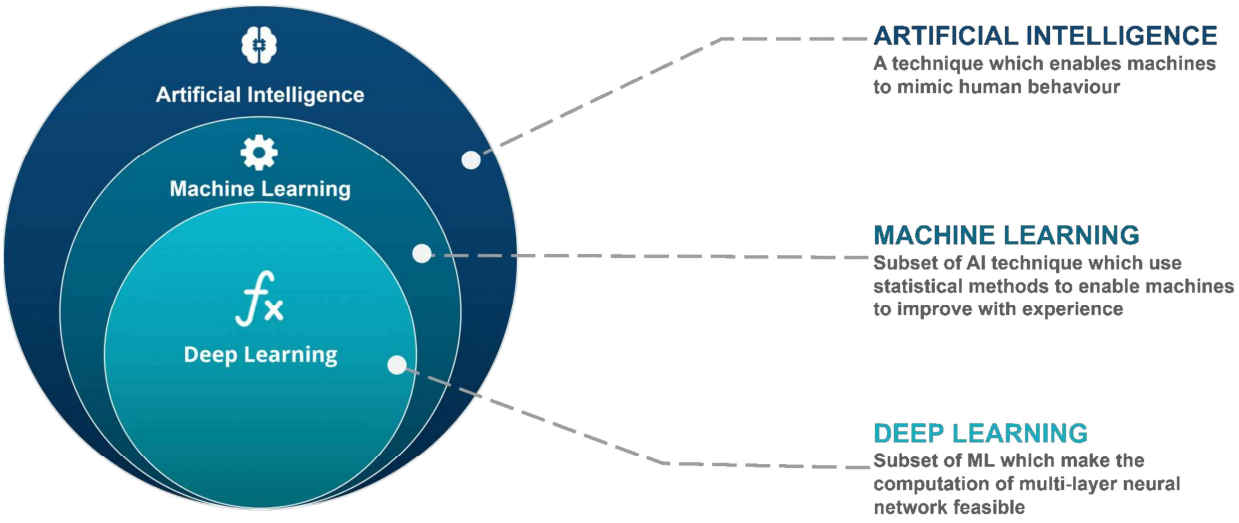
Deep Learning

Q1. What do you mean by Deep Learning?

Deep Learning is nothing but a paradigm of machine learning which has shown incredible promise in recent years. This is because of the fact that Deep Learning shows a great analogy with the functioning of the neurons in the human brain.



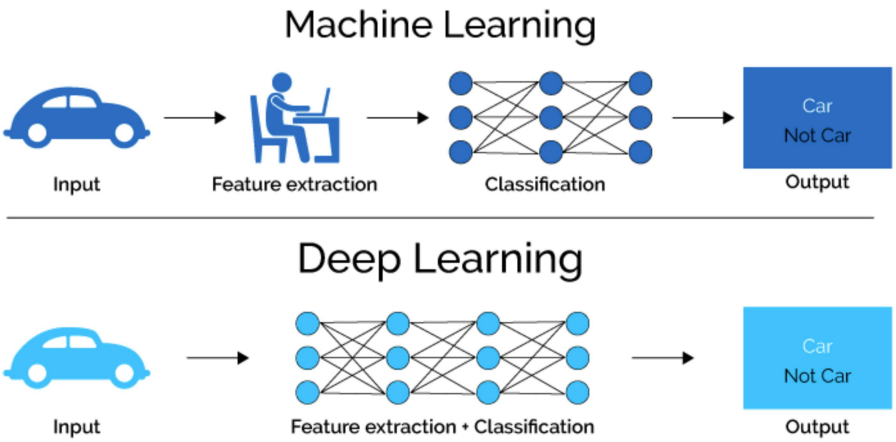
Q2. What is the difference between machine learning and deep learning?

https://parsers.me/deep-learning-machine-learning-whats-the-difference/

Machine learning is a field of computer science that gives computers the ability to learn without being explicitly programmed. Machine learning can be categorized in the following four categories.

1. Supervised machine learning,
2. Semi-supervised machine learning,
3. Unsupervised machine learning,
4. Reinforcement learning.

Deep Learning is a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks.



- The main difference between deep learning and machine learning is due to the way data is presented in the system. Machine learning algorithms almost always require structured data, while deep learning networks rely on layers of ANN (artificial neural networks).
- Machine learning algorithms are designed to "learn" to act by understanding labeled data and then use it to produce new results with more datasets. However, when the result is incorrect, there is a need to "teach them". Because machine learning algorithms require bulleted data, they are not suitable for solving complex queries that involve a huge amount of data.
- Deep learning networks do not require human intervention, as multilevel layers in neural networks place data in a hierarchy of different concepts, which ultimately learn from their own mistakes. However, even they can be wrong if the data quality is not good enough.
- Data decides everything. It is the quality of the data that ultimately determines the quality of the result.
- Both of these subsets of AI are somehow connected to data, which makes it possible to represent a certain form of "intelligence." However, you should be aware that deep learning requires much more data than a traditional machine learning algorithm. The reason for this is that deep learning networks can identify different elements in neural network layers only when more than a million data points interact. Machine learning algorithms, on the other hand, are capable of learning by pre-programmed criteria.

Q3. What, in your opinion, is the reason for the popularity of Deep Learning in recent times?

Now although Deep Learning has been around for many years, the major breakthroughs from these techniques came just in recent years. This is because of two main reasons:

- The increase in the amount of data generated through various sources
- The growth in hardware resources required to run these models

GPUs are multiple times faster and they help us build bigger and deeper deep learning models in comparatively less time than we required previously.

Q4. What is reinforcement learning?

Reinforcement Learning allows to take actions to max cumulative reward. It learns by trial and error through reward/penalty system. Environment rewards agent so by time agent makes better decisions.
Ex: robot=agent, maze=environment. Used for complex tasks (self-driving cars, game AI).

RL is a series of time steps in a Markov Decision Process:

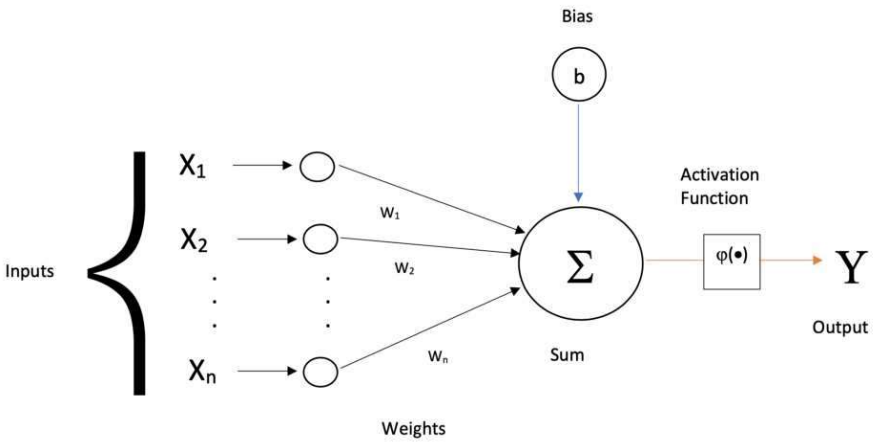
1. Environment: space in which RL operates
2. State: data related to past action RL took
3. Action: action taken
4. Reward: number taken by agent after last action
5. Observation: data related to environment: can be visible or partially shadowed

Q5. What are Artificial Neural Networks?

Artificial Neural networks are a specific set of algorithms that have revolutionized machine learning. They are inspired by biological neural networks. Neural Networks can adapt to changing the input, so the network generates the best possible result without needing to redesign the output criteria.

Q6. Describe the structure of Artificial Neural Networks?

Artificial Neural Networks works on the same principle as a biological Neural Network. It consists of inputs which get processed with weighted sums and Bias, with the help of Activation Functions.



Q7. How Are Weights Initialized in a Network?

There are two methods here: we can either initialize the weights to zero or assign them randomly.

Initializing all weights to 0: This makes your model similar to a linear model. All the neurons and every layer perform the same operation, giving the same output and making the deep net useless.

Initializing all weights randomly: Here, the weights are assigned randomly by initializing them very close to 0. It gives better accuracy to the model since every neuron performs different computations. This is the most commonly used method.

Q8. What Is the Cost Function?

Also referred to as "loss" or "error," cost function is a measure to evaluate how good your model's performance is. It's used to compute the error of the output layer during backpropagation. We push that error backwards through the neural network and use that during the different training functions.
The most known one is the mean sum of squared errors.

$$\text{MSE} = \underbrace{\frac{1}{n}}_{\text{test set}} \sum_{i=1}^{n} \underbrace{(y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i)^2}_{\text{actual value}}$$

$$\hat{y}_i = \phi(\sum(w_i x_i) + b)$$

Q9. What Are Hyperparameters?

With neural networks, you're usually working with hyperparameters once the data is formatted correctly. A hyperparameter is a parameter whose value is set before the learning process begins. It determines how a network is trained and the structure of the network (such as the number of hidden units, the learning rate, epochs, batches, etc.).

