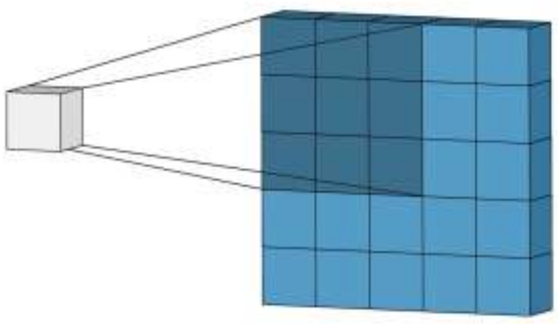


Fig: Illustration of Convolution Operation

During the forward pass, the kernel slides across the height and width of the image-producing the image representation of that receptive region. This produces a two-dimensional representation of the image known as an activation map that gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called a stride.

If we have an input of size W x W x D and Dout number of kernels with a spatial size of F with stride S and amount of padding P, then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F + 2P}{S} + 1$$

Fig: Formula for Convolution Layer

This will yield an output volume of size ***Wout* x *Wou*t x *Dout*.**

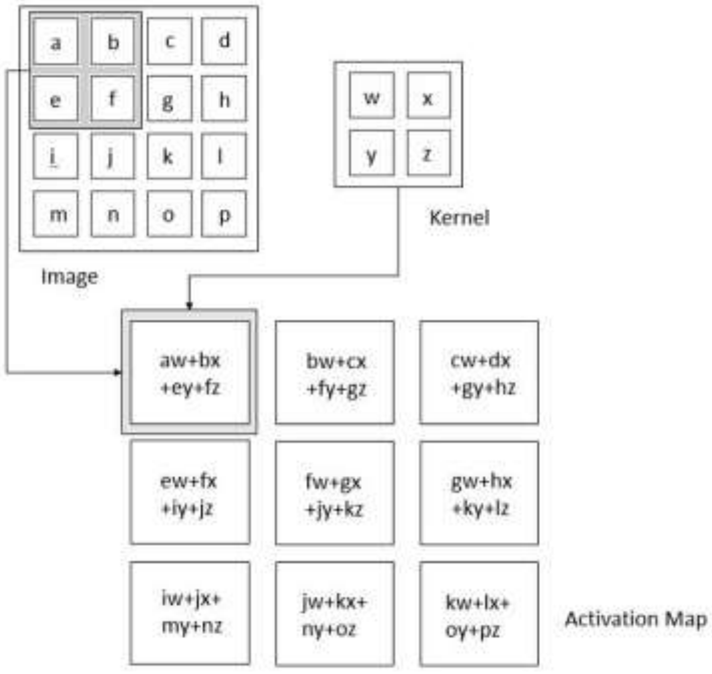


Figure: Convolution Operation

Pooling Layer: The pooling layer replaces the output of the network at certain locations by deriving a summary statistic of the nearby outputs. This helps in reducing the spatial size of the representation, which decreases the required amount of computation and weights. The pooling operation is processed on every slice of the representation individually.

There are several pooling functions such as the average of the rectangular neighborhood, L2 norm of the rectangular neighborhood, and a weighted average based on the distance from the central pixel. However, the most popular process is max pooling, which reports the maximum output from the neighborhood.

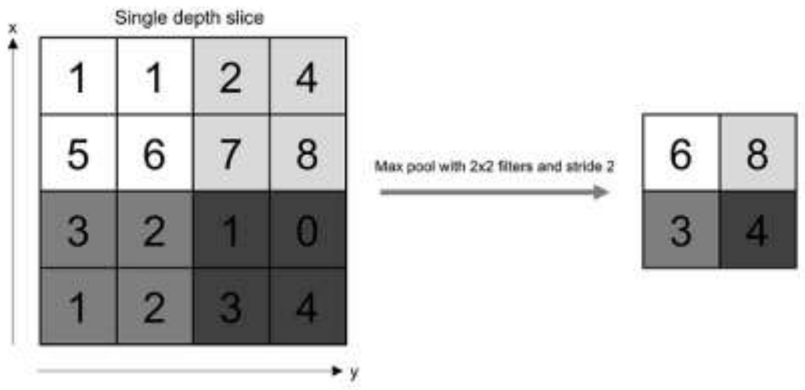


Figure: Pooling Operation

If we have an activation map of size *W* x *W* x *D*, a pooling kernel of spatial size *F*, and stride *S*, then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F}{S} + 1$$

Formula for Padding Layer

This will yield an output volume of size ***Wout* x *Wout* x *D*.** In all cases, pooling provides some translation invariance which means that an object would be recognizable regardless of where it appears on the frame.

Fully Connected Layer: Neurons in this layer have full connectivity with all neurons in the preceding and succeeding layer as seen in regular FCNN. This is why it can be computed as usual by a matrix multiplication followed by a bias effect. The FC layer helps to map the representation between the input and the output.

Since convolution is a linear operation and images are far from linear, non-linearity layers are often placed directly after the convolutional layer to introduce non-linearity to the activation map. There are several types of non-linear operations, the popular ones being:**1. Sigmoid, 2. Tanh, 3. ReLU**

3) Give Brief note on Motivations behind Convolution?

Motivation behind Convolution: Convolution leverages three important ideas that motivated computer vision researchers, they are

- Sparse interactions
- Parameter sharing
- Equivariant representations

Spare interaction: Sparse interaction or sparse weights is implemented by using kernels or feature detector smaller than the input image. If we have an input image of the size 256 by 256 then it becomes difficult to detect edges in the image may occupy only a smaller subset of pixels in the image. If we use smaller feature detectors then we can easily identify the edges as we focus on the local feature identification. one more advantage is computing output requires fewer operations making it statistically efficient.

Parameter Sharing: Parameter Sharing is used to control the number of parameters or weights used in CNN. In traditional neural networks each weight is used exactly once however in CNN we assume that if the one feature detector is useful to compute one spatial position, then it can be used to compute a different spatial position. As we share parameters across the CNN, it reduces the number of parameters to be learnt and also reduces the computational needs.

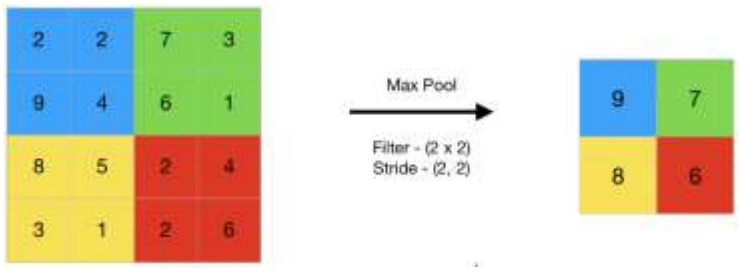
Equivariant representation: It means that object detection is invariant to the changes in illumination, change of position, but internal representation is equivariance to these changes represent(rose) = represent(transform(rose)).

4) Explain Pooling in CNN?

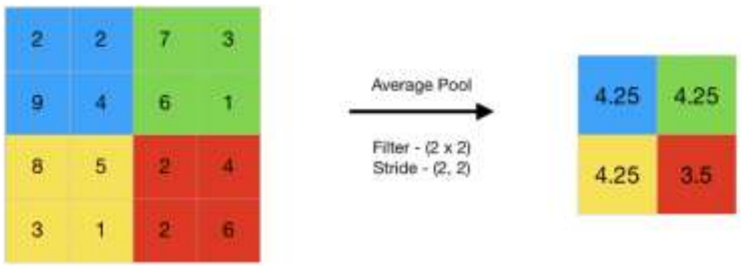
A) In convolutional neural networks (CNNs), the pooling layer is a common type of layer that is typically added after convolutional layers. The pooling layer is used to reduce the spatial dimensions (i.e., the width and height) of the feature maps, while preserving the depth (i.e., the number of channels).

The pooling layer works by dividing the input feature map into a set of non-overlapping regions, called pooling regions. Each pooling region is then transformed into a single output value, which represents the presence of a particular feature in that region. There are four types of pooling layers - **Max, Min, Average, and Global Pooling**. The most common types of pooling operations are max pooling and average pooling.

Max Pooling: In max pooling, the output value for each pooling region is simply the maximum value of the input values within that region. This has the effect of preserving the most salient features in each pooling region, while discarding less relevant information. Max pooling is often used in CNNs for object recognition tasks, as it helps to identify the most distinctive features of an object, such as its edges and corners.



Average Pooling: In average pooling, the output value for each pooling region is the average of the input values within that region. This has the effect of preserving more information than max pooling, but may also dilute the most salient features. Average pooling is often used in CNNs for tasks such as image segmentation and object detection, where a more fine-grained representation of the input is required.



Min Pooling: The Min Pooling layer summarizes the features in a region represented by the minimum value in that region. Contrary to Max Pooling in CNN, this type is mainly used for images with a light background to focus on darker pixels.

Global Pooling: The pooling technique reduces each feature map channel to a single value. This value depends on the type of global pooling, which can be any of the previously explained pooling types. In other words, applying global pooling is similar to using a filter of the exact dimensions of the feature map.

Pooling layers are typically used in conjunction with convolutional layers in a CNN, with each pooling layer reducing the spatial dimensions of the feature maps, while the convolutional layers extract increasingly complex features from the input. The resulting feature maps are then passed to a fully connected layer, which performs the final classification or regression task.

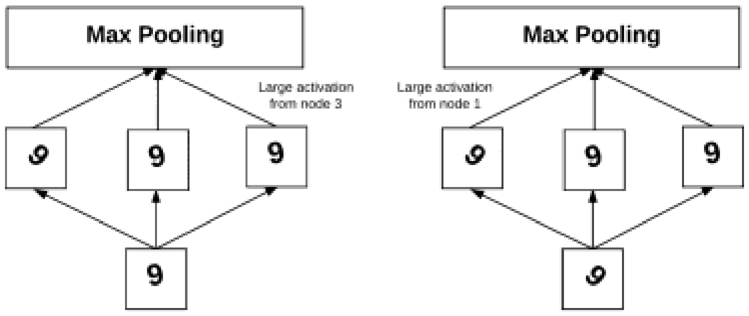
Advantages of Pooling Layer:

1. **Dimensionality reduction**: The main advantage of pooling layers is that they help in reducing the spatial dimensions of the feature maps. This reduces the computational cost and also helps in avoiding overfitting by reducing the number of parameters in the model.
2. **Translation invariance**: Pooling layers are also useful in achieving translation invariance in the feature maps. This means that the position of an object in the image does not affect the classification result, as the same features are detected regardless of the position of the object.
3. **Feature selection**: Pooling layers can also help in selecting the most important features from the input, as max pooling selects the most salient features and average pooling preserves more information.

