Explorer User Guide

QπAI

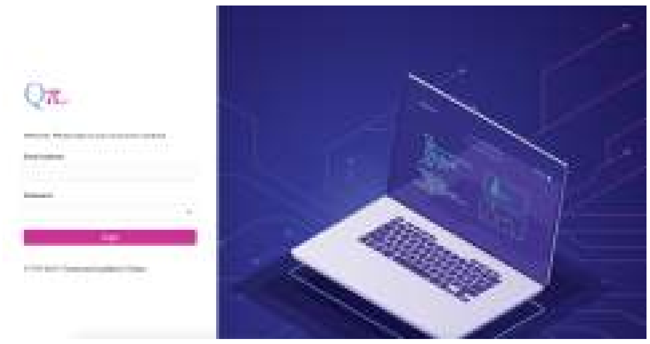
Bengaluru

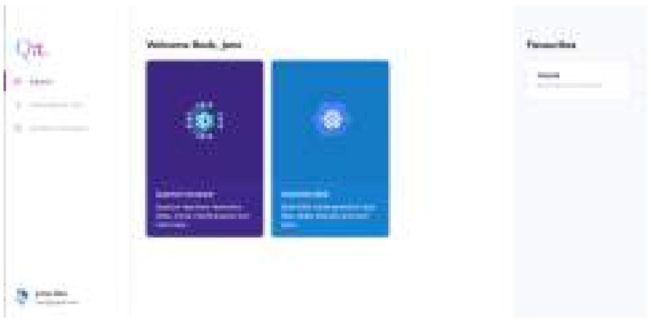
Contents

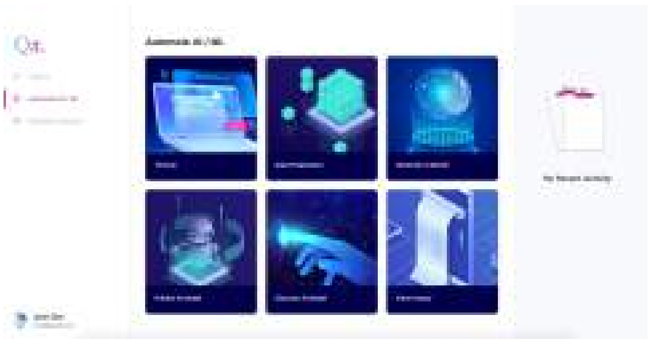
1 INTRODUCTION TO QPIAI^{TM}-EXPLORER

$QpiAI^{TM}$-Explorer is an entry level version of $QpiAI^{TM}$ platform for AI/ML/Quantum ML. This version of the QpiAI platform is meant to be used to skill Corporate users in AI/ML/Quantum ML, Education purposes to skill students and early corporate users in machine learning and as a path to more advanced QpiAI platforms $QpiAI^{TM}$-Pro, $QpiAI^{TM}$-Explorer and $QpiAI^{TM}$-Quantum for more advanced commercial applications. $QpiAI^{TM}$-Explorer contains various machine learning models which can be subject to autoML optimization and has capability to discover simple new models by installing software on desktop and laptops. The Explorer version contains , built in libraries of 100s of simple base models , which users can experiment with and it contains various tutorials on best practices for modelling and quantum computation. This entry stage platform can simulate simple Quantum Circuits and applications.

The landing page looks like :





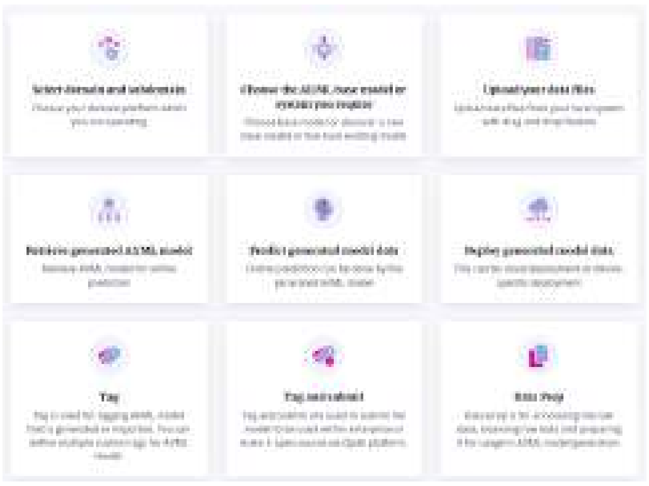


2 INTRODUCTION TO AUTOMATED AI/ML

AutoML is different, in this approach we don't need a human. Because program makes feature engineering, model trains by automate. AutoML is open domain — we can run the same algorithm on any type of data like credit scoring, sales stock, text classifications, and others.

Auto ML services provide machine learning at the click of a button, or, at the very least, promise to keep algorithm implementation, data pipelines, and code, in general, hidden from view.

2.1 INTRODUCTION TO AUTO AI/ML MODEL BUILDING PIPELINE


Select domain and subdomain
Choose the AI/ML base model or system you require
Upload your data files
Retrieve generated AI/ML model
Predict generated model data
Deploy generated model data
Tag
Tag and submit
Data Prep

- **Select Domain and Sub-Domain :** The very first is to select the domain (say transport, healthcare,finance etc) and the respective sub-domain of the same.
- **Data Preparation :** The most important part before modelling lies in Data Preparation.It includes data cleaning, data scaling, feature engineering. These step are the building blocks for modelling.This module contains data preprocessing for Text,Image, Video, Voice such as scaling, blurring, edge detection, lemmatization, stemming, gray scaling, MFCC feature etc.
- **Upload Data Files :** Once the data is ready we need to upload them to start modelling.
- **Choose AI/ML base model or system you require :** Now we are good to go for modelling.The next step is to select the models from the list stated in section 2.2 and start training your model.
- **Retrieve generate AI/ML model :** Once the training is done you can retrieve the model file from the target device(your local system in case of Explorer).
- **Predict generated AI model :** Now you can test the model with your test data and get to know how much your model is learned from the performance metrics.

2.2 INTRODUCTION TO VARIOUS MODELS SUPPORTED ON $QPIAI^{TM}$ -EXPLORER VERSION

User can select these models as base models in the model generation function or they can predict using trained models. Also users can discover new models using discovery. Only entry level discovery is supported in explorer for learning purposes. The full-fledged commercial grade discovery, deployment and model monitoring can be obtained using QpiAI-pro and

QpiAI-Enterprise.

2.2.1 GENERAL ML MODELS

- **Regression**

  Regression can be thought of as "Relationship" between two things.For example, imagine you stay on the ground and the temperature is 70°F. You start climbing a hill and as you climb, you realize that you are feeling colder and the temperature is dropping. When you reach the hilltop which is 500 meters above ground level and you measure the temperature is 60°F. We can conclude that the height above sea level influences temperature. Hence, there is a relationship between height and temperature. This is termed "regression" in statistics. The temperature depends on height and hence is the "dependent" variable, whereas height is the "independent" variable. There may be various factors influencing the temperature such as humidity, pressure, even air pollution levels etc.

  Formal definition of Regression
  Any equation, that is a function of the dependent variables and a set of weights is called a regression function.
  y~$f(x; w)$ where "y" is the dependent variable (in the above example, temperature), "x" are the independent variables (humidity, pressure etc) and "w" are the weights of the equation (coefficients of x terms).

  Linear regression
  Linear regression is probably the simplest approach for statistical learning. It is a good starting point for more advanced approaches, and in fact, many fancy statistical learning techniques can be seen as an extension of linear regression. Therefore, understanding this simple model will build a good base before moving on to more complex approaches. Linear regression is very good to answer the following questions:

  - Is there a relationship between 2 variables?
  - How strong is the relationship?
  - Which variable contributes the most?
  - How accurately can we estimate the effect of each variable?
  - How accurately can we predict the target?
  - Is the relationship linear?
  - Is there an interaction effect?

Estimating the Linear Regression
Let's assume we only have one variable and one target. Then, linear regression is expressed as:

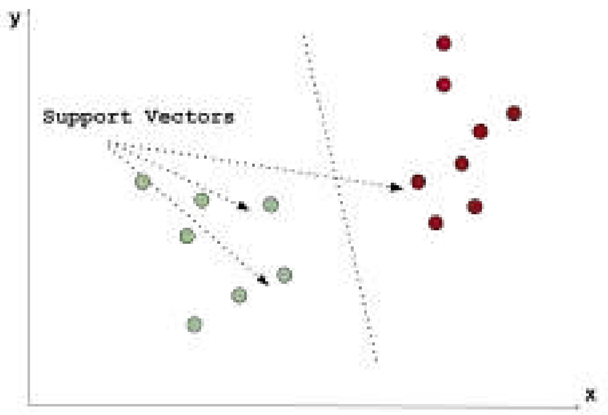
$$Y = \beta_0 + \beta_1 X$$

Assumptions of Linear Regression There are 5 basic assumptions of Linear Regression Algorithm :

- Linear Relationship between the features and target : According to this assumption there is a linear relationship between the features and target.Linear regression captures only linear relationships .It can be validated by plotting Scatter Plot between features and the target.
- Little or no Multicollinearity between the features : Multicollinearity is a state of very high inter-correlations or inter-associations among the independent variables.It is therefore a type of disturbance in the data if present weakens the statistical power of the regression model.Pair plots and heatmaps(correlation matrix) can be used for identifying highly correlated features.
- Homoscedasticity Assumption: Homoscedasticity describes a situation in which the error term (that is, the "noise" or random disturbance in the relationship between the features and the target) is the same across all values of the independent variables.A scatter plot of residual values vs predicted values is a goodway to check for homoscedasticity..There should be no clear pattern in the distribution and if there is a specific pattern,the data is heteroscedastic.
- Normal distribution of error terms: The fourth assumption is that the error(residuals) follow a normal distribution.Normal distribution of the residuals can be validated by plotting a q-q plot.
- Little or No autocorrelation in the residuals: Autocorrelation occurs when the residual errors are dependent on each other.The presence of correlation in error terms drastically reduces model's accuracy.This usually occurs in time series models where the next instant is dependent on previous instant.Autocorrelation can be tested with the help of Durbin-Watson test.

- **Support Vector Machine(SVM)**
  "Support Vector Machine" (SVM) is a supervised machine learning algorithm which can be used for both classification or regression challenges. However, it is mostly used in

classification problems. In the SVM algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiates the two classes very well (look at the below snapshot).



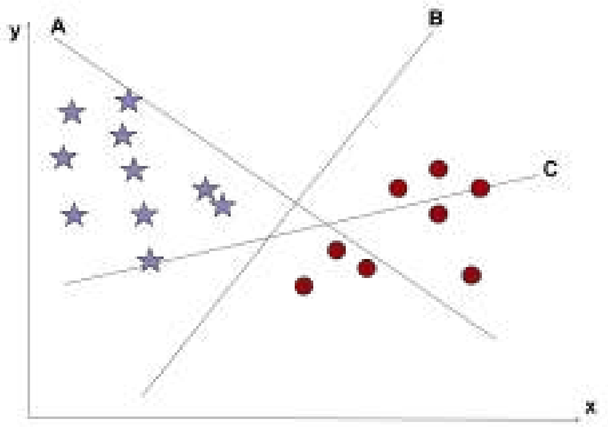
Support Vectors are simply the coordinates of individual observation. The SVM classifier is a frontier which best segregates the two classes (hyper-plane/ line).

How does it work?

Above, we got accustomed to the process of segregating the two classes with a hyper-plane. Now the burning question is "How can we identify the right hyper-plane?".