Q10. What Will Happen If the Learning Rate Is Set inaccurately (Too Low or Too High)?

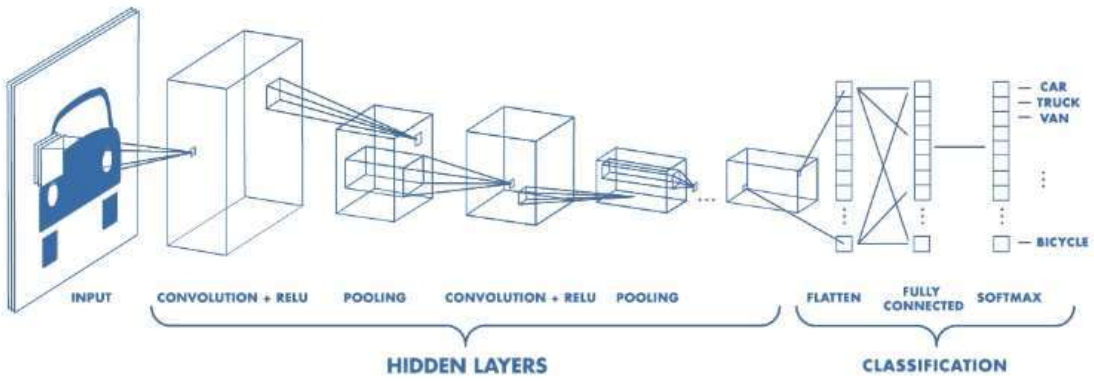
When your learning rate is too low, training of the model will progress very slowly as we are making minimal updates to the weights. It will take many updates before reaching the minimum point.
If the learning rate is set too high, this causes undesirable divergent behavior to the loss function due to drastic updates in weights. It may fail to converge (model can give a good output) or even diverge (data is too chaotic for the network to train).

Q11. What Is The Difference Between Epoch, Batch, and Iteration in Deep Learning?

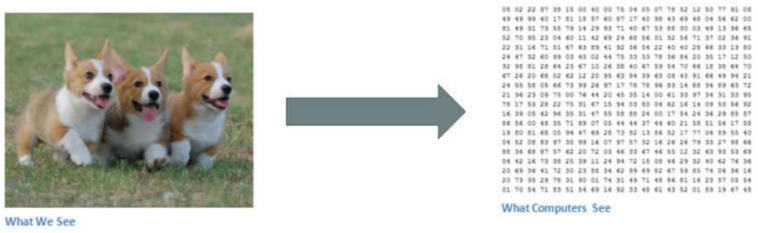
- Epoch – Represents one iteration over the entire dataset (everything put into the training model).
- Batch – Refers to when we cannot pass the entire dataset into the neural network at once, so we divide the dataset into several batches.
- Iteration – if we have 10,000 images as data and a batch size of 200. then an epoch should run 50 iterations (10,000 divided by 50).

Q12. What Are the Different Layers on CNN?

https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add



The Convolutional neural networks are regularized versions of multilayer perceptron (MLP). They were developed based on the working of the neurons of the animal visual cortex.



Let's say we have a color image in JPG form and its size is 480 x 480. The representative array will be 480 x 480 x 3. Each of these numbers is given a value from 0 to 255 which describes the pixel intensity at that point. RGB intensity values of the image are visualized by the computer for processing.

The objective of using the CNN:

The idea is that you give the computer this array of numbers and it will output numbers that describe the probability of the image being a certain class (.80 for a cat, .15 for a dog, .05 for a bird, etc.). It works similar to how our brain works. When we look at a picture of a dog, we can classify it as such if the picture has identifiable features such as paws or 4 legs. In a similar way, the computer is able to perform image classification by looking for low-level features such as edges and curves and then building up to more abstract concepts through a series of convolutional layers. The computer uses low-level features obtained at the initial levels to generate high-level features such as paws or eyes to identify the object.

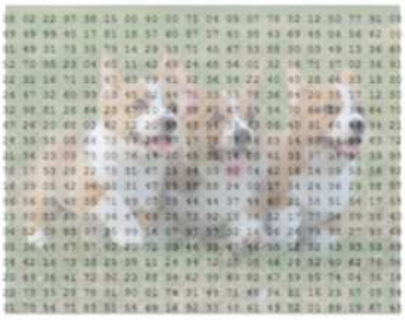
There are four layers in CNN:

1. **Convolutional Layer –** the layer that performs a convolutional operation, creating several smaller picture windows to go over the data.
2. **Activation Layer (ReLU Layer)** – it brings non-linearity to the network and converts all the negative pixels to zero. The output is a rectified feature map. It follows each convolutional layer.
3. **Pooling Layer –** pooling is a down-sampling operation that reduces the dimensionality of the feature map. Stride = how much you slide, and you get the max of the $n \; x \; n$ matrix
4. **Fully Connected Layer –** this layer recognizes and classifies the objects in the image.

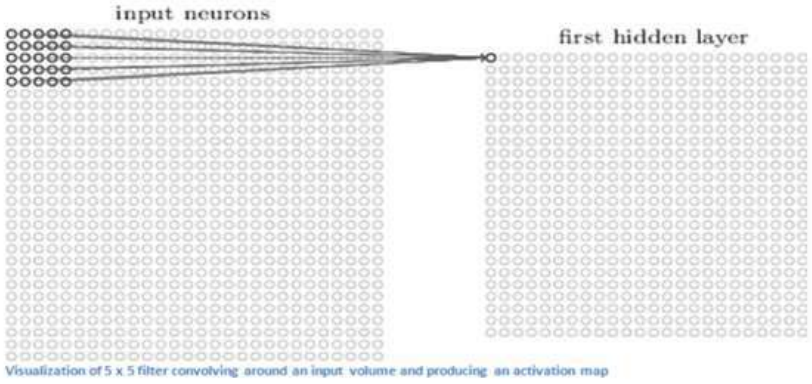
Convolution Operation

First Layer:

1. **Input to a convolutional layer**
   The image is resized to an optimal size and is fed as input to the convolutional layer.
   Let us consider the input as 32x32x3 array of pixel values.



2. **There exists a filter or neuron or kernel which lays over some of the pixels of the input image depending on the dimensions of the Kernel size.**
   Let the dimensions of the kernel of the filter be 5x5x3.



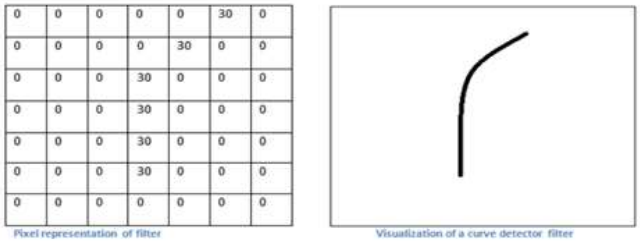
Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

3. **The Kernel actually slides over the input image; thus, it is multiplying the values in the filter with the original pixel values of the image (aka computing element-wise multiplications).**
   The multiplications are summed up generating a single number for that particular receptive field and hence for sliding the kernel a total of 784 numbers are mapped to 28x28 array known as the feature map.

   ***Now if we consider two kernels of the same dimension then the obtained first layer feature map will be (28x28x2).*

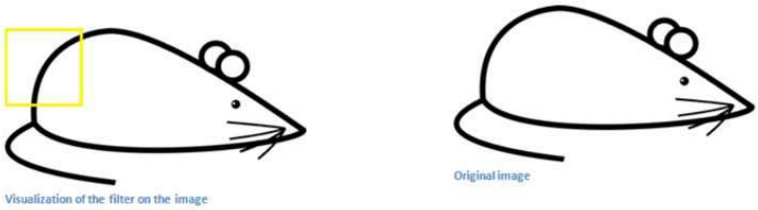
High-level Perspective

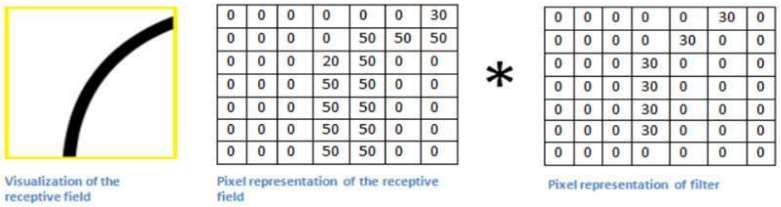
- Let us take a kernel of size (7x7x3) for understanding. Each of the kernels is considered to be a feature identifier, hence say that our filter will be a curve detector.



0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

- The original image and the visualization of the kernel on the image.