Disadvantages of Pooling Layer:

1. **Information loss**: One of the main disadvantages of pooling layers is that they discard some information from the input feature maps, which can be important for the final classification or regression task.
2. **Over-smoothing**: Pooling layers can also cause over-smoothing of the feature maps, which can result in the loss of some fine-grained details that are important for the final classification or regression task.
3. **Hyperparameter tuning**: Pooling layers also introduce hyperparameters such as the size of the pooling regions and the stride, which need to be tuned in order to achieve optimal performance. This can be time-consuming and requires some expertise in model building.

Example of using Max pooling to learn invariances in images: A CNN *as a whole* can *learn* filters that fire when a pattern is presented at a particular orientation. For example, consider following **Figure**,



CNN as a whole learns filters that will fire when a pattern is presented at a particular orientation. On the *left*, the digit *9* has been rotated ≈ 10˚. This rotation is similar to node three, which has learned what the digit *9* looks like when rotated in this manner. This node will have a higher activation than the other two nodes — the max pooling operation will detect this. On the *right* we have a second example, only this time the *9* has been rotated ≈ −45˚, causing the first node to have the highest activation

Here, we see the digit *"9"* (bottom) presented to the CNN along with a set of filters the CNN has learned (middle). Since there is a filter inside the CNN that has "learned" what a *"9"* looks like, rotated by 10 degrees, it fires and emits a strong activation. This large activation is captured during the pooling stage and ultimately reported as the final classification.

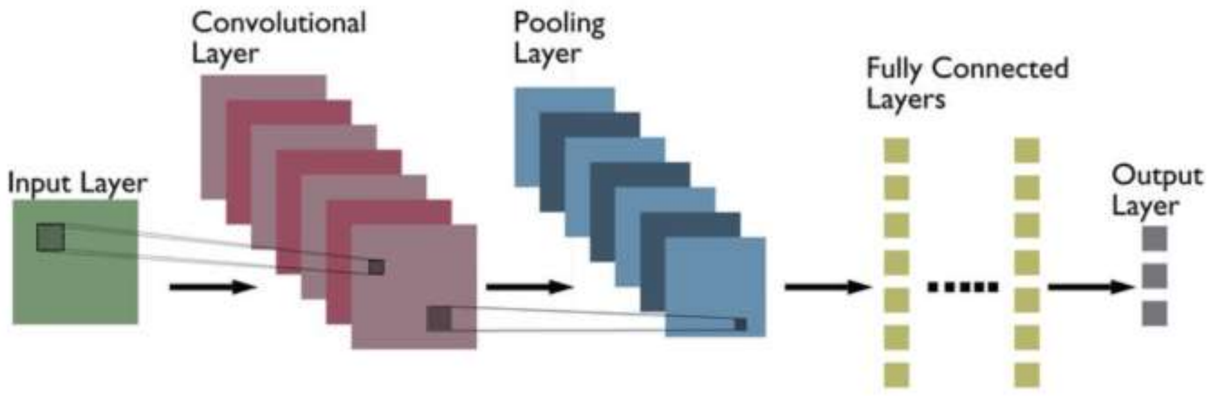
The same is true for the second example (**Figure**, *right*). Here we see the *"9"* rotated by −45 degrees, and since there is a filter in the CNN that has learned what a *"9"* looks like when it is rotated by −45 degrees, the neuron activates and fires. Again, these filters themselves are *not* rotation invariant — it's just that the CNN has learned what a *"9"* looks like under *small rotations* that exist in the training set.

Unless your training data includes digits that are rotated across the full 360-degree spectrum, your CNN is *not* truly rotation invariant.

5) Give the list of various CNN architectures?

A) Convolutional Neural Networks, commonly referred to as CNNs, are a specialized kind of neural network architecture that is designed to process data with a grid-like topology. This makes them particularly well-suited for **dealing with spatial and temporal data**, like **images** and **videos**, that maintain a high degree of correlation between adjacent elements.

CNNs are similar to other neural networks, but they have an added layer of complexity due to the fact that they use a **series of convolutional layers**. Convolutional layers perform a **mathematical operation** called **convolution**, a sort of **specialized matrix multiplication**, on the input data. The convolution operation helps to preserve the spatial relationship between pixels by learning image features using small squares of input data. . The picture below represents a typical CNN architecture

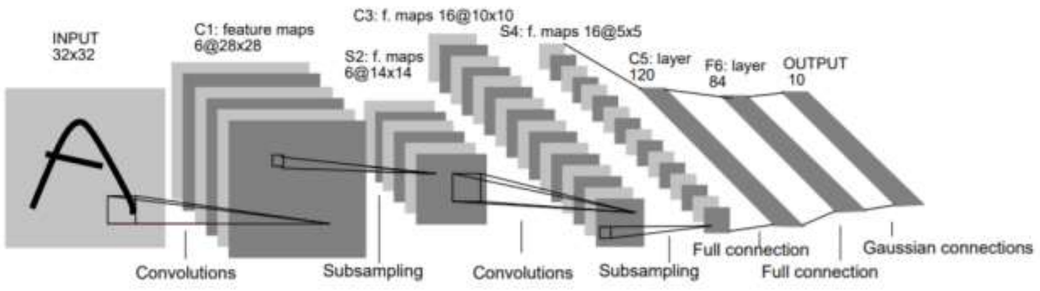


<u>**Different types of CNN Architectures:**</u> The following is a list of different types of CNN architectures:

- **LeNet**
- **AlexNet**
- **ZF Net**
- **GoogLeNet**
- **VGGNet**

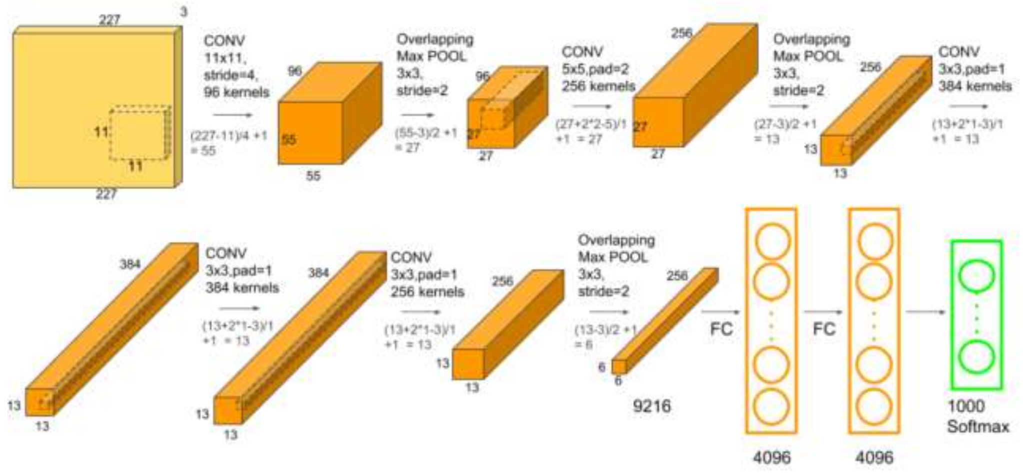
- **ResNet**
- **MobileNets**
- **GoogLeNet_DeepDream**

LeNet: LeNet is the first CNN architecture. It was developed in 1998 by Yann LeCun, Corinna Cortes, and Christopher Burges for handwritten digit recognition problems. LeNet was one of the first successful CNNs and is often considered the "Hello World" of deep learning. It is one of the earliest and most widely-used CNN architectures and has been successfully applied to tasks such as handwritten digit recognition. The LeNet architecture consists of multiple convolutional and pooling layers, followed by a fully-connected layer. The model has five convolution layers followed by two fully connected layers. LeNet was the beginning of CNNs in deep learning for computer vision problems. However, LeNet could not train well due to the vanishing gradients problem. To solve this issue, a shortcut connection layer known as max-pooling is used between convolutional layers to reduce the spatial size of images which helps prevent overfitting and allows CNNs to train more effectively. The diagram below represents LeNet-5 architecture.



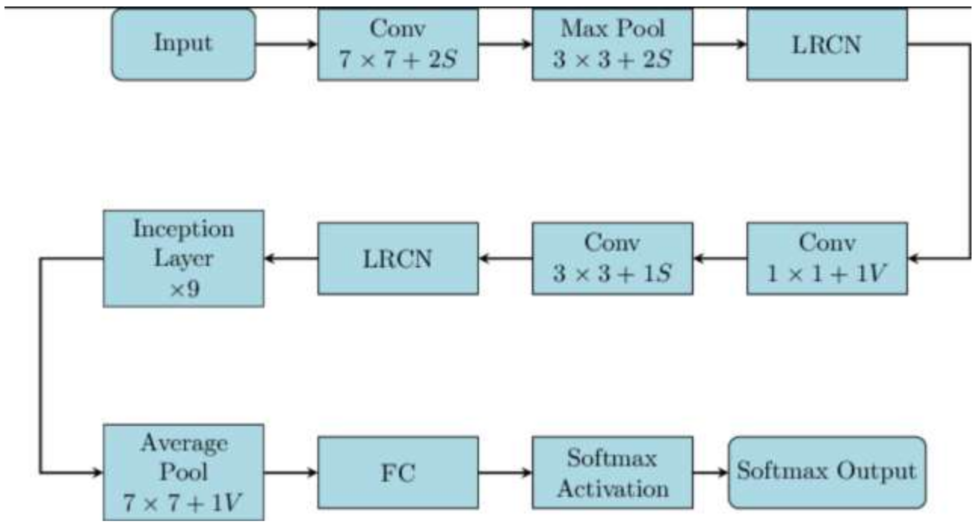
The LeNet CNN is a simple yet powerful model that has been used for various tasks such as handwritten digit recognition, traffic sign recognition, and face detection. Although LeNet was developed more than 20 years ago, its architecture is still relevant today and continues to be used.

