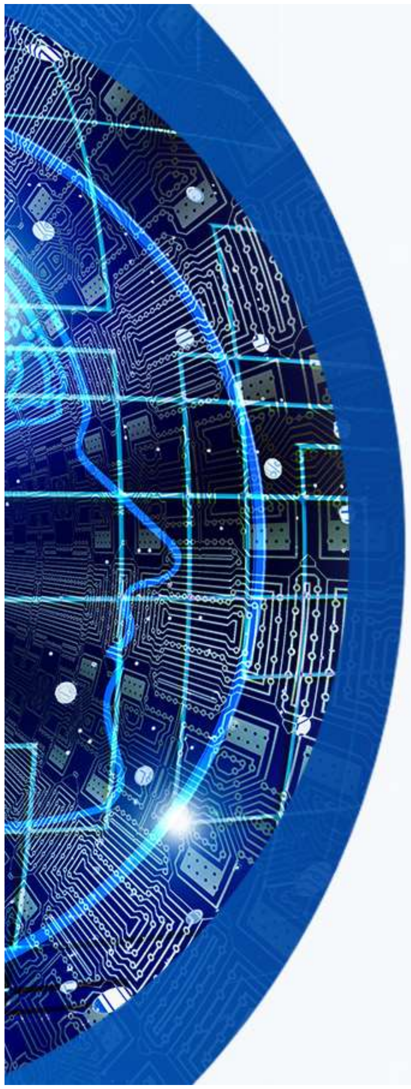
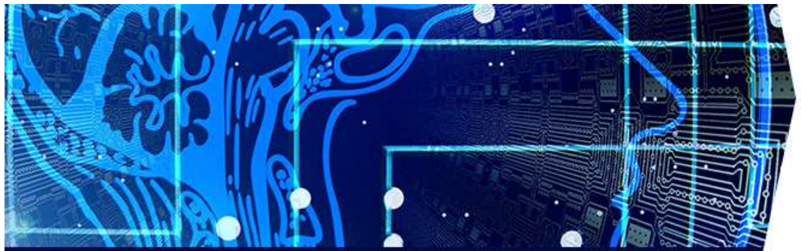
Deep Learning and Object Detection

https://www.amrita.edu/faculty/t-senthilkumar



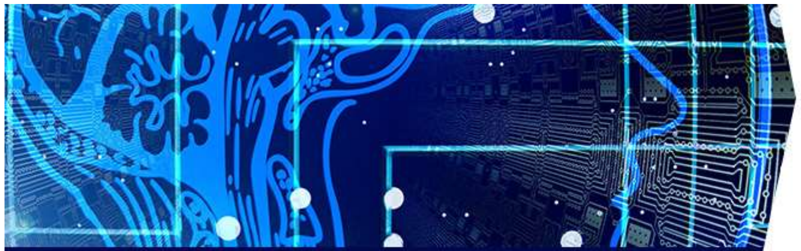
Introduction
Fundamentals
Common Neural Network Process
Convolutional Neural Network (CNN)
Recurrent Neural Networks (RNN)
Gated Recurrent Unit (GRU)
Long Short-Term Memory (LSTM)
Processing of RNN, GRU, LSTM
Comparative Study - CNN, RNN, GRU and LSTM
Algorithm Performance Analysis
Comparative Study on Activation Functions
Backpropagation, Error Calculation with gradient descent
Q&A



Introduction

Artificial Intelligence	•AI is the study of pattern recognition and mimicking human behavior.AI powered computers has started simulating the human brain work style sensation, actions, interaction, perception, and cognitive abilities.
Machine Learning	•A subset of AI that incorporates math and statistics in such a way that allows the application to learn from data.
Deep Learning	•A subset of ML that uses neural network to learn from unstructured or unlabeled data.

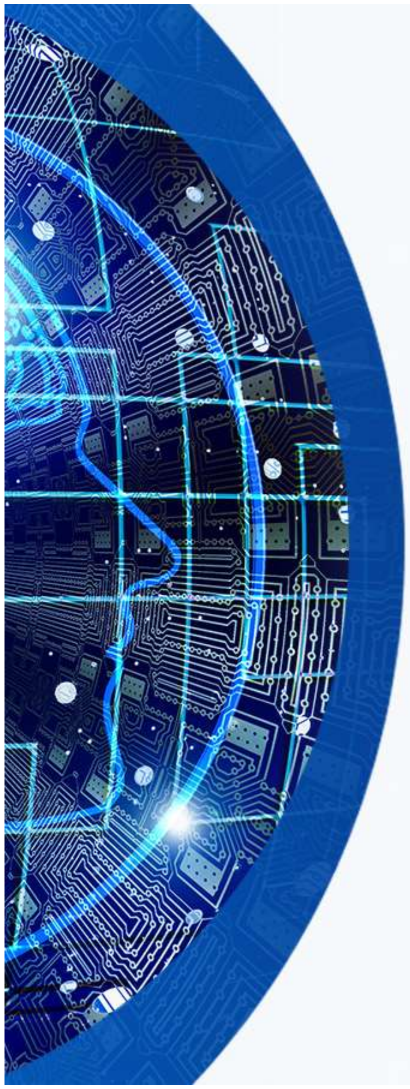
Feature	•A measurable attribute of data, determined to be valuable in the learning process.
Neural Network	•A set of algorithms inspired by neural connections in the human brain, consisting of thousands to millions of connected processing nodes.
Classificatio n	•Identifying to which category a given data point belongs.



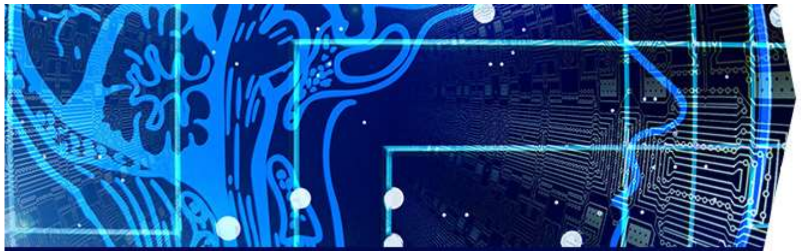
Machine Learning Vs Deep Learning

Description	Machine Learning	Deep Learning
Operation	ML algorithms get train data set and learn how to predict similar events in future which is usually as test set.	DL is mostly based on neural network which is one of ML algorithm. DL works mostly on feature selection.
Methods	Supervised and Unsupervised	Supervised and Unsupervised
Data	A few thousands, can train on lesser data	More than million, Requires large data.
Accuracy	Lesser accuracy	High accurcy
Algorithm	Linear and Logistic Regression, Support Vector Machine (SEVI), Naive Bayes(NB), K-Nearest Neighborhood (KNN), Decision Tree Random Forest, Neural Network(NN)	Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Long Short Term Memory(LSTM)
Relationship	Machine Learning is child of Artificial Intelligence and parent of Deep Learning	Convolution Neural Network (CNN), Recurrent Neural Network (RNN), Long Short Term Memory(LSTM)

What's the difference between the two?
Simply explained, both machine learning and deep learning mimic the way the human brain learns. Its main difference is therefore the type of algorithms used in each case, although deep learning is more similar to human learning as it works with neurons. Machine learning usually uses decision trees and deep learning neural networks, which are more evolved. In addition, both can learn in a supervised or unsupervised way.



Fundamentals



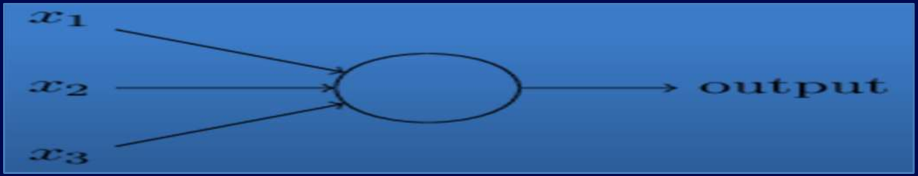
Fundamentals

Forward Propagation

Gradient Descent

Backward Propagation

Perceptron



By directly combining the input and computing the output based on a threshold value. for eg: Take x1=0, x2=1, x3=1 and setting a threshold =0. So, if x1+x2+x3>0, the output is 1 otherwise 0. You can see that in this case, the perceptron calculates the output as 1.

Next, let us add weights to the inputs. Weights give importance to an input. For example, you assign w1=2, w2=3, and w3=4 to x1, x2, and x3 respectively. To compute the output, we will multiply input with respective weights and compare with threshold value as w1*x1 + w2*x2 + w3*x3 > threshold. These weights assign more importance to x3 in comparison to x1 and x2.

Let us add bias: Each perceptron also has a bias which is thought of as how much flexible the perceptron is. It is similar to constant *b* of a linear function *y = ax + b. It allows us to move the lineup and down to fit the prediction with the data better. Without b the line will always go through the origin (0, 0) and you may get a poorer fit.* For example, a perceptron may have two inputs, in that case, it requires three weights. One for each input and one for the bias. Now linear representation of input will look like, w1*x1 + w2*x2 + w3*x3 + 1*b.

Perceptrons used for Linear. A neuron applies non-linear transformations (activation function) to the inputs and biases.

Fundamentals

Activation function

Activation Function takes the sum of weighted input (w1*x1 + w2*x2 + w3*x3 + 1*b) as an argument and returns the output of the neuron. In the below equation, we have represented 1 as x0 and b as w0

$$a = f\left(\sum_{i=0}^{N} w_i x_i\right)$$

It is used to make a non-linear transformation that allows us to fit nonlinear hypotheses or to estimate the complex functions. Like “Sigmoid”, “Tanh”, ReLu and many others.

Forward Propagation

Backward Propagation

Gradient Descent

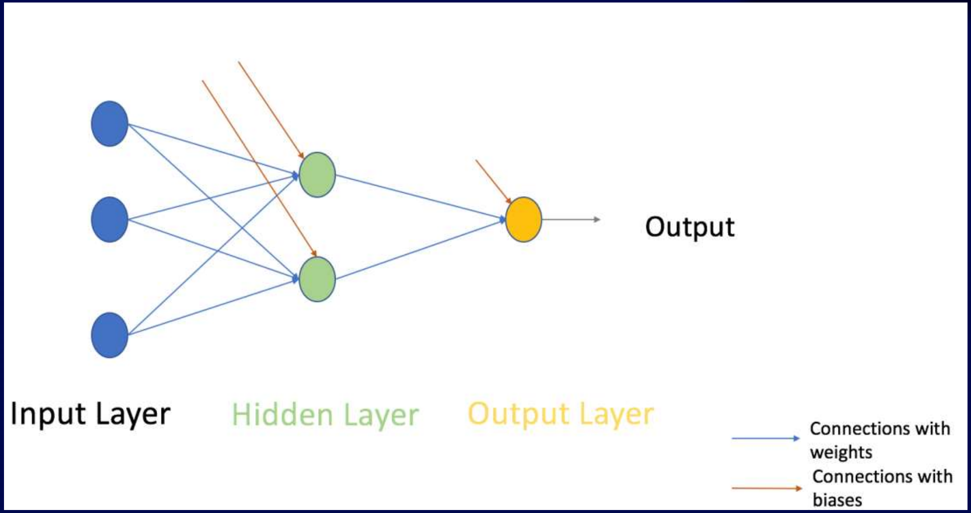
Epoch

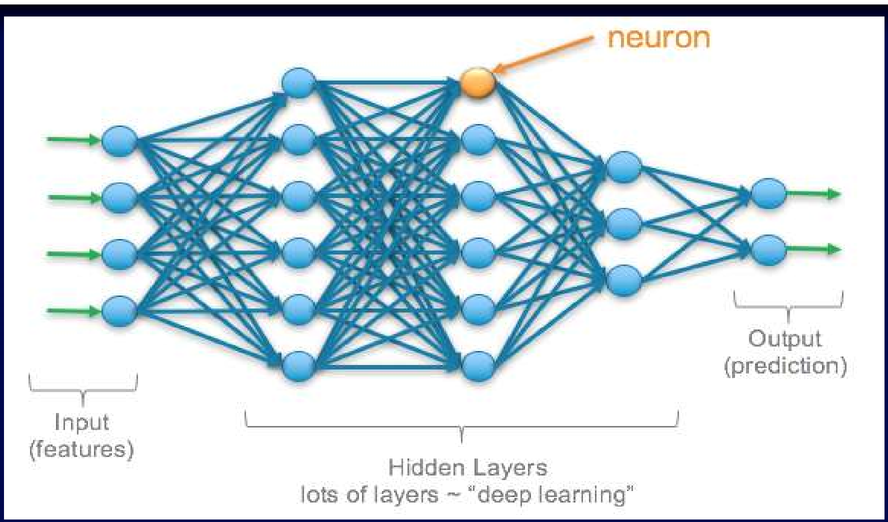
Multi-layer perceptron

The different components are:
I. Xi, XN: Inputs to the neuron. These can either be the actual observations from the input layer or an intermediate value from one Of the hidden layers.
2. Xo: Bias unit. This is a constant value added to the input of the activation function. It works similar to an intercept term and typically has +1 value.
3. w0,w1,w2,w3…wN: Weights on each input. Note that even the bias unit has a weight.

f is known as an activation function. This makes a Neural Network extremely flexible and imparts the capability to estimate complex non-linear relationships in data. It can be a gaussian function, logistic function, hyperbolic function or even a linear function in simple cases.

Fundamentals





Hidden Layer
A single hidden layer in green but in practice can contain multiple hidden layers. In addition, another point to remember in case of an MLP is that all the layers are fully connected i.e every node in a layer(except the input and the output layer) is connected to every node in the previous layer and the following layer.

Full Batch Gradient Descent

Stochastic Gradient Descent

Fundamentals

Model Parameters :
The properties of training data that will learn on its own during training by the classifier or other ML model. For example, weights and biases, or split points in Decision Tree.

Model Hyperparameters:
They are instead properties that govern the entire training process. Hyperparameters are important since they directly control behavior of the training algo, having important impact on performance of the model under training.
The variables which determines the network structure (for example, Number of Hidden Units)
The variables which determine how the network is trained (for example, Learning Rate)
Model hyperparameters are set before training (before optimizing the weights and bias).

- Learning Rate
- Number of Epochs
- Hidden Layers
- Hidden Units
- Activations Functions

PARAMETERS	HYPERPARAMETER
They are required for making predictions	They are required for estimating the model parameters
They are estimated by optimization algorithms(Gradient Descent, Adam, Adagrad)	They are estimated by hyperparameter tuning
They are not set manually	They are set manually
The final parameters found after training will decide how the model will perform on unseen data	The choice of hyperparameters decide how efficient the training is. In gradient descent the learning rate decide how efficient and accurate the optimization process is in estimating the parameters

Fundamentals

Underfitting refers to a model that can neither model the training dataset nor generalize to new dataset. An underfit machine learning model is not a suitable model and will be obvious as it will have poor performance on the training dataset.