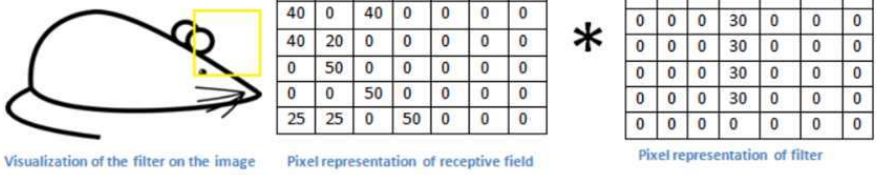
0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

$*$

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

The sum of the multiplication value that is generated is $= 4 * (50 * 30) + (20 * 30) = 6600$ *(large number).*

- Now when the kernel moves to the other part of the image.



0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

$*$

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

The sum of the multiplication value that is generated is = 0 (small number).

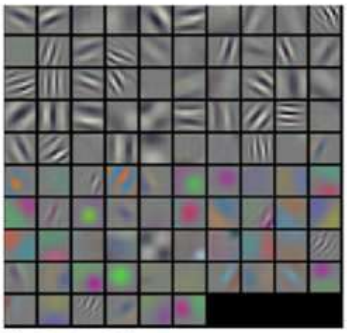
The use of the small and the large value

1. The value is much lower! This is because there wasn't anything in the image section that responded to the curve detector filter. Remember, the output of this convolution layer is an activation map. So, in the simple case of a one filter convolution (and if that filter is a curve detector), the activation map will show the areas in which there at most likely to be curved in the picture.

2. *In the previous example, the top-left value of our 26 x 26 x 1 activation map (26 because of the 7x7 filter instead of 5x5) will be 6600. This high value means that it is likely that there is some sort of curve in the input volume that caused the filter to activate. The top right value in our activation map will be 0 because there wasn't anything in the input volume that caused the filter to activate. This is just for one filter.*

3. This is just a filter that is going to detect lines that curve outward and to the right. We can have other filters for lines that curve to the left or for straight edges. The more filters, the greater the depth of the activation map, and the more information we have about the input volume.

   In the picture, we can see some examples of actual visualizations of the filters of the first conv. layer of a trained network. Nonetheless, the main argument remains the same. The filters on the first layer convolve around the input image and "activate" (or compute high values) when the specific feature it is looking for is in the input volume.



Visualizations of filters

Sequential convolutional layers after the first one

1. When we go through another conv. layer, the output of the first conv. layer becomes the input of the 2nd conv. layer.

2. However, when we're talking about the 2nd conv. layer, the input is the activation map(s) that result from the first layer. So, each layer of the input is basically describing the locations in the original image for where certain low-level features appear.

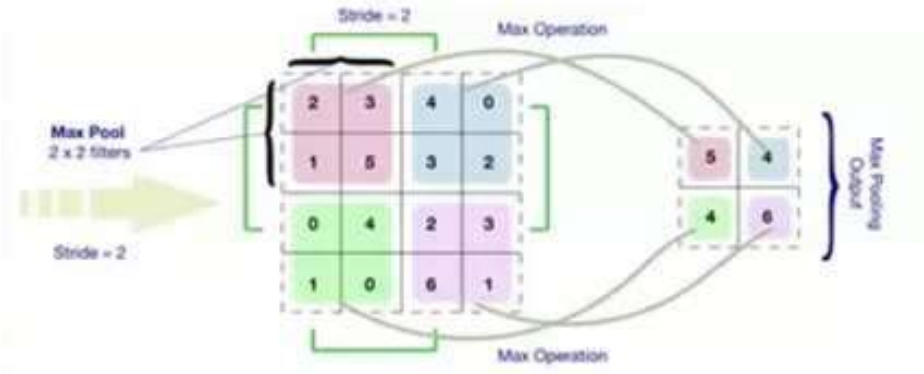
3. Now when you apply a set of filters on top of that (pass it through the 2nd conv. layer), the output will be activations that represent higher-level features. Types of these features could be semicircles (a combination of a curve and straight edge) or squares (a combination of several straight edges). As you go through the network and go through more convolutional layers, you get activation maps that represent more and more complex features.

4. By the end of the network, you may have some filters that activate when there is handwriting in the image, filters that activate when they see pink objects, etc.

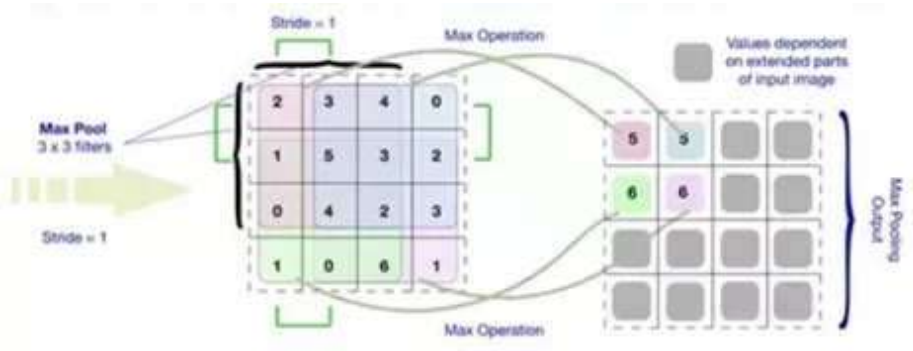
Pooling Operation

It consists in getting the largest number out of a matrix to get the most important number and reduce the dimention.

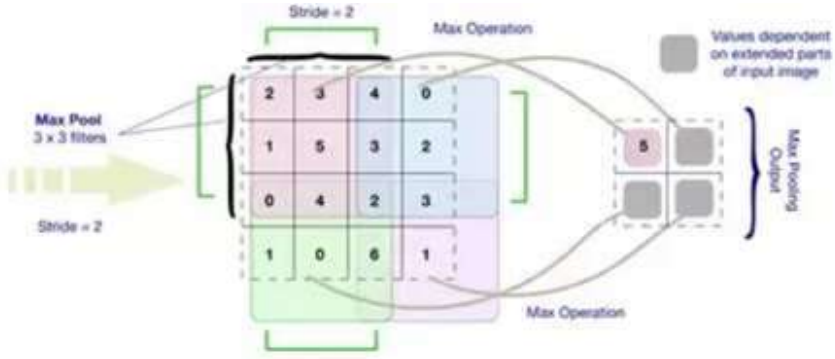
Max Pooling example



2x2 filters with stride = 2 (maximum value) is considered



3x3 filters with stride = 1 (maximum value) is considered



3x3 filters with stride = 2 (maximum value) is considered

Classification

1. **Flatten:** The pooled matrix is converted to a vector**.**

2. **Fully Connected layer:** The way this fully connected layer works is that it looks at the output of the previous layer (which as we remember should represent the activation maps of high-level features) and the number of classes p (10 for digit classification). *For example, if the program is predicting that some image is a dog, it will have high values in the activation maps that represent high-level features like a paw or 4 legs, etc. Basically, an FC layer looks at what high level features most strongly correlate to a particular class and has particular weights so that when you compute the products between the weights and the previous layer, you get the correct probabilities for the different classes.*
3. **Soft-max approach:** The output of a fully connected layer is as follows [0 .1 .1 .75 0 0 0 0 0 .05], then this represents a 10% probability that the image is a 1, a 10% probability that the image is a 2, a 75% probability that the image is a 3, and a 5% probability that the image is a 9 (SoftMax approach) for digit classification.

Training

§We know kernels also known as feature identifiers, used for identification of specific features. But how the kernels are initialized with the specific weights or how do the filters know what values to have.

Hence comes the important step of training. The training process is also known as backpropagation, which is further separated into 4 distinct sections or processes.

- Forward Pass
- Loss Function
- Backward Pass
- Weight Update

The Forward Pass

For the first epoch or iteration of the training the initial kernels of the first convolutional layer are initialized with random values. Thus, after the first iteration output will be something like [.1.1.1.1.1.1.1.1.1.1], which does not give preference to any class as the kernels don't have specific weights.

The Loss Function

The training involves images along with labels, hence the label for the digit 3 will be [0 0 0 1 0 0 0 0 0 0], whereas the output after a first epoch is very different, hence we will calculate loss (MSE — Mean Squared Error)

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

The objective is to minimize the loss, which is an optimization problem in calculus. It involves trying to adjust the weights to reduce the loss.

The Backward Pass

It involves determining which weights contributed most to the loss and finding ways to adjust them so that the loss decreases. It is computed using $\frac{\partial L}{\partial W}$ (or $\nabla_w L$), where L is the loss and the W is the weights of the corresponding kernel.

The weights update

This is where the weights of the kernel are updated using the following equation.

$$w = w_i - \eta \frac{dL}{dW}$$

w = Weight
w_i = Initial Weight
η = Learning Rate

Here the Learning Rate is chosen by the programmer. Larger value of the learning rate indicates much larger steps towards optimization of steps and larger time to convolve to an optimized weight.

Testing

Finally, to see whether or not our CNN works, we have a different set of images and labels (can't double dip between training and test!) and pass the images through the CNN. We compare the outputs to the ground truth and see if our network works!

Q13. What Is Pooling on CNN, and How Does It Work?