AlexNet: AlexNet is the deep learning architecture that popularized CNN. It was developed by Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. AlexNet network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other. AlexNet was the first large-scale CNN and was used to win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. The AlexNet architecture was designed to be used with large-scale image datasets and it achieved state-of-the-art results at the time of its publication. AlexNet is composed of 5 convolutional layers with a combination of max-pooling layers, 3 fully connected layers, and 2 dropout layers. The activation function used in all layers is **Relu.** The activation function used in the output layer is **Softmax**. The total number of parameters in this architecture is around 60 million.

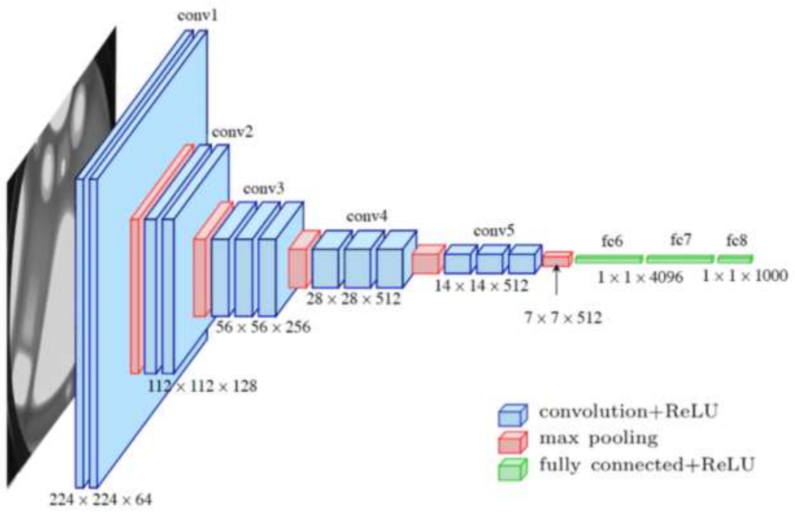

227
3
227
11
11
CONV 11x11, stride=4, 96 kernels
(227-11)/4 +1 = 55
96
55
55
Overlapping Max POOL 3x3, stride=2
(55-3)/2 +1 = 27
96
27
27
CONV 5x5,pad=2 256 kernels
(27+2*2-5)/1 +1 = 27
256
27
27
Overlapping Max POOL 3x3, stride=2
(27-3)/2 +1 = 13
256
13
13
CONV 3x3,pad=1 384 kernels
(13+2*1-3)/1 +1 = 13
384
13
13
CONV 3x3,pad=1 384 kernels
(13+2*1-3)/1 +1 = 13
384
13
13
CONV 3x3,pad=1 256 kernels
(13+2*1-3)/1 +1 = 13
256
13
13
Overlapping Max POOL 3x3, stride=2
(13-3)/2 +1 = 6
256
6
6
9216
FC
4096
FC
4096
1000 Softmax

ZF Net: ZFnet is the CNN architecture that uses a combination of fully-connected layers and CNNs. ZF Net was developed by Matthew Zeiler and Rob Fergus. It was the ILSVRC 2013 winner. The network has relatively fewer parameters than AlexNet, but still outperforms it on ILSVRC 2012 classification task by achieving top accuracy with only 1000 images per class. It was an improvement on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller. It is based on the Zeiler and Fergus model, which was trained on the ImageNet dataset. ZF Net CNN architecture consists of a total of seven layers: Convolutional layer, max-pooling layer (downscaling), concatenation layer, convolutional layer with linear activation function, and stride one, dropout for regularization purposes applied before the fully connected output. This CNN model is computationally more efficient than AlexNet by introducing an approximate inference stage through deconvolutional layers in the middle of CNNs.

GoogLeNet: GoogLeNet is the CNN architecture used by Google to win ILSVRC 2014 classification task. It was developed by Jeff Dean, Christian Szegedy, Alexandro Szegedy et al.. It has been shown to have a notably reduced error rate in comparison with previous winners AlexNet (Ilsvrc 2012 winner) and ZF-Net (Ilsvrc 2013 winner). In terms of error rate, the error is significantly lesser than VGG (2014 runner up). It achieves deeper architecture by employing a number of distinct techniques, including 1×1 convolution and global average pooling. GoogleNet CNN architecture is computationally expensive. To reduce the parameters that must be learned, it uses heavy unpooling layers on top of CNNs to remove spatial redundancy during training and also features shortcut connections between the first two convolutional layers before adding new filters in later CNN layers. Real-world applications/examples of GoogLeNet CNN architecture include Street View House Number (SVHN) digit recognition task, which is often used as a proxy for roadside object detection. Below is the simplified block diagram representing GoogLeNet CNN architecture:



VGGNet: VGGNet is the CNN architecture that was developed by Karen Simonyan, Andrew Zisserman et al. at Oxford University. VGGNet is a 16-layer CNN with up to 95 million parameters and trained on over one billion images (1000 classes). It can take large input images of 224 x 224-pixel size for which it has 4096 convolutional features. CNNs with such large filters are expensive to train and require a lot of data, which is the main reason why CNN architectures like GoogLeNet (AlexNet architecture) work better than VGGNet for most image classification tasks where input images have a size between 100 x 100-pixel and 350 x 350 pixels. Real-world applications/examples of VGGNet CNN architecture include the ILSVRC 2014 classification task, which was also won by GoogleNet CNN architecture. The VGG CNN model is computationally efficient and serves as a strong baseline for many applications in computer vision due to its applicability for numerous tasks including object detection. Its deep feature representations are used across multiple neural network architectures like YOLO, SSD, etc. The diagram below represents the standard VGG16 network architecture diagram:



<u>ResNet</u>: ResNet is the CNN architecture that was developed by Kaiming He et al. to win the ILSVRC 2015 classification task with a top-five error of only 15.43%. The network has 152 layers and over one million parameters, which is considered deep even for CNNs because it would have taken more than 40 days on 32 GPUs to train the network on the ILSVRC 2015 dataset. CNNs are mostly used for image classification tasks with 1000 classes, but ResNet proves that CNNs can also be used successfully to solve natural language processing problems like sentence completion or machine comprehension, where it was used by the Microsoft Research Asia team in 2016 and 2017 respectively. Real-life applications/examples of ResNet CNN architecture include Microsoft's machine comprehension system, which has used CNNs to generate the answers for more than 100k questions in over 20 categories. The CNN architecture ResNet is computationally efficient and can be scaled up or down to match the computational power of GPUs.

<u>MobileNets</u>: MobileNets are CNNs that can be fit on a mobile device to classify images or detect objects with low latency. MobileNets have been developed by Andrew G Trillion et al.. They are usually very small CNN architectures, which makes them easy to run in real-time using embedded devices like smartphones and drones. The architecture is also flexible so it has been tested on CNNs with 100-300 layers and it still works better than other architectures like VGGNet. Real-life examples of MobileNets CNN architecture include CNNs that is built into Android phones to run Google's Mobile Vision API, which can automatically identify labels of popular objects in images.

<u>GoogLeNet_DeepDream</u>: GoogLeNet_DeepDream is a deep dream CNN architecture that was developed by Alexander Mordvintsev, Christopher Olah, etc. It uses the Inception network to generate images based on CNN features. The architecture is often used with the ImageNet dataset to generate psychedelic images or create abstract artworks using human imagination at the ICLR 2017 workshop by David Ha, et al.

6) Give Brief notes on CNN Visualization Techniques?

A) <u>CNN Visualization Techniques:</u> Convolutional Neural Network (CNN) visualizations refer to techniques for visually representing the learned features or patterns captured by different layers of a CNN. These visualizations can help to gain insights into how the network is processing and classifying input images.

There are several types of visualizations for CNNs, including feature map visualization, activation maximization, integrated gradients, saliency maps, etc.

1. **Feature maps** show the output of each filter in a given layer of the network for a particular input image. These visualizations can help to understand the features that the network is detecting at each layer.
2. **Activation maximization** involves generating an image that maximally activates a particular neuron or group of neurons in a given layer of the network. This technique can be used to visualize the features learned by the network at different levels of abstraction.
3. **Integrated Gradients** is a technique used to measure the importance of each input feature (e.g., pixel values) for a given output class of a deep neural network. The technique involves computing the gradient of the output class with respect to the input features, and then integrating this gradient over a path from a baseline input (e.g., an all-zero image) to the actual input image. The resulting integrated gradient map highlights the important input features that contributed to the output class.
4. **Saliency maps** highlight the regions of an input image that are most important for a particular output class. This technique can help to identify which parts of the input image the network is focusing on when making its classification decision.

CNN visualizations are powerful tools for understanding how these complex neural networks are processing and classifying images and can be useful for tasks such as network debugging, feature engineering, and model interpretability.

7) Sequence Modeling and Recurrent Neural Networks?

A) Sequence Models: Sequence models are the machine learning models that input or output sequences of data. Sequential data includes text streams, audio clips, video clips, time-series data and etc. Recurrent Neural Networks (RNNs) is a popular algorithm used in sequence models.

Applications of Sequence Models:

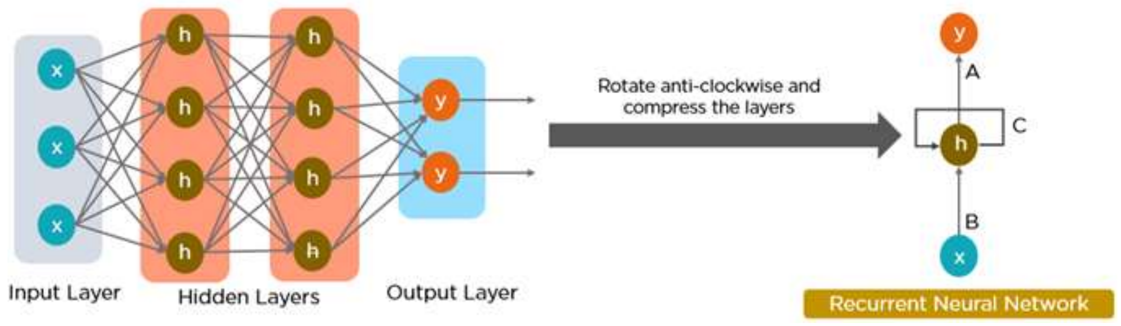
1. Speech recognition: In speech recognition, an audio clip is given as an input and then the model has to generate its text transcript. Here both the input and output are sequences of data.