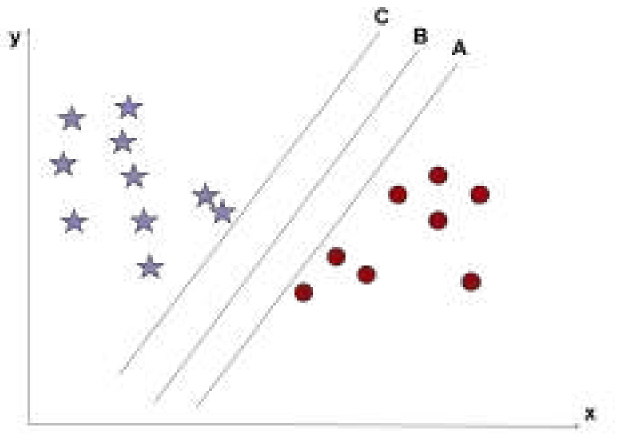
Let's understand:

- Identify the right hyper-plane (Scenario-1): Here, we have three hyper-planes (A, B and C). Now, identify the right hyper-plane to classify star and circle. "Select the

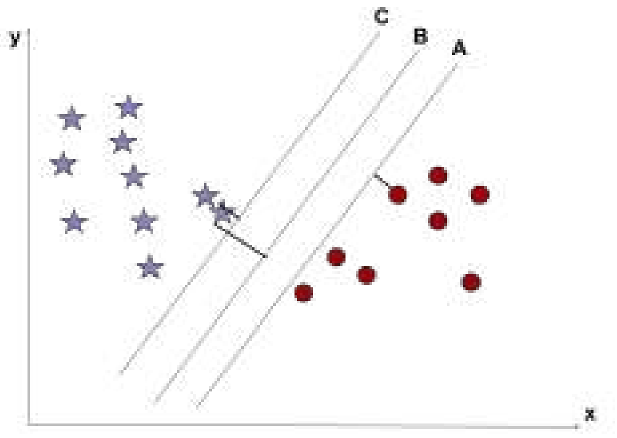


hyper-plane which segregates the two classes better". In this scenario, hyper-plane "B" has excellently performed this job.

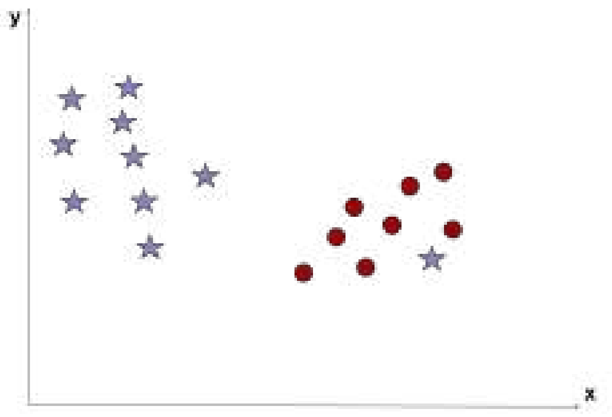
- Identify the right hyper-plane (Scenario-2): Here, we have three hyper-planes (A, B and C) and all are segregating the classes well. Now, How can we identify the right hyper-plane?



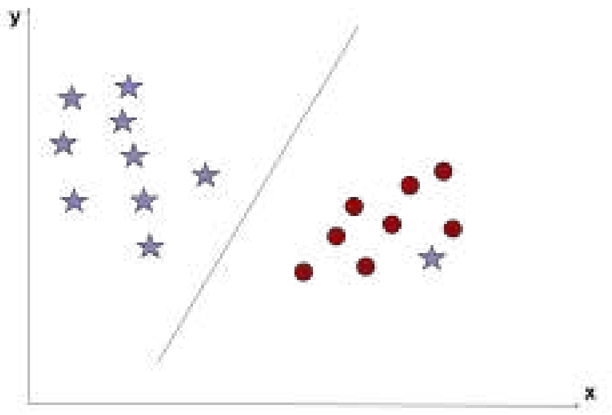
- Here, maximizing the distances between the nearest data point (either class) and hyper-plane will help us to decide the right hyper-plane. This distance is called Margin.



- Above, you can see that the margin for hyper-plane C is high as compared to both A and B. Hence, we name the right hyper-plane as C. Another lightning reason for selecting the hyper-plane with higher margin is robustness. If we select a hyper-plane having low margin then there is a high chance of miss-classification.
- Can we classify two classes (Scenario-3)?: Below, I am unable to segregate the two classes using a straight line, as one of the stars lies in the territory of another(circle) class as an outlier.



- As I have already mentioned, one star at the other end is like an outlier for star class. The SVM algorithm has a feature to ignore outliers and find the hyper-plane that has the maximum margin. Hence, we can say, SVM classification is robust to outliers.



Pros of SVM:

- It works really well with a clear margin of separation
- It is effective in high dimensional spaces.
- It is effective in cases where the number of dimensions is greater than the number of samples.
- It uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.

Cons of SVM:

- It doesn't perform well when we have large data set because the required training time is higher

- It also doesn't perform very well, when the data set has more noise i.e. target classes are overlapping
- SVM doesn't directly provide probability estimates, these are calculated using an expensive five-fold cross-validation.

- **K-Means Clustering**
  K-means clustering is one of the simplest and popular unsupervised machine learning algorithms. Typically, unsupervised algorithms make inferences from datasets using only input examples without referring to known, or labelled, outcomes.

  Clustering helps us understand our data in a unique way – by grouping things together into clusters. Clustering is the process of dividing the entire data into groups (also known as clusters) based on the patterns in the data. In clustering, we do not have a target to predict. We look at the data and then try to club similar observations and form different groups. Hence it is an unsupervised learning problem.

  The objective of K-means is simple: group similar data points together and discover underlying patterns. To achieve this objective, K-means looks for a fixed number (k) of clusters in a dataset. A cluster refers to a collection of data points aggregated together because of certain similarities. The property of these clusters are:

  - The members in a cluster should be similar to each other
  - The members from different clusters should be as different as possible

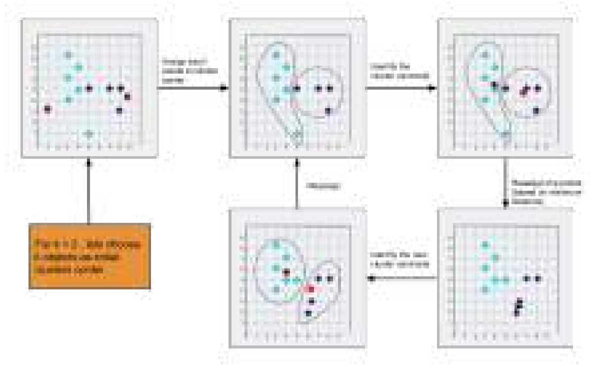
  Formal Definition: K-means is a centroid-based algorithm, or a distance-based algorithm, where we calculate the distances to assign a point to a cluster. In K-Means, each cluster is associated with a centroid. A centroid is the imaginary or real location representing the center of the cluster.The main objective of the K-Means algorithm is to minimize the sum of distances between the points and their respective cluster centroid.

  In other words, the K-means algorithm identifies the k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible. To process the learning data, the K-means algorithm in data mining starts with a first group of randomly selected centroids, which are used as the beginning points for every cluster, and then performs iterative (repetitive) calculations to optimize the positions of the centroids.

  The process for k-means clustering are as follows:

- Pick the number of clusters, k
- Randomly select the centroid for each cluster. Let's say we want to have 2 clusters, so k is equal to 2 here. We then randomly select the centroid
- Assign each point to the closest cluster centroid
- Now, once assigned all of the points to either cluster, the next step is to compute the centroids of newly formed clusters
- Repeat Steps 3 and 4.

Below figure explains the process:



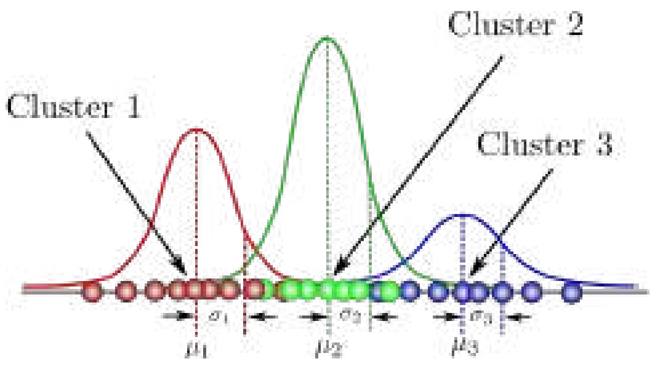
- **Gaussian Mixture Models(GMM)**
  One important characteristic of K-means is that it is a hard clustering method, which means that it will associate each point to one and only one cluster. A limitation to this approach is that there is no uncertainty measure or probability that tells us how much a data point is associated with a specific cluster. It also assigns data to a respective circular distribution around centroids as it is based on distance measure. GMM uses distribution(a probabilistic measure) to assign data rather than a distance measure and hence it is a probabilistic model.

  GMM assumes a mixture of gaussian distributions to have generated the data. It uses soft-assignment of data points to clusters(i.e. probabilistic and therefore better) contrasting with the K-means approach of hard-assignments of data points to clusters with the assumption of a circular distribution of data around centroids.

  Let's say we have three Gaussian distributions (more on that in the next section) – GD1, GD2, and GD3. These have a certain mean (⍰1, ⍰2, ⍰3) and variance (⍰1, ⍰2, ⍰3) value respectively. For a given set of data points, our GMM would identify the probability of each data point belonging to each of these distributions. Gaussian Mixture Models are probabilistic models and use the soft clustering approach for distributing the points in different clusters.

Formal Definition : A Gaussian Mixture is a function that is composed of several Gaussians, each identified by k ϵ 1,..., K, where K is the number of clusters of our dataset. Each Gaussian k in the mixture is comprised of the following parameters:

- A mean μ that defines its centre.
- A covariance $\sum$ that defines its width. This would be equivalent to the dimensions of an ellipsoid in a multivariate scenario.
- A mixing probability π that defines how big or small the Gaussian function will be i.e Density of the distribution(how many data points will be associated with each distribution)



The mean and variance are calculated by an algorithm called Expectation Maximization Algorithm which goes as follows:

- E-step:For each point xi, calculate the probability that it belongs to cluster/distribution c1, c2, ... ck. This is done using the below formula: This value will be high when the

$$r_{ic} = \frac{\text{Probability Xi belongs to c}}{\text{Sum of probability Xi belongs to } c_1, c_2, \dots c_k} = \frac{\pi_c \mathcal{N}(x_i \; ; \; \mu_c, \Sigma_c)}{\sum_{c'} \pi_{c'} \mathcal{N}(x_i \; ; \; \mu_{c'}, \Sigma_{c'})}$$

  point is assigned to the right cluster and lower otherwise.
- M-step:Post the E-step, we go back and update the ⍰, ⍰ and ⍰ values. These are updated in the following manner:
  - The new density is defined by the ratio of the number of points in the cluster and the total number of points:
  - The mean and the covariance matrix are updated based on the values assigned to the distribution, in proportion with the probability values for the data point. Hence, a data point that has a higher probability of being a part of that distribution will contribute a larger portion:

$$\Pi = \frac{\text{Number of points assigned to cluster}}{\text{Total number of points}}$$

$$\mu = \frac{1}{\text{Number of points assigned to cluster}} \Sigma_i \; r_{ic} x_i$$

$$\Sigma_c = \frac{1}{\text{Number of points assigned to cluster}} \Sigma_i \; r_{ic} \, (x_i - \mu_c)^T \, (x_i - \mu_c)$$

Based on the updated values generated from this step, we calculate the new probabilities for each data point and update the values iteratively. Moreover we can say that k-means only considers the mean to update the centroid while GMM takes into account the mean as well as the variance of the data.

It is used in Clustering Applications wherever soft clustering is required.

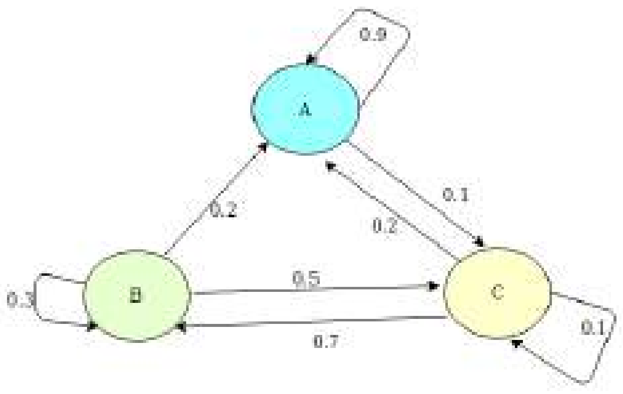
- **Markov Chain**
  Markov chains are used to model probabilities using information that can be encoded in the current state. Something transitions from one state to another semi-randomly, or stochastically. Each state has a certain probability of transitioning to each other state, so each time you are in a state and want to transition, a markov chain can predict outcomes based on pre-existing probability data. More technically, information is put into a matrix and a vector - also called a column matrix - and with many iterations, a collection of probability vectors makes up Markov chains. To determine the transition probabilities, you have to ”train” your Markov Chain on some input corpus.