Overfitting is that a machine learning model can't generalize or fit well on unseen dataset. The model's error on the testing or validation dataset is much greater than the error on training dataset. The model / function corresponds too closely to a dataset. As a result, overfitting may fail to fit additional data, and this may affect the accuracy of predicting future observations.
A model learns the detail and noise in the training dataset to the extent that it negatively impacts the performance of the model on a new dataset.

Methods to prevent Overfitting

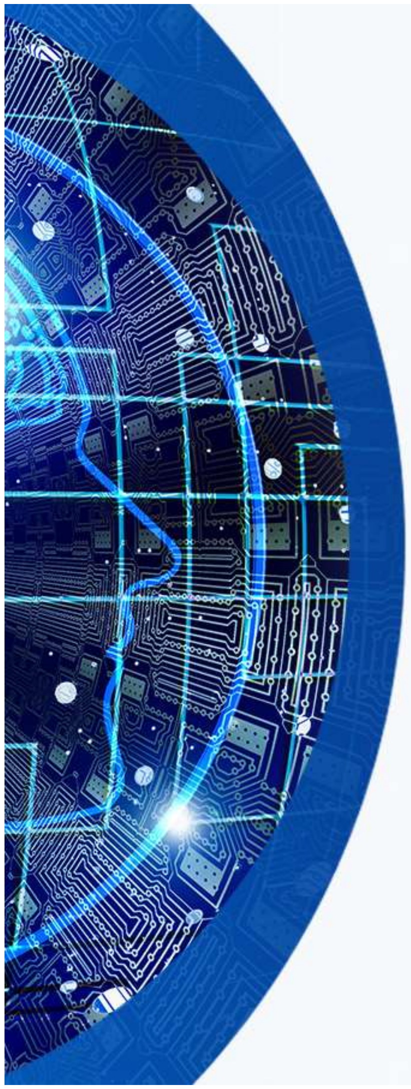
Cross-validation:
Initial training data to generate multiple mini train-test splits. Use these splits to tune the model. Tune hyperparameters with only original training dataset. This allows to keep the test dataset as a truly unseen dataset.

More training data:
More data into the model, it will be unable to overfit all the samples and will be forced to generalize to obtain results, also increases accuracy.

Data augmentation:
It makes a data sample look slightly different every time it is processed by the model. The process makes each data set appear unique to the model and prevents the model from learning the characteristics of the data sets.

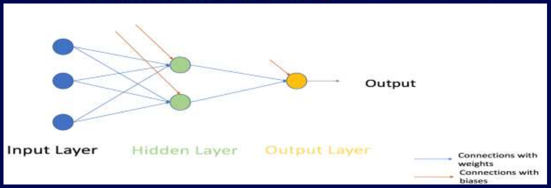
Reduce Complexity or Data Simplification:
Reduce overfitting by decreasing the complexity of the model. Reduce the number of parameters in a Neural Networks, and using dropout on a Neural Networks.

Ensembling:
Machine learning methods for combining predictions from multiple separate models.
Boosting attempts to improve the predictive flexibility of simple models.
Bagging attempts to reduce the chance of overfitting complex models.



Step-by-Step Procedure of Neural Network Operation Methodology

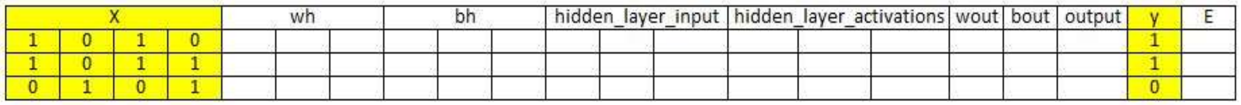
Visualization of steps for Neural Network Operation



Let's look at the step by step building methodology of Neural Network (MLP with one hidden layer, similar to above-shown architecture). At the output layer, we have only one neuron as we are solving a binary classification problem (predict 0 or 1). We could also have two neurons for predicting each of both classes.

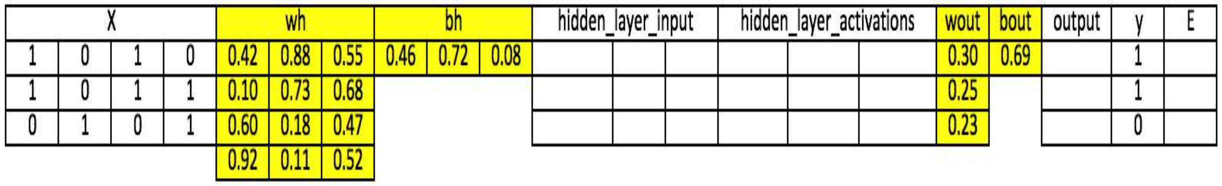
Yellow filled cells represent current active cell. Orange cell represents the input used to populate the values of the current cell

0.) We take input and output
X as an input matrix
y as an output matrix



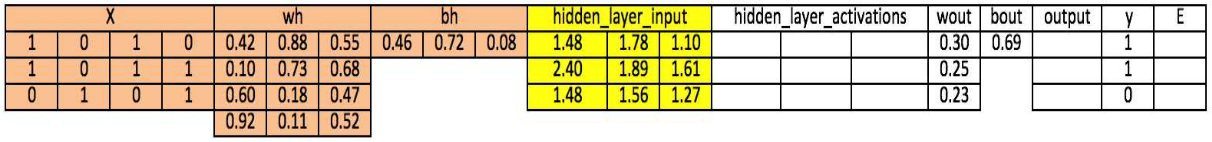
X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0																1	
1	0	1	1																1	
0	1	0	1																0	

1.) Then we initialize weights and biases with random values (one-time initiation. Next iteration, use updated weights, and biases). Let us define:
wh as a weight matrix to the hidden layer
bh as bias matrix to the hidden layer
wout as a weight matrix to the output layer
bout as bias matrix to the output layer

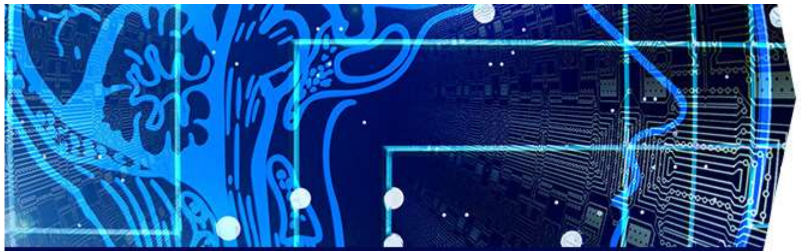


X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0	0.42	0.88	0.55	0.46	0.72	0.08							0.30	0.69		1	
1	0	1	1	0.10	0.73	0.68										0.25			1	
0	1	0	1	0.60	0.18	0.47										0.23			0	
				0.92	0.11	0.52														

2.) Then we take matrix dot product of input and weights assigned to edges between the input and hidden layer then add biases of the hidden layer neurons to respective inputs, this is known as linear transformation:
hidden_layer_input= matrix_dot_product(X,wh) + bh

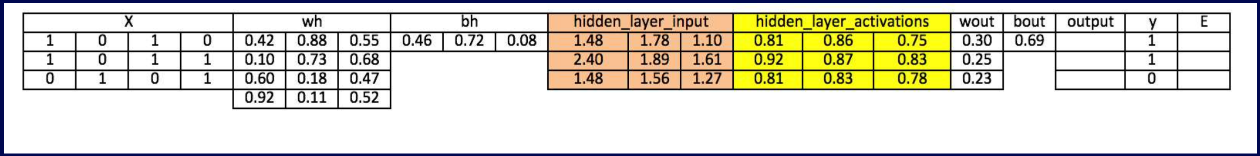


X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0	0.42	0.88	0.55	0.46	0.72	0.08	1.48	1.78	1.10				0.30	0.69		1	
1	0	1	1	0.10	0.73	0.68				2.40	1.89	1.61				0.25			1	
0	1	0	1	0.60	0.18	0.47				1.48	1.56	1.27				0.23			0	
				0.92	0.11	0.52														



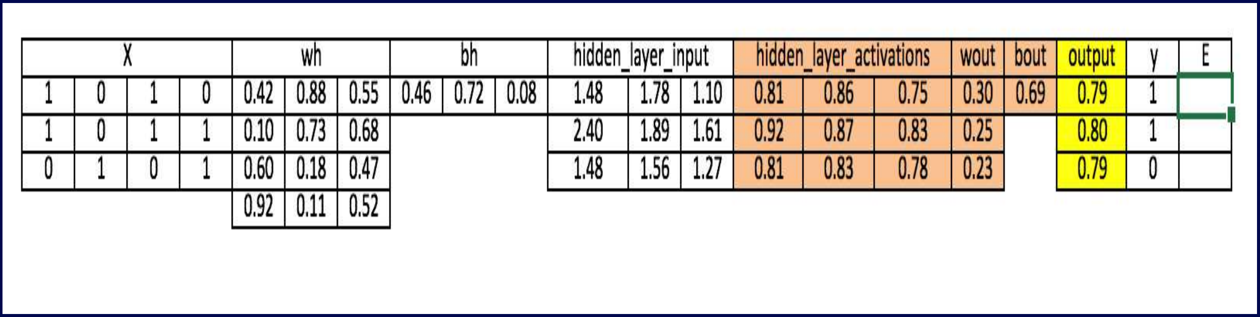
Visualization of steps for Neural Network Operation

3) Perform non-linear transformation using an activation function (Sigmoid). Sigmoid will return the output as 1/(1 + exp(-x)).
hiddenlayer_activations = sigmoid(hidden_layer_input)



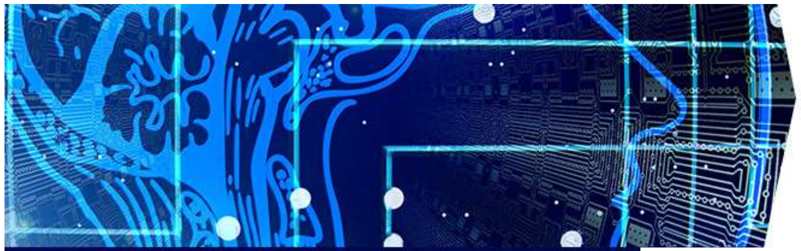
X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0	0.42	0.88	0.55	0.46	0.72	0.08	1.48	1.78	1.10	0.81	0.86	0.75	0.30	0.69		1	
1	0	1	1	0.10	0.73	0.68				2.40	1.89	1.61	0.92	0.87	0.83	0.25			1	
0	1	0	1	0.60	0.18	0.47				1.48	1.56	1.27	0.81	0.83	0.78	0.23			0	
				0.92	0.11	0.52														

4.) Then perform a linear transformation on hidden layer activation (take matrix dot product with weights and add a bias of the output layer neuron) then apply an activation function (again used sigmoid, but can use any activation function depending upon task) to predict the output
output_layer_input = matrix_dot_product (hiddenlayer_activations * wout) + bout
output = sigmoid(output_layer_input)



X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0	0.42	0.88	0.55	0.46	0.72	0.08	1.48	1.78	1.10	0.81	0.86	0.75	0.30	0.69	0.79	1	
1	0	1	1	0.10	0.73	0.68				2.40	1.89	1.61	0.92	0.87	0.83	0.25		0.80	1	
0	1	0	1	0.60	0.18	0.47				1.48	1.56	1.27	0.81	0.83	0.78	0.23		0.79	0	
				0.92	0.11	0.52														

All the above steps are known as “Forward Propagation“



Visualization of steps for Neural Network Operation

5.) Compare prediction with actual output and calculate the gradient of error (Actual – Predicted)

Error is the mean square loss = ((Y-t)^2)/2

E = y – output

X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0	0.42	0.88	0.55	0.46	0.72	0.08	1.48	1.78	1.10	0.81	0.86	0.75	0.30	0.69	0.79	1	0.21
1	0	1	1	0.10	0.73	0.68				2.40	1.89	1.61	0.92	0.87	0.83	0.25		0.80	1	0.20
0	1	0	1	0.60	0.18	0.47				1.48	1.56	1.27	0.81	0.83	0.78	0.23		0.79	0	-0.79
				0.92	0.11	0.52														

6.) Compute the slope/gradient of hidden and output layer neurons (To find the slope, calculate the derivatives of non-linear activations x at each layer for each neuron). The gradient of sigmoid can be returned as x * (1 – x).

slope_output_layer = derivatives_sigmoid(output)

slope_hidden_layer = derivatives_sigmoid(hiddenlayer_activations)

X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0	0.42	0.88	0.55	0.46	0.72	0.08	1.48	1.78	1.10	0.81	0.86	0.75	0.30	0.69	0.79	1	0.21
1	0	1	1	0.10	0.73	0.68				2.40	1.89	1.61	0.92	0.87	0.83	0.25		0.80	1	0.20
0	1	0	1	0.60	0.18	0.47				1.48	1.56	1.27	0.81	0.83	0.78	0.23		0.79	0	-0.79
				0.92	0.11	0.52														

Slope hidden layer		
0.15	0.12	0.19
0.08	0.11	0.14
0.15	0.14	0.17

Slope Output
0.17
0.16
0.17

7.) Then compute change factor(delta) at the output layer, dependent on the gradient of error multiplied by the slope of output layer activation

d_output = E * slope_output_layer

X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0	0.42	0.88	0.55	0.46	0.72	0.08	1.48	1.78	1.10	0.81	0.86	0.75	0.30	0.69	0.79	1	0.21
1	0	1	1	0.10	0.73	0.68				2.40	1.89	1.61	0.92	0.87	0.83	0.25		0.80	1	0.20
0	1	0	1	0.60	0.18	0.47				1.48	1.56	1.27	0.81	0.83	0.78	0.23		0.79	0	-0.79
				0.92	0.11	0.52														

Slope hidden layer			error at hidden layer		
0.15	0.12	0.19	0.010	0.009	0.008
0.08	0.11	0.14	0.010	0.008	0.008
0.15	0.14	0.17	-0.039	-0.033	-0.031

Slope Output
0.17
0.16
0.17

E
0.21
0.20
-0.79

delta output
0.04
0.03
-0.13

8.) At this step, the error will propagate back into the network which means error at the hidden layer. For this, take the dot product of the output layer delta with the weight parameters of edges between the hidden and output layer (wout.T).

Error_at_hidden_layer = matrix_dot_product(d_output, wout.Transpose)

Slope hidden layer			error at hidden layer		
0.15	0.12	0.19	0.010	0.009	0.008
0.08	0.11	0.14	0.010	0.008	0.008
0.15	0.14	0.17	-0.039	-0.033	-0.031

Slope Output
0.17
0.16
0.17

E
0.21
0.20
-0.79

delta output
0.04
0.03
-0.13

9.) Compute change factor(delta) at hidden layer, multiply the error at hidden layer with slope of hidden layer activation

*d_hiddenlayer = Error_at_hidden_layer * slope_hidden_layer*

10.) Then update weights at the output and hidden layer: The weights in the network can be updated from the errors calculated for training example(s).

wout = wout + matrix_dot_product(hiddenlayer_activations.Transpose, d_output)*learning_rate

wh = wh + matrix_dot_product(X.Transpose,d_hiddenlayer)*learning_rate

learning_rate: The amount that weights are updated is controlled by a configuration parameter called the learning rate)

11.) Finally, update biases at the output and hidden layer: The biases in the network can be updated from the aggregated errors at that neuron.

bias at output_layer =bias at output_layer + sum of delta of output_layer at row-wise * learning_rate

bias at hidden_layer =bias at hidden_layer + sum of delta of output_layer at row-wise * learning_rate

*bh = bh + sum(d_hiddenlayer, axis=0) * learning_rate*

*bout = bout + sum(d_output, axis=0)*learning_rate*

Above, you can see that there is still a good error not close to the actual target value because we have completed only one training iteration. If we will train the model multiple times then it will be a very close actual outcome. I have completed thousands iteration and my result is close to actual target values ([[0.98032096] [0.96845624] [0.04532167]]).