2. Sentiment Classification: In sentiment classification opinions expressed in a piece of text is categorized. Here the input is a sequence of words.

3. Video Activity Recognition: In video activity recognition, the model needs to identify the activity in a video clip. A video clip is a sequence of video frames, therefore in case of video activity recognition input is a sequence of data.

Recurrent Neural Networks (RNN): Neural networks imitate the function of the human brain in the fields of AI, machine learning, and deep learning, allowing computer programs to recognize patterns and solve common issues.

RNNs are a type of neural network that can be used to model sequence data. RNNs, which are formed from feedforward networks, are similar to human brains in their behavior. Simply said, recurrent neural networks can anticipate sequential data in a way that other algorithms can’t.



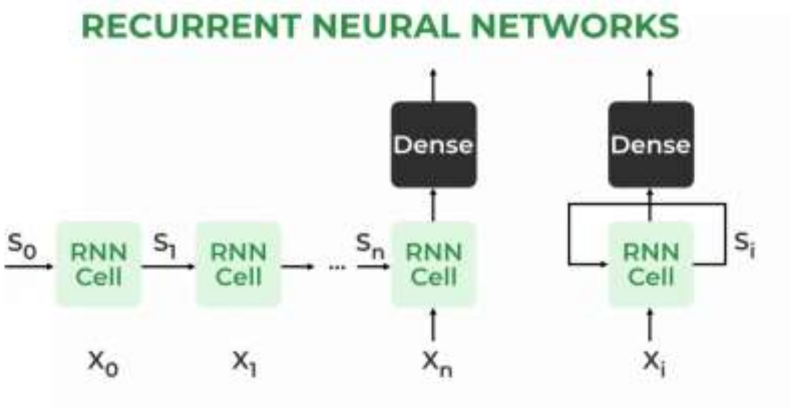
All of the inputs and outputs in standard neural networks are independent of one another, however in some circumstances, such as when predicting the next word of a phrase, the prior words are necessary, and so the previous words must be remembered. As a result, RNN was created, which used a Hidden Layer to overcome the problem. The most important component of RNN is the Hidden state, which remembers specific information about a sequence.

RNNs have a Memory that stores all information about the calculations. It employs the same settings for each input since it produces the same outcome by performing the same task on all inputs or hidden layers.

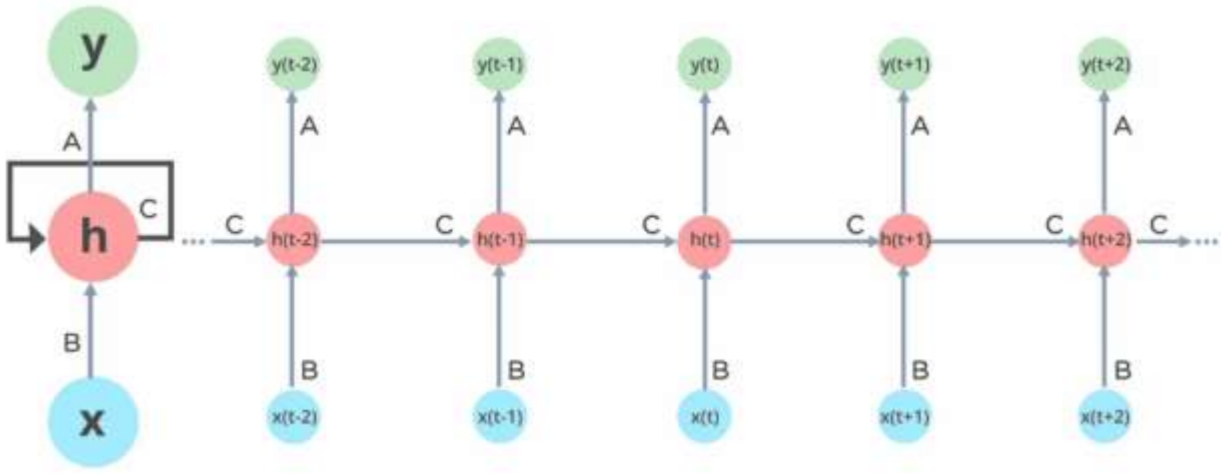
Architecture Of Recurrent Neural Network : RNNs have the same input and output architecture as any other deep neural architecture. However, differences arise in the way information flows from input to output. Unlike Deep neural networks where we have different weight matrices for each Dense network in RNN, the weight across the network remains the same. It calculates state hidden **state** H_i **for every input** X_i. By using the following formulas:

h= σ(UX + Wh$_{-1}$ + B)
Y = O(Vh + C) Hence
Y = f (X, h , W, U, V, B, C)

Here S is the State matrix which has element si as the state of the network at timestep i The parameters in the network are W, U, V, c, b which are shared across timestep



RNN working: In Recurrent Neural networks, the information cycles through a loop to the middle hidden layer.



The input layer ‘x’ takes in the input to the neural network and processes it and passes it onto the middle layer. The middle layer ‘h’ can consist of multiple hidden layers, each with its own activation functions and weights and biases. If you have a neural network where the various parameters of different hidden layers are not affected by the previous layer, ie: the neural network does not have memory, then you can use a recurrent neural network.

The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit. This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past. The hidden state is updated using the following recurrence relation:-

The formula for calculating the current state:

$$h_t = f(h_{t-1}, x_t)$$

where:

h_t -> current state
h_{t-1} -> previous state
x_t -> input state

Formula for applying Activation function(tanh):

$$h_t = \tanh (W_{hh}h_{t-1} + W_{xh}x_t)$$

where:

w_{hh} -> weight at recurrent neuron
w_{xh} -> weight at input neuron

The formula for calculating output:

$$y_t = W_{hy}h_t$$

Y_t -> output

W_{hy} -> weight at output layer

These parameters are updated using Backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as Backpropagation through time.

Training through RNN

1. A single-time step of the input is provided to the network.
2. Then calculate its current state using a set of current input and the previous state.
3. The current ht becomes ht-1 for the next time step.
4. One can go as many time steps according to the problem and join the information from all the previous states.
5. Once all the time steps are completed the final current state is used to calculate the output.
6. The output is then compared to the actual output i.e the target output and the error is generated.
7. The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained using Backpropagation through time.

Advantages of Recurrent Neural Network

1. An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short-Term Memory.
2. Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.

Disadvantages of Recurrent Neural Network

1. Gradient vanishing and exploding problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using tanh or relu as an activation function.

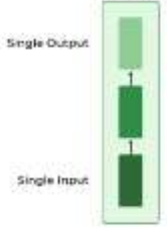
Applications of Recurrent Neural Network

1. Language Modelling and Generating Text
2. Speech Recognition
3. Machine Translation
4. Image Recognition, Face detection
5. Time series Forecasting

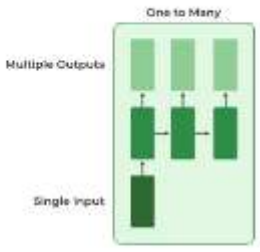
<u>**Types Of RNN**</u>**:** There are four types of RNNs based on the number of inputs and outputs in the network.

1. One to One
2. One to Many
3. Many to One
4. Many to Many

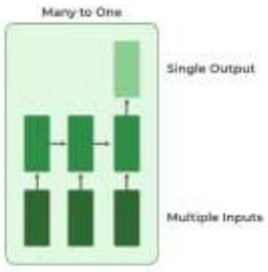
One to One : This type of RNN behaves the same as any simple Neural network it is also known as Vanilla Neural Network. In this Neural network, there is only one input and one output.



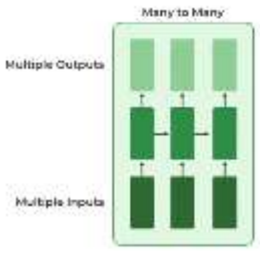
One To Many : In this type of RNN, there is one input and many outputs associated with it. One of the most used examples of this network is Image captioning where given an image we predict a sentence having Multiple words.



Many to One: In this type of network, many inputs are fed to the network at several states of the network generating only one output. This type of network is used in the problems like sentimental analysis. Where we give multiple words as input and predict only the sentiment of the sentence as output.



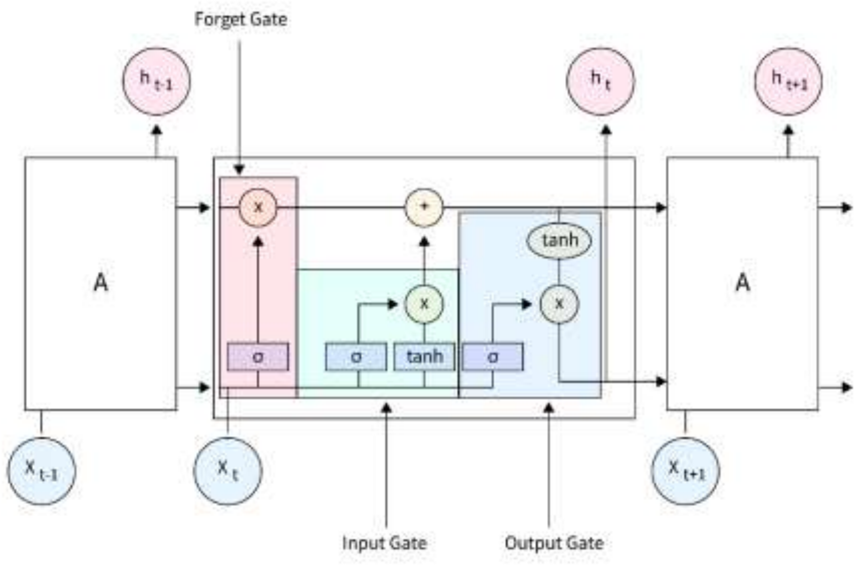
Many to Many : In this type of neural network, there are multiple inputs and multiple outputs corresponding to a problem. One Example of this Problem will be language translation. In language translation, we provide multiple words from one language as input and predict multiple words from the second language as output.



8) Describe about LSTM?