  Markov Matrix
  A Markov Matrix, or stochastic matrix, is a square matrix in which the elements of each row sum to 1. It can be seen as an alternative representation of the transition probabilities of a Markov chain. Representing a Markov chain as a matrix allows for calculations to be performed in a convenient manner. For example, for a given Markov chain P, the probability of transition from state i to state j in k steps is given by the (i, j)th element of Pk.

  Markov Model
  A Markov model is a stochastic model with the property that future states are determined only by the current state – in other words, the model has no memory; it only knows what state it’s in now, not any of the states which occurred previously.

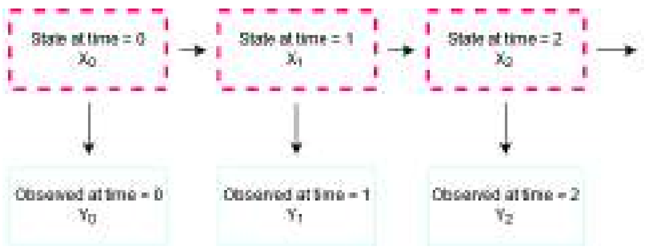
A Markov chain is one example of a Markov model, but other examples exist. One other example commonly used in the field of artificial intelligence is the Hidden Markov model, which is a Markov chain for which the state is not directly observable. Observations can be made, but these observations are not usually sufficient to uniquely determine the state of the model. Hidden Markov models are used, for example, in speech recognition: the audio waveform of the speech is the direct observation, and the actual state of the system is the spoken text.

An Easy Example of a Markov Chain The easiest way to explain a Markov chain is by



simply looking at one. In this example, we can see we have three states(events) : “A”, “B” and “C”. Let’s say the current event is A, and we want to know what the chances are that the next event will be. We can see that the Markov chain indicates that there is a .9, or 90%, chance that the next event will again be “A” , and a small chance of 0.1 or 10% that it will be “C” . Following this pattern, we can see that there will probably be many “A” events lumped together.
Obviously, this is not a real world example - this is not how real-world weather models work. However, if one were building a video game that had a “real” weather component, a Markov chain like this may be useful when fed several years of real-time data.

- **Hidden Markov Models(HMM)** In very simple terms, the HMM is a probabilistic model to infer unobserved information from observed data. HMM helps us to solve where we have to deal with data which consists of sequences. Here, for each observed data, we have an associated hidden truth. In formal HMM discussion, people call each ‘hidden truth’ as a ‘state variable’, and each ‘observed data’ as a ‘observed variable’. Let’s look at the figure



above. In this HMM architecture, we have a sequence of states (hidden truths) ‘linked’ with each other. The arrow has a meaning of dependence here. For example, the state at time = 1 depends on the state at time = 0. This is the very important assumption in the Markov model that makes it so simple — any state depends ONLY on some of its previous state(s).
The main difference between Markov Chain and HMM is Markov Chains deals directly with the predictable states which aren’t visible for us now, whereas HMM uses the visible states to predict the hidden states.

As a motivating example, consider a robot that wants to know where it is. Unfortunately, its sensor is noisy, so instead of reporting its true location, the sensor sometimes reports nearby locations. These reported locations are the observations, and the true location is the state of the system.

Formal Definition: We define HMM using three parameters, first transition matrix which gives probability of going from one hidden state to another, second emission matrix which stores the probabilities of a state i resulting in an observed value j, third π which is the initial state of the system.

Our model is defined as ▯ = (▯, ▯, π) where ▯ is the transition probability matrix, ▯ is the emission probability matrix , π is the initial state the system starts on. Mainly, HMMs are used to solve 3 problems:-

– Observations Probability Estimation: Given a model like above, observations, hidden states, transition probabilities and emission probabilities, estimate the probability of observation sequence given the model. To calculate this, the method of Forward-Backward Algorithm is used which calculates the forward and backward probabilities for every state and adds them together
– Optimal Hidden State Sequence: Given a model with sequence of observations, determine the optimal path or sequence of the hidden states (Predicting)
– HMM Parameters Estimation: Choose the model parameters that maximizes the probability of a specific observation given a specific state(Training)

For training the HMM, Baum-Welch algorithm(Forward-Backward Algorithm) is used and a special case of Expectation maximization algorithm. Its purpose is to tune the parameters of the HMM, namely the state transition matrix A, the emission matrix B, and the initial state distribution π▯, such that the model is maximally like the observed data. There are a few phases for this algorithm, including the initial phase, the forward phase, the backward phase, and the update phase. The forward and the backward phase form the E-step of the EM algorithm, while the update phase itself is the M-step.

In the E-step, the forward and the backward formulas tell us the expected hidden states given the observed data and the set of parameter matrices before-tuned. The M-step update formulas then tune the parameter matrices to best fit the observed data and the expected hidden states. And these two steps are then iterated over and over again until the parameters converged.

For obtaining inference given a trained HMM model, Viterbi algorithm is used. It works by asking a question: given the trained parameter matrices and data, what is the choice of states such that the joint probability reaches maximum? In other words, what is the most likely choice given the data and the trained model? The Viterbi Algorithm does so by calculating with the below formula:

$$X^*_{0:T} = \operatorname*{argmax}_{X_{0:T}} P[X_{0:T}|Y_{0:T}]$$

It means that we have to find the states that maximize the conditional probability of states given data.

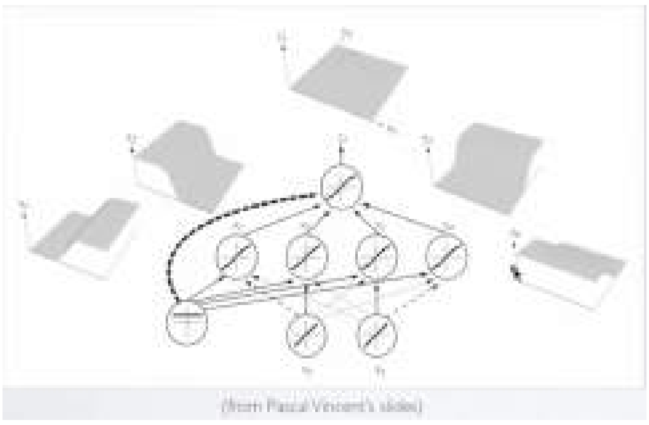
2.2.2 DEEP LEARNING MODELS

- Basic Typologies or building blocks of deep architectures:
  - **Deep Feed Forward Network** Deep Feedforward networks or also known multi-layer perceptrons are the foundation of most deep learning models. Networks like CNNs and RNNs are just some special cases of Feedforward networks. These networks are mostly used for supervised machine learning tasks where we already know the target function ie the result we want our network to achieve and are extremely important for practicing machine learning and form the basis of many commercial applications, areas such as computer vision and NLP were greatly affected by the presence of these networks.

    The reason these networks are called feedforward is that the flow of information takes place in the forward direction, as input is used to calculate some intermediate function in the hidden layer which in turn is used to calculate the specific output. In this, if we add feedback from the last hidden layer to the first hidden layer it would represent a recurrent neural network.

Formal Definition : The main goal of a feedforward network is to approximate some function f*. For example, a regression function y = f *(x) maps an input x to a value y. A feedforward network defines a mapping y = f (x; θ) and learns the value of the parameters θ that result in the best function approximation.

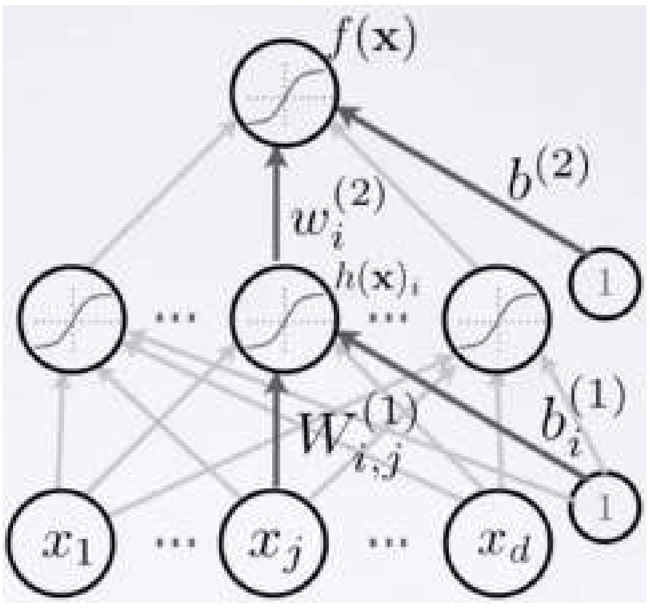
The layers between the input layer and output layers are known as hidden layers, as the training data does not show the desired output for these layers. A network can contain any number of hidden layers with any number of hidden units. A unit basically resembles a neuron which takes input from units of previous layers and computes its own activation value.
As we can see by applying a neural network we can overcome the problem of non



linearity(Fig). As we can observe in the figure that the neural network has trained a model by combining different distributions in one desirable distribution considering the non linearity of the problem.

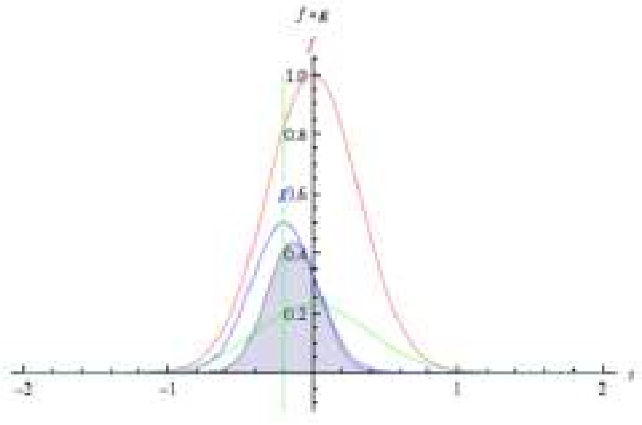
The main idea of the feedforward neural networks is to bring nonlinearity to the function approximation tasks. These networks are trained with the help of Gradient Based Learning and setting a cost function with respect to the output.

Architecture:The leftmost layer in this network is called the input layer, and the neurons within the layer are called input neurons. The rightmost or output layer contains the output neurons, or, as in this case, a single output neuron. The middle layer is called a hidden layer, since the neurons in this layer are neither inputs nor outputs.



– **Convolutional Neural Networks(CNN)**
  This is the basic building block of model architectures that handles mostly image based dataset. It's a supervised model which can be used in a wide variety of tasks like classification, object detection, segmentation and action recognition to name a few.

  Definition : From the Latin convolvere, "to convolve" means to roll together. For mathematical purposes, a convolution is the integral measuring how much two functions overlap as one passes over the other. Think of a convolution as a way of mixing two functions by multiplying them. A CNN is a subset of Neural Networks most commonly used for analyzing visual imagery following the same principle stated above. It's the basic building block for several complex model architectures.
  Working Explanation : The first thing to know about convolutional networks is



that they don't perceive images like humans do. Therefore, you are going to have to think in a different way about what an image means as it is fed to and processed by a convolutional network.

Convolutional networks perceive images as volumes; i.e. three-dimensional objects, rather than flat canvases to be measured only by width and height. That's because digital color images have a red-blue-green (RGB) encoding, mixing those three colors to produce the color spectrum humans perceive. A convolutional network ingests such images as three separate strata of color stacked one on top of the other.

So a convolutional network receives a normal color image as a rectangular box whose width and height are measured by the number of pixels along those dimensions, and whose depth is three layers deep, one for each letter in RGB. Those depth layers are referred to as channels.

As images move through a convolutional network, we will describe them in terms of input and output volumes, expressing them mathematically as matrices of multiple dimensions in this form: 30x30x3. From layer to layer, their dimensions change for reasons that will be explained below.

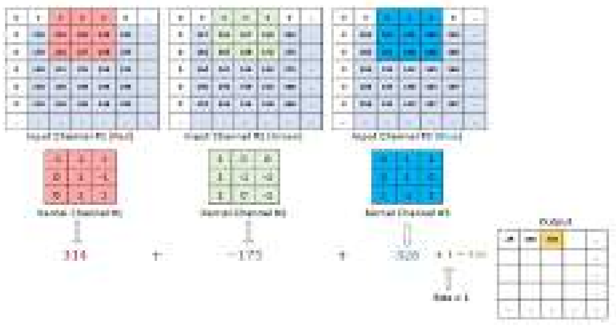
You will need to pay close attention to the precise measures of each dimension of the image volume, because they are the foundation of the linear algebra operations used to process images.