Steps from 5 to 11 are known as "Backward Propagation "One forward and backward propagation iteration is considered as one training cycle

X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0	0.42	0.88	0.55	0.46	0.72	0.08	1.48	1.78	1.10	0.81	0.86	0.75	0.30	0.69	0.79	1	0.21
1	0	1	1	0.10	0.73	0.68				2.40	1.89	1.61	0.92	0.87	0.83	0.25		0.80	1	0.20
0	1	0	1	0.60	0.18	0.47				1.48	1.56	1.27	0.81	0.83	0.78	0.23		0.79	0	-0.79
				0.92	0.11	0.52														

Slope hidden layer			error at hidden layer		
0.15	0.12	0.19	0.010	0.009	0.008
0.08	0.11	0.14	0.010	0.008	0.008
0.15	0.14	0.17	-0.039	-0.033	-0.031

Slope Output
0.17
0.16
0.17

E
0.21
0.20
-0.79

delta hidden layer		
0.002	0.001	0.002
0.001	0.001	0.001
-0.006	-0.005	-0.005

delta output
0.04
0.03
-0.13

X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0	0.42	0.88	0.55	0.46	0.72	0.08	1.48	1.78	1.10	0.81	0.86	0.75	0.29	0.69	0.79	1	0.21
1	0	1	1	0.10	0.73	0.68				2.40	1.89	1.61	0.92	0.87	0.83	0.25		0.80	1	0.20
0	1	0	1	0.60	0.18	0.47				1.48	1.56	1.27	0.81	0.83	0.78	0.23		0.79	0	-0.79
				0.92	0.11	0.51														

Slope hidden layer			error at hidden layer		
0.15	0.12	0.19	0.010	0.009	0.008
0.08	0.11	0.14	0.010	0.008	0.008
0.15	0.14	0.17	-0.039	-0.033	-0.031

Slope Output
0.17
0.16
0.17

E
0.21
0.20
-0.79

Learning Rate	0.1

delta hidden layer		
0.002	0.001	0.002
0.001	0.001	0.001
-0.006	-0.005	-0.005

delta output
0.035
0.033
-0.131

X				wh			bh			hidden_layer_input			hidden_layer_activations			wout	bout	output	y	E
1	0	1	0	0.42	0.88	0.55	0.46	0.72	0.08	1.48	1.78	1.10	0.81	0.86	0.75	0.29	0.68	0.79	1	0.21
1	0	1	1	0.10	0.73	0.68				2.40	1.89	1.61	0.92	0.87	0.83	0.25		0.80	1	0.20
0	1	0	1	0.60	0.18	0.47				1.48	1.56	1.27	0.81	0.83	0.78	0.23		0.79	0	-0.79
				0.92	0.11	0.51														

Slope hidden layer			error at hidden layer		
0.15	0.12	0.19	0.010	0.009	0.008
0.08	0.11	0.14	0.010	0.008	0.008
0.15	0.14	0.17	-0.039	-0.033	-0.031

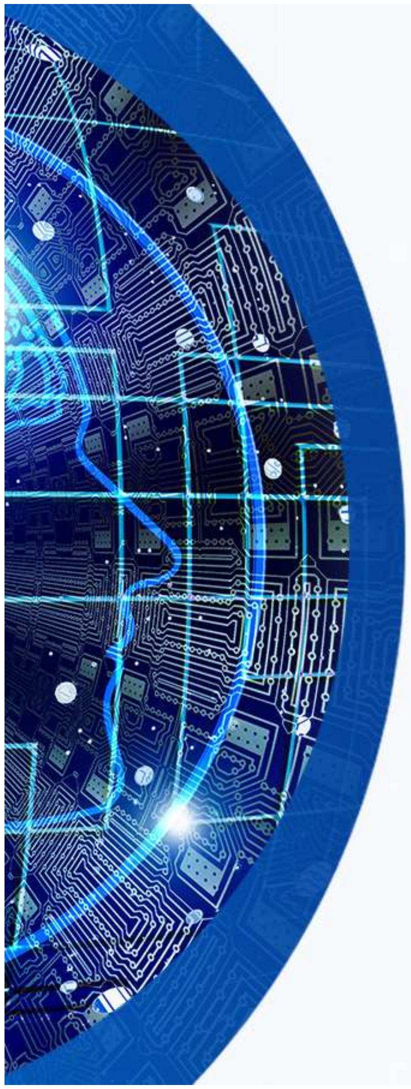
Slope Output
0.17
0.16
0.17

E
0.21
0.20
-0.79

Learning Rate	0.1

delta hidden layer		
0.002	0.001	0.002
0.001	0.001	0.001
-0.006	-0.005	-0.005

delta output
0.035
0.033
-0.131



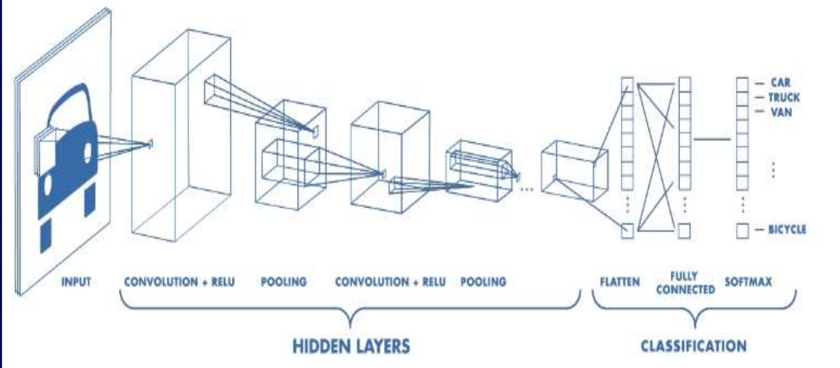
Convolutional Neural Network

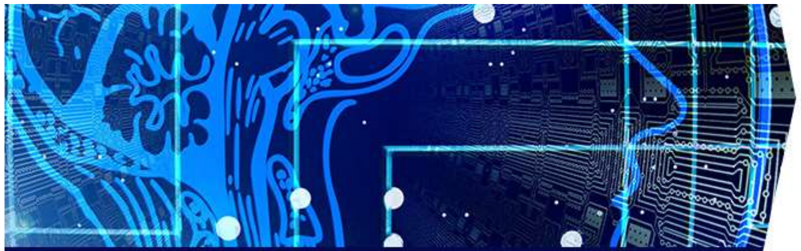
Convolutional Neural Network (CNN) Architecture

Advantages of Convolution Neural Network (CNN):
•CNN learns the filters automatically without mentioning it explicitly. These filters help in extracting the right and relevant features from the input data.
•CNN captures the from an image. Spatial features refer to the arrangement of pixels and the relationship between them in an image. They help us in identifying the object accurately, the location of an object, as well as its relation with other objects in an image.
•CNN also follows the concept of parameter sharing. A single filter is applied across different parts of an input to produce a feature map.

The ConvNet architecture consists of three types of layers: Convolutional Layer, Pooling Layer, and Fully-Connected Layer.

INPUT layer : hold the input image as a 3-D array of pixel values.
CONV layer : Will compute the dot product between the kernel and sub-array of an input image same size as a kernel. Then it'll sum all the values resulted from the dot product and this will be the single pixel value of an output image. This process is repeated until the whole input image is covered and for all the kernels.
RELU layer : will apply an activation function max(0,x) on all the pixel values of an output image.
POOL layer : Perform down sampling along the width and height of an image resulting in reducing the dimension of an image.
FC (Fully-Connected) layer : Compute the class score for each of the classification category.

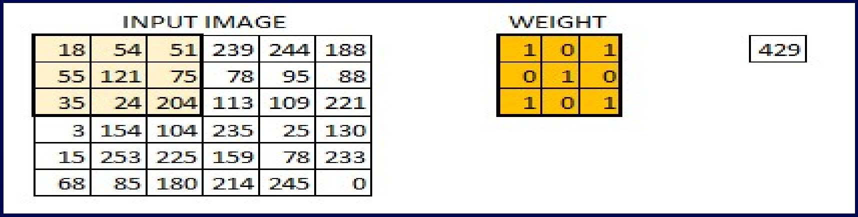




Convolutional Neural Network Step-by-Step Process

The Convolution Layer

Consider we have an image of size 6*6.
We define a weight matrix which extracts certain features from the images



INPUT IMAGE

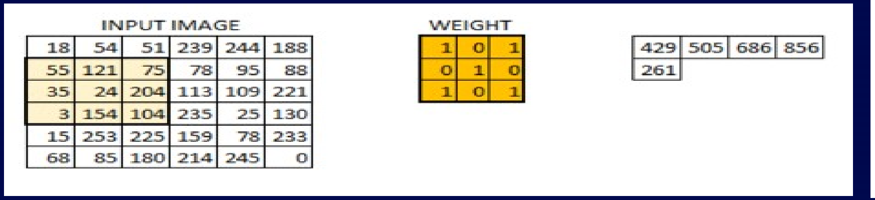
18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

WEIGHT

1	0	1
0	1	0
1	0	1

429

We have initialized the weight(Filter) as a 3*3 matrix. This weight shall now run across the image such that all the pixels are covered at least once, to give a convolved output. The value 429 above, is obtained by the adding the values obtained by element wise multiplication of the weight matrix and the highlighted 3*3 part of the input image.



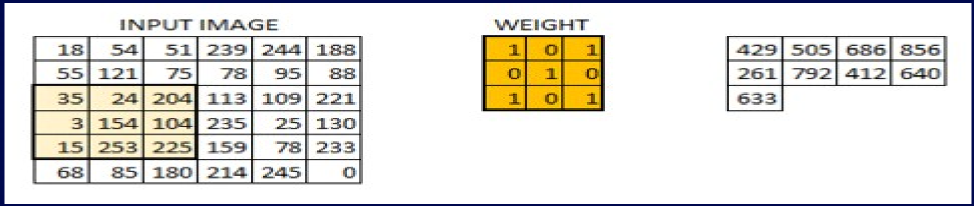
INPUT IMAGE

18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

WEIGHT

1	0	1
0	1	0
1	0	1

429	505	686	856
261			



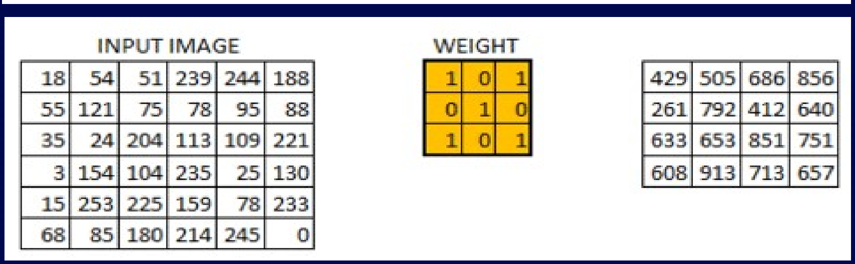
INPUT IMAGE

18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

WEIGHT

1	0	1
0	1	0
1	0	1

429	505	686	856
261	792	412	640
633			



INPUT IMAGE

18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

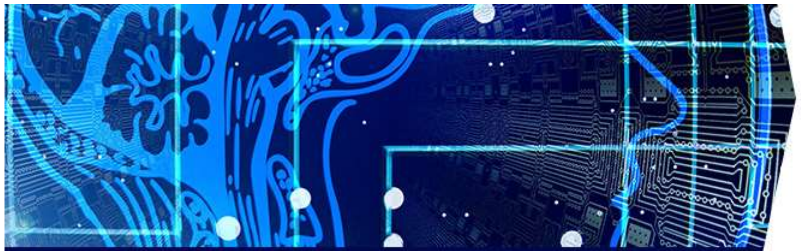
WEIGHT

1	0	1
0	1	0
1	0	1

429	505	686	856
261	792	412	640
633	653	851	751
608	913	713	657

The 6*6 image is now converted into a 4*4 image. Pixel values are used again when the weight matrix moves along the image. This basically enables parameter sharing in a convolutional neural network

weights are learnt to extract features from the original image which help the network in correct prediction

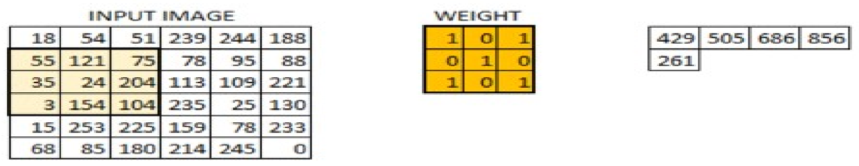


Convolutional Neural Network

Stride

The filter or the weight matrix, was moving across the entire image moving **n** pixel at a time, n is stride.