A) Long Short-Term Memory (LSTM): LSTM is a type of Neural Network used in the field of Deep Learning. LSTM stands for **Long-Short-term-Memory**. LSTM is an improved version of the RNN (Recurrent Neural Network). LSTM is mainly used in **Time series** and **Sequence** data because RNN doesn't perform efficiently as the gap length rises. LSTM differs from conventional Feedforward Networks as it uses previous data and its output to affect the current predictions. LSTM is also better at retaining information for longer periods when compared with RNN. Long Short-Term Memory uses Gated Cells to remember or forget previous information.

LSTM was designed by **Hochreiter & Schmidhuber**. It's a challenging task to get your head around LSTM as it belongs to the complex area of Deep Learning. LSTM deals with algorithms to uncover the underlying relationships in the given sequential data. The **chain structure LSTM** contains four neural networks and different memory blocks called **cells**. The LSTM may keep information for a long time by default, and information is retained by the cells, and the three gates do the memory manipulations.



Need of LSTM: LSTM was introduced to tackle the problems and challenges in Recurrent Neural Networks. It touches on the topic of RNN. RNN is a type of Neural Network that stores the previous output to help improve its future predictions. Vanilla RNN has a "short-term" memory. The input at the beginning of the sequence doesn't affect the output of the Network after a while, maybe 3 or 4 inputs. This is called a **long-term dependency issue**.

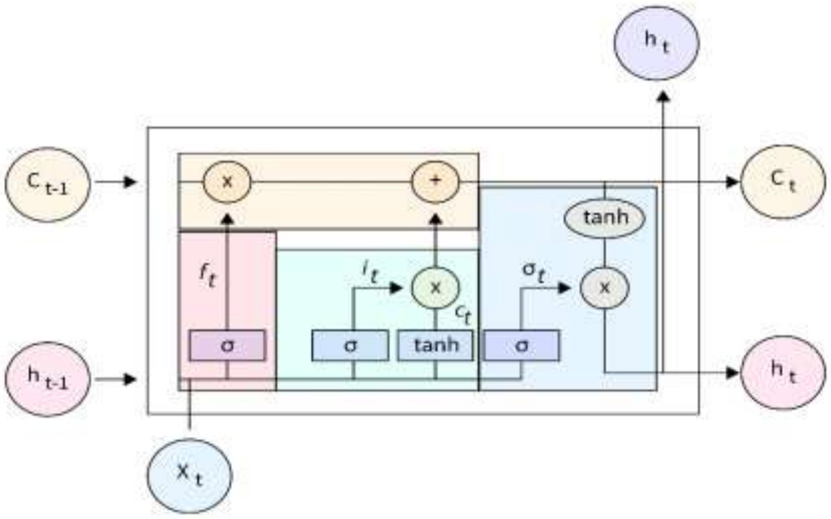
Example:
Let's take this sentence.
The Sun rises in the ______.
An RNN could easily return the correct output that the sun rises in the East as all the necessary information is nearby.

Example2:
I was born in Japan, ……… and I speak fluent ______.

In this sentence, the RNN would be unable to return the correct output as it requires remembering the word Japan for a long duration. Since RNN only has a "Short-term" memory, it doesn't work well. LSTM solves this problem by enabling the Network to remember **Long-term dependencies**.

The other RNN problems are the **Vanishing Gradient** and **Exploding Gradient**. It arises during the Backpropagation of the Neural Network. For example, suppose the gradient of each layer is contained between 0 and 1. As the value gets multiplied in each layer, it gets smaller and smaller, ultimately, a value very close to 0. This is the Vanishing gradient problem. The converse, when the values are greater than 1, exploding gradient problem occurs, where the value gets really big, disrupting the training of the Network. Again, these problems are tackled in LSTMs.

Structure of LSTM: LSTM is a cell that consists of 3 gates. A **forget gate**, **input gate**, and **output gate**. The gates decide which information is important and which information can be forgotten. The cell has two states **Cell State** and **Hidden State**. They are continuously updated and carry the information from the previous to the current time steps. The cell state is the “long-term” memory, while the hidden state is the “short-term” memory.



Forget Gate: Forget gate is responsible for deciding what information should be **removed from the cell state**. It takes in the hidden state of the previous time-step and the current input and passes it to a **Sigma** Activation Function, which outputs a value between 0 and 1, where 0 means forget and 1 means keep.

Forget Gate

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f$$

Input Gate: The Input Gate considers the current input and the hidden state of the previous time step. The input gate is used to **update the cell state** value. It has two parts. The first part contains the **Sigma activation** function. Its purpose is to decide what percent of the information is required. The second part passes the two values to a **Tanh activation** function. It aims to map the data between -1 and 1. To obtain the relevant information required from the output of Tanh, we multiply it by the output of the Sigma function. This is the output of the Input gate, which updates the cell state.

Input Gate

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$C_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_f)$$

Output Gate: The output gate returns the hidden state for the next time stamp. The output gate has two parts. The first part is a **Sigma function**, which serves the same purpose as the other two gates, to decide the percent of the relevant information required. Next, the newly updated cell state is passed through a **Tanh function** and multiplied by the output from the sigma function. This is now the **new hidden state**.

Output Gate

$$o_t = \sigma(W_o \cdot [h_{t-1} \cdot x_t] + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

Cell State: The forget gate and input gate update the cell state. The cell state of the previous state is multiplied by the output of the forget gate. The output of this state is then summed with the output of the input gate. This value is then used to calculate **hidden state** in the output gate.

Cell State

$$C_t = f_t * C_{t-1} \cdot i_t * \bar{C}_t$$

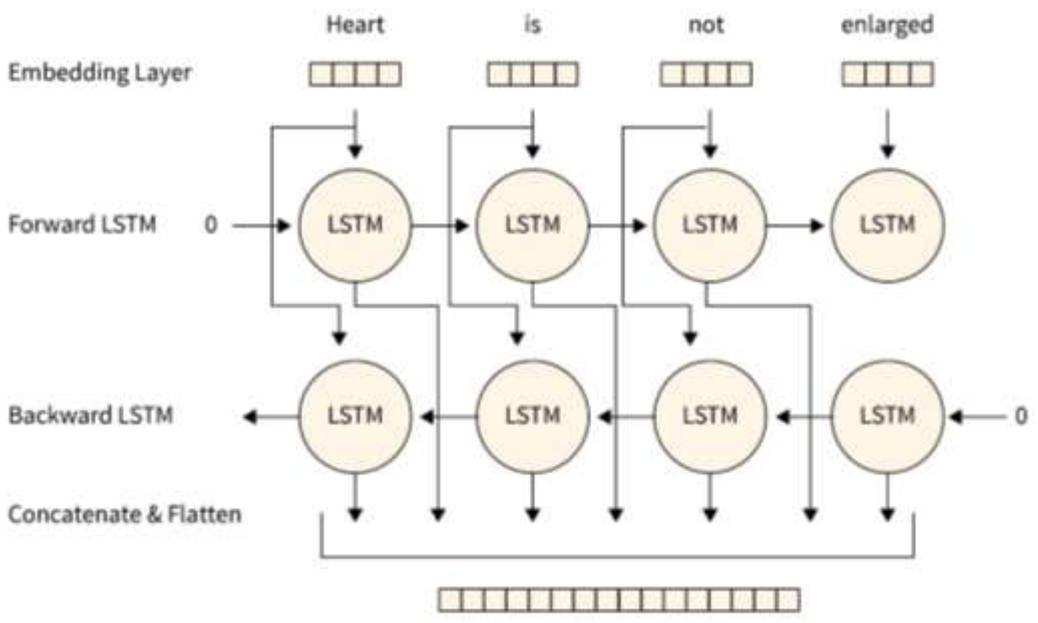
LSTM Working: The LSTM architecture is similar to RNN, but instead of the feedback loop has an LSTM cell. The sequence of LSTM cells in each layer is fed with the **output of the last cell**. This enables the cell to get the previous inputs and sequence information. A cyclic set of steps happens in each LSTM cell

- The Forget gate is computed.
- The Input gate value is computed.
- The Cell state is updated using the above two outputs.
- The output (hidden state) is computed using the output gate.

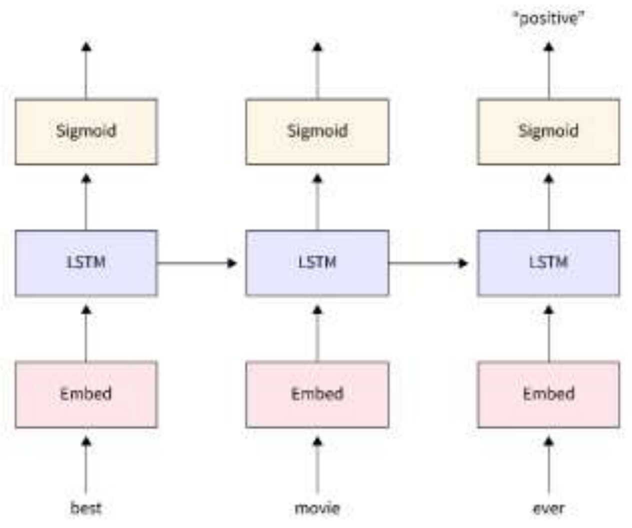
These series of steps occur in every LSTM cell. The intuition behind LSTM is that the Cell and Hidden states carry the previous information and pass it on to future time steps. The Cell state is **aggregated** with all the past data information and is the **long-term** information retainer. The Hidden state carries the output of the last cell, i.e. **short-term memory**. This combination of Long term and short-term memory techniques enables LSTM's to perform well In time series and sequence data.