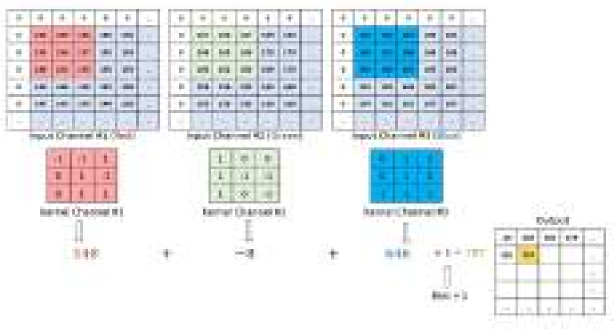
Now, for each pixel of an image, the intensity of R, G and B will be expressed by a number, and that number will be an element in one of the three, stacked two-dimensional matrices, which together form the image volume.

Those numbers are the initial, raw, sensory features being fed into the convolutional network, and the ConvNets purpose is to find which of those numbers are significant signals that actually help it classify images more accurately. (Just like other feedforward networks we have discussed.)

Rather than focus on one pixel at a time, a convolutional net takes in square patches of pixels and passes them through a filter. That filter is also a square matrix smaller than the image itself, and equal in size to the patch. It is also called a kernel, which will ring a bell for those familiar with support-vector machines, and the job of the filter is to find patterns in the pixels.

We are going to take the dot product of the filter with this patch of the image channel. If the two matrices have high values in the same positions, the dot product's output will be high. If they don't, it will be low. In this way, a single value – the out-





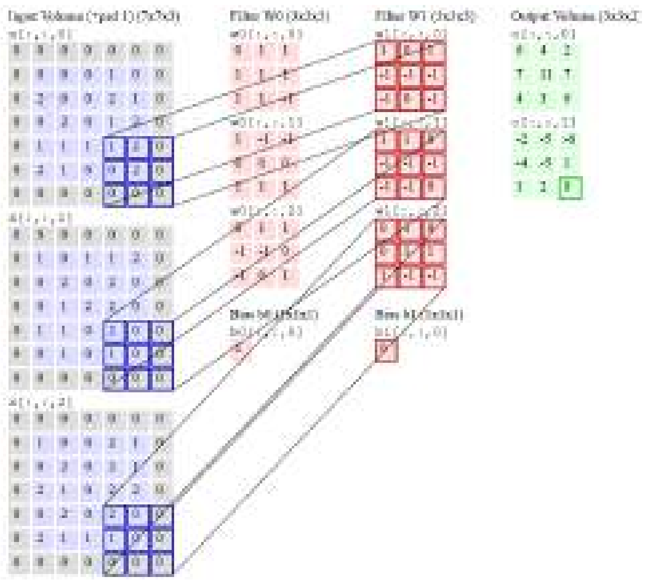
put of the dot product – can tell us whether the pixel pattern in the underlying image matches the pixel pattern expressed by our filter.

Let's imagine that our filter expresses a horizontal line, with high values along its second row and low values in the first and third rows. Now picture that we start in the upper lefthand corner of the underlying image, and we move the filter across the image step by step until it reaches the upper righthand corner. The size of the step is known as stride. You can move the filter to the right one column at a time, or you can choose to make larger steps.

At each step, you take another dot product, and you place the results of that dot product in a third matrix known as an activation map. The width, or number of columns, of the activation map is equal to the number of steps the filter takes to traverse the underlying image. Since larger strides lead to fewer steps, a big stride will produce a smaller activation map. This is important, because the size of the matrices that convolutional networks process and produce at each layer is directly proportional to how computationally expensive they are and how much time they take to train. A larger stride means less time and compute.

A filter superimposed on the first three rows will slide across them and then begin again with rows 4-6 of the same image. If it has a stride of three, then it will produce a matrix of dot products that is 10x10. That same filter representing a horizontal line can be applied to all three channels of the underlying image, R, G and B. And the three 10x10 activation maps can be added together, so that the aggregate activation map for a horizontal line on all three channels of the underlying image is also 10x10.

Now, because images have lines going in many directions, and contain many different kinds of shapes and pixel patterns, you will want to slide other filters across the underlying image in search of those patterns. You could, for example, look for 96 different patterns in the pixels. Those 96 patterns will create a stack of 96 activation maps, resulting in a new volume that is 10x10x96. In the diagram below, we've relabeled the input image, the kernels and the output activation maps to make sure we're clear.

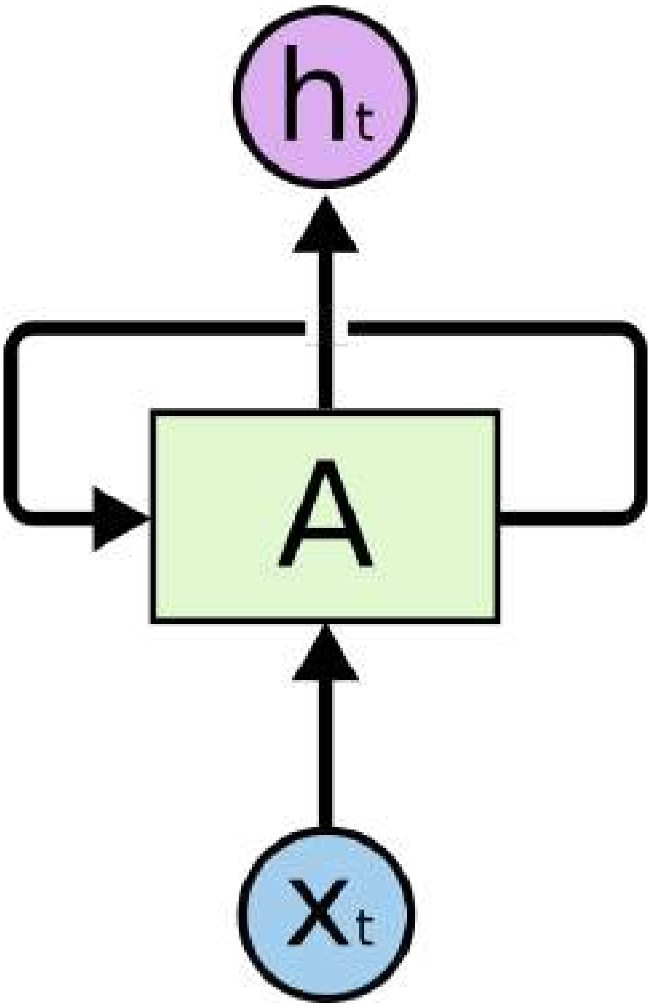


Why CNN over Fully Connected Layers? An image is nothing but a matrix of pixel values, right? So why not just flatten the image (e.g. 3x3 image matrix into a 9x1 vector) and feed it to a Multi-Level Perceptron for classification purposes? Uh.. not really.

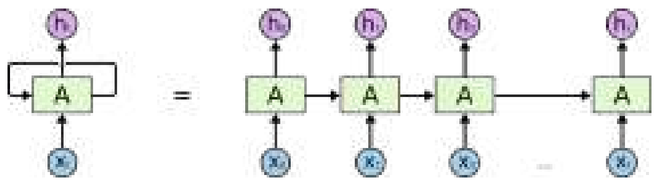
In cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

A ConvNet is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

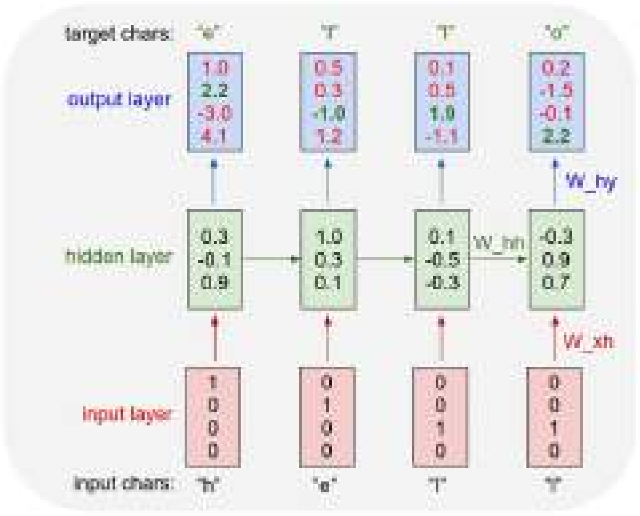
- **Recurrent Neural Networks(RNN)**
  Humans don't start their thinking from scratch every second. As you read this essay, you understand each word based on your understanding of previous words. You don't throw everything away and start thinking from scratch again. Your thoughts have persistence.
  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.



In the above diagram, a chunk of neural network, A, looks at some input xt and outputs a value ht. 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:

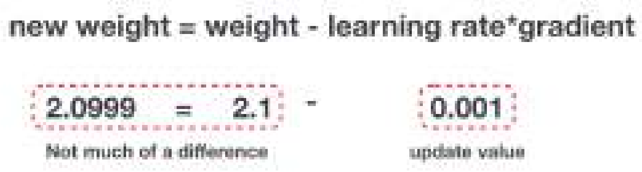


This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. They're the natural architecture of neural network to use for such data.
And they certainly are used! In the last few years, there have been incredible success applying RNNs to a variety of problems: speech recognition, language modeling, translation, image captioning... The list goes on.
Character-Level Language Models Okay, so we have an idea about what RNNs are, why they are super exciting, and how they work. We'll now ground this in a fun application: We'll train RNN character-level language models. That is, we'll give the RNN a huge chunk of text and ask it to model the probability distribution of the next character in the sequence given a sequence of previous characters. This will then allow us to generate new text one character at a time.
As a working example, suppose we only had a vocabulary of four possible letters "helo", and wanted to train an RNN on the training sequence "hello". This training sequence is in fact a source of 4 separate training examples: 1. The probability of "e" should be likely given the context of "h", 2. "l" should be likely in the context of "he", 3. "l" should also be likely given the context of "hel", and finally 4. "o" should be likely given the context of "hell".
Concretely, we will encode each character into a vector using 1-of-k encoding (i.e. all zero except for a single one at the index of the character in the vocabulary), and feed them into the RNN one at a time with the step function. We will then observe a sequence of 4-dimensional output vectors (one dimension per character), which we interpret as the confidence the RNN currently assigns to each character coming next in the sequence. Here's a diagram:



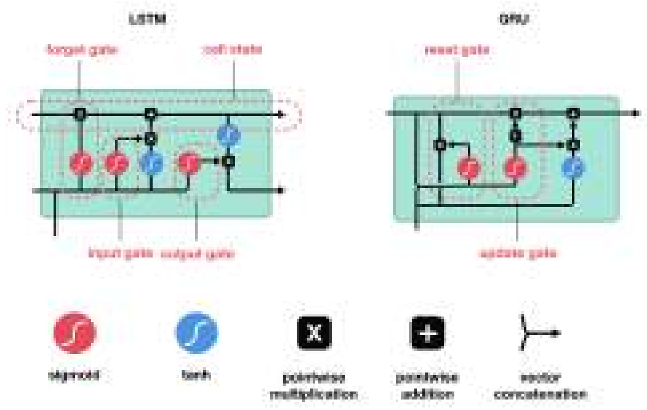
An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters ”hell” as input. The output layer contains confidences the RNN assigns for the next character (vocabulary is ”h,e,l,o”); We want the green numbers to be high and red numbers to be low.
For example, we see that in the first time step when the RNN saw the character “h” it assigned confidence of 1.0 to the next letter being “h”, 2.2 to letter “e”, -3.0 to “l”, and 4.1 to “o”. Since in our training data (the string “hello”) the next correct character is “e”, we would like to increase its confidence (green) and decrease the confidence of all other letters (red). Similarly, we have a desired target character at every one of the 4 time steps that we’d like the network to assign a greater confidence to. Since the RNN consists entirely of differentiable operations we can run the backpropagation algorithm (this is just a recursive application of the chain rule from calculus) to figure out in what direction we should adjust every one of its weights to increase the scores of the correct targets (green bold numbers). We can then perform a parameter update, which nudges every weight a tiny amount in this gradient direction. If we were to feed the same inputs to the RNN after the parameter update we would find that the scores of the correct characters (e.g. “e” in the first time step) would be slightly higher (e.g. 2.3 instead of 2.2), and the scores of incorrect characters would be slightly lower. We then repeat this process over and over many times until the network converges and its predictions are eventually consistent with the training data in that correct characters are always predicted next.
A more technical explanation is that we use the standard Softmax classifier (also commonly referred to as the cross-entropy loss) on every output vector simultaneously. The RNN is trained with mini-batch Stochastic Gradient Descent and I like to use RMSProp or Adam (per-parameter adaptive learning rate methods) to stabilize the updates.