Stride = 1



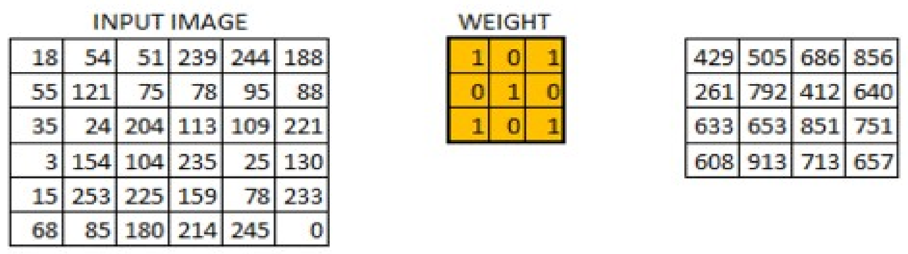
INPUT IMAGE

18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

WEIGHT

1	0	1
0	1	0
1	0	1

429	505	686	856
261			



INPUT IMAGE

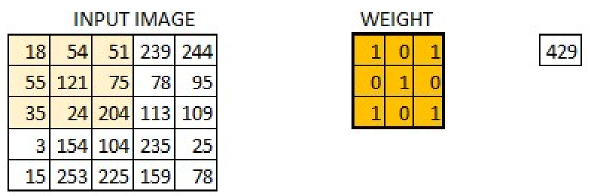
18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

WEIGHT

1	0	1
0	1	0
1	0	1

429	505	686	856
261	792	412	640
633	653	851	751
608	913	713	657

Stride = 2



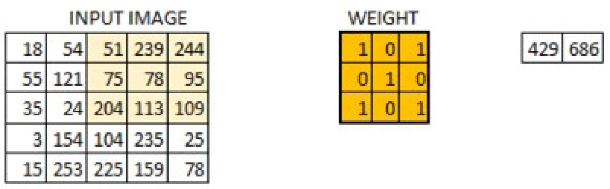
INPUT IMAGE

18	54	51	239	244
55	121	75	78	95
35	24	204	113	109
3	154	104	235	25
15	253	225	159	78

WEIGHT

1	0	1
0	1	0
1	0	1

429



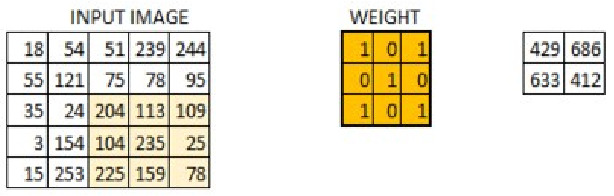
INPUT IMAGE

18	54	51	239	244
55	121	75	78	95
35	24	204	113	109
3	154	104	235	25
15	253	225	159	78

WEIGHT

1	0	1
0	1	0
1	0	1

429	686



INPUT IMAGE

18	54	51	239	244
55	121	75	78	95
35	24	204	113	109
3	154	104	235	25
15	253	225	159	78

WEIGHT

1	0	1
0	1	0
1	0	1

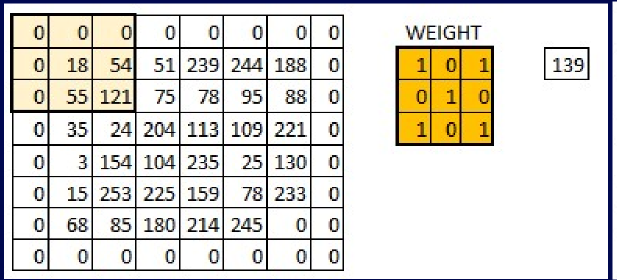
429	686
633	412

The size of image keeps on reducing as we increase the stride value.
This is defined as hyperparameter, as to how we would want the weight matrix to move across the image. If the weight matrix moves 1 pixel at a time, we call it as a stride of 1.

Convolutional Neural Network

Padding

Padding the input image with zeros across maintains the output image size from stride. We can also add more than one layer of zeros around the image in case of higher stride values.

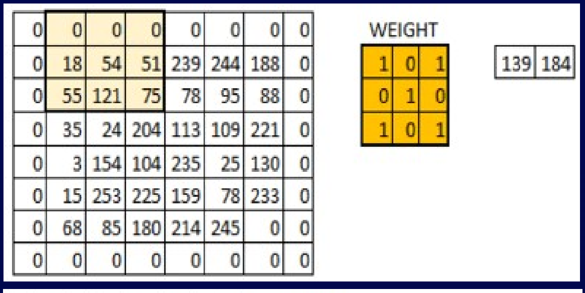


0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

WEIGHT

1	0	1
0	1	0
1	0	1

139

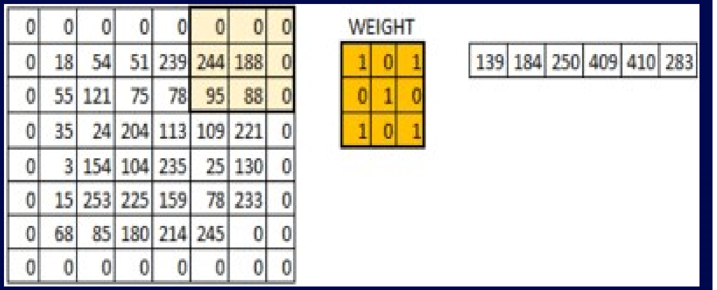


0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

WEIGHT

1	0	1
0	1	0
1	0	1

139	184

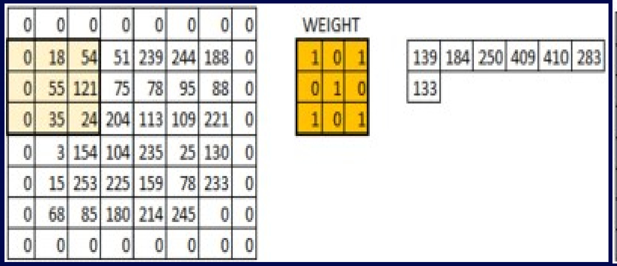


0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

WEIGHT

1	0	1
0	1	0
1	0	1

139	184	250	409	410	283

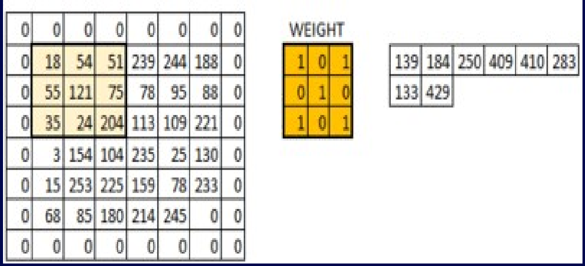


0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

WEIGHT

1	0	1
0	1	0
1	0	1

139	184	250	409	410	283
133					

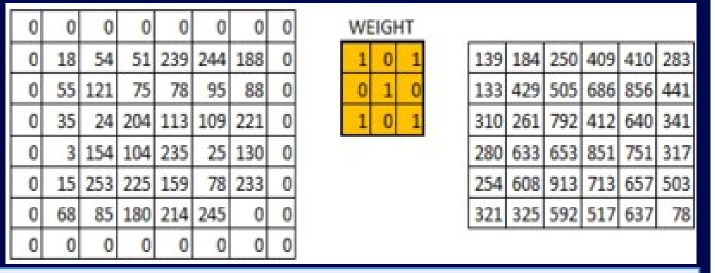


0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

WEIGHT

1	0	1
0	1	0
1	0	1

139	184	250	409	410	283
133	429				



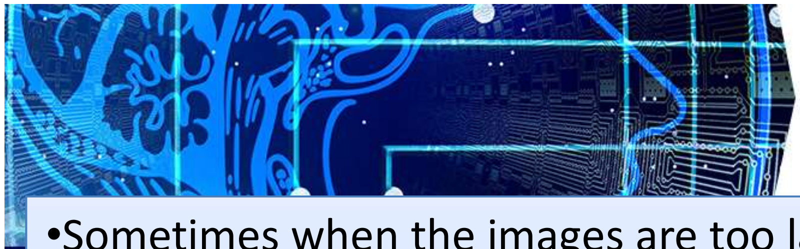
0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

WEIGHT

1	0	1
0	1	0
1	0	1

139	184	250	409	410	283
133	429	505	686	856	441
310	261	792	412	640	341
280	633	653	851	751	317
254	608	913	713	657	503
321	325	592	517	637	78

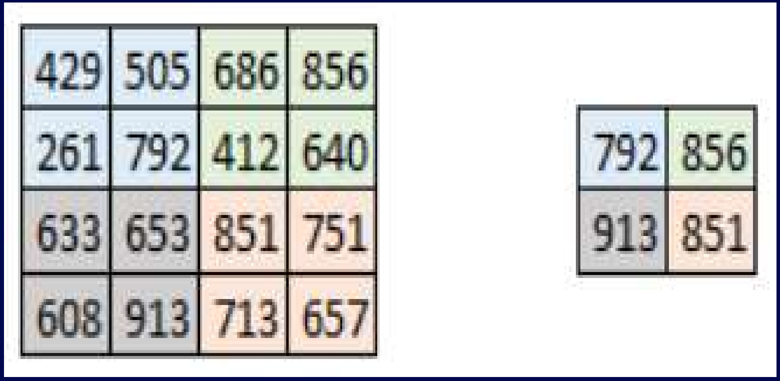
The initial shape of the image is retained after we padded the image with a zero. This is known as same padding since the output image has the same size as the input, which means that we considered only the valid pixels of the input image. The middle 4*4 pixels would be the same. Here we have retained more information from the borders and have also preserved the size of the image.



Convolutional Neural Network

Pooling

•Sometimes when the images are too large, we would need to reduce the number of trainable parameters.
•It is then desired to periodically introduce pooling layers between subsequent convolution layers.
•Pooling is done for the sole purpose of reducing the spatial size of the image.
•Pooling is done independently on each depth dimension, therefore the depth of the image remains unchanged.
•The most common form of pooling layer generally applied is the max pooling.



429	505	686	856
261	792	412	640
633	653	851	751
608	913	713	657

792	856
913	851

Here stride as 2, while pooling size also as 2.
The max operation is applied to each depth dimension of the convolved output.
The 4*4 convolved output has become 2*2 after the max pooling operation. Convoluted image and applied max pooling reduce the parameters.

Convolutional Neural Network

Output Dimensions & Output Layer

Output dimensions :

Filters / Depth: The number of filters The depth of the output volume will be equal to the number of filter applied. The depth of the activation map will be equal to the number of filters.

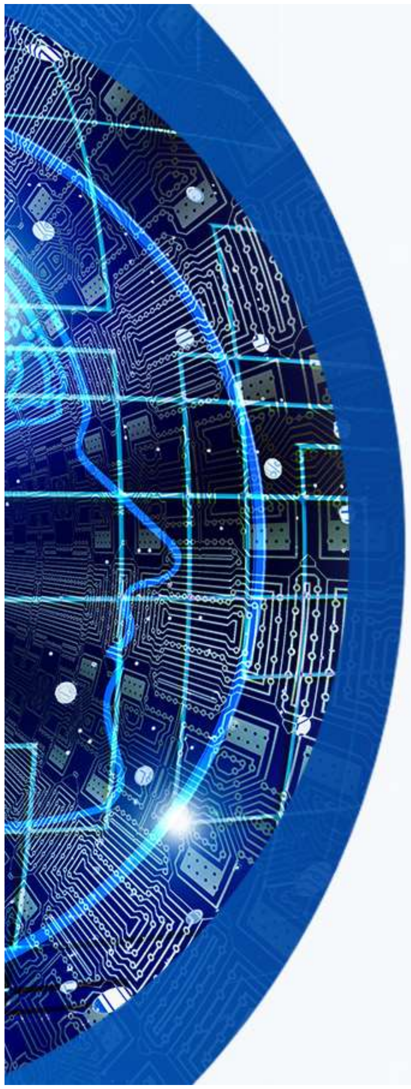
Stride: For the stride of one we move across and down a single pixel. With higher stride values, we move large number of pixels at a time and hence produce smaller output volumes.

Zero padding: This helps us to preserve the size of the input image. If a single zero padding is added, a single stride filter movement would retain the size of the original image.

Formula to calculate the output dimensions.

The spatial size of the output image = ([W-F+2P]/S)+1.
W is the input volume size
F is the size of the filter
P is the number of padding applied
S is the number of strides.
Suppose we have an input image of size 32*32*3, we apply 10 filters of size 3*3*3, with single stride and no zero padding.
Here W=32, F=3, P=0 and S=1. The output depth will be equal to the number of filters applied i.e. 10. The size of the output volume will be ([32-3+0]/1)+1 = 30. Therefore the output volume will be 30*30*10.

Output layer:
The convolution and pooling layers would only be able to extract features and reduce the number of parameters from the original images. However, to generate the final output we need to apply a fully connected layer to generate an output equal to the number of classes we need. The output layer has a loss function like categorical cross-entropy, to compute the error in prediction. Once the forward pass is complete the back propagation begins to update the weight and biases for error and loss reduction.



Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNN) Architecture

Recurrent Neural Networks (RNN)

A recurrent neuron stores the **state of a previous input** and **combines with the current input** thereby **preserving some relationship of the current input with the previous input.** The input layer receives the input, the first hidden layer activations are applied and then these activations are sent to the next hidden layer, and successive activations through the layers to produce the output. Each hidden layer is characterized by its own weights and biases.
RNNs can be used for mapping inputs to outputs of varying types, lengths and are fairly generalized in their application.

The formula for the current state:

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

ht is the new state, **ht-1** is the previous state, **xt** is the current input. We now have a state of the previous input instead of the input itself, because the input neuron would have applied the transformations on our previous input. So each successive input is called as a time step.

Simple RNN, the activation function is tanh and the equation for the state at

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

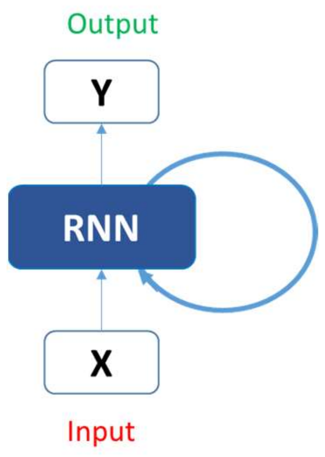
Whh the weight at recurrent neuron, **Wxh** is the weight at input neuron
The Recurrent neuron is just taking the immediate previous state into consideration. For longer sequences the equation can involve multiple such states. Once the final state is calculated we can go on to produce the output

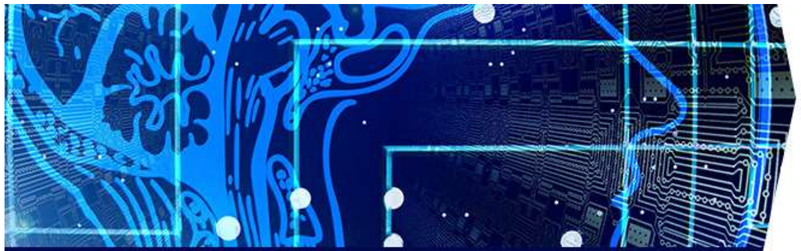
Once the current state is calculated we can calculate

$$y_t = W_{hy}h_t$$

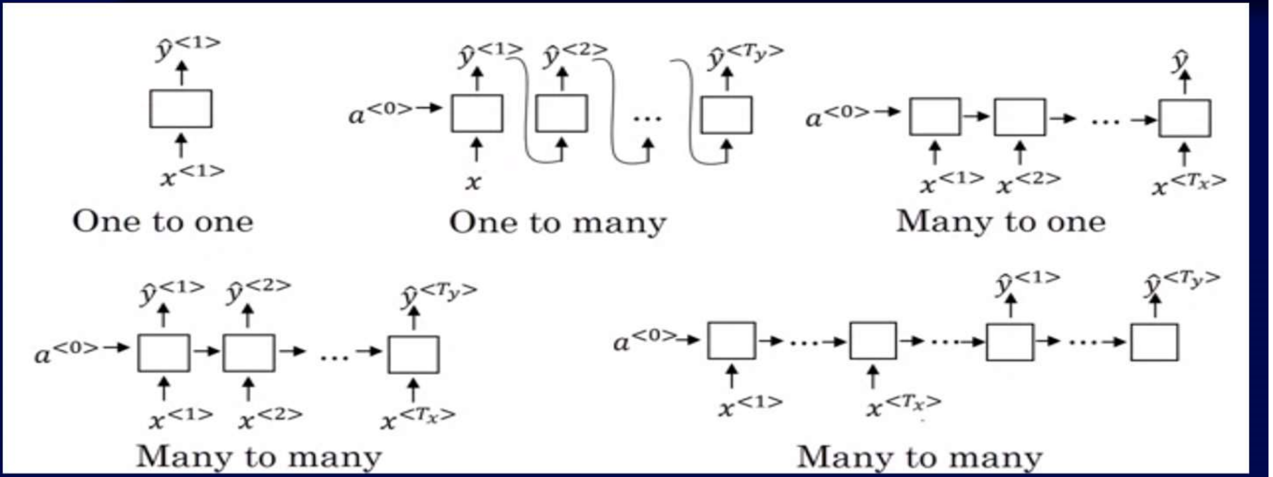
Yt Final output
ht is the new state
RNN models used in the fields of natural language processing and speech recognition

Recurrent Neuron





Recurrent Neural Networks (RNN) Architecture



$\hat{y}^{<1>}$
$x^{<1>}$
One to one
$a^{<0>}$
$\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<T_y>}$
x
One to many
$\hat{y}$
$x^{<1>}$ $x^{<2>}$ $x^{<T_x>}$
Many to one
$\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<T_y>}$
$x^{<1>}$ $x^{<2>}$ $x^{<T_x>}$
Many to many
$\hat{y}^{<1>}$ $\hat{y}^{<T_y>}$
$x^{<1>}$ $x^{<T_x>}$
Many to many

One to One RNN (Tx=Ty=1) is the most basic and traditional type of Neural network giving a **single output for a single input**, as can be seen in the above image.