Applications of LSTM:

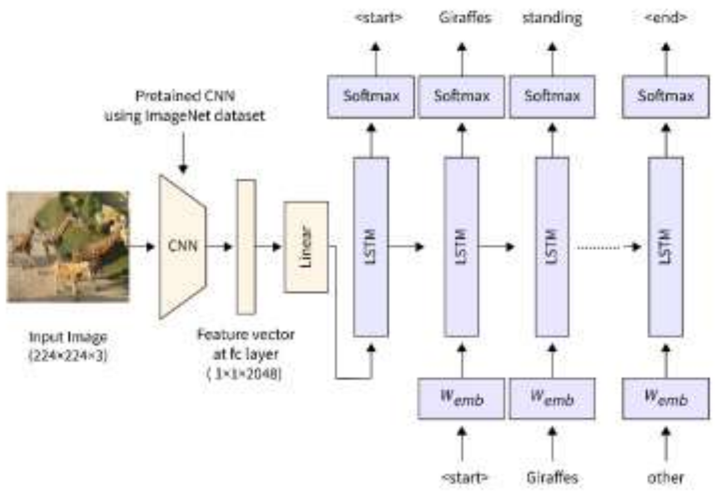
- **Language Modeling:** LSTMs have been used to build language models that can generate natural language text, such as in machine translation systems or chatbots.



- **Time series prediction:** LSTMs have been used to model time series data and predict future values in the series. For example, LSTMs have been used to predict stock prices or traffic patterns.
- **Sentiment analysis:** LSTMs have been used to analyze text sentiments, such as in social media posts or customer reviews.



- **Speech recognition:** LSTMs have been used to build speech recognition systems that can transcribe spoken language into text.
- **Image captioning:** LSTMs have been used to generate descriptive captions for images, such as in image search engines or automated image annotation systems.



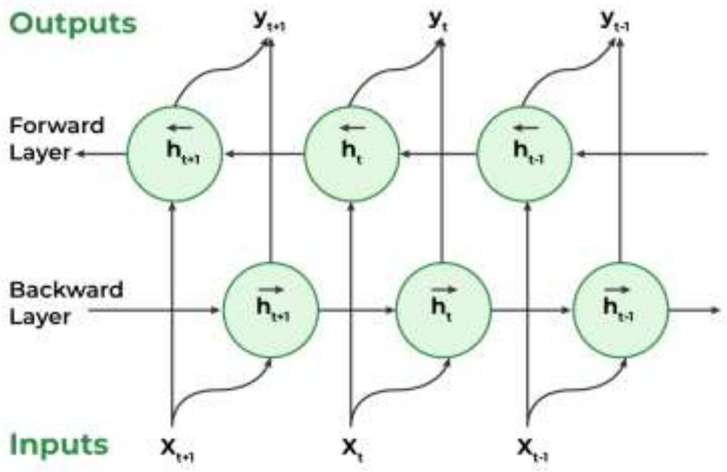
9) Describe about Bidirectional RNN and Bidirectional LSTM?

A) **Bi-directional Recurrent Neural Network:** An architecture of a neural network called a bidirectional recurrent neural network (BRNN) is made to process sequential data. In order for the network to use information from both the past and future context in its predictions, BRNNs process input sequences in both the forward and backward directions. This is the main distinction between BRNNs and conventional recurrent neural networks.

A BRNN has two distinct recurrent hidden layers, one of which processes the input sequence forward and the other of which processes it backward. After that, the results from these hidden layers are collected and input into a prediction-making final layer. Any recurrent neural network cell, such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit, can be used to create the recurrent hidden layers.

The BRNN functions similarly to conventional recurrent neural networks in the forward direction, updating the hidden state depending on the current input and the prior hidden state at each time step. The backward hidden layer, on the other hand, analyses the input sequence in the opposite manner, updating the hidden state based on the current input and the hidden state of the next time step.

In order to update the model parameters, the gradients are computed for both the forward and backward passes of the backpropagation through the time technique that is typically used to train BRNNs. The input sequence is processed by the BRNN in a single forward pass at inference time, and predictions are made based on the combined outputs of the two hidden layers. layers.



Working of Bidirectional Recurrent Neural Network

1. **Inputting a sequence:** A sequence of data points, each represented as a vector with the same dimensionality, are fed into a BRNN. The sequence might have different lengths.
2. **Dual Processing:** Both the forward and backward directions are used to process the data. On the basis of the input at that step and the hidden state at step t-1, the hidden state at time step t is determined in the forward direction. The input at step t and the hidden state at step t+1 are used to calculate the hidden state at step t in a reverse way.
3. **Computing the hidden state:** A non-linear activation function on the weighted sum of the input and previous hidden state is used to calculate the hidden state at each step. This creates a memory mechanism that enables the network to remember data from earlier steps in the process.
4. **Determining the output:** A non-linear activation function is used to determine the output at each step from the weighted sum of the hidden state and a number of output weights. This output has two options: it can be the final output or input for another layer in the network.
5. **Training:** The network is trained through a supervised learning approach where the goal is to minimize the discrepancy between the predicted output and the actual output. The network adjusts its weights in the input-to-hidden and hidden-to-output connections during training through backpropagation.

To calculate the output from an RNN unit, we use the following formula:

H_t (Forward) = A(X_t * W_{XH} (forward) + H_{t-1} (Forward) * W_{HH} (Forward) + b_H (Forward)
H_t (Backward) = A(X_t * W_{XH} (Backward) + H_{t+1} (Backward) * W_{HH} (Backward) + b_H (Backward)

where,
A = activation function,
W = weight matrix
b = bias

The hidden state at time t is given by a combination of H_t (Forward) and H_t (Backward). The output at any given hidden state is :

$$\mathbf{Y_t = H_t * W_{AY} + b_y}$$

The training of a BRNN is similar to backpropagation through a time algorithm. BPTT algorithm works as follows:

- Roll out the network and calculate errors at each iteration
- Update weights and roll up the network.

However, because forward and backward passes in a BRNN occur simultaneously, updating the weights for the two processes may occur at the same time. This produces inaccurate outcomes. Thus, the following approach is used to train a BRNN to accommodate forward and backward passes individually.

Applications of Bidirectional Recurrent Neural Network

Bi-RNNs have been applied to various natural language processing (NLP) tasks, including:

1. **Sentiment Analysis:** By taking into account both the prior and subsequent context, BRNNs can be utilized to categorize the sentiment of a particular sentence.
2. **Named Entity Recognition:** By considering the context both before and after the stated thing, BRNNs can be utilized to identify those entities in a sentence.
3. **Part-of-Speech Tagging:** The classification of words in a phrase into their corresponding parts of speech, such as nouns, verbs, adjectives, etc., can be done using BRNNs.
4. **Machine Translation:** BRNNs can be used in encoder-decoder models for machine translation, where the decoder creates the target sentence and the encoder analyses the source sentence in both directions to capture its context.
5. **Speech Recognition:** When the input voice signal is processed in both directions to capture the contextual information, BRNNs can be used in automatic speech recognition systems.

Advantages of Bidirectional RNN

- **Context from both past and future:** With the ability to process sequential input both forward and backward, BRNNs provide a thorough grasp of the full context of a sequence. Because of this, BRNNs are effective at tasks like sentiment analysis and speech recognition.
- **Enhanced accuracy:** BRNNs frequently yield more precise answers since they take both historical and upcoming data into account.
- **Efficient handling of variable-length sequences:** When compared to conventional RNNs, which require padding to have a constant length, BRNNs are better equipped to handle variable-length sequences.

- **Resilience to noise and irrelevant information:** BRNNs may be resistant to noise and irrelevant data that are present in the data. This is so because both the forward and backward paths offer useful information that supports the predictions made by the network.
- **Ability to handle sequential dependencies:** BRNNs can capture long-term links between sequence pieces, making them extremely adept at handling complicated sequential dependencies.

Disadvantages of Bidirectional RNN

- **Computational complexity:** Given that they analyze data both forward and backward, BRNNs can be computationally expensive due to the increased amount of calculations needed.
- **Long training time**: BRNNs can also take a while to train because there are many parameters to optimize, especially when using huge datasets.
- **Difficulty in parallelization:** Due to the requirement for sequential processing in both the forward and backward directions, BRNNs can be challenging to parallelize.
- **Overfitting:** BRNNs are prone to overfitting since they include many parameters that might result in too complicated models, especially when trained on short datasets.
- **Interpretability:** Due to the processing of data in both forward and backward directions, BRNNs can be tricky to interpret since it can be difficult to comprehend what the model is doing and how it is producing predictions.

Bidirectional LSTM (BiLSTM): Bidirectional LSTM or BiLSTM is a term used for a sequence model which contains two LSTM layers, one for processing input in the forward direction and the other for processing in the backward direction. It is usually used in NLP-related tasks. The intuition behind this approach is that by processing data in both directions, the model is able to better understand the relationship between sequences (e.g. knowing the following and preceding words in a sentence).

For example, The first statement is “Server can you bring me this dish” and the second statement is “He crashed the server”. In both these statements, the word server has different meanings and this relationship depends on the following and preceding words in the statement. The bidirectional LSTM helps the machine to understand this relationship better than compared with unidirectional LSTM. This ability of BiLSTM makes it a suitable architecture for tasks like **sentiment analysis, text classification, and machine translation**.

Architecture: The architecture of bidirectional LSTM comprises of two unidirectional LSTMs which process the sequence in both forward and backward directions. This architecture can be interpreted as having two separate LSTM networks, one gets the sequence of tokens as it is while the other gets in the reverse order. Both of these LSTM network returns a probability vector as output and the final output is the combination of both of these probabilities. It can be represented as:

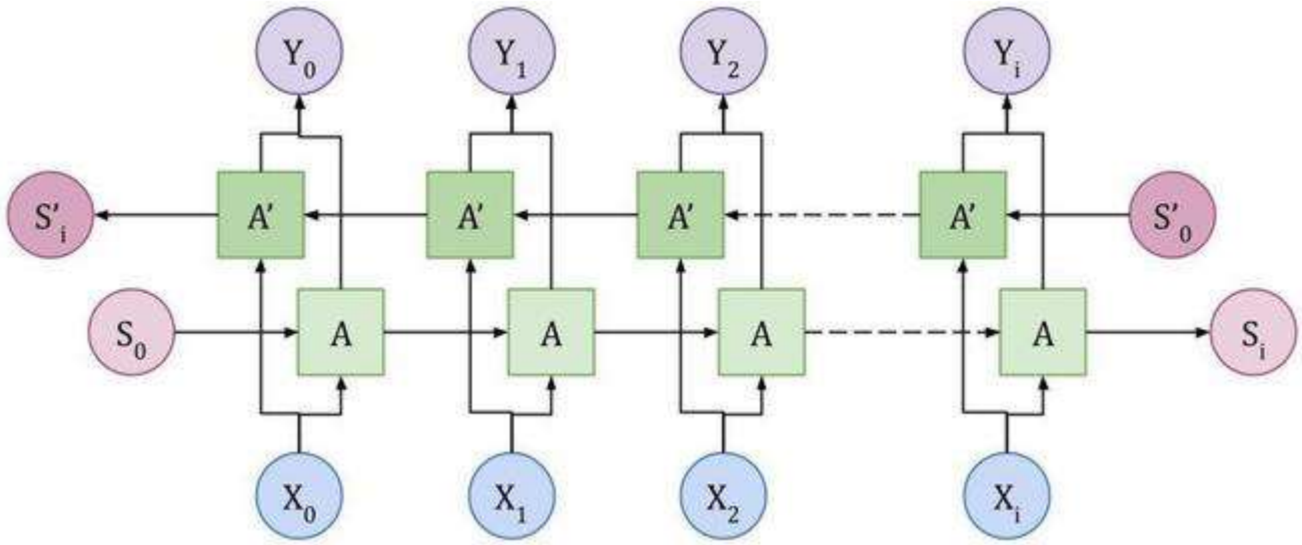
$$++p_t = p_t^f + p_t^b$$

where,

p_t : Final probability vector of the network.

p_t^f : Probability vector from the forward LSTM network.