Notice also that the first time the character “l” is input, the target is “l”, but the second time the target is “o”. The RNN therefore cannot rely on the input alone and must use its recurrent connection to keep track of the context to achieve this task. At test time, we feed a character into the RNN and get a distribution over what characters are likely to come next. We sample from this distribution, and feed it right back in to get the next letter. Repeat this process and you’re sampling text! Lets now train an RNN on different datasets and see what happens.
The Problem, Short-term Memory Recurrent Neural Networks suffer from short-term memory. If a sequence is long enough, they’ll have a hard time carrying information from earlier time steps to later ones. So if you are trying to process a paragraph of text to do predictions, RNN’s may leave out important information from the beginning. During back propagation, recurrent neural networks suffer from the vanishing gradient problem.
Gradients are values used to update a neural networks weights. The vanishing gradient problem is when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it doesn’t contribute too much learning.
Gradient Update Rule



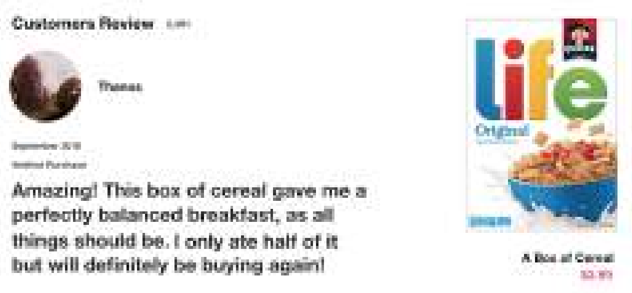
So in recurrent neural networks, layers that get a small gradient update stops learning. Those are usually the earlier layers. So because these layers don’t learn, RNN’s can forget what it seen in longer sequences, thus having a short-term memory.

* **LSTM and GRU**
  LSTM ’s and GRU’s were created as the solution to short-term memory. They have internal mechanisms called gates that can regulate the flow of information.

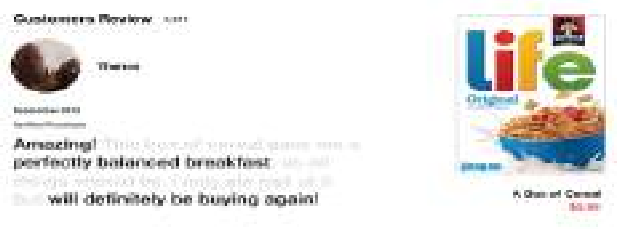


These gates can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions. Almost all state of the art results based on recurrent neural networks are achieved with these two networks. LSTM's and GRU's can be found in speech recognition, speech synthesis, and text generation. You can even use them to generate captions for videos.

Intuition Ok, Let's start with a thought experiment. Let's say you're looking at reviews online to determine if you want to buy Life cereal (don't ask me why). You'll first read the review then determine if someone thought it was good or if it was bad. When you read the review, your brain subconsciously only remem-


Customers Review
Thomas
Amazing! This box of cereal gave me a perfectly balanced breakfast, as all things should be. I only ate half of it but will definitely be buying again!
life
Original
A Box of Cereal

bers important keywords. You pick up words like "amazing" and "perfectly balanced breakfast". You don't care much for words like "this", "gave", "all", "should", etc. If a friend asks you the next day what the review said, you probably wouldn't remember it word for word. You might remember the main points though like "will definitely be buying again". If you're a lot like me, the other words will fade away from memory. And that is essentially what an LSTM or GRU does. It can

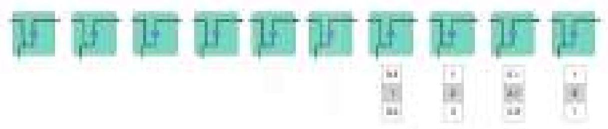

Customers Review
Thomas
Amazing!
perfectly balanced breakfast
will definitely be buying again!
life
Original
A Box of Cereal

learn to keep only relevant information to make predictions, and forget non relevant data. In this case, the words you remembered made you judge that it was good.

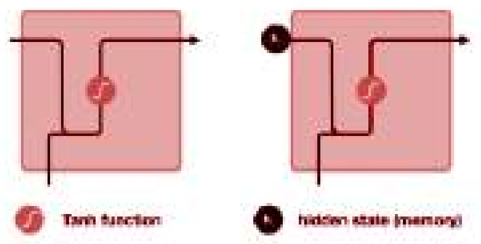
To understand how LSTM's or GRU's achieves this, let's review the recurrent neural network. An RNN works like this; First words get transformed into machine-readable vectors. Then the RNN processes the sequence of vectors one by one.

Processing sequence one by one

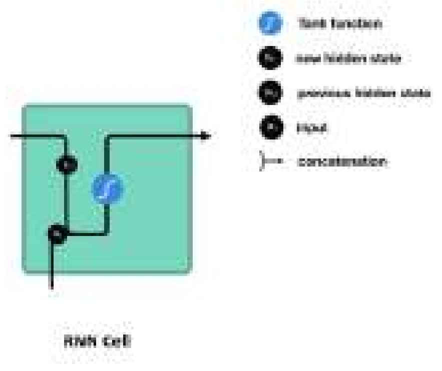
While processing, it passes the previous hidden state to the next step of the



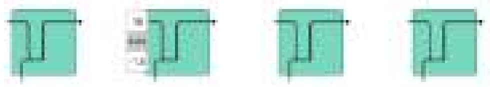
sequence. The hidden state acts as the neural networks memory. It holds information on previous data the network has seen before.



Passing hidden state to next time step
Let's look at a cell of the RNN to see how you would calculate the hidden state. First, the input and previous hidden state are combined to form a vector. That vector now has information on the current input and previous inputs. The vector goes through the tanh activation, and the output is the new hidden state, or the memory of the network. When vectors are flowing through a neural

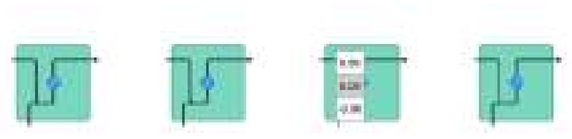


network, it undergoes many transformations due to various math operations. So imagine a value that continues to be multiplied by let's say 3. You can see how some values can explode and become astronomical, causing other values to seem insignificant. A tanh function ensures that the values stay between -



1 and 1, thus regulating the output of the neural network. You can see how

the same values from above remain between the boundaries allowed by the tanh function. So that's an RNN. It has very few operations internally but works



pretty well given the right circumstances (like short sequences). RNN's uses a lot less computational resources than it's evolved variants, LSTM's and GRU's.

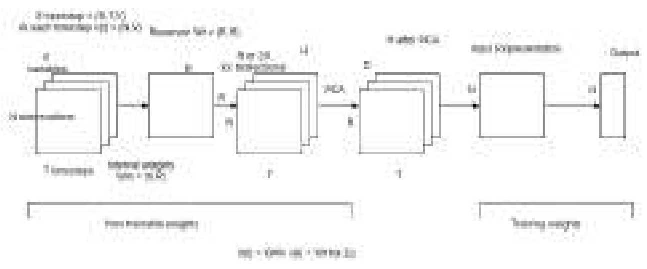
∗ **Echo State Network**

Echo state network is a type of Recurrent Neural Network, part of the reservoir computing framework, which has the following particularities:

- the weights between the input -the hidden layer (the 'reservoir') : Win and also the weights of the 'reservoir': Wr are randomly assigned and not trainable
- the weights of the output neurons (the 'readout' layer) are trainable and can be learned so that the network can reproduce specific temporal patterns
- the hidden layer (or the 'reservoir') is very sparsely connected (typically < 10% connectivity) the reservoir architecture creates a recurrent non linear embedding (H on the image below) of the input which can be then connected to the desired output and these final weights will be trainable
- it is possible to connect the embedding to a different predictive model (a trainable NN or a ridge regressor/SVM for classification problems)

Reservoir Computing Reservoir computing is an extension of neural networks in which the input signal is connected to a fixed (non-trainable) and random dynamical system (the reservoir), thus creating a higher dimension representation (embedding). This embedding is then connected to the desired output via trainable units.

Echo State Networks are recurrent networks. f is a nonlinear function (such as tanh) which makes the current state dependent on the previous state and the current input



For an input of shape N, T, V, where N is the number of observations, T is the number of time steps and V is the number of variables we will:

- choose the size of the reservoir R and other parameters governing the level of sparsity of connection, if we want to model a leakage, the ideal number of components after the dimensionality reduction, etc generate (V, R) input weights Win by sampling from a random binomial distribution
- generate (R, R) reservoir weights Wr by sampling from an uniform distribution with a given density, parameter which sets the level of sparsity
- calculate the high dimensional state representation H as a nonlinear function (typically tanh) of the input at the current time step (N, V) multiplied by the internal weights plus the previous state multiplied by the reservoir matrix (R, R)
- optionally we can run a dimensionality reduction algorithm such as PCA to D components, which brings H to (N, T, D)
- create an input representation either by using for example the entire reservoir and training a regressor to map states t to t+1: one representation could be the matrix of all calculated slopes and intercepts. Another option could be to use the mean or the last value of H
- connect this embedding to the desired output, either by using a NN structure which will be trainable or by using other types of predictors.

Why and when should you use Echo State Networks?

- Traditional NN architectures suffer from the vanishing/exploding gradient problem and as such, the parameters in the hidden layers either don't change that much or they lead to numeric instability and chaotic behavior. Echo state networks don't suffer from this problem.
- Traditional NN architectures are computationally expensive, Echo State Networks are very fast as there is no back propagation phase on the reservoir.
- Traditional NN can be disrupted by bifurcations.
- ESN is well adapted for handling chaotic time series.

∗ **Liquid State Machines**
A liquid state machine (LSM) is a particular kind of spiking neural netwrok. An LSM consists of a large collection of units (called nodes, or neurons). Each node receives time varying input from external sources (the inputs) as well as from other nodes. Nodes are randomly connected to each other. The recurrent nature of the connections turns the time varying input into a spatio-temporal pattern of activations in the network nodes. The spatio-temporal patterns of activation are read out by linear discriminant units.

The soup of recurrently connected nodes will end up computing a large variety of nonlinear functions on the input. Given a large enough variety of such nonlinear functions, it is theoretically possible to obtain linear combinations (using the read out units) to perform whatever mathematical operation is needed to perform a certain task, such as speech recognition or computer vision.
The word liquid in the name comes from the analogy drawn to dropping a stone into a still body of water or other liquid. The falling stone will generate ripples in the liquid. The input (motion of the falling stone) has been converted into a spatio-temporal pattern of liquid displacement (ripples). LSMs have been put forward as a way to explain the operation of brains. LSMs are argued to be an improvement over the theory of artificial neural networks because:

- Circuits are not hard coded to perform a specific task.
- Continuous time inputs are handled ”naturally”.
- Computations on various time scales can be done using the same network.
- The same network can perform multiple computations.

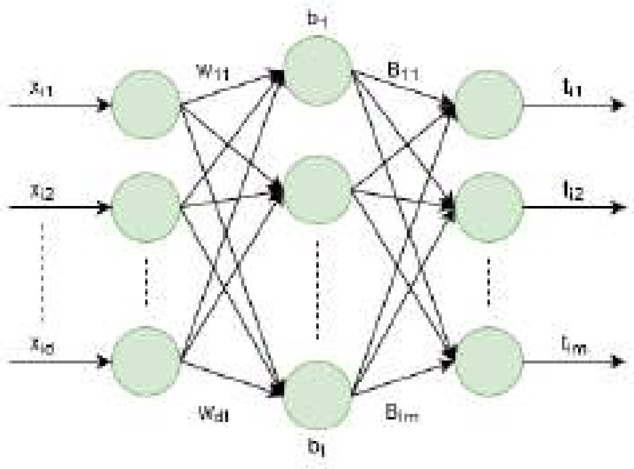
Criticisms of LSMs as used in computational neuroscience are that

- LSMs don’t actually explain how the brain functions. At best they can replicate some parts of brain functionality.
- There is no guaranteed way to dissect a working network and figure out how or what computations are being performed.
- Very little control over the process.