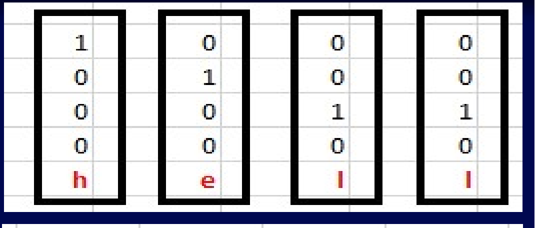
One to Many (Tx=1,Ty>1) - that give **multiple output for a single input**. A basic example application is **Music generation**. Generate a music piece(multiple output) from a single musical note(single input).

Many-to-one RNN architecture **(Tx>1,Ty=1)** Sentiment analysis model as a common example. Take for example The **Twitter sentiment analysis model**. In that model, a text input (words as multiple inputs) gives its fixed sentiment (single output). Another example is **movie ratings model** that takes review texts as input to provide a rating to a movie that may **range from 1 to 5.**

Many-to-Many RNN (Tx>1,Ty>1) Architecture takes multiple input and gives multiple output. **1.Tx=Ty:** Input and output layers have the **same size**. This can be also understood as every input having a output, and a common application can be found in **Named-entity Recognition**. **2.Tx!=Ty:** Input and output layers are of **different size**. Example is **Machine Translation**. For example, "I Love you", the 3 words of English translates to only 2 in Spanish, "te amo". Machine translation models are capable of returning words more or less than the input string because of a non-equal Many-to-Many RNN architecture works in the background.

Recurrent Neural Networks (RNN) Step-by-Step Process

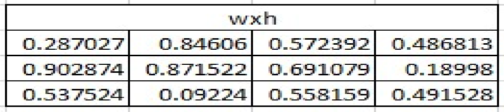
Let's take a character level RNN where we have a **word "Hello".**
So we provide the first 4 letters i.e. h,e,l,l and let the network to predict the last letter i.e.'o'.
So here the vocabulary of the task is just 4 letters {h,e,l,o}.



1	0	0	0
0	1	0	0
0	0	1	1
0	0	0	0
h	e	l	l

The inputs are one hot encoded. Our entire vocabulary is {h,e,l,o} and hence we can easily one hot encode the inputs.

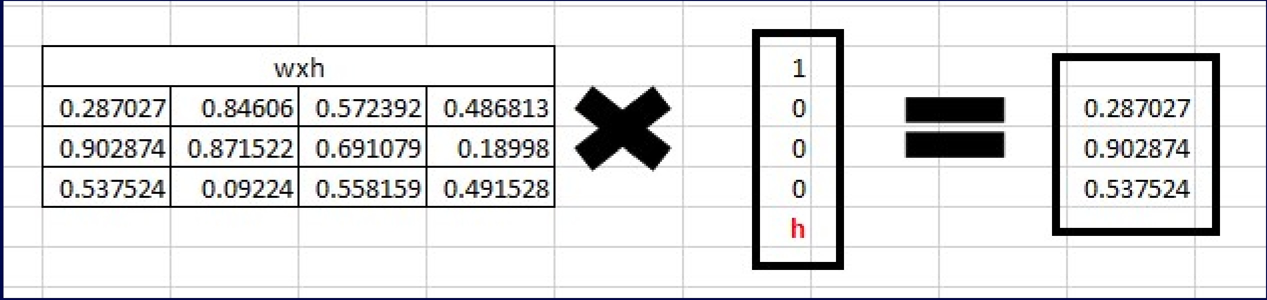
Now the input neuron would transform the input to the hidden state using the weight wxh. We have randomly initialized the weights as a 3*4 matrix –



wxh			
0.287027	0.84606	0.572392	0.486813
0.902874	0.871522	0.691079	0.18998
0.537524	0.09224	0.558159	0.491528

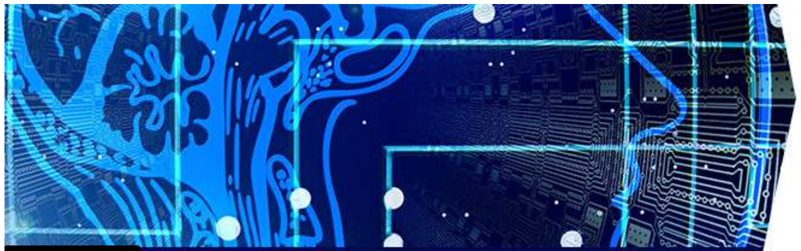
Step 1:

Now for the letter "h", for the the hidden state we would need Wxh*Xt. By matrix multiplication, we get it as –



$$\begin{array}{cccc} & \text{wxh} & & \\ 0.287027 & 0.84606 & 0.572392 & 0.486813 \\ 0.902874 & 0.871522 & 0.691079 & 0.18998 \\ 0.537524 & 0.09224 & 0.558159 & 0.491528 \end{array} \times \begin{array}{c} 1 \\ 0 \\ 0 \\ 0 \\ \text{h} \end{array} = \begin{array}{c} 0.287027 \\ 0.902874 \\ 0.537524 \end{array}$$

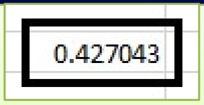
In real case scenarios involving natural language processing, the vocabularies include the words in entire wikipedia database, or all the words in a language. Here for simplicity we have taken a very small set of vocabulary.



Recurrent Neural Networks (RNN) Step-by-Step Process

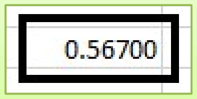
Step 2:

Now moving to the recurrent neuron, we have Whh as the weight which is a 1*1 matrix as



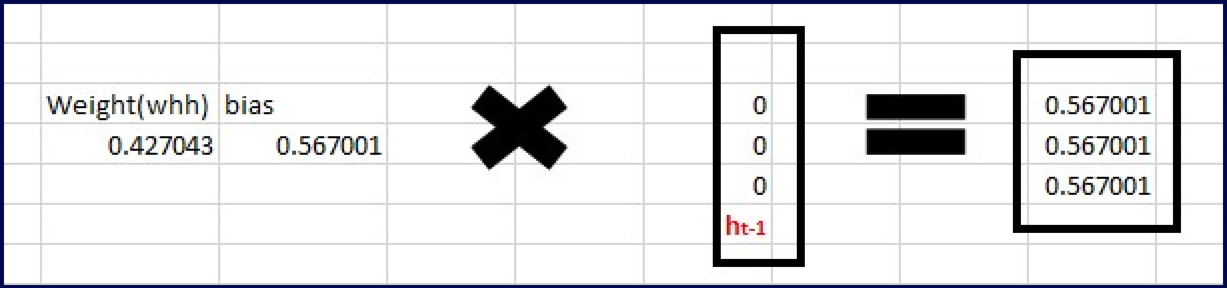
0.427043

and the bias which is also a 1*1 matrix as

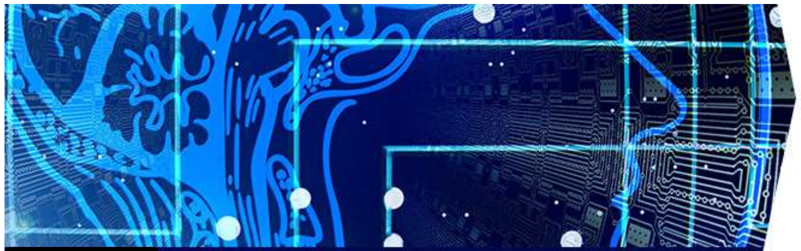


0.56700

For the letter "h", the previous state is [0,0,0] since there is no letter prior to it. So to calculate -> (whh*ht-1+bias)



Weight(whh)	bias		ht-1		
0.427043	0.567001	✖	0	=	0.567001
			0		0.567001
			0		0.567001



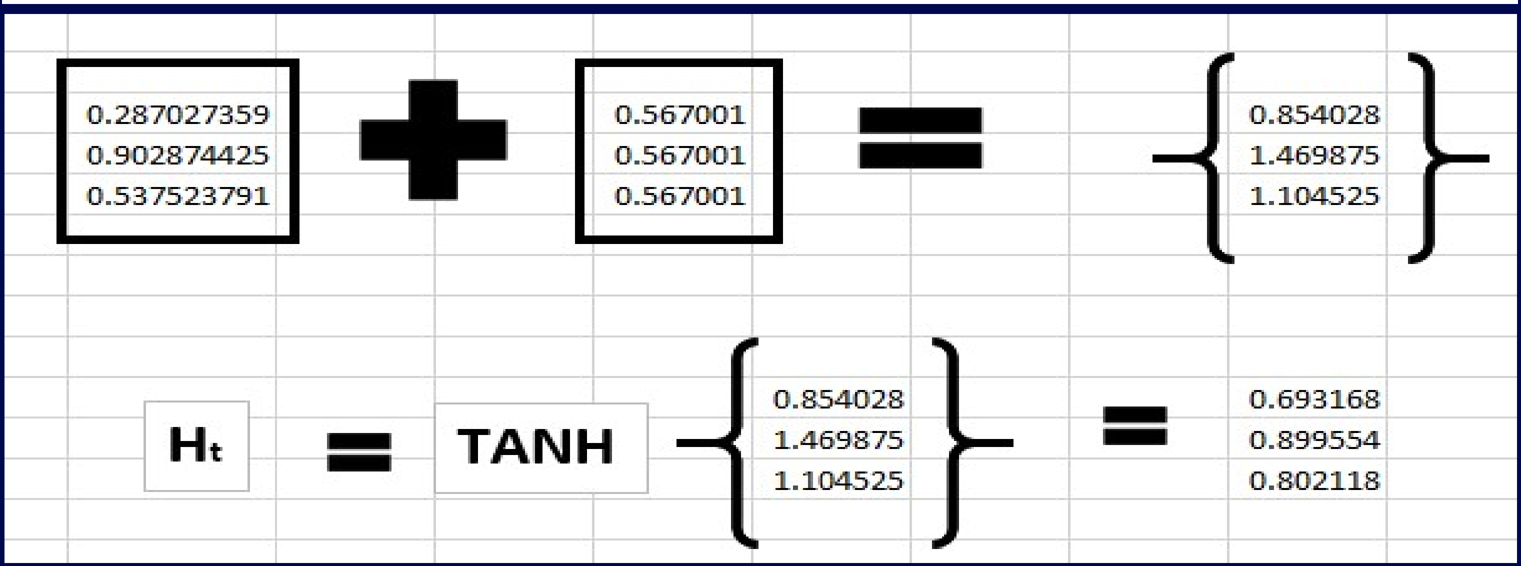
Recurrent Neural Networks (RNN) Step-by-Step Process

Step 3:

Now we can get the current state as –

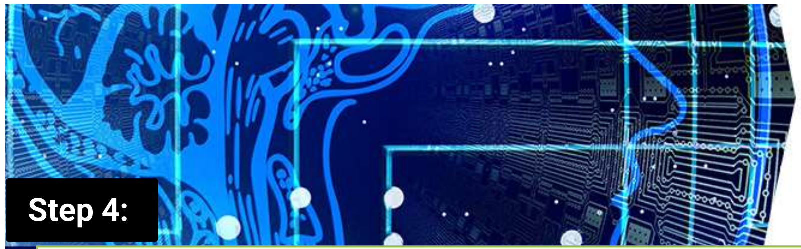
$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{xh}x_t\right)$$

Since for h, there is no previous hidden state we apply the tanh function to this output and get the current state –



$$\boxed{\begin{matrix}0.287027359\\0.902874425\\0.537523791\end{matrix}} + \boxed{\begin{matrix}0.567001\\0.567001\\0.567001\end{matrix}} = \left\{\begin{matrix}0.854028\\1.469875\\1.104525\end{matrix}\right\}$$

$$H_t = \text{TANH}\left\{\begin{matrix}0.854028\\1.469875\\1.104525\end{matrix}\right\} = \begin{matrix}0.693168\\0.899554\\0.802118\end{matrix}$$



Recurrent Neural Networks (RNN) Step-by-Step Process

Step 4:

Now we go on to the next state. “e” is now supplied to the network. The processed output of ht, now becomes ht-1, while the one hot encoded e, is xt. Let’s now calculate the current state ht.