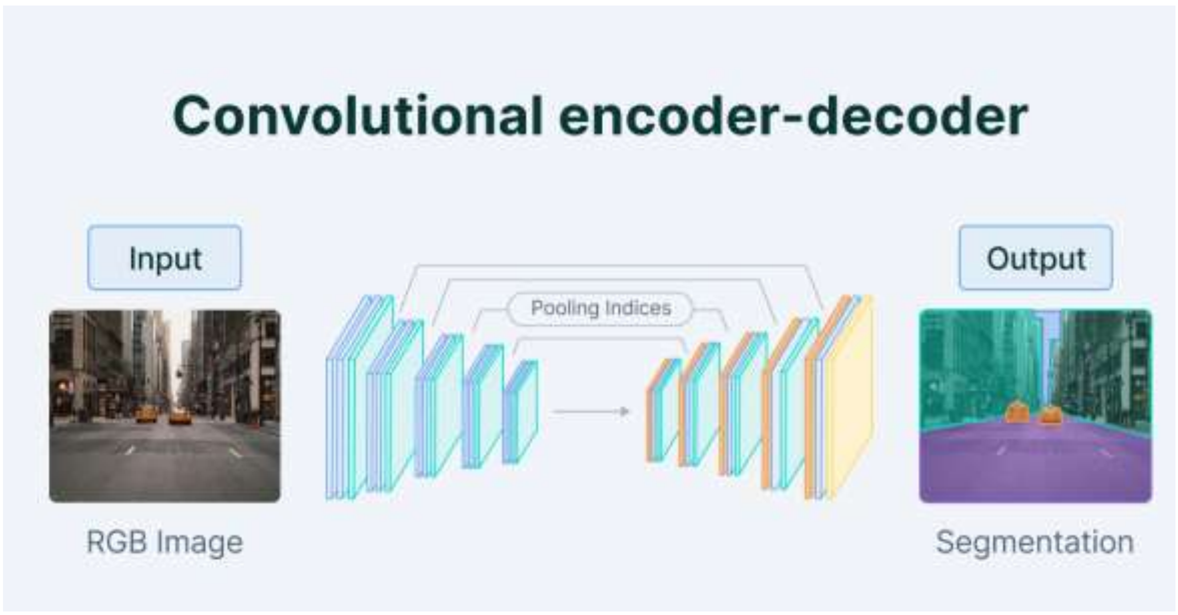
- : Probability vector from the backward LSTM network.



where X_i is the input token, Y_i is the output token, and A and A' are LSTM nodes. The final output of $Y_i$is the combination of A and A' LSTM nodes.

10) What is an autoencoder?

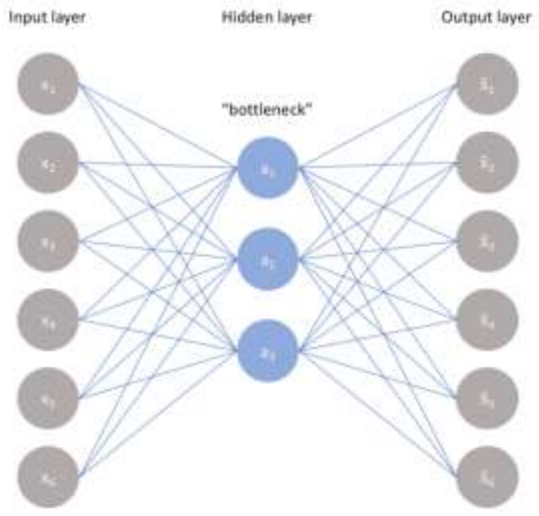
An autoencoder is a type of artificial neural network used to learn data encodings in an unsupervised manner. The aim of an autoencoder is to learn a lower-dimensional representation (encoding) for a higher-dimensional data, typically for dimensionality reduction, by training the network to capture the most important parts of the input image.



Autoencoders are self-supervised machine learning models which are used to reduce the size of input data by recreating it. These models are trained as supervised machine learning models and during inference, they work as unsupervised models that's why they are called self-supervised models. **Autoencoder is made up of 3 components namely,**

1. **Encoder**: A module that compresses the train-validate-test set input data into an encoded representation that is typically several orders of magnitude smaller than the input data.
2. **Bottleneck:** A module that contains the compressed knowledge representations and is therefore the most important part of the network.
3. **Decoder:** A module that helps the network "decompress" the knowledge representations and reconstructs the data back from its encoded form. The output is then compared with a ground truth.

The architecture as a whole look something like this:



Training autoencoders: we need to set 4 hyperparameters before *training* an autoencoder:

1. **Code size:** The code size or the size of the bottleneck is the most important hyperparameter used to tune the autoencoder. The bottleneck size decides how much the data has to be compressed. This can also act as a regularization term.
2. **Number of layers:** Like all neural networks, an important hyperparameter to tune autoencoders is the depth of the encoder and the decoder. While a higher depth increases model complexity, a lower depth is faster to process.
3. **Number of nodes per layer:** The number of nodes per layer defines the weights we use per layer. Typically, the number of nodes decreases with each subsequent layer in the autoencoder as the input to each of these layers becomes smaller across the layers.
4. **Reconstruction Loss:** The loss function we use to train the autoencoder is highly dependent on the type of input and output we want the autoencoder to adapt to. If we are working with image data, the most popular loss functions for reconstruction are MSE Loss and L1 Loss. In case the inputs and outputs are within the range [0,1], as in MNIST, we can also make use of Binary Cross Entropy as the reconstruction loss.

Types of Autoencoders

- **Under Complete Autoencoders:** Under complete autoencoders is an unsupervised neural network that you can use to generate a compressed version of the input data. It is done by taking in an image and trying to predict the same image as output, thus reconstructing the image from its compressed bottleneck region. The primary use for autoencoders like these is generating a latent space or bottleneck, which forms a compressed substitute of the input data and can be easily decompressed back with the help of the network when needed.

- **Sparse Autoencoders:** Sparse autoencoders are controlled by changing the number of nodes at each hidden layer. Since it is impossible to design a neural network with a flexible number of nodes at its hidden layers, sparse autoencoders work by penalizing the activation of some neurons in hidden layers. It means that a penalty directly proportional to the number of neurons activated is applied to the loss function. As a means of regularizing the neural network, the sparsity function prevents more neurons from being activated. There are two types of regularizes used:

    1. The L1 Loss method is a general regularizer we can use to add magnitude to the model.

2. The KL-divergence method considers the activations over a collection of samples at once rather than summing them as in the L1 Loss method. We constrain the average activation of each neuron over this collection.

- **Contractive Autoencoders:** The input is passed through a bottleneck in a contractive autoencoder and then reconstructed in the decoder. The bottleneck function is used to learn a representation of the image while passing it through. The contractive autoencoder also has a regularization term to prevent the network from learning the identity function and mapping input into output. To train a model that works along with this constraint, we need to ensure that the derivatives of the hidden layer activations are small concerning the input**.**

- **Denoising Autoencoders**: Have you ever wanted to remove noise from an image but didn't know where to start? If so, then denoising autoencoders are for you! Denoising autoencoders are similar to regular autoencoders in that they take an input and produce an output. However, they differ because they don't have the input image as their ground truth. Instead, they use a noisy version. The loss function usually used with these networks is L2 or L1 loss.

- **Variational Autoencoders:** Variational autoencoders (VAEs) are models that address a specific problem with standard autoencoders. When you train an autoencoder, it learns to represent the input just in a compressed form called the latent space or the bottleneck. However, this latent space formed after training is not necessarily continuous and, in effect, might not be easy to interpolate.

Applications: Autoencoders have various applications like:

- **Anomaly detection**: autoencoders can identify data anomalies using a loss function that penalizes model complexity. It can be helpful for anomaly detection in financial markets, where you can use it to identify unusual activity and predict market trends.
- **Data denoising image and audio:** autoencoders can help clean up noisy pictures or audio files. You can also use them to remove noise from images or audio recordings.
- **Image inpainting**: autoencoders have been used to fill in gaps in images by learning how to reconstruct missing pixels based on surrounding pixels. For example, if you're trying to restore an old photograph that's missing part of its right side, the autoencoder could learn how to fill in the missing details based on what it knows about the rest of the photo.
- **Information retrieval**: autoencoders can be used as content-based image retrieval systems that allow users to search for images based on their content.

11) What is a Boltzmann machine? Briefly explain about Deep Boltzmann machines (DBM)?

A) A Boltzmann machine is an unsupervised deep learning model in which every node is connected to every other node. It is a type of recurrent neural network, and the nodes make binary decisions with some level of bias. These machines are not deterministic deep learning models, they are stochastic or generative deep learning models. They are representations of a system.