– **Extreme Learning Machine**

Extreme learning machines are feedforward neural networks for classification, regression, clustering, sparse approximation, compression and feature learning with a single layer or multiple layers of hidden nodes, where the parameters of hidden nodes (not just the weights connecting inputs to hidden nodes) need not be tuned. These hidden nodes can be randomly assigned and never updated (i.e. they are random projection but with nonlinear transforms), or can be inherited from their ancestors without being changed. In most cases, the output weights of hidden nodes are usually learned in a single step, which essentially amounts to learning a linear model. The name ”extreme learning machine” (ELM) was given to such models by its main inventor Guang-Bin Huang.

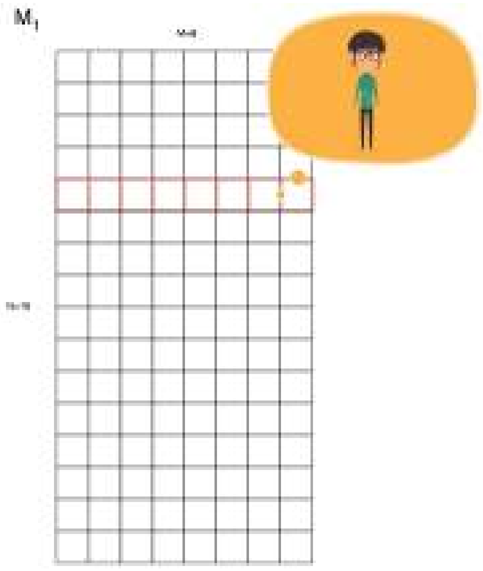
In most cases, ELM is used as a single hidden layer feedforward network (SLFN) including but not limited to sigmoid networks, RBF networks, threshold networks, fuzzy inference networks, complex neural networks, wavelet networks, Fourier transform, Laplacian transform, etc. Due to its different learning algorithm implementations for regression, classification, sparse coding, compression, feature learning and clustering, multi ELMs have been used to form multi hidden layer networks,



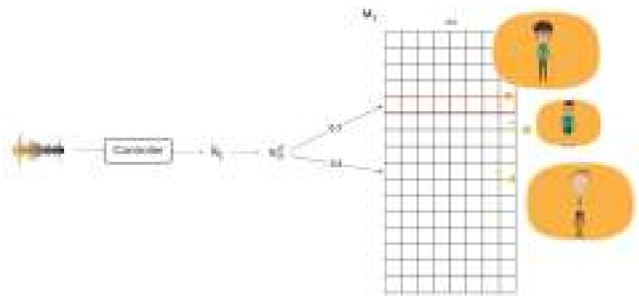
The Classical ELM architecture

deep learning or hierarchical networks. A hidden node in ELM is a computational element, which need not be considered as classical neuron. A hidden node in ELM can be classical artificial neurons, basis functions, or a subnetwork formed by some hidden nodes.

- **Neural Turing Machine**
  Memory is a crucial part of the brain and the computer. For example, in question and answer, we memorize information that we have processed and use them to answer questions.In layman terms, we create a memory structure, typically an array, and we read and write from it. Sound simple: not exactly. First, we do not have an unlimited memory capacity to hold all images or voices we encountered, and we access information by similarity or relevancy, but not by array indexing. **Memory Structure** Deep Learning (DL) memory structure Mt contains N rows each with M elements. Each row represents a piece of information (memory), for example, the latent factors you use to remember your cousin. **Reading**



In conventional programming, we access memory by index Mt[i]. But for AI, we access information by similarity. So we derive a reading mechanism using weights that measure the similarities between the input and each memory row. Our recalled memory will be a weighted sum of the memory rows. You may immediately ask what purpose it serves. Let's go through an example. A friend hands you a drink. It tastes like tea and feels like milk. By extracting our memory profile on tea and milk, we apply linear algebra to interpolate the final result and find out that it is a boba tea. Sounds like magic. But in word embedding, we use the same kind of linear algebra to manipulate relationships. In problems like question and answer, we can use this mechanism to merge previously learned knowledge. So how do we create those weights? A controller extracts features (kt) from the input using a deep network and we use it to compute the weights. For example, you take a phone call but you cannot recognize the voice immediately. The voice sounds a whole lot like your cousin but it also resembles the voice of your elder brother. Through linear algebra, we conclude that it is your high school classmate even though the voice is not what you remember exactly.



Mathematically, to compute the weight w⍰, we measure the similarity between kt and each of our memory entries. We calculate a score K using cosine similarity. Here, u is our extracted feature kt, and v is each individual rows in our memory.

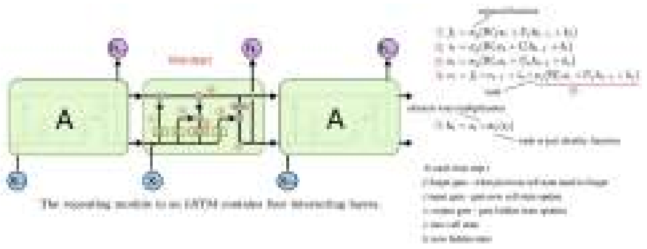
$$K[\mathbf{u},\mathbf{v}] = \frac{\mathbf{u}\cdot\mathbf{v}}{\|\mathbf{u}\|\cdot\|\mathbf{v}\|}$$

$$w_t(i) \longleftarrow \frac{\exp\left(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(i)]\right)}{\sum_j \exp\left(\beta_t K[\mathbf{k}_t, \mathbf{M}_t(j)]\right)}$$

We apply a softmax function on the score K to compute the weight w⍰. ⍰t is added to amplify or attenuate the difference in scores. For example, if it is greater than one, it amplifies the difference. w retrieves information based on similarity and we call this content addressing.

Writing

So how we write information into memory. In LSTM, the internal state of a cell is a combination of the previous state and a new input state It trains and computes the


A
A
The repeating module in an LSTM contains four interacting layers.

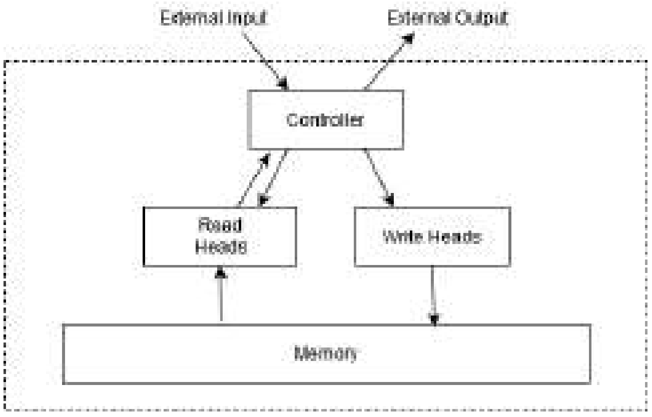
forget gate f to control what previous states should be forgotten (or erase) and the input gate i to control what states should be added to the current cell.
Borrow from the same intuition, the memory writing process comprises previous state and new input. Here, we erase part of the previous state with where et is an

$$\tilde{M}_t(i) \longleftarrow M_{t-1}(i)[1 - w_t(i)e_t]$$

erase vector — acts like the forget gate in LSTM.
Then, we write our new information. where at is what we want to add — acts like

$$M_t(i) \longleftarrow \tilde{M}_t(i) + w_t(i)a_t$$

the input gate in LSTM.
Here, through a controller that generates w, we read and write from our memory.
Addressing Mechanisms
Our controller computes w to extract information. But extraction by similarity (content addressing) can be further improved. **Interpolation**

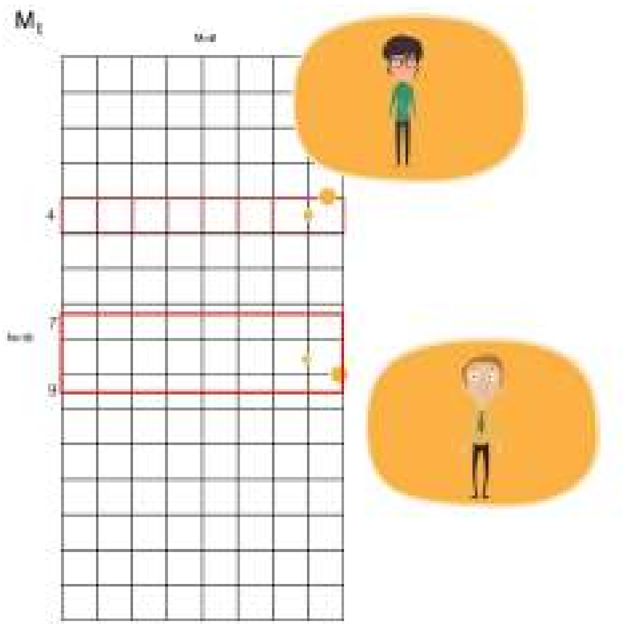


w represents what is our current focus (attention) in our memory. In content addressing, our focus is only based on the current input. However, this does not account for our previous encounter. For example, if your classmate texts you an hour ago, you should be more likely to recall him. How do we accomplish previous attention in extracting information? We compute a new merged weight based on the current content focus as well as our previous focus. Yes, this sounds like the gating mechanism in LSTM or GRU. where g is the gate computed from the previous focus

$$w_t^g \longleftarrow g_t w_t^c + (1 - g_t)w_{t-1}$$

and our current input.
Next, we will look at some operators that can be applied to the memory network.

Convolution shift Convolution shift handles a shift of focus. It is not specifically designed for deep learning. Instead, it shows how an NTM can perform basic algorithms like copying and sorting. For example, instead of accessing w[4], we want to shift every focus by 3 rows. i.e. w[i] ⍰ w[i+3].
In convolution shift, we can shift our focus to a range of rows, i.e. w[i] ⍰ convolution(w[i+3], w[i+4], w[i+5]). Usually, the convolution is just a linear weighted sum of rows say 0.3 × w[i+3] + 0.5 × w[i+4] + 0.2 × w[i+5].
This is the mathematical formulation to shift our focus:



Sharpening

$$\tilde{w}_t(i) \longleftarrow \sum_{j=0}^{N-1} w_t^g(j)\, s_t(i-j)$$

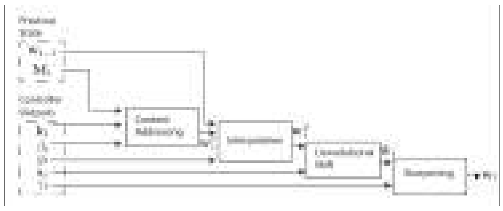
Our convolution shift behaves like a convolutional blurring filter. So we apply the sharpening technique to our weights to counterplay the blurring if needed. ⍰ will be another parameter output by the controller to sharpen our focus.

Putting it together

$$w_t(i) \longleftarrow \frac{\tilde{w}_t(i)^{\gamma_t}}{\sum_j \tilde{w}_t(j)^{\gamma_t}}$$

We retrieve information from our memory using the weight w. w includes factors like our current input, previous focus, possible shifting, and blurring. Here is the generic system diagram in which a controller outputs the necessary parameters to be used in calculating w at different stages.

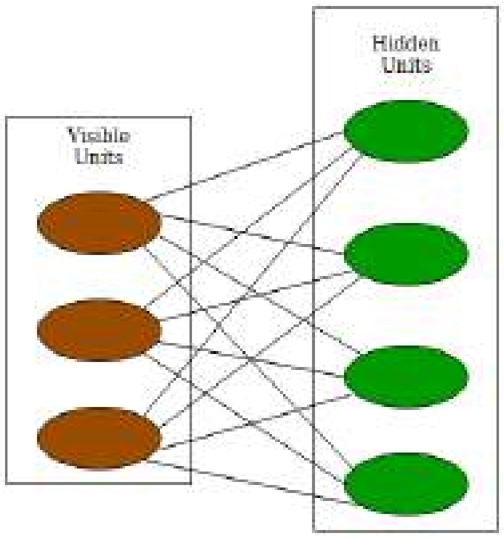
- **Basic Architectures specifically used for Unsupervised Learning**



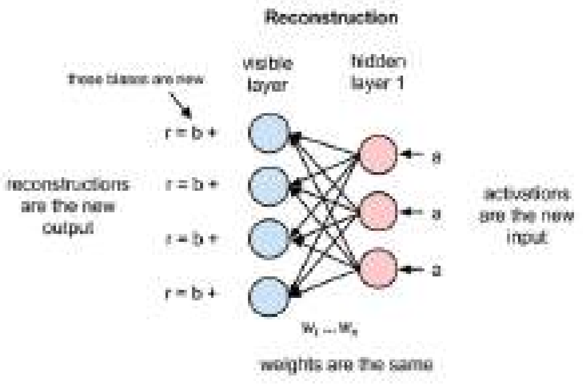
- **Boltzmann Machine** Boltzmann machines are stochastic and generative neural networks capable of learning internal representations and are able to represent and (given sufficient time) solve difficult combinatorial problems.