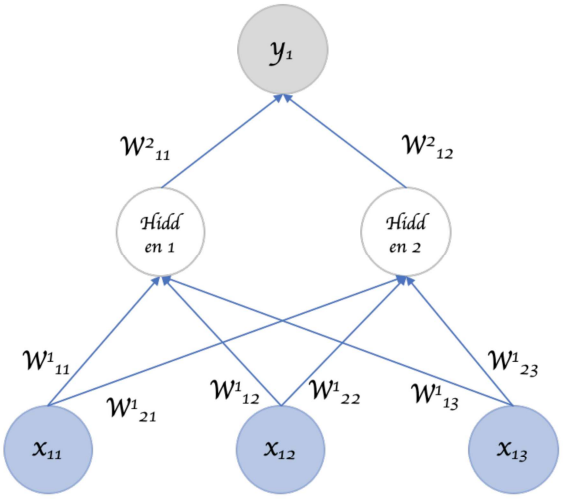
Pooling is used to reduce the spatial dimensions of a CNN. It performs down-sampling operations to reduce the dimensionality and creates a pooled feature map by sliding a filter matrix over the input matrix.

Q14. What are Recurrent Neural Networks (RNNs)?

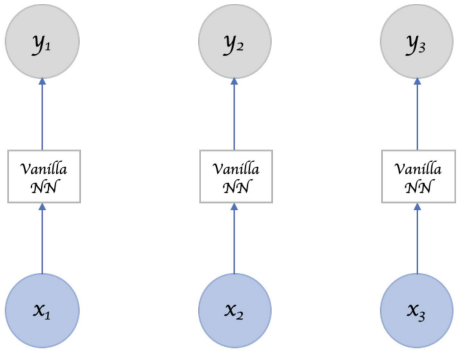
https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce

RNNs are a type of artificial neural networks designed to recognize the pattern from the sequence of data such as Time series, stock market and government agencies etc.

Recurrent Neural Networks (RNNs) add an interesting twist to basic neural networks. A vanilla neural network takes in a fixed size vector as input which limits its usage in situations that involve a 'series' type input with no predetermined size.

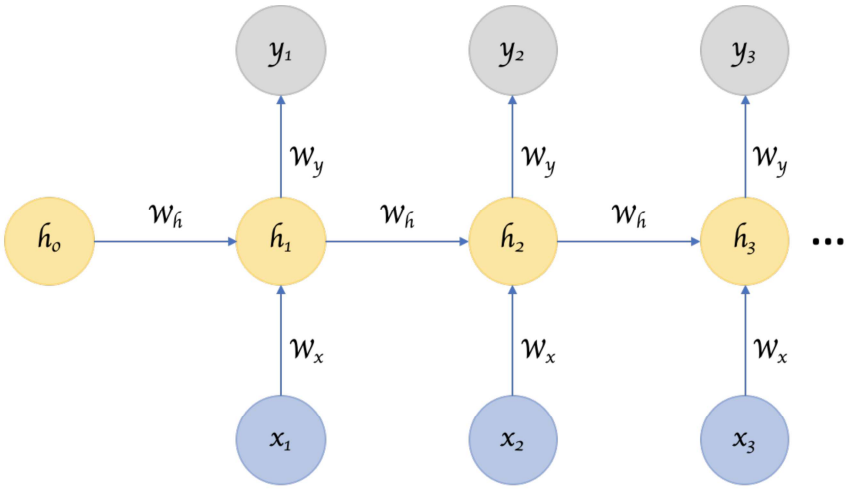


RNNs are designed to take a series of input with no predetermined limit on size. One could ask what's the big deal, I can call a regular NN repeatedly too?



Sure can, but the 'series' part of the input means something. A single input item from the series is related to others and likely has an influence on its neighbors. Otherwise it's just "many" inputs, not a "series" input (duh!).

Recurrent Neural Network remembers the past and its decisions are influenced by what it has learnt from the past. Note: Basic feed forward networks "remember" things too, but they remember things they learnt during training. For example, an image classifier learns what a "1" looks like during training and then uses that knowledge to classify things in production.
While RNNs learn similarly while training, in addition, they remember things learnt from prior input(s) while generating output(s). RNNs can take one or more input vectors and produce one or more output vectors and the output(s) are influenced not just by weights applied on inputs like a regular NN, but also by a "hidden" state vector representing the context based on prior input(s)/output(s). So, the same input could produce a different output depending on previous inputs in the series.



In summary, in a vanilla neural network, a fixed size input vector is transformed into a fixed size output vector. Such a network becomes "recurrent" when you repeatedly apply the transformations to a series of given input and produce a series of output vectors. There is no pre-set limitation to the size of the vector. And, in addition to generating the output which is a function of the input and hidden state, we update the hidden state itself based on the input and use it in processing the next input.

Parameter Sharing

You might have noticed another key difference between Figure 1 and Figure 3. In the earlier, multiple different weights are applied to the different parts of an input item generating a hidden layer neuron, which in turn is transformed using further weights to produce an output. There seems to be a lot of weights in play here. Whereas in Figure 3, we seem to be applying the same weights over and over again to different items in the input series.

I am sure you are quick to point out that we are kind of comparing apples and oranges here. The first figure deals with "a" single input whereas the second figure represents multiple inputs from a series. But nevertheless, intuitively speaking, as the number of inputs increase, shouldn't the number of weights in play increase as well? Are we losing some versatility and depth in Figure 3?

Perhaps we are. We are sharing parameters across inputs in Figure 3. If we don't share parameters across inputs, then it becomes like a vanilla neural network where each input node requires weights of their own. This introduces the constraint that the length of the input has to be fixed and that makes it impossible to leverage a series type input where the lengths differ and is not always known.

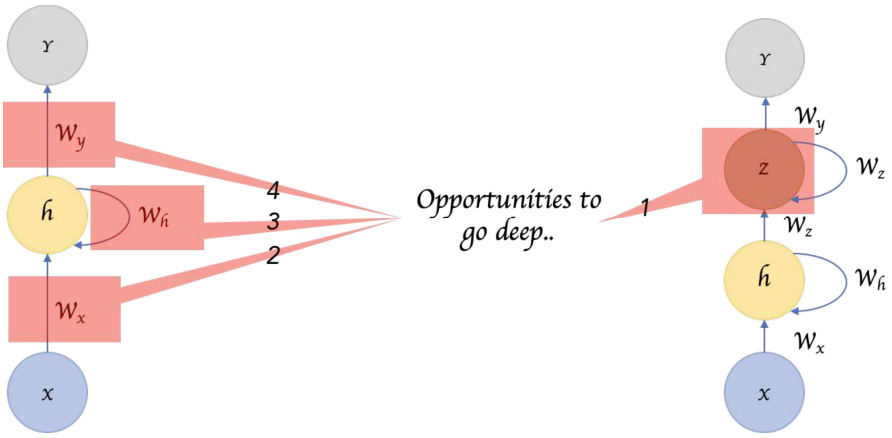
But what we seemingly lose in value here, we gain back by introducing the "hidden state" that links one input to the next. The hidden state captures the relationship that neighbors might have with each other in a serial input and it keeps changing in every step, and thus effectively every input undergoes a different transition!

Image classifying CNNs have become so successful because the 2D convolutions are an effective form of parameter sharing where each convolutional filter basically extracts the presence or absence of a feature in an image which is a function of not just one pixel but also of its surrounding neighbor pixels.

In other words, the success of CNNs and RNNs can be attributed to the concept of "parameter sharing" which is fundamentally an effective way of leveraging the relationship between one input item and its surrounding neighbors in a more intrinsic fashion compared to a vanilla neural network.

Deep RNNs

While it's good that the introduction of hidden state enabled us to effectively identify the relationship between the inputs, is there a way we can make an RNN "deep" and gain the multi-level abstractions and representations we gain through "depth" in a typical neural network?

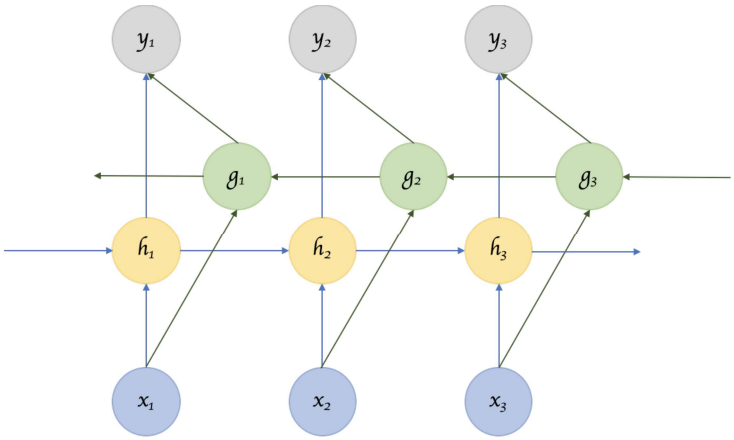


Here are four possible ways to add depth.