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

Whh*ht-1 +bias will be –

$$\boxed{Whh^*H_{t-1}+Bias} = \boxed{0.427043} \times \begin{array}{r} 0.69316804 \\ 0.89955366 \\ 0.8021184 \end{array} + \boxed{0.567001} = \begin{array}{r} 0.863013 \\ 0.951149 \\ 0.90954 \end{array}$$

Wxh*xt will be –>

$$\begin{array}{|r|r|r|r|} \hline \multicolumn{4}{|c|}{wxh} \\ \hline 0.287027359 & 0.84606 & 0.572392 & 0.486813 \\ \hline 0.902874425 & 0.871522 & 0.691079 & 0.18998 \\ \hline 0.537523791 & 0.09224 & 0.558159 & 0.491528 \\ \hline \end{array} \times \boxed{\begin{array}{c} 0 \\ 1 \\ 0 \\ 0 \\ e \end{array}} = \begin{array}{|r|} \hline 0.84606 \\ \hline 0.871522 \\ \hline 0.09224 \\ \hline \end{array}$$

Recurrent Neural Networks (RNN) Step-by-Step Process

Step 5:

Now calculating ht for the letter “e”, →

Now this would become ht-1 for the next state and the recurrent neuron would use this along with the new character to predict the next one.

$$H_t = \text{TANH}\left\{ \begin{bmatrix} 0.863013 \\ 0.951149 \\ 0.90954 \end{bmatrix} + \begin{bmatrix} 0.84606 \\ 0.871522 \\ 0.09224 \end{bmatrix} \right\} = \begin{bmatrix} 0.93653372 \\ 0.94910403 \\ 0.76234056 \end{bmatrix}$$

Step 6:

At each state, the recurrent neural network would produce the output as well. Let’s calculate yt for the letter e ->

$$y_t = W_{hy}h_t$$

$$\begin{array}{ccc} & \text{why} & \\ 0.37168 & 0.974829459 & 0.830034886 \\ 0.39141 & 0.282585823 & 0.659835709 \\ 0.64985 & 0.09821557 & 0.334287084 \\ 0.91266 & 0.32581642 & 0.144630018 \end{array} \times \begin{array}{c} \text{Ht} \\ 0.936534 \\ 0.949104 \\ 0.762341 \end{array} = \begin{array}{c} \text{yt} \\ 1.90607732 \\ 1.13779113 \\ 0.95666016 \\ 1.27422602 \end{array}$$

Step 7:

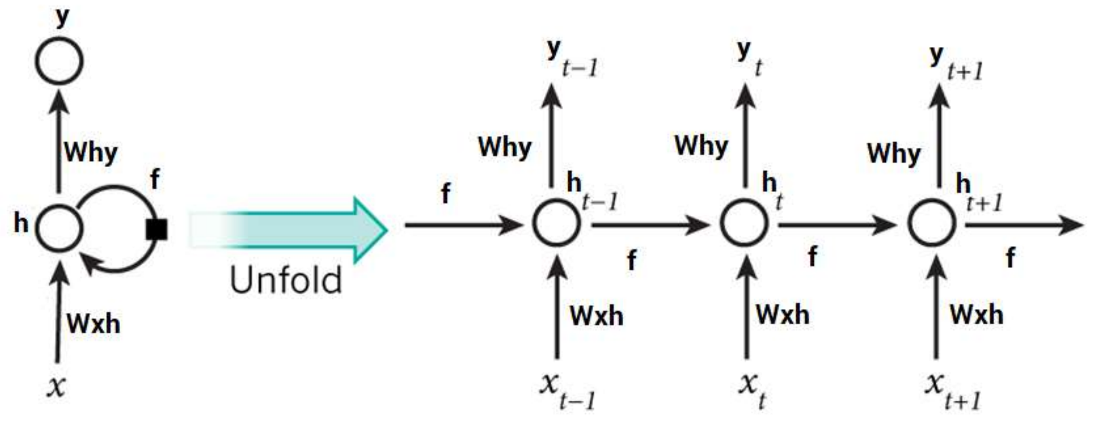
The probability for a particular letter from the vocabulary can be calculated by applying the softmax function. so we shall have softmax(yt)

$$\text{Classwise Probabilities for the next letter} = \text{Softmax}\left\{ \begin{bmatrix} 0.419748 \\ 0.194682 \\ 0.162429 \\ 0.223141 \end{bmatrix} \right\}$$

If we convert these probabilities to understand the prediction, model says that the letter after “e” should be h, since the highest probability is for the letter “h”. Does this mean we have done something wrong? No, so here we have hardly trained the network. We have just shown it two letters. So it pretty much hasn’t learnt anything yet.

Recurrent Neural Networks (RNN) Step-by-Step Process

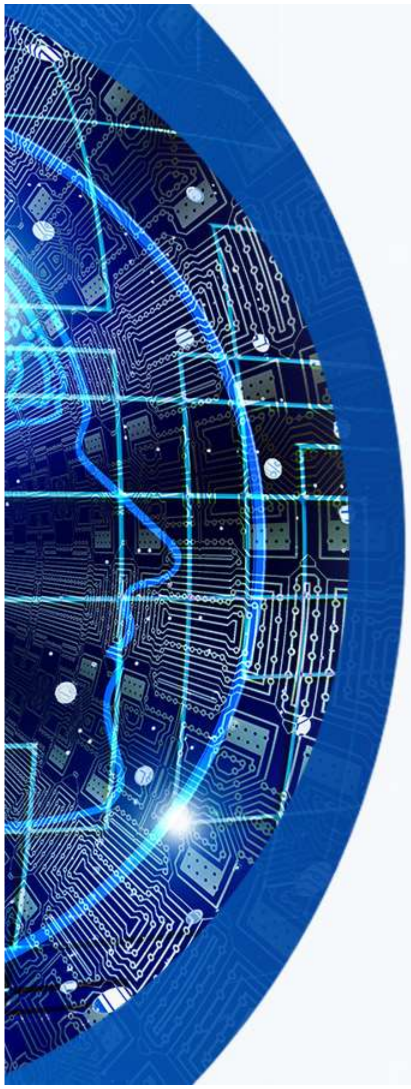
Back propagation in a Recurrent Neural Network(BPTT)
In an RNN we may or may not have outputs at each time step.
In case of a forward propagation, the inputs enter and move forward at each time step. In case of a backward propagation in this case, we are figuratively going back in time to change the weights, hence we call it the Back propagation through time(BPTT).



In case of an RNN, if yt is the predicted value ȳt is the actual value, the error is calculated as a **cross entropy loss –>**
We typically treat the full sequence (word) as one training example, so the total error is just the sum of the errors at each time step (character). The weights as we can see are the same at each time step.

$$E_t(\bar{y}_t, y_t) = -\bar{y}_t \log(y_t)$$
$$E(\bar{y}, y) = -\sum \bar{y}_t \log(y_t)$$

Let’s summarize the steps for backpropagation:
1. The cross entropy error is first computed using the current output and the actual output
2. Remember that the network is unrolled for all the time steps
3. For the unrolled network, the gradient is calculated for each time step with respect to the weight parameter
4. Now that the weight is the same for all the time steps the gradients can be combined together for all time steps
5. The weights are then updated for both recurrent neuron and the dense layers



Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) Architecture

Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) are two special architectures of RNN
Strong on processing semantics of a Natural Language Processing (NLP).

Gated Recurrent Unit (GRU), is a variant of the **RNN architecture**

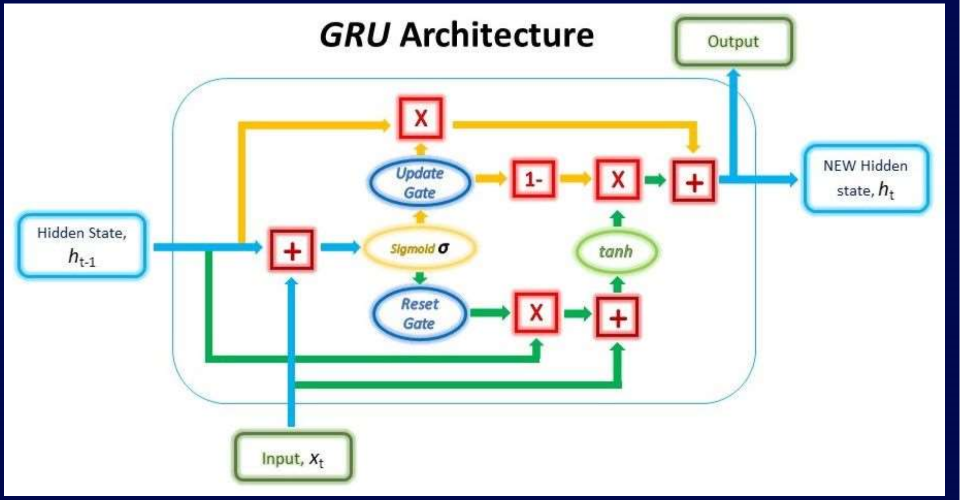
Uses **gating mechanisms** to control and manage the flow of information between cells in the neural network

These **gates are responsible** for regulating the information to be **kept or discarded** at each time step.

Update gate and the Reset gate.
Trained to **selectively filter** out any irrelevant information.
These gates are **vectors** containing values between **0 to 1**, which will be multiplied with the input data and/or hidden state

1. A **0 value** in the gate vectors indicates that the corresponding data in the input or hidden state is **unimportant** and **discarded**
2. a **1 value** in the gate vector means that the corresponding data is **important** and **will be used**

Adaptively capture dependencies from **large sequences of data** without discarding information from earlier parts of the sequence.



Gated Recurrent Unit (GRU) Architecture – Reset Gate

Multiplying the **previous hidden state** and **current input** with their respective weights and summing them before passing the sum through a ***sigmoid*** function.

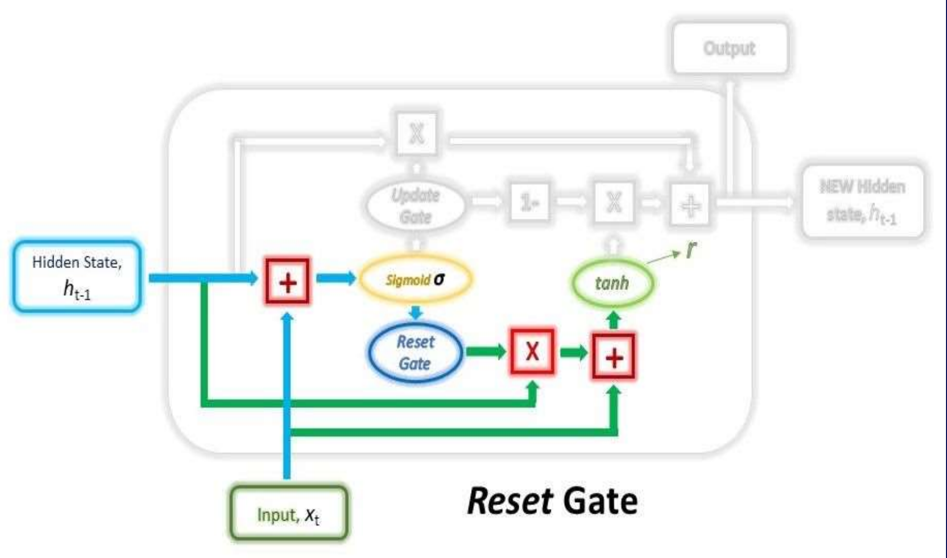
The ***sigmoid*** function will transform the values to fall between *0* and *1*, allowing the gate to filter between the less-important and more-important information in the subsequent steps.

$$gate_{reset} = \sigma(W_{input_{reset}} \cdot x_t + W_{hidden_{reset}} \cdot h_{t-1})$$

The **previous hidden state** will first **be multiplied by a trainable weight** and will then undergo an **element-wise multiplication** (Hadamard product) with the **reset vector**. This operation will decide which information is to be kept from the previous time steps together with the new inputs. At the same time, the **current input** will also be multiplied by a trainable weight before being summed with the product of the **reset vector** and **previous hidden state** above. Lastly, a **non-linear activation *tanh* function** will be applied to the final result to obtain ***r*** in the equation below.

$$r = tanh(gate_{reset} \odot (W_{h_1} \cdot h_{t-1}) + W_{x_1} \cdot x_t)$$

This gate is derived and calculated using both the **hidden state from the previous time step and the input data at the current time step.**



When the entire network is trained through back-propagation, the **weights** in the equation will be updated such that the **vector** will learn to retain only the useful features.

Gated Recurrent Unit (GRU) Architecture – Update Gate

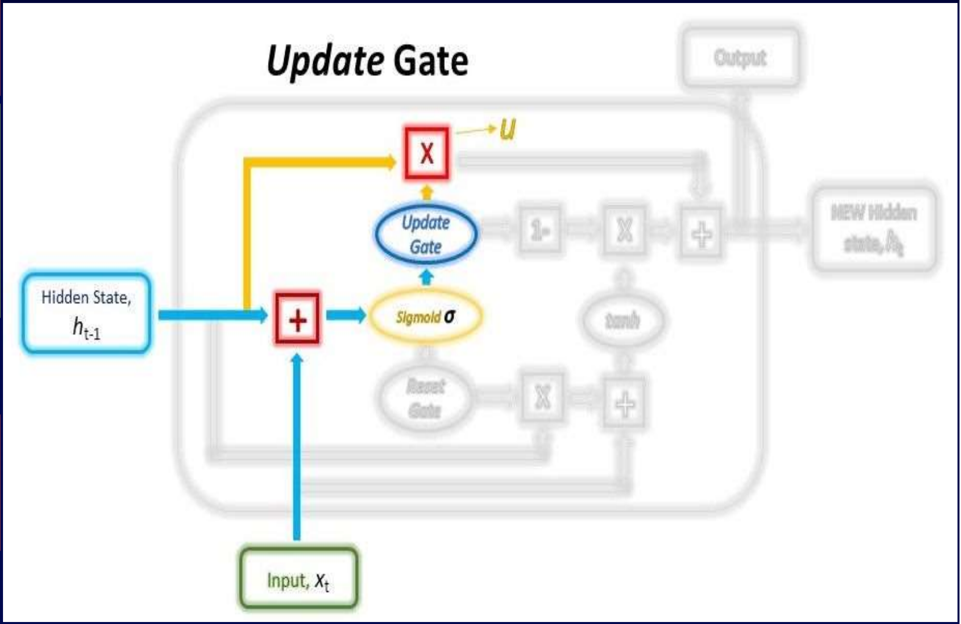
Update gate is computed using the previous hidden state and current input data as mentioned in reset gate.

Both the ***Update*** and ***Reset*** gate vectors are created using the same formula, **but, the weights multiplied with the input and hidden state are unique to each gate**, which means that the final vectors for each gate are different. This allows the gates to serve their specific purposes.

$$gate_{update} = \sigma(W_{input_{update}} \cdot x_t + W_{hidden_{update}} \cdot h_{t-1})$$

The ***Update*** vector will then undergo element-wise multiplication with the **previous hidden state** to obtain ***u*** in our equation below, which will be used to compute our final output later.

$$u = gate_{update} \odot h_{t-1}$$



The ***Update*** vector will also be used in another operation later when obtaining our final output. The purpose of the ***Update*** gate here is to help the model determine how much of the past information stored in the **previous hidden state** needs to be retained for the future.