  Boltzmann machines are non-deterministic (or stochastic) generative Deep Learning models with only two types of nodes — hidden and visible nodes. There are no output nodes

  * **Restricted Boltzmann Machine**
    RBMs are a two-layered artificial neural network with generative capabilities. They have the ability to learn a probability distribution over its set of input and can be used for dimensionality reduction, classification, regression, collaborative filtering, feature learning, and topic modeling.
    RBMs are a special class of Boltzmann machines and they are restricted in terms of the connections between the visible and the hidden units. RBM is a Stochas-



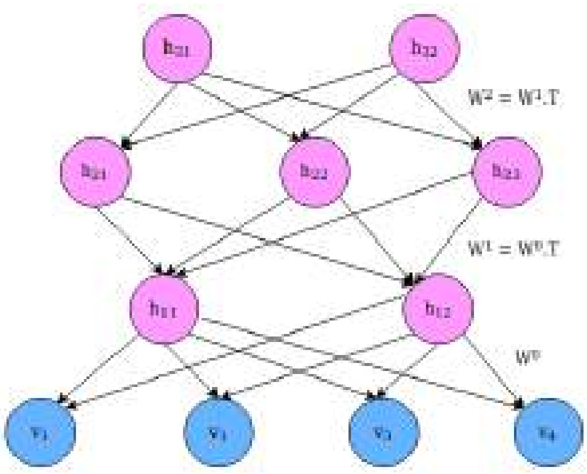
    tic Neural Network which means that each neuron will have some random behavior when activated. There are two other layers of bias units (hidden bias and visible bias) in an RBM. This is what makes RBMs different from autoencoders. The hidden bias RBM produces the activation on the forward pass and the visible bias helps RBM to reconstruct the input during a backward pass. The reconstructed input is always different from the actual input as there are no connections among the visible units and therefore, no way of transferring information among themselves.



Reconstruction is different from regression or classification in that it estimates the probability distribution of the original input instead of associating a continuous/discrete value to an input example. This means it is trying to guess multiple values at the same time. This is known as generative learning as opposed to discriminative learning that happens in a classification problem (mapping input to labels).

Let us try to see how the algorithm reduces loss or simply put, how it reduces the error at each step. Assume that we have two normal distributions, one from the input data (denoted by p(x)) and one from the reconstructed input approximation (denoted by q(x)). The difference between these two distributions is our error in the graphical sense and our goal is to minimize it, i.e., bring the graphs as close as possible. This idea is represented by a term called the Kullback–Leibler divergence. KL-divergence measures the non-overlapping areas under the two graphs and the RBM's optimization algorithm tries to minimize this difference by changing the weights so that the reconstruction closely resembles the input.

∗ **Deep Belief Network**
Deep belief networks are algorithms that use probabilities and unsupervised learning to produce outputs. They are composed of binary latent variables, and they contain both undirected layers and directed layers.
Unlike other models, each layer in deep belief networks learns the entire input. In convolutional neural networks, the first layers only filter inputs for basic features, such as edges, and the later layers recombine all the simple patterns found by the previous layers. Deep belief networks, on the other hand, work globally and regulate each layer in order.
Architecture of deep belief networks The network is like a stack of Restricted Boltzmann Machines (RBMs), where the nodes in each layer are connected to



all the nodes in the previous and subsequent layer. However, unlike RBMs, nodes in a deep belief network do not communicate laterally within their layer. A network of symmetrical weights connect different layers.

The connections in the top layers are undirected and associative memory is formed from the connections between them. The connections in the lower levels are directed.

The nodes in the hidden layer fulfill two roles⍰they act as a hidden layer to nodes that precede it and as visible layers to nodes that succeed it. These nodes identify the correlations in the data.

How do deep belief networks work?

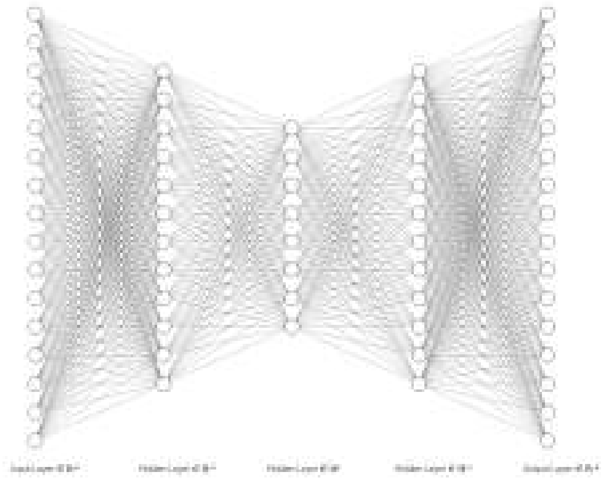
Greedy learning algorithms are used to pre-train deep belief networks. This is a problem-solving approach that involves making the optimal choice at each layer in the sequence, eventually finding a global optimum.

Greedy learning algorithms start from the bottom layer and move up, fine-tuning the generative weights. The learning takes place on a layer-by-layer basis, meaning the layers of the deep belief networks are trained one at a time. Therefore, each layer also receives a different version of the data, and each layer uses the output from the previous layer as their input.

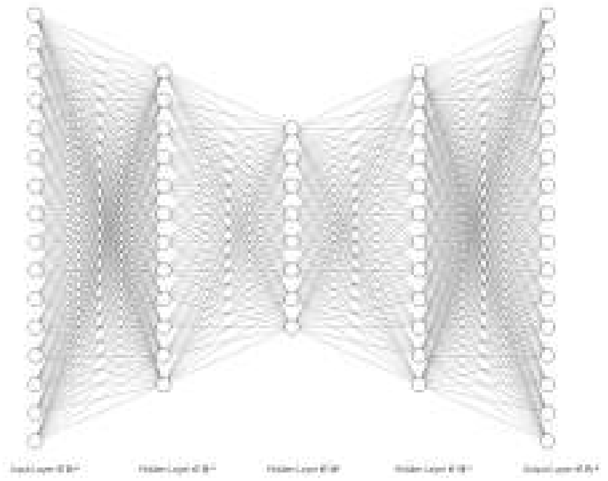
Greedy learning algorithms are used to train deep belief networks because they are quick and efficient. Moreover, they help to optimize the weights at each layer.

– **Autoencoders**

Autoencoders are an unsupervised learning technique in which we leverage neural networks for the task of representation learning. Specifically, we'll design a neural network architecture such that we impose a bottleneck in the network which forces a compressed knowledge representation of the original input. If the input features were each independent of one another, this compression and subsequent reconstruction would be a very difficult task. However, if some sort of structure exists in the data (ie. correlations between input features), this structure can be learned and consequently leveraged when forcing the input through the network's bottleneck. As visualized above, we can take an unlabeled dataset and frame it as a



supervised learning problem tasked with outputting x, a reconstruction of the original input x. This network can be trained by minimizing the reconstruction error,L(x, x), which measures the differences between our original input and the consequent reconstruction. The bottleneck is a key attribute of our network design; without the presence of an information bottleneck, our network could easily learn to simply memorize the input values by passing these values along through the network (visualized below). A bottleneck constrains the amount of information that can traverse



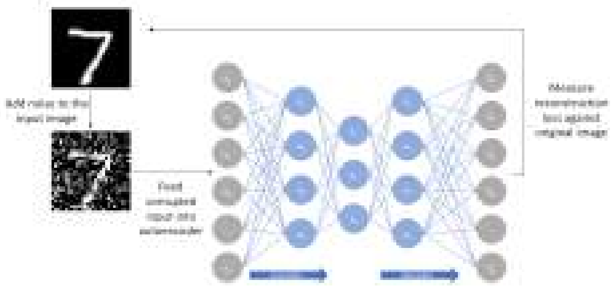
the full network, forcing a learned compression of the input data.
The ideal autoencoder model balances the following:

* Sensitive to the inputs enough to accurately build a reconstruction.
* Insensitive enough to the inputs that the model doesn't simply memorize or overfit the training data.

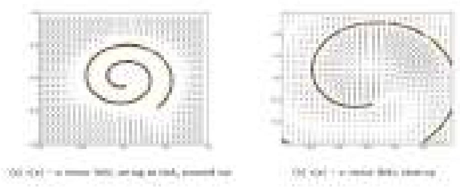
This trade-off forces the model to maintain only the variations in the data required to reconstruct the input without holding on to redundancies within the input. For most cases, this involves constructing a loss function where one term encourages our model to be sensitive to the inputs and a second term discourages memorization/overfitting (ie. an added regularizer).

* Variational Autoencoders
* **Denoising Autoencoders**

  So far I've discussed the concept of training a neural network where the input and outputs are identical and our model is tasked with reproducing the input as closely as possible while passing through some sort of information bottleneck. Recall that I mentioned we'd like our autoencoder to be sensitive enough to recreate the original observation but insensitive enough to the training data such that the model learns a generalizable encoding and decoding. Another approach towards developing a generalizable model is to slightly corrupt the input data but still maintain the uncorrupted data as our target output. With this


Add noise to the input image
Feed corrupted input into autoencoder
Measure reconstruction loss against original image

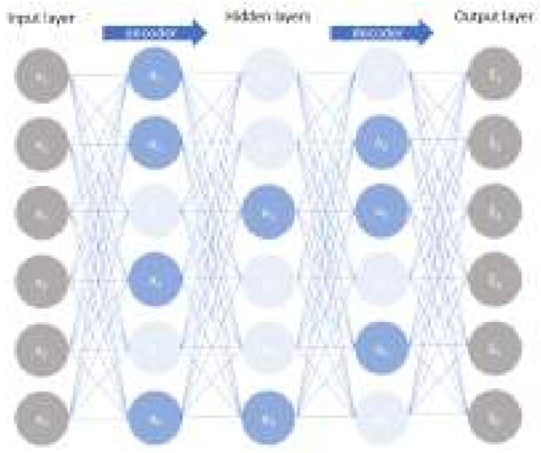
  approach, our model isn't able to simply develop a mapping which memorizes the training data because our input and target output are no longer the same. Rather, the model learns a vector field for mapping the input data towards a lower-dimensional manifold (recall from my earlier graphic that a manifold describes the high density region where the input data concentrates); if this manifold accurately describes the natural data, we've effectively "canceled out" the added noise.



  The above figure visualizes the vector field described by comparing the reconstruction of x with the original value of x. The yellow points represent training examples prior to the addition of noise. As you can see, the model has learned

to adjust the corrupted input towards the learned manifold.
It's worth noting that this vector field is typically only well behaved in the regions where the model has observed during training. In areas far away from the natural data distribution, the reconstruction error is both large and does not always point in the direction of the true distribution.

∗ **Sparse Autoencoders**
Sparse autoencoders offer us an alternative method for introducing an information bottleneck without requiring a reduction in the number of nodes at our hidden layers. Rather, we'll construct our loss function such that we penalize activations within a layer. For any given observation, we'll encourage our network to learn an encoding and decoding which only relies on activating a small number of neurons. It's worth noting that this is a different approach towards regularization, as we normally regularize the weights of a network, not the activations.
A generic sparse autoencoder is visualized below where the opacity of a node corresponds with the level of activation. It's important to note that the individual nodes of a trained model which activate are data-dependent, different inputs will result in activations of different nodes through the network.



One result of this fact is that we allow our network to sensitize individual hidden layer nodes toward specific attributes of the input data. Whereas an undercomplete autoencoder will use the entire network for every observation, a sparse autoencoder will be forced to selectively activate regions of the network depending on the input data. As a result, we've limited the network's capacity

to memorize the input data without limiting the network's capability to extract features from the data. This allows us to consider the latent state representation and regularization of the network separately, such that we can choose a latent state representation (ie. encoding dimensionality) in accordance with what makes sense given the context of the data while imposing regularization by the sparsity constraint.
There are two main ways by which we can impose this sparsity constraint; both involve measuring the hidden layer activations for each training batch and adding some term to the loss function in order to penalize excessive activations.