1) We can add hidden states, one on top of another, feeding the output of one to the next.
2) We can also add additional nonlinear hidden layers between input to hidden state.
3) We can increase depth in the hidden to hidden transition.
4) We can increase depth in the hidden to output transition.

Bidirectional RNNs

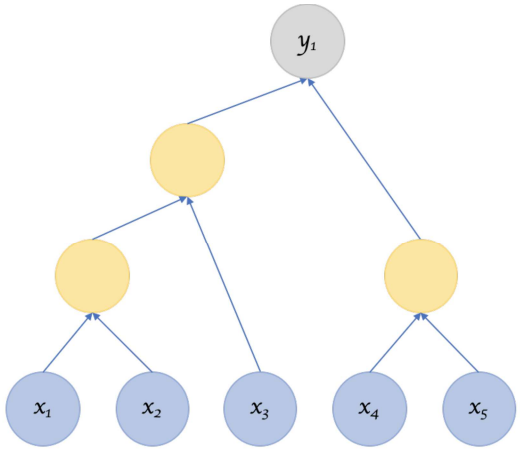
Sometimes it's not just about learning from the past to predict the future, but we also need to look into the future to fix the past. In speech recognition and handwriting recognition tasks, where there could be considerable ambiguity given just one part of the input, we often need to know what's coming next to better understand the context and detect the present.



This does introduce the obvious challenge of how much into the future we need to look into, because if we have to wait to see all inputs then the entire operation will become costly. And in cases like speech recognition, waiting till an entire sentence is spoken might make for a less compelling use case. Whereas for NLP tasks, where the inputs tend to be available, we can likely consider entire sentences all at once. Also, depending on the application, if the sensitivity to immediate and closer neighbors is higher than inputs that come further away, a variant that looks only into a limited future/past can be modeled.

Recursive Neural Network

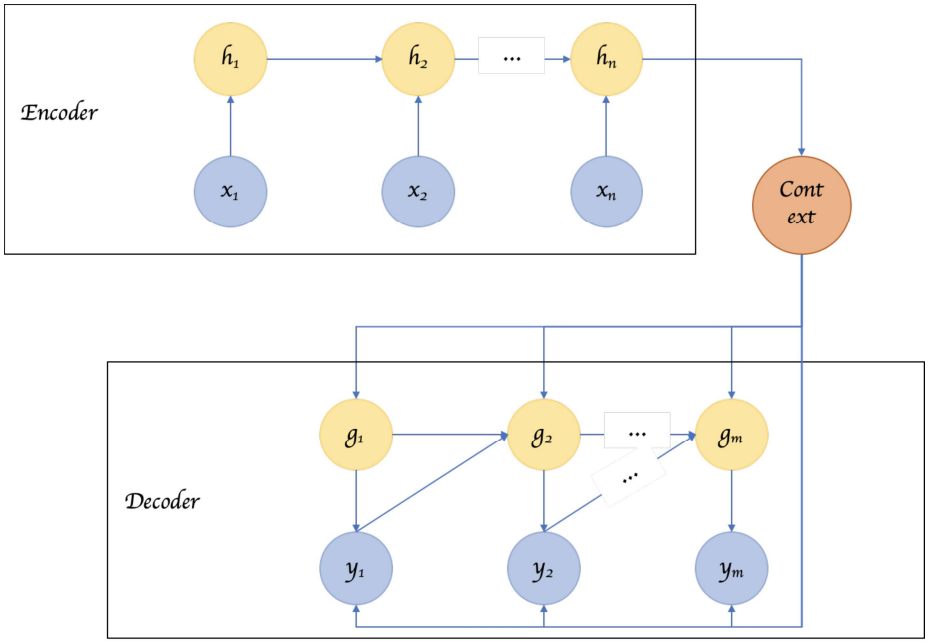
A recurrent neural network parses the inputs in a sequential fashion. A recursive neural network is similar to the extent that the transitions are repeatedly applied to inputs, but not necessarily in a sequential fashion. Recursive Neural Networks are a more general form of Recurrent Neural Networks. It can operate on any hierarchical tree structure. Parsing through input nodes, combining child nodes into parent nodes and combining them with other child/parent nodes to create a tree like structure. Recurrent Neural Networks do the same, but the structure there is strictly linear. i.e. weights are applied on the first input node, then the second, third and so on.



But this raises questions pertaining to the structure. How do we decide that? If the structure is fixed like in Recurrent Neural Networks then the process of training, backprop, makes sense in that they are similar to a regular neural network. But if the structure isn't fixed, is that learnt as well?

Encoder Decoder Sequence to Sequence RNNs

Encoder Decoder or Sequence to Sequence RNNs are used a lot in translation services. The basic idea is that there are two RNNs, one an encoder that keeps updating its hidden state and produces a final single "Context" output. This is then fed to the decoder, which translates this context to a sequence of outputs. Another key difference in this arrangement is that the length of the input sequence and the length of the output sequence need not necessarily be the same.



LSTMs

LSTM is not a different variant of RNN architecture, but rather it introduces changes to how we compute outputs and hidden state using the inputs.
In a vanilla RNN, the input and the hidden state are simply passed through a single tanh layer. LSTM (Long-Short-Term Memory) networks improve on this simple transformation and introduces additional gates and a cell state, such that it fundamentally addresses the problem of keeping or resetting context, across sentences and regardless of the distance between such context resets. There are variants of LSTMs including GRUs that utilize the gates in different manners to address the problem of long-term dependencies.

Q15. How Does an LSTM Network Work?

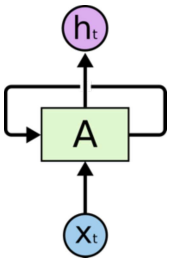
http://karpathy.github.io/2015/05/21/rnn-effectiveness
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long-Short-Term Memory (LSTM) is a special kind of recurrent neural network capable of learning long-term dependencies, remembering information for long periods as its default behavior. There are three steps in an LSTM network:

- Step 1: The network decides what to forget and what to remember.
- Step 2: It selectively updates cell state values.
- Step 3: The network decides what part of the current state makes it to the output.

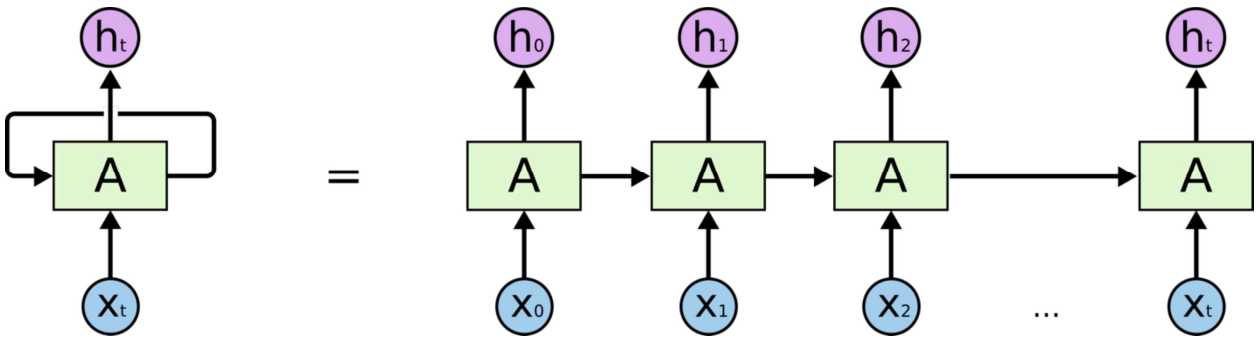
Recurrent Neural Networks

Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.
Traditional neural networks can't do this, and it seems like a major shortcoming. *For example, imagine you want to classify what kind of event is happening at every point in a movie. It's unclear how a traditional neural network could use its reasoning about previous events in the film to inform later ones.*
Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.



Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, A, looks at some input x_t and outputs a value h_t. A loop allows information to be passed from one step of the network to the next.
These loops make recurrent neural networks seem kind of mysterious. However, if you think a bit more, it turns out that they aren't all that different than a normal neural network. A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. Consider what happens if we unroll the loop:



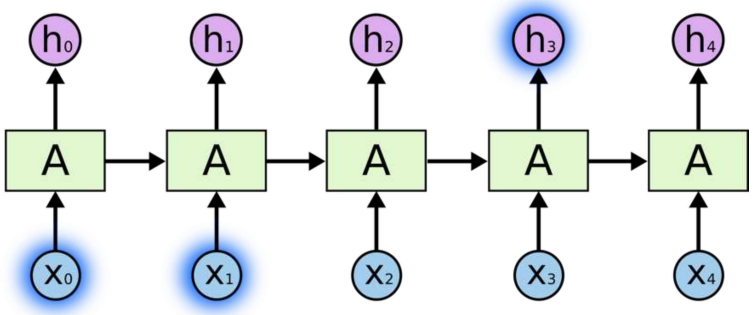
An unrolled recurrent neural network.