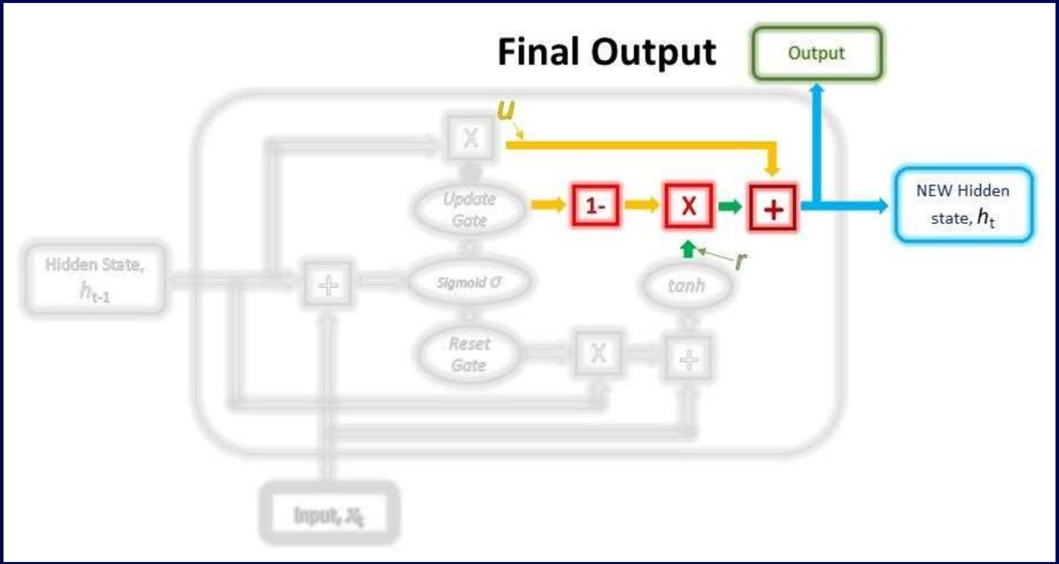
Gated Recurrent Unit (GRU) Architecture – Output

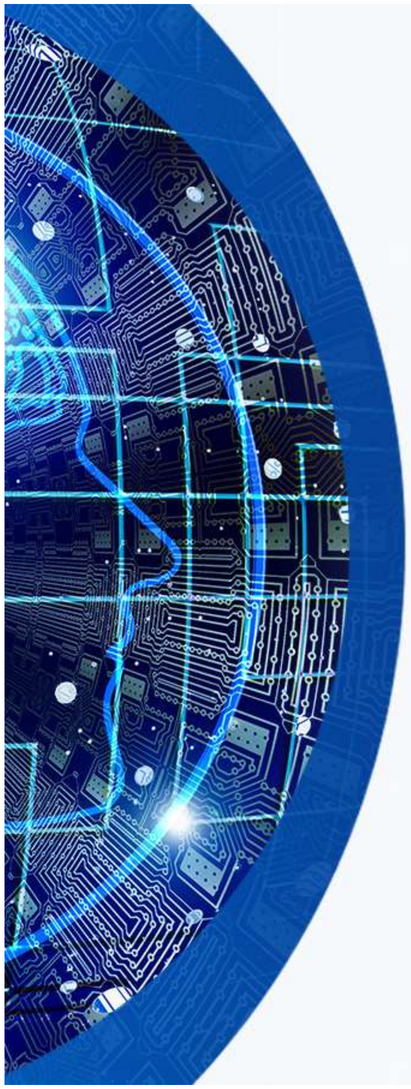
Take element-wise inverse version of the same ***Update*** vector (*1 - **Update** gate*) and doing an element-wise multiplication with output from the ***Reset*** gate, ***r***. The purpose of this operation is for the ***Update*** gate to determine what portion of the new information should be stored in the **hidden state**.

Lastly, the result from the above operations will be summed with our output from the ***Update*** gate in the previous step, ***u***. This will give us our **new** and **updated hidden state**

$$h_t = r \odot (1 - gate_{update}) + u$$

We can use this **new** hidden state as our output for that time step as well by passing it through a linear activation layer.





Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) Architecture

Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM)s have property **of selectively remembering** patterns for long durations of time.
LSTM networks are a type of recurrent neural network capable of learning order dependence in sequence prediction problems.

Disadvantage of RNN:
In order to add a new information in RNN, **it transforms the existing information completely by applying a function**. Because of this, the entire information is modified, on the whole, i. e. there is no consideration for 'important' information and 'not so important' information.

Strength of LSTM:
LSTMs on the other hand, make small modifications to the information by multiplications and additions. With LSTMs, the information flows through a mechanism known as **cell states**. This way, LSTMs can selectively remember or forget things. The information at a particular cell state has three different dependencies.

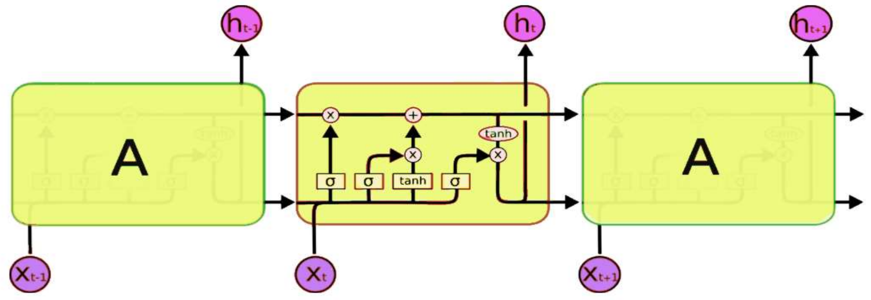
We'll visualize this with an example.
Let's take the example of predicting stock prices for a particular stock. The stock price of today will depend upon:
The trend that the stock has been following in the previous days, maybe a downtrend or an uptrend.
The price of the stock on the previous day, because many traders compare the stock's previous day price before buying it.
The factors that can affect the price of the stock for today. This can be a new company policy that is being criticized widely, or a drop in the company's profit, or maybe an unexpected change in the senior leadership of the company.

These dependencies can be generalized to any problem as:
The previous cell state (i.e. the information that was present in the memory after the previous time step)
The previous hidden state (i.e. this is the same as the output of the previous cell)
The input at the current time step (i.e. the new information that is being fed in at that moment)

LSTMs have two different states passed between the cells — the **cell state** & **hidden state**, carry long and short-term memory, respectively

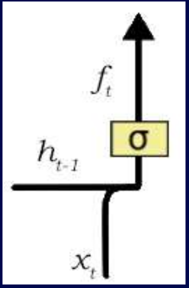
Long Short-Term Memory (LSTM) Architecture

LSTM network is comprised of different memory blocks called **cells** (the rectangles that we see in the image). There are two states that are being transferred to the next cell; the **cell state** and the **hidden state**. The memory blocks are responsible for remembering things and manipulations to this memory is done through three major mechanisms, called **gates**



Forget Gate

A forget gate is responsible for removing information from the cell state. The information that is no longer required for the LSTM to understand things or the information that is of less importance is removed via multiplication of a filter. This is required for optimizing the performance of the LSTM network.



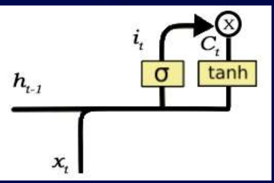
This gate takes in two inputs; h_t-1 and x_t.
h_t-1 is the hidden state from the previous cell or the output of the previous cell
x_t is the input at that particular time step.

1. The given inputs are multiplied by the weight matrices and a bias is added.
2. Following this, the sigmoid function is applied to this value.
3. The sigmoid function outputs a vector, with values ranging from 0 to 1, corresponding to each number in the cell state.
4. Basically, the sigmoid function is responsible for deciding which values to keep and which to discard.
5. If a '0' is output for a particular value in the cell state, it means that the forget gate wants the cell state to forget that piece of information completely.
6. Similarly, a '1' means that the forget gate wants to remember that entire piece of information. This vector output from the sigmoid function is multiplied to the cell state.

Long Short-Term Memory (LSTM) Architecture

Input Gate

This process of adding some new information can be done via the input gate. The input gate is responsible for the addition of information to the cell state.



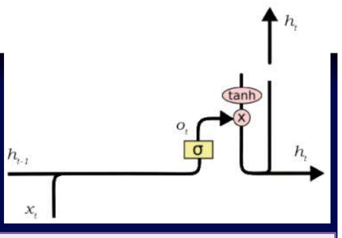
This addition of information is basically three-step process:

1. Regulating what values need to be added to the cell state by involving a sigmoid function. This is basically very similar to the forget gate and acts as a filter for all the information from h_t-1 and x_t.
2. Creating a vector containing all possible values that can be added (as perceived from h_t-1 and x_t) to the cell state. This is done using the tanh function, which outputs values from -1 to +1.
3. Multiplying the value of the regulatory filter (the sigmoid gate) to the created vector (the tanh function) and then adding this useful information to the cell state via addition operation.

Once this three-step process is done, ensure that only information is added to the cell state that is important and is not redundant.

Output Gate

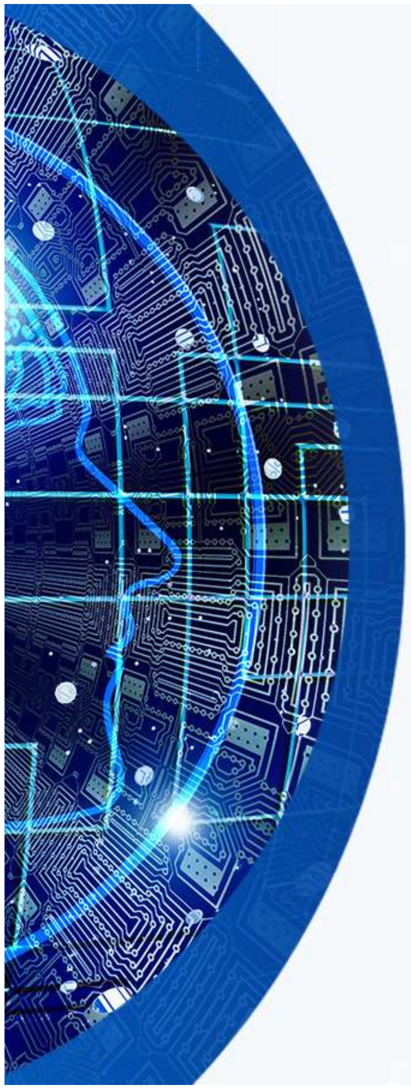
This job of selecting useful information from the current cell state and showing it out as an output is done via the output gate.



The functioning of an output gate can again be broken down to three steps:

1. Creating a vector after applying tanh function to the cell state, thereby scaling the values to the range -1 to +1.
2. Making a filter using the values of h_t-1 and x_t, such that it can regulate the values that need to be output from the vector created above. This filter again employs a sigmoid function.
3. Multiplying the value of this regulatory filter to the vector created in step 1, and sending it out as a output and also to the hidden state of the next cell.

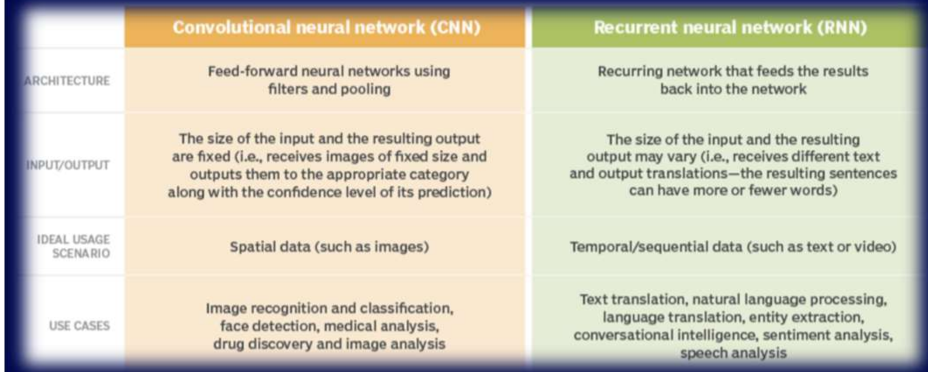
The filter in the above example will make sure that it diminishes all other values but 'Bob'. Thus the filter needs to be built on the input and hidden state values and be applied on the cell state vector.



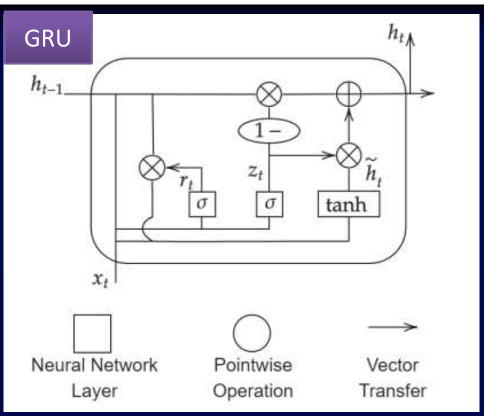
Comparative Study
CNN, RNN, GRU and LSTM

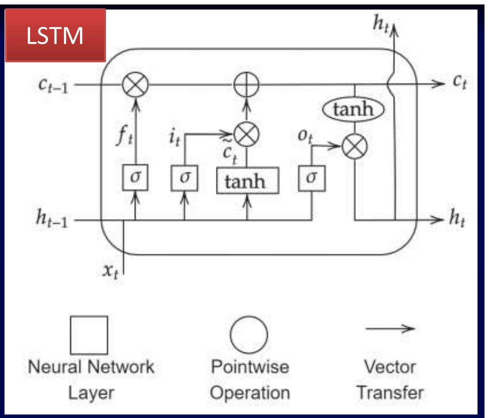
Comparative Study
CNN Vs RNN
GRU Vs LSTM

CNN	RNN
CNN is considered a more powerful tool than RNN.	RNN has fewer features and low capabilities compared to CNN.
The interconnection consumes a finite set of input and generates a finite set of output according to the input.	RNN can allow arbitrary input length and output length.
CNN is a clockwise type of feed-forward artificial neural network with a variety of multiple layers of perceptron which is specially designed to utilize the minimum amount of pre-processing.	RNN works on a loop network which uses their internal memory to handle the arbitrary input sequences.
CNN's are special for video processing and image processing.	RNN works primarily on time series information on the past influence of the consumer.
CNN follows interconnectivity patterns between the neurons which is inspired by the animal visual cortex, where the individual neurons are organized in a way that they respond to overlapping areas tilling the visual field.	RNN works primarily on speech analysis and text analysis.