– **GANs(Genrative Adversarial Networks)**
Generative adversarial networks (GANs) are an exciting recent innovation in machine learning. GANs are generative models: they create new data instances that resemble your training data. For example, GANs can create images that look like photographs of human faces, even though the faces don't belong to any real person. These images were created by a GAN:
GANs achieve this level of realism by pairing a generator, which learns to produce the target output, with a discriminator, which learns to distinguish true data from the output of the generator. The generator tries to fool the discriminator, and the discriminator tries to keep from being fooled.

Generative Model
What does "generative" mean in the name "Generative Adversarial Network"? "Generative" describes a class of statistical models that contrasts with discriminative models.
Informally:

∗ Generative models can generate new data instances.
∗ Discriminative models discriminate between different kinds of data instances.

A generative model could generate new photos of animals that look like real animals, while a discriminative model could tell a dog from a cat. GANs are just one kind of generative model.
More formally, given a set of data instances X and a set of labels Y:

∗ Generative models capture the joint probability p(X, Y), or just p(X) if there are no labels.
∗ Discriminative models capture the conditional probability p(Y | X).

A generative model includes the distribution of the data itself, and tells you how likely a given example is. For example, models that predict the next word in a sequence are typically generative models (usually much simpler than GANs) because

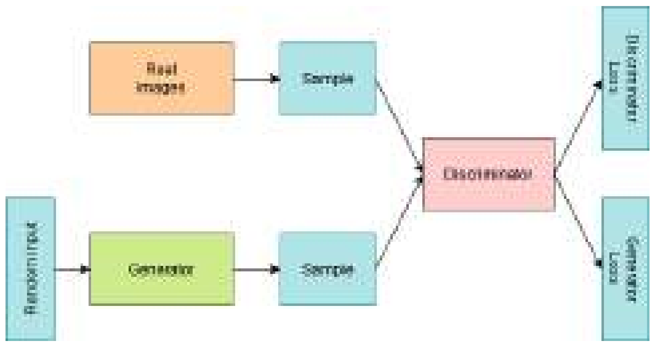
they can assign a probability to a sequence of words.
A discriminative model ignores the question of whether a given instance is likely, and just tells you how likely a label is to apply to the instance.
Note that this is a very general definition. There are many kinds of generative models. GANs are just one kind of generative model.

Overview of GAN Structure
A generative adversarial network (GAN) has two parts:

∗ The generator learns to generate plausible data. The generated instances become negative training examples for the discriminator.

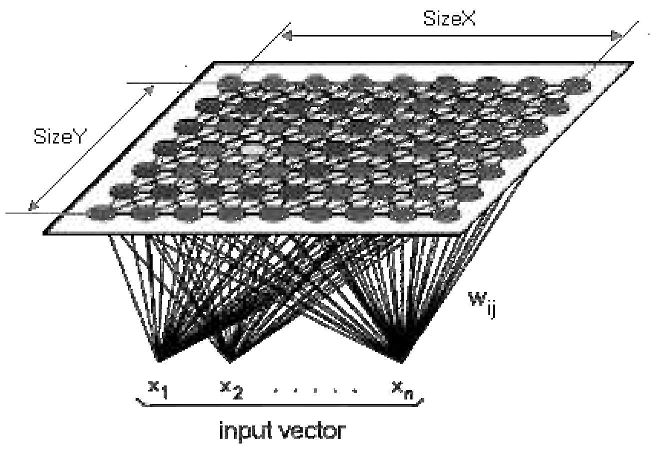
∗ The discriminator learns to distinguish the generator's fake data from real data. The discriminator penalizes the generator for producing implausible results.

When training begins, the generator produces obviously fake data, and the discriminator quickly learns to tell that it's fake. Here's a picture of the whole system:



– **Self Organizing Maps or Kohonen Network**
Kohonen, a self-organising map is an unsupervised learning model, intended for applications in which maintaining a topology between input and output spaces is of importance. The notable characteristic of this algorithm is that the input vectors that are close — similar — in high dimensional space are also mapped to nearby nodes in the 2D space. It is in essence a method for dimensionality reduction, as it maps high-dimension inputs to a low (typically two) dimensional discrete representation and conserves the underlying structure of its input space. A valuable detail is that the entire learning occurs without supervision i.e. the nodes are self-organising. They are also called feature maps, as they are essentially retraining the features of the input data, and simply grouping themselves according to the similarity between one another. This has a pragmatic value for visualising complex or large quantities of high dimensional data and representing the relationship between them into a low, typically two-dimensional, field to see if the given unlabelled data has any structure to it.

A Self-Organizing Map (SOM) differs from typical ANNs both in its architecture



and algorithmic properties. Firstly, its structure comprises a single-layer linear 2D grid of neurons, instead of a series of layers. All the nodes on this grid are connected directly to the input vector, but not to one another, meaning the nodes do not know the values of their neighbours, and only update the weight of their connections as a function of the given inputs. The grid itself is the map that organises itself at each iteration as a function of the input of the input data. As such, after clustering, each node has its own (i,j) coordinate, which allows one to calculate the Euclidean distance between 2 nodes by means of the Pythagorean theorem.

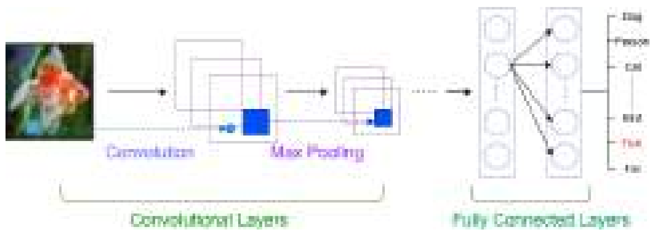
- **Architectures based on CNN and categorized according to tasks**

  - **Image Classification**
    Image Classification is a supervised task where we assign a category to a particular image. It is similar to any other classification tasks where we classify an image whether it is something or not (Binary Classifier), or assign a class from a number of different classes(Multi-Class classifier). Eg:- Classifying an image whether it's cat or not, ImageNet 1000 class challenge, etc.

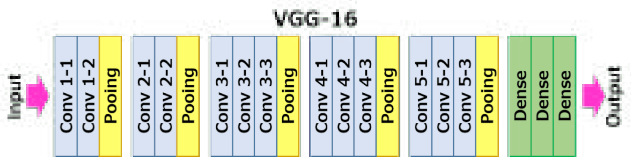
    It is a very generic Computer Vision task and is unable to solve several other aspects such location of the object, no. of objects and also fails to detect if different objects are present. The above tasks are solved with Image Classification being the basic building blocks of the solution algorithm.

    General Architecture:
    Basic blocks of Image Recognition consists of CNN layers. Various pooling, sampling and residual blocks are also added specific to each model to increase the accuracy, downsample feature space, etc. Finally there are a few fully-connected layers which end with a logistic regressor(Binary classification) or softmax unit(Multi-Class task)

to get the final class of the image.



* **VGG**
  Type Task : It is a supervised task mostly used for Image classification. **Definition :** VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford. It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. It was one of the first models which showed the significance of deeper models when it came to increasing the accuracy of Image based tasks.

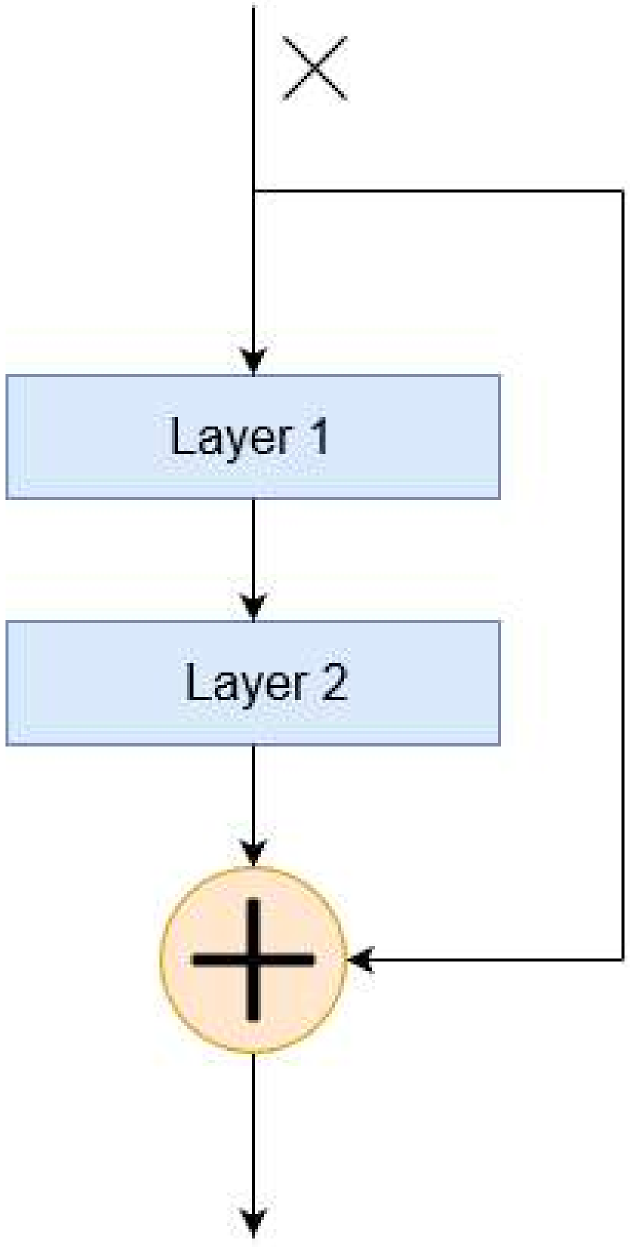


* **ResNet**

  Type Task : It's a type of supervised model which is specifically designed for Image Classification tasks. It can also be tweaked and fine-tuned for Object Detection and several other tasks as well.

  Definition : With the Inception Network developed by Google achieving the contemporary best results for the ImageNet challenge, it was clear that the deeper the model architecture is, the better will be the classification accuracy. Future attempts to make the model deeper faced with various shortcomings due to restrictions in computation power available at the time. Even after the development of better hardwares the model architectures were not able to improve accuracy as they faced issues like overfitting, vanishing gradients, etc. To tackle this problem He et al. at Microsoft Research Asia came with his paper describing the Residual Network architecture.

  A basic block of the ResNet has two convolutional and pooling layers. The output of the second layer is added with the input of the block. Hence a residue of the input skips a couple layers and gets added to the output layer. This helped in tackling the vanishing gradient problem to a greater extent and thus achieving the best accuracy to the ImageNet challenge which is the SOTA till date



∗ **DenseNet**
Type Task : It's also a type of supervised model which is mainly used for Image Classification tasks. It can also be tweaked and fine-tuned for Semantic Segmentation and several semi-supervised tasks as well.

Motivation : Several problems arise as CNNs go deeper. This is because the path for information from the input layer until the output layer (and for the gradient in the opposite direction) becomes so big, that they can vanish before reaching the other side. DenseNet simply connects every layer directly with each other and hence solves the above problem by ensuring a maximum information flow. Instead of drawing representational power from extremely deep or wide architectures, DenseNets exploit the potential of the network through feature reuse.

Counter-intuitively, by connecting this way DenseNets require fewer parameters than an equivalent traditional CNN, as there is no need to learn redundant feature maps. Furthermore, some variations of ResNets have proven that many layers are barely contributing and can be dropped. In fact, the number of parameters of ResNets are big because every layer has its weights to learn. Instead, DenseNets layers are very narrow (e.g. 12 filters), and they just add a small set of new feature-maps.

Another problem with very deep networks was the problems to train, because of the mentioned flow of information and gradients. DenseNets solve this issue since each layer has direct access to the gradients from the loss function and the original input image.

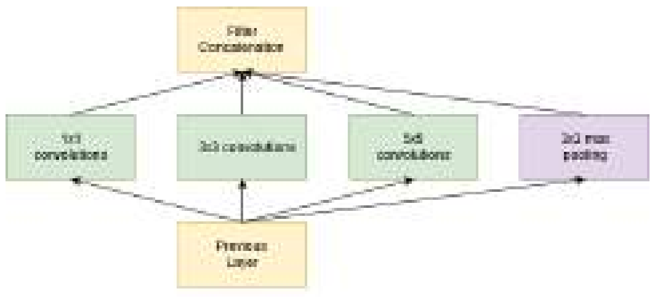
∗ **AlexNet**

Type Task : It is a supervised task mostly used for Image classification.
Definition : AlexNet is the name of a convolutional neural network which has had a large impact on the field of machine learning, specifically in the application of deep learning to machine vision. It famously won the