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.
And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning…

Essential to these successes is the use of "LSTMs," a very special kind of recurrent neural network which works, for many tasks, much better than the standard version. Almost all exciting results based on recurrent neural networks are achieved with them.

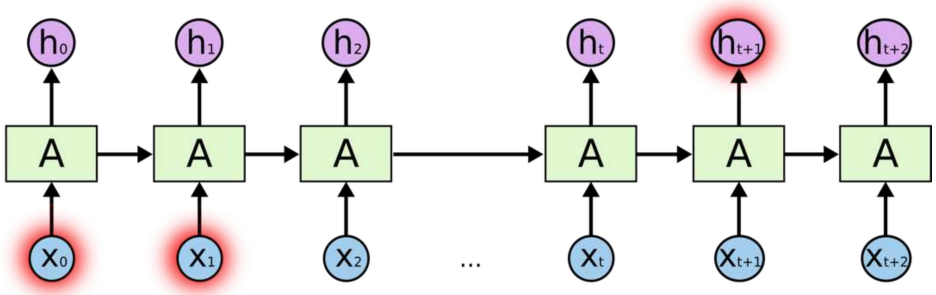
The Problem of Long-Term Dependencies

One of the appeals of RNNs is the idea that they might be able to connect previous information to the present task, such as using previous video frames might inform the understanding of the present frame. If RNNs could do this, they'd be extremely useful. But can they? It depends.
Sometimes, we only need to look at recent information to perform the present task. *For example, consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in "the clouds are in the* sky*," we don't need any further context – it's pretty obvious the next word is going to be sky. In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information.*



But there are also cases where we need more context. Consider trying to predict the last word in the text "I grew up in France… I speak fluent French*." Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of France, from further back. It's entirely possible for the gap between the relevant information and the point where it is needed to become very large.*

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



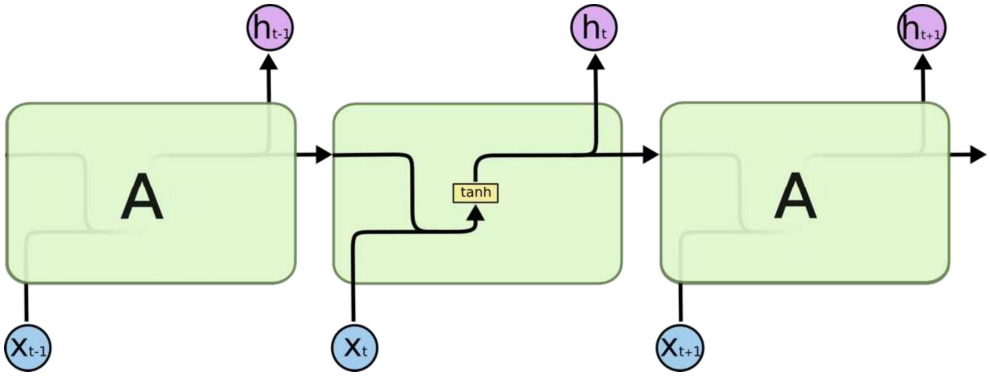
In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them. Thankfully, LSTMs don't have this problem!

LSTM Networks

Long Short-Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They work tremendously well on a large variety of problems and are now widely used.

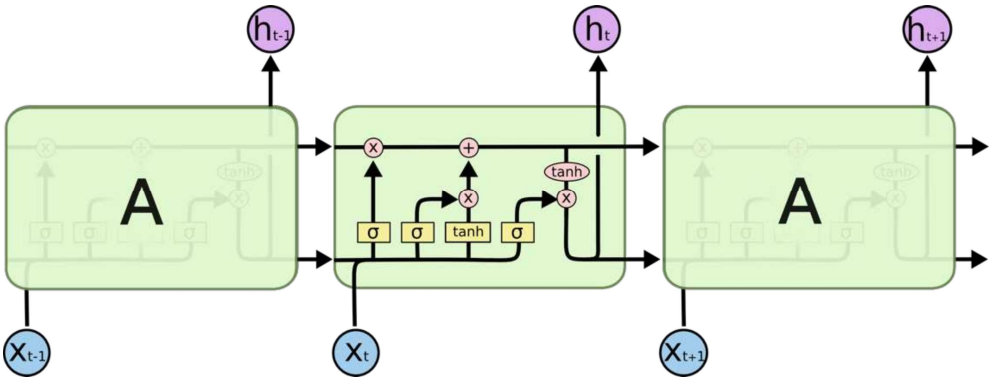
LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

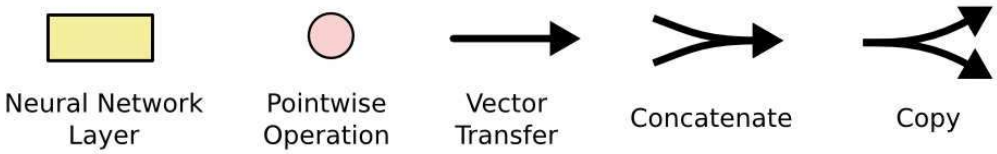


The repeating module in a standard RNN contains a single layer.

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



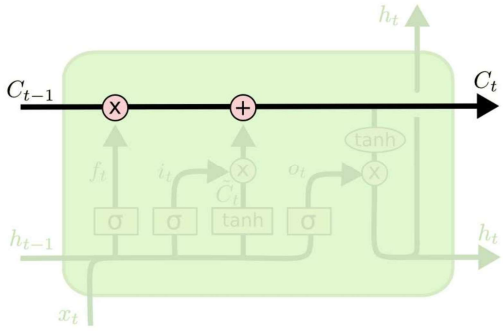
The repeating module in an LSTM contains four interacting layers.



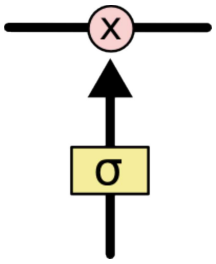
In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denotes its content being copied and the copies going to different locations.

The Core Idea Behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.
The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.
Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.



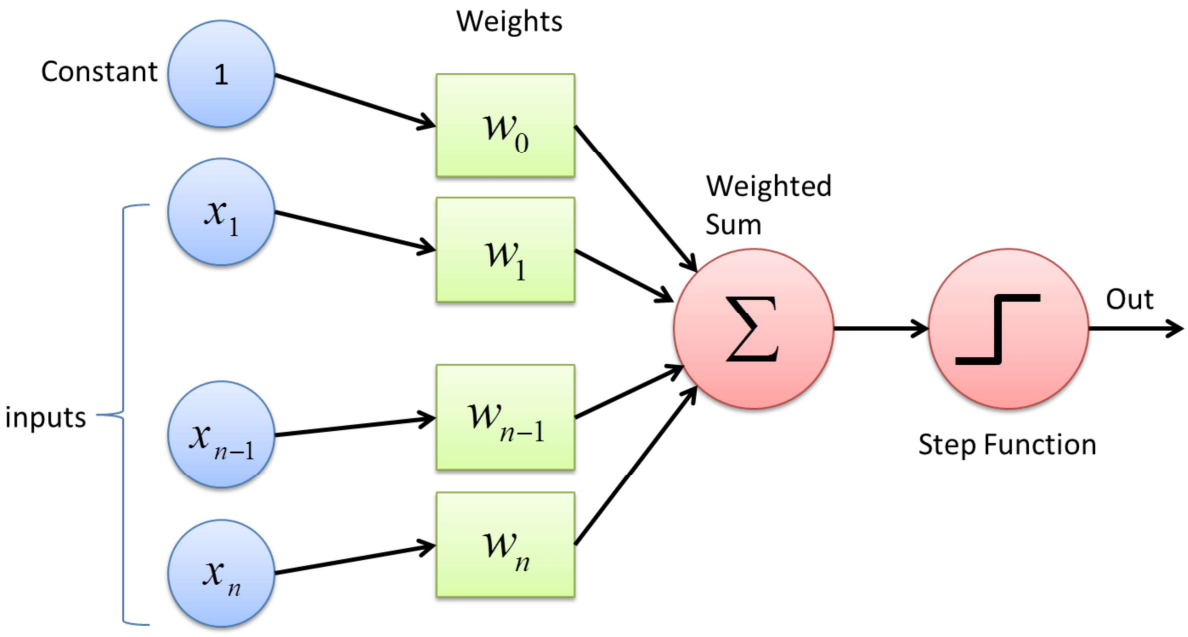
The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”
An LSTM has three of these gates, to protect and control the cell state.

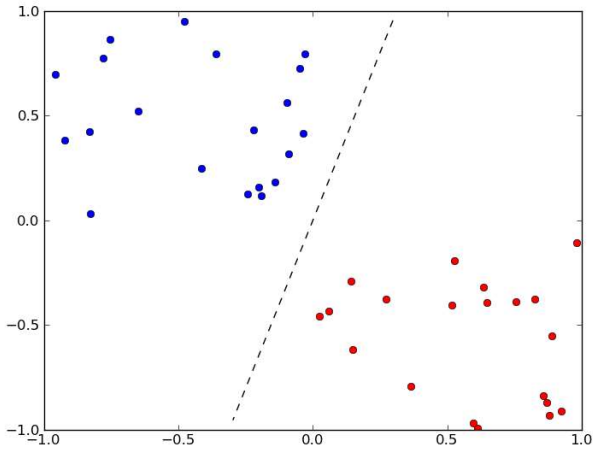
Q16. What Is a Multi-layer Perceptron (MLP)?

https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53

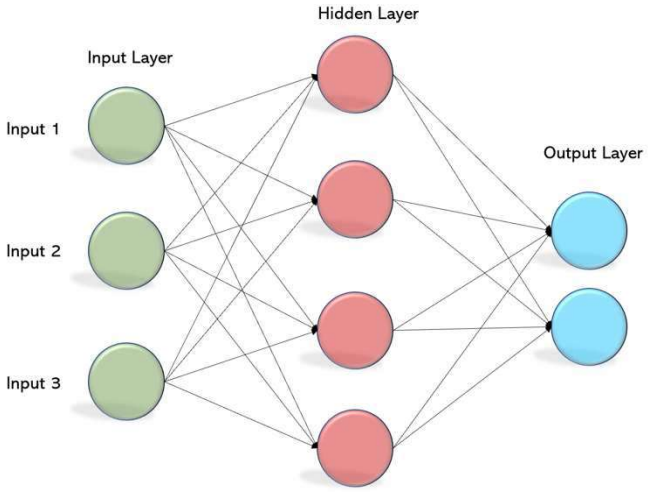
As in Neural Networks, MLPs have an input layer, a hidden layer, and an output layer. It has the same structure as a single layer perceptron with one or more hidden layers.

Perceptron is a single layer neural network and a multi-layer perceptron is called Neural Networks.
A (single layer) perceptron is a single layer neural network that works as a linear binary classifier. Being a single layer neural network, it can be trained without the use of more advanced algorithms like back propagation and instead can be trained by "stepping towards" your error in steps specified by a learning rate. When someone says perceptron, I usually think of the single layer version.





A single layer perceptron can classify only linear separable classes with binary output {0,1} or {-1,1}, but MLP can classify nonlinear classes. **The activation functions are used to map the input between the required values like {0, 1} or {-1, 1}**.



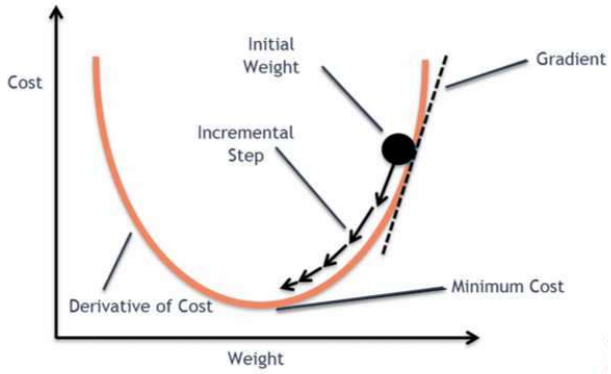
Except for the input layer, each node in the other layers uses a nonlinear activation function. This means the input layers, the data coming in, and the activation function is based upon all nodes and weights being added together, producing the output. MLP uses a supervised learning method called "backpropagation." In backpropagation, the neural network calculates the error with the help of cost function. It propagates this error backward from where it came (adjusts the weights to train the model more accurately).

Usually, RELU is in hidden layers (it does not classify), and Soft-max or tanh is in output layers.

Q17. Explain Gradient Descent.

Let's first explain what a gradient is. A gradient is a mathematical function. When calculated on a point of a function, it gives the hyperplane (or slope) of the directions in which the function increases more. The gradient vector can be interpreted as the "direction and rate of fastest increase". If the gradient of a

function is non-zero at a point *p*, the direction of the gradient is the direction in which the function increases most quickly from *p*, and the magnitude of the gradient is the rate of increase in that direction. Further, the gradient is the zero vector at a point if and only if it is a stationary point (where the derivative vanishes).
In DS, it simply measures the change in all weights with regard to the change in error, as we are partially derivating by w the loss function.



Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function.

The goal of the gradient descent is to minimize a given function which, in our case, is the loss function of the neural network. To achieve this goal, it performs two steps iteratively.

1. Compute the slope (gradient) that is the first-order derivative of the function at the current point
2. Move-in the opposite direction of the slope increase from the current point by the computed amount

So, the idea is to pass the training set through the hidden layers of the neural network and then update the parameters of the layers by computing the gradients using the training samples from the training dataset.

Think of it like this. Suppose a man is at top of the valley and he wants to get to the bottom of the valley. So, he goes down the slope. He decides his next position based on his current position and stops when he gets to the bottom of the valley which was his goal.

Q18. What is exploding gradients?

https://machinelearningmastery.com/exploding-gradients-in-neural-networks/

While training an RNN, if you see exponentially growing (very large) error gradients which accumulate and result in very large updates to neural network model weights during training, they're known as exploding gradients. At an extreme, the values of weights can become so large as to overflow and result in NaN values. The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1.0.

This has the effect of your model is unstable and unable to learn from your training data.

There are some subtle signs that you may be suffering from exploding gradients during the training of your network, such as:

- The model is unable to get traction on your training data (e.g. poor loss).
- The model is unstable, resulting in large changes in loss from update to update.
- The model loss goes to NaN during training.
- The model weights quickly become very large during training.
- The error gradient values are consistently above 1.0 for each node and layer during training.

Solutions

1. **Re-Design the Network Model:**
    a. In deep neural networks, exploding gradients may be addressed by redesigning the network to have fewer layers. There may also be some benefit in using a smaller batch size while training the network.
    b. In RNNs, updating across fewer prior time steps during training, called truncated Backpropagation through time, may reduce the exploding gradient problem.

2. **Use Long Short-Term Memory Networks:** In RNNs, exploding gradients can be reduced by using the Long Short-Term Memory (LSTM) memory units and perhaps related gated-type neuron structures. Adopting LSTM memory units is a new best practice for recurrent neural networks for sequence prediction.

3. **Use Gradient Clipping:** Exploding gradients can still occur in very deep Multilayer Perceptron networks with a large batch size and LSTMs with very long input sequence lengths. If exploding gradients are still occurring, you can check for and limit the size of gradients during the training of your network. This is called gradient clipping. Specifically, the values of the error gradient are checked against a threshold value and clipped or set to that threshold value if the error gradient exceeds the threshold.

4. **Use Weight Regularization:** another approach, if exploding gradients are still occurring, is to check the size of network weights and apply a penalty to the networks loss function for large weight values. This is called weight regularization and often an L1 (absolute weights) or an L2 (squared weights) penalty can be used.

Q19. What is vanishing gradients?

While training an RNN, your slope can become either too small; this makes the training difficult. When the slope is too small, the problem is known as a Vanishing Gradient. It leads to long training times, poor performance, and low accuracy.

- Hyperbolic tangent and Sigmoid/Soft-max suffer vanishing gradient.
- RNNs suffer vanishing gradient, LSTM no (so it is perfect to predict stock prices). In fact, the propagation of error through previous layers makes the gradient get smaller so the weights are not updated.

Solutions

1. **Choose RELU**
2. **Use LSTM (for RNNs)**
3. **Use ResNet (Residual Network)** → after some layers, add x again: $F(x) \to \cdots \to F(x) + x$
4. **Multi-level hierarchy:** pre-train one layer at the time through unsupervised learning, then fine-tune via backpropagation
5. **Gradient checking:** debugging strategy used to numerically track and assess gradients during training.

Q20. What is Back Propagation and Explain it Works.

Backpropagation is a training algorithm used for neural network. In this method, we update the weights of each layer from the last layer recursively, with the formula:

$$w_{previous\ layer} = w_{layer} - \eta \nabla_w L(w)$$

It has the following steps:

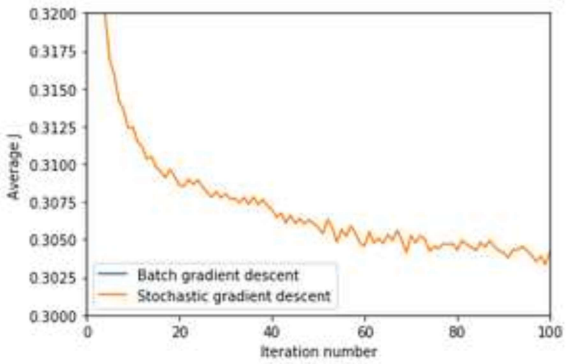
- Forward Propagation of Training Data (initializing weights with random or pre-assigned values)
- Gradients are computed using output weights and target
- Back Propagate for computing gradients of error from output activation
- Update the Weights

Q21. What are the variants of Back Propagation?

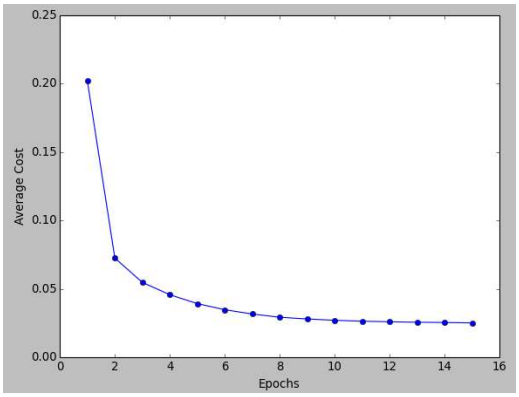
https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a

- **Stochastic Gradient Descent:** In Batch Gradient Descent we were considering all the examples for every step of Gradient Descent. But what if our dataset is very huge. Deep learning models crave for data. The more the data the more chances of a model to be good. *Suppose our dataset has 5 million examples, then just to take one step the model will have to calculate the gradients of all the 5 million examples. This does not seem an efficient way.* To tackle this problem, we have Stochastic Gradient Descent. In Stochastic Gradient Descent (SGD), we consider just one example at a time to take a single step. We do the following steps in **one epoch** for SGD:
    1. Take an example
    2. Feed it to Neural Network
    3. Calculate its gradient
    4. Use the gradient we calculated in step 3 to update the weights
    5. Repeat steps 1–4 for all the examples in training dataset

    Since we are considering just one example at a time the cost will fluctuate over the training examples and it will **not** necessarily decrease. But in the long run, you will see the cost decreasing with fluctuations. Also, because the cost is so fluctuating, it will never reach the minimum, but it will keep dancing around it. SGD can be used for larger datasets. It converges faster when the dataset is large as it causes updates to the parameters more frequently.



- **Batch Gradient Descent:** all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch. Batch Gradient Descent is great for convex or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution. The graph of cost vs epochs is also quite smooth because we are averaging over all the gradients of training data for a single step. The cost keeps on decreasing over the epochs.



- **Mini-batch Gradient Descent:** It's one of the most popular optimization algorithms. It's a variant of Stochastic Gradient Descent and here instead of single training example, mini batch of samples is used. Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large. Batch Gradient Descent converges directly to minima. SGD converges faster for larger datasets. But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations. To tackle this problem, a mixture of Batch Gradient Descent and SGD is used. Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. Doing this helps us achieve the advantages of both the former variants we saw. So, after creating the mini-batches of fixed size, we do the following steps in **one epoch:**
    1. Pick a mini-batch
    2. Feed it to Neural Network
    3. Calculate the mean gradient of the mini-batch
    4. Use the mean gradient we calculated in step 3 to update the weights
    5. Repeat steps 1–4 for the mini-batches we created

Just like SGD, the average cost over the epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time. So, when we are using the mini-batch gradient descent we are updating our parameters frequently as well as we can use vectorized implementation for faster computations.

Q22. What are the different Deep Learning Frameworks?

- PyTorch: PyTorch is an open source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab. It is free and open-source software released under the Modified BSD license.
- TensorFlow: TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library and is also used for machine learning applications such as neural networks. Licensed by Apache License 2.0. Developed by Google Brain Team.
- Microsoft Cognitive Toolkit: Microsoft Cognitive Toolkit describes neural networks as a series of computational steps via a directed graph.
- Keras: Keras is an open-source neural-network library written in Python. It is capable of running on top of TensorFlow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML. Designed to enable fast experimentation with deep neural networks, it focuses on being user-friendly, modular, and extensible. Licensed by MIT.

Q23. What is the role of the Activation Function?

The Activation function is used to introduce non-linearity into the neural network helping it to learn more complex function. Without which the neural network would be only able to learn linear function which is a linear combination of its input data. An activation function is a function in an artificial neuron that delivers an output based on inputs.

Q24. Name a few Machine Learning libraries for various purposes.

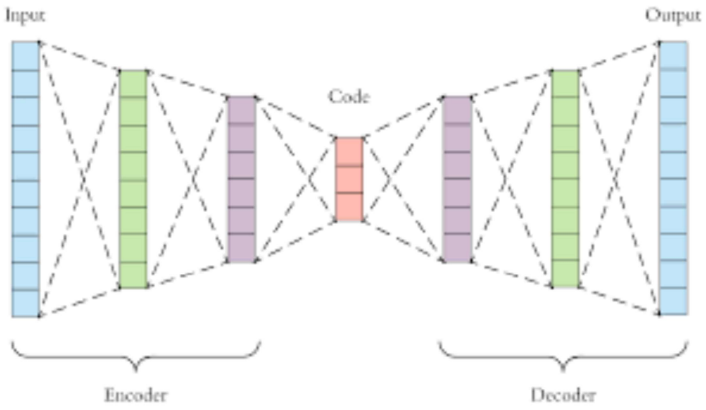
Purpose	Libraries
Scientific Computation	Numpy
Tabular Data	Pandas, GeoPandas
Data Modelling & Preprocessing	Scikit Learn
Time-Series Analysis	Statsmodels
Text processing	NTLK, Regular Expressions
Deep Learning	TensorFlow, Pytorch
Visualization	Bokeh, Seaborn
Plotting	Matplot

Q25. What is an Auto-Encoder?

https://www.quora.com/What-is-an-autoencoder-What-are-its-applications

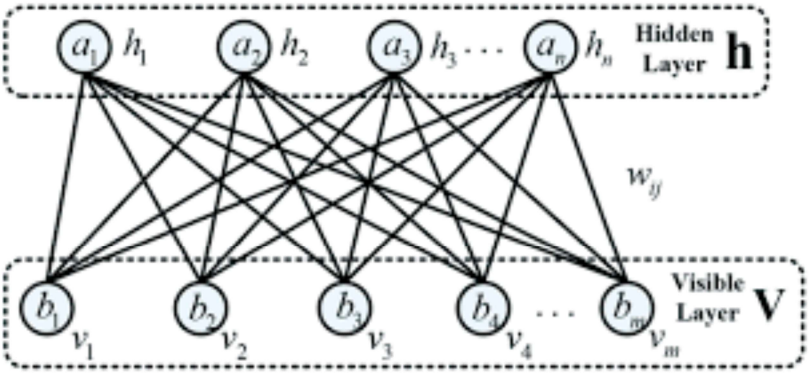
Auto-encoders are simple learning networks that aim to transform inputs into outputs with the minimum possible error. This means that we want the output to be as close to input as possible. We add a couple of layers between the input and the output, and the sizes of these layers are smaller than the input layer. The auto-encoder receives unlabeled input which is then encoded to reconstruct the input.

An **autoencoder** is a type of artificial neural network used to learn efficient data coding in an unsupervised manner. The aim of an **autoencoder** is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore signal "noise". Along with the reduction side, a reconstructing side is learnt, where the autoencoder tries to generate from the reduced encoding a representation as close as possible to its original input, hence its name. Several variants exist to the basic model, with the aim of forcing the learned representations of the input to assume useful properties. Autoencoders are effectively used for solving many applied problems, from face recognition to acquiring the semantic meaning of words.



Q26. What is a Boltzmann Machine?

Boltzmann machines have a simple learning algorithm that allows them to discover interesting features that represent complex regularities in the training data. The Boltzmann machine is basically used to optimize the weights and the quantity for the given problem. The learning algorithm is very slow in networks with many layers of feature detectors. "Restricted Boltzmann Machines" algorithm has a single layer of feature detectors which makes it faster than the rest.



Q27. What Is Dropout and Batch Normalization?

Dropout is a technique of dropping out hidden and visible nodes of a network randomly to prevent overfitting of data (typically dropping 20 per cent of the nodes). It doubles the number of iterations needed to converge the network. It used to avoid overfitting, as it increases the capacity of generalization.

Batch normalization is the technique to improve the performance and stability of neural networks by normalizing the inputs in every layer so that they have mean output activation of zero and standard deviation of one.

Q28. Why Is TensorFlow the Most Preferred Library in Deep Learning?

TensorFlow provides both C++ and Python APIs, making it easier to work on and has a faster compilation time compared to other Deep Learning libraries like Keras and PyTorch. TensorFlow supports both CPU and GPU computing devices.

Q29. What Do You Mean by Tensor in TensorFlow?

A tensor is a mathematical object represented as arrays of higher dimensions. Think of a n-D matrix. These arrays of data with different dimensions and ranks fed as input to the neural network are called “Tensors.”

Q30. What is the Computational Graph?

Everything in a TensorFlow is based on creating a computational graph. It has a network of nodes where each node operates. Nodes represent mathematical operations, and edges represent tensors. Since data flows in the form of a graph, it is also called a “DataFlow Graph.”

Q31. How is logistic regression done?

Logistic regression measures the relationship between the dependent variable (our label of what we want to predict) and one or more independent variables (our features) by estimating probability using its underlying logistic function (sigmoid).