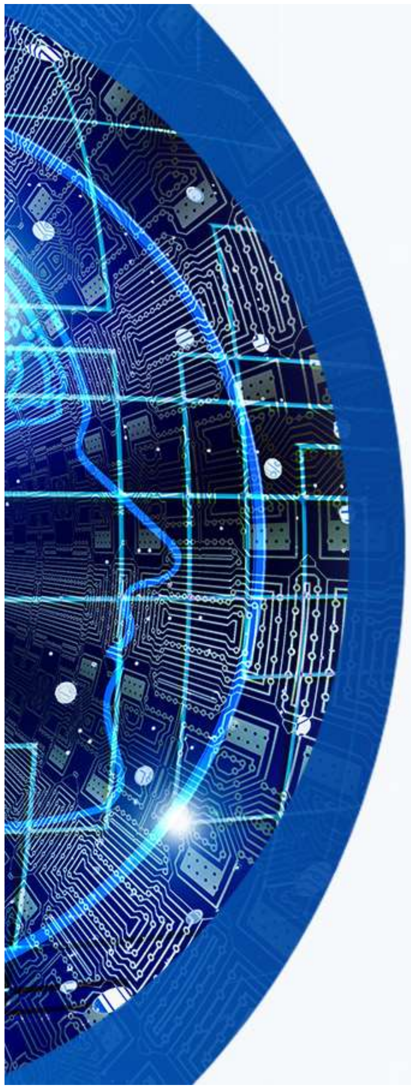


	Convolutional neural network (CNN)	Recurrent neural network (RNN)
ARCHITECTURE	Feed-forward neural networks using filters and pooling	Recurring network that feeds the results back into the network
INPUT/OUTPUT	The size of the input and the resulting output are fixed (i.e., receives images of fixed size and outputs them to the appropriate category along with the confidence level of its prediction)	The size of the input and the resulting output may vary (i.e., receives different text and output translations—the resulting sentences can have more or fewer words)
IDEAL USAGE SCENARIO	Spatial data (such as images)	Temporal/sequential data (such as text or video)
USE CASES	Image recognition and classification, face detection, medical analysis, drug discovery and image analysis	Text translation, natural language processing, language translation, entity extraction, conversational intelligence, sentiment analysis, speech analysis





GRU	LSTM
Uses less training parameters	More parameters than GRU
Uses less memory	Memory intensive
Execute faster and train faster than LSTM's.	Execute slower and train slower than GRU.
Less accuracy on longer sequence	LSTM is more accurate on dataset using longer sequence.
Applied if less memory consumption and faster operation required	Applied if sequence is large or accuracy is very critical, please go for LSTM



Algorithm Performance Analysis

Well Known 10 Evaluation Metrics for Classification Models

Predicted: Outcome of the model on the validation set
Actual: Values seen in the training set
Positive (P): Observation is positive
Negative (N): Observation is not positive
True Positive (TP): Observation is positive, and is predicted correctly
False Negative (FN): Observation is positive, but predicted wrongly
True Negative (TN): Observation is negative, and predicted correctly
False Positive (FP): Observation is negative, but predicted wrongly

1. Confusion matrix is a metric used to quantify the performance of a machine learning classifier. Confusion matrices are used to visualize important predictive analytics like recall, specificity, accuracy, and precision.

2. Accuracy : It defines your total number of true predictions in total dataset. Accuracy is the number of correct predictions over the output size. Accuracy = TP + TN / TP + TN + FP + FN

3. Detection rate : This metric basically shows the number of correct positive class predictions made as a proportion of all of the predictions made. Detection Rate = TP / TP + FP + FN + TN

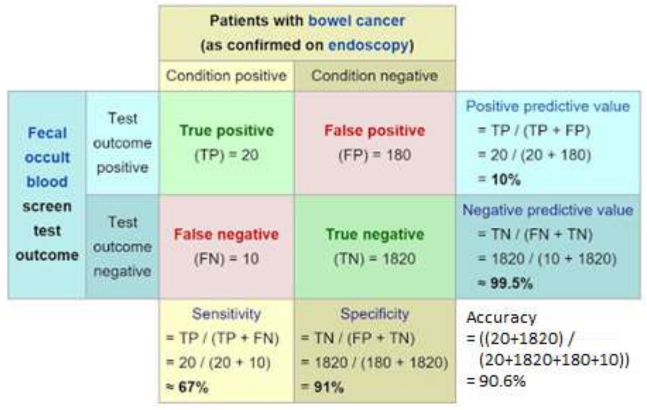
4. Logarithmic loss: log loss, functions **by penalizing all false/incorrect classifications.** Assign a specific probability to each class for all samples. The formula:

$$LogarithmicLoss = \frac{-1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} * \log(p_{ij})$$

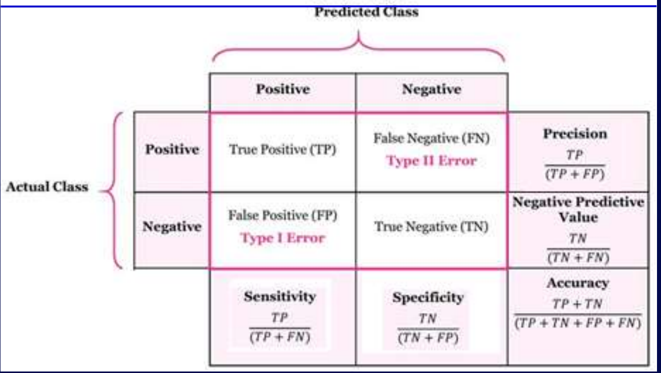
- Yij – Indicates if sample i belongs to class j or not
- Pij – Indicates the probability of sample i belonging to class j

5. Sensitivity (true positive rate): The true positive rate, corresponds to the proportion of positive data points that are correctly considered as positive, with respect to all positive data points. Sensitivity = TP / FN + TP

6. Specificity (false positive rate): Corresponds to the proportion of negative data points that are mistakenly considered as positive, with respect to all negative data points. Specificity = FP / FP + TN : Please note that both FPR and TPR have values in the range of 0 to 1.



		Patients with bowel cancer (as confirmed on endoscopy)		
		Condition positive	Condition negative	
Fecal occult blood screen test outcome	Test outcome positive	True positive (TP) = 20	False positive (FP) = 180	Positive predictive value = TP / (TP + FP) = 20 / (20 + 180) = 10%
	Test outcome negative	False negative (FN) = 10	True negative (TN) = 1820	Negative predictive value = TN / (FN + TN) = 1820 / (10 + 1820) ≈ 99.5%
		Sensitivity = TP / (TP + FN) = 20 / (20 + 10) ≈ 67%	Specificity = TN / (FP + TN) = 1820 / (180 + 1820) = 91%	Accuracy = ((20+1820) / (20+1820+180+10)) = 90.6%



		Predicted Class		
		Positive	Negative	
Actual Class	Positive	True Positive (TP)	False Negative (FN) Type II Error	Precision $\frac{TP}{(TP+FP)}$
	Negative	False Positive (FP) Type I Error	True Negative (TN)	Negative Predictive Value $\frac{TN}{(TN+FN)}$
		Sensitivity $\frac{TP}{(TP+FN)}$	Specificity $\frac{TN}{(TN+FP)}$	Accuracy $\frac{TP+TN}{(TP+TN+FP+FN)}$

Well Known 10 Evaluation Metrics for Classification Models

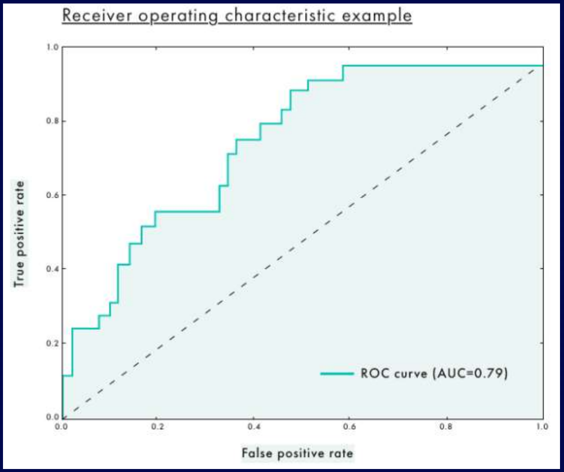
7. Precision
This metric is the number of correct positive results divided by the number of positive results predicted by the classifier.
Precision = TP / TP + FP

8. Recall
Recall is the number of correct positive results divided by the number of all samples that should have been identified as positive.
Recall = TP / TP + FN

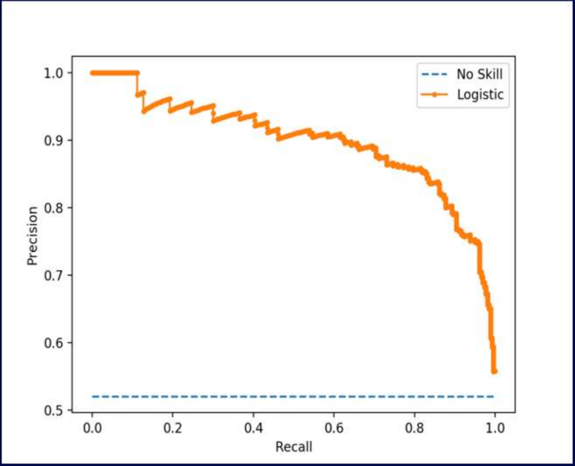
9. F1 score : The F1 score is basically the harmonic mean between precision and recall. It is used to measure the accuracy of tests and is a direct indication of the model's performance. The range of the F1 score is between 0 to 1, with the goal being to get as close as possible to 1. It is calculated as per:

$$F1 = 2 * \frac{1}{\frac{1}{precision} + \frac{1}{recall}}$$

10. Receiver operating characteristic curve (ROC) / area under curve (AUC) score
The ROC curve is basically a graph that displays the classification model's performance at all thresholds. As the name suggests, the AUC is the entire area below the two-dimensional area below the ROC curve. This curve basically generates two important metrics: sensitivity and specificity.



Precision-Recall Curve (PRC)
As the name suggests, this curve is a direct representation of the precision(y-axis) and the recall(x-axis).
This is particularly useful for the situations where we have an imbalanced dataset and the number of negatives is much larger than the positives.

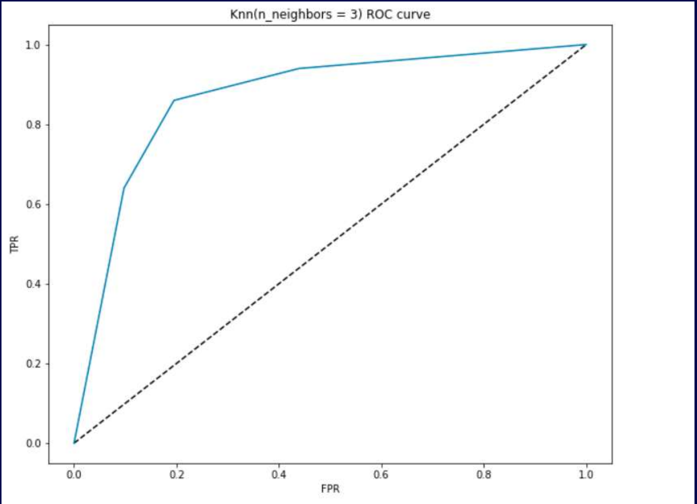


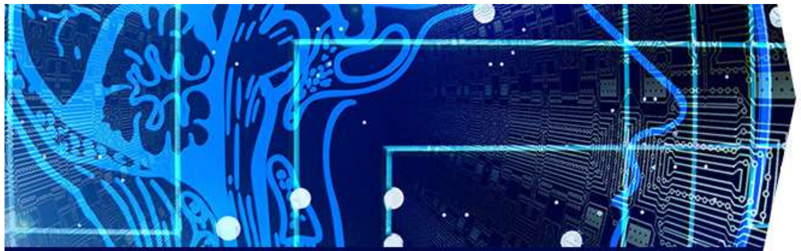
Interpretation of ROC Curves (Receiver Operating Characteristic Curve)

ROC Curve:

1. It is the plot between the TPR(y-axis) and FPR(x-axis).
2. Consider the model classifies the patient as having heart disease or not based on the probabilities generated for each class, we can decide the threshold of the probabilities as well.
3. For example, we want to set a threshold value of 0.4. This means that the model will classify the datapoint/patient as having heart disease if the probability of the patient having a heart disease is greater than 0.4.
4. This will obviously give a high recall value and reduce the number of False Positives. Similarly, we can visualize how our model performs for different threshold values using the ROC curve.
5. Let us generate a ROC curve for our model with k = 3.

1. At the lowest point, i.e. at (0, 0)- the threshold is set at 1.0. This means our model classifies all patients as not having a heart disease.
2. At the highest point i.e. at (1, 1), the threshold is set at 0.0. This means our model classifies all patients as having a heart disease.
3. The rest of the curve is the values of FPR and TPR for the threshold values between 0 and 1. At some threshold value, we observe that for FPR close to 0, we are achieving a TPR of close to 1. This is when the model will predict the patients having heart disease almost perfectly.
4. The area with the curve and the axes as the boundaries is called the Area Under Curve(AUC). It is this area which is considered as a metric of a good model. With this metric ranging from 0 to 1, we should aim for a high value of AUC. Models with a high AUC are called as **models with good skill**. Let us compute the AUC score of our model and the above plot: 0.868
5. We get a value of 0.868 as the AUC which is a pretty good score! This means that the model will be able to distinguish the patients with heart disease and those who don't 87% of the time.
6. The diagonal line is a random model with an AUC of 0.5, a model with no skill, which just the same as making a random prediction



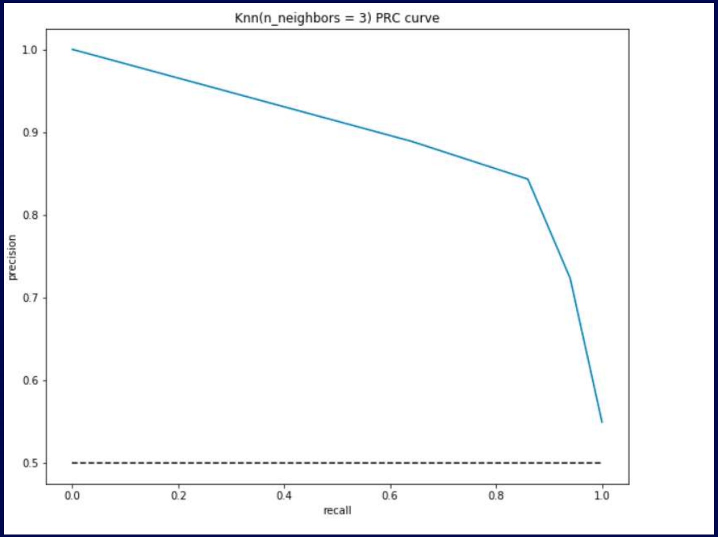


Interpretation of Precision-Recall Curve (PRC)

As the name suggests, this curve is a direct representation of the **precision(y-axis) and the recall(x-axis).**
If you observe our definitions and formulae for the Precision and Recall above, you will notice that at no point are we using the True Negatives(the actual number of people who don't have heart disease).
This is particularly useful for the situations where we have an imbalanced dataset and the number of negatives is much larger than the positives(or when the number of patients having no heart disease is much larger than the patients having it).
In such cases, our higher concern would be detecting the patients with heart disease as correctly as possible and would not need the TNR.

PRC Interpretation:

1. At the lowest point, i.e. at (0, 0)- the threshold is set at 1.0. This means our model makes no distinctions between the patients who have heart disease and the patients who don't.
2. At the highest point i.e. at (1, 1), the threshold is set at 0.0. This means that both our precision and recall are high and the model makes distinctions perfectly.
3. The rest of the curve is the values of Precision and Recall for the threshold values between 0 and 1. Our aim is to make the curve as close to (1, 1) as possible- meaning a good precision and recall.
4. Similar to ROC, the area with the curve and the axes as the boundaries is the Area Under Curve(AUC). Consider this area as a metric of a good model. The AUC ranges from 0 to 1. Therefore, we should aim for a high value of AUC. Let us compute the AUC for our model and the above plot: 0.8957
5. As before, we get a good AUC of around 90%. Also, the model can achieve high precision with recall as 0 and would achieve a high recall by compromising the precision of 50%.


Knn(n_neighbors = 3) PRC curve
precision
recall