A Boltzmann machine has two kinds of nodes

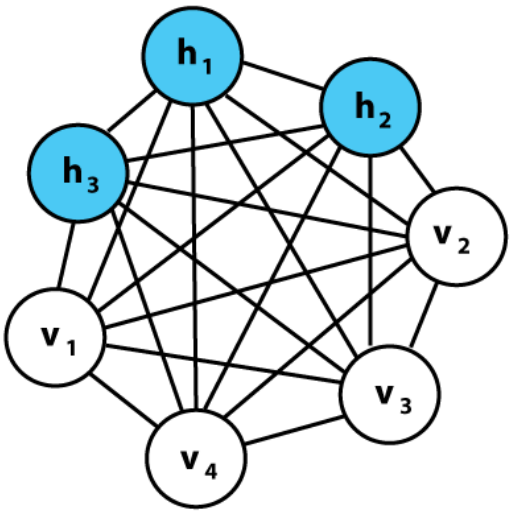
- Visible nodes: These are nodes that can be measured and are measured.
- Hidden nodes: These are nodes that cannot be measured or are not measured.

A Boltzmann machine can be called a stochastic Hopfield network which has hidden units. It has a network of units with an 'energy' defined for the overall network. Boltzmann machines seek to reach thermal equilibrium. It essentially looks to optimize global distribution of energy. But the temperature and energy of the system are relative to laws of thermodynamics and are not literal.

A Boltzmann machine is made up of a learning algorithm that enables it to discover interesting features in datasets composed of binary vectors. The learning algorithm tends to be slow in networks that have many layers of feature detectors but it is possible to make it faster by implementing a learning layer of feature detectors.

They use stochastic binary units to reach probability distribution equilibrium (to minimize energy). It is possible to get multiple Boltzmann machines to collaborate together to form far more sophisticated systems like deep belief networks.

The Boltzmann machine is named after **Ludwig Boltzmann**, an Austrian scientist who came up with the Boltzmann distribution. However, this type of network was first developed by **Geoff Hinton**, a Stanford Scientist.



Working of Boltzmann machine: Boltzmann machines are non-deterministic (stochastic) generative Deep Learning models that only have two kinds of nodes - hidden and visible nodes. They don't have any output nodes, and that's what gives them the non-deterministic feature. They learn patterns without the **typical 1 or 0 type** output through which patterns are learned and optimized using Stochastic Gradient Descent.

A major difference is that unlike other traditional networks (A/C/R) which don't have any connections between the input nodes, Boltzmann Machines have connections among the input nodes. Every node is connected to all other nodes irrespective of whether they are input or hidden nodes. This enables them to share information among themselves and self-generate subsequent data. You'd only measure what's on the visible nodes and not what's on the hidden nodes. After the input is provided, the Boltzmann machines are able to capture all the parameters, patterns and correlations among the data. It is because of this that they are known as deep generative models and they fall into the class of Unsupervised Deep Learning.

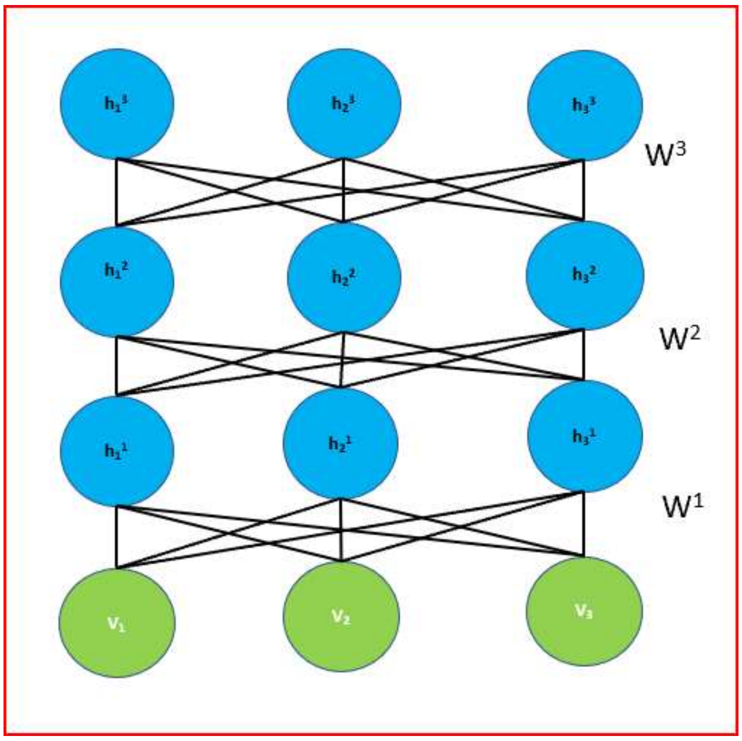
Types of Boltzmann machines: There are three types of Boltzmann machines. These are:

- Restricted Boltzmann Machines (RBMs)
- Deep Belief Networks (DBNs)
- Deep Boltzmann Machines (DBMs)

1. Restricted Boltzmann Machines (RBMs): While in a full Boltzmann machine all the nodes are connected to each other and the connections grow exponentially, an RBM has certain restrictions with respect to node connections. In a Restricted Boltzmann Machine, hidden nodes cannot be connected to each other while visible nodes are connected to each other.

2. Deep Belief Networks (DBNs): In a Deep Belief Network, you could say that multiple Restricted Boltzmann Machines are stacked, such that the outputs of the first RBM are the inputs of the subsequent RBM. The connections within individual layers are undirected, while the connections between layers are directed. However, there is an exception here. The connection between the top two layers is undirected. A deep belief network can either be trained using a Greedy Layer-wise Training Algorithm or a Wake-Sleep Algorithm.

3. Deep Boltzmann Machines (DBMs): Deep Boltzmann Machines are very similar to Deep Belief Networks. The difference between these two types of Boltzmann machines is that while connections between layers in DBNs are directed, in DBMs, the connections within layers, as well as the connections between the layers, are all undirected.



Deep Boltzmann Machines (DBMs) are a type of generative probabilistic model that belong to the family of deep learning architectures. They were introduced as a way to model complex, high-dimensional probability distributions.

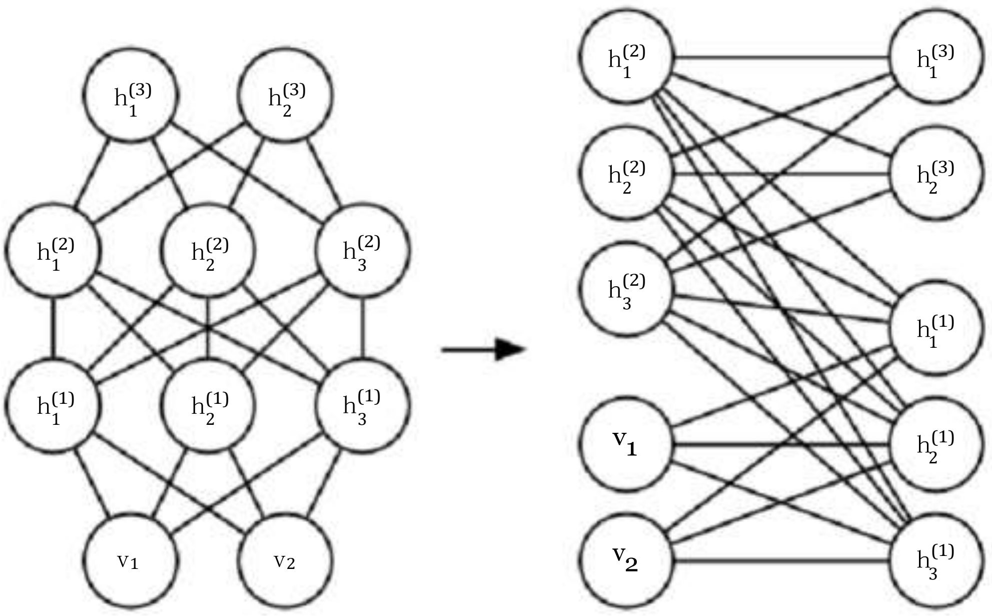


Figure: A deep Boltzmann machine, re-arranged to reveal its bipartite graph structure.

In comparison to the RBM energy function, the DBM energy function includes connections between the hidden units (latent variables) in the form of the weight matrices ($\boldsymbol{W}^{(2)}$ and $\boldsymbol{W}^{(3)}$). As we will see, these connections have significant consequences for both the model behavior as well as how we go about performing inference in the model. The energy function of a Deep Boltzmann Machine (DBM) is a crucial aspect of its probabilistic modeling framework. The energy function is used to define the joint probability distribution over the visible and hidden units in the network. Let's break down the energy function for a standard two-layer DBM, consisting of visible units (v) and two sets of hidden units (h1 and h2). The energy function (E) for a Deep Boltzmann Machine is given by:

$$E(v, h_1, h_2) = -\sum_i \sum_j w_{ij}^{(1)} v_i h_j^{(1)} - \sum_j \sum_k w_{jk}^{(2)} h_j^{(1)} h_k^{(2)} - \sum_i b_i v_i - \sum_j c_j^{(1)} h_j^{(1)} - \sum_k c_k^{(2)} h_k^{(2)}$$

Here's a breakdown of the terms in the energy function:

1. v_i represents the state (binary or real-valued) of visible unit i.
2. $h_j^{(1)}$ and $h_k^{(2)}$ represent the states of hidden units in the first and second hidden layers, respectively.
3. $w_{ij}^{(1)}$ are the weights connecting visible unit i to hidden unit j in the first hidden layer.
4. $w_{jk}^{(2)}$ are the weights connecting hidden unit j in the first hidden layer to hidden unit k in the second hidden layer.
5. b_i, $c_j^{(1)}$, and $c_k^{(2)}$ are biases for the visible units, first hidden layer units, and second hidden layer units, respectively.

The energy function determines the compatibility between different configurations of visible and hidden units. Configurations with lower energy are assigned higher probabilities in the model. The negative exponential of the energy function is used to define the probability distribution over the joint configuration space:

$$P(v, h_1, h_2) = \frac{e^{-E(v,h_1,h_2)}}{Z}$$

Where Z is the partition function, a normalization term calculated as the sum of the exponential of the negative energy over all possible configurations.

$$Z = \sum_{v,h_1,h_2} e^{-E(v,h_1,h_2)}$$

Training a DBM involves adjusting the weights and biases to maximize the likelihood of the training data. This often involves techniques like Contrastive Divergence and layer-wise pre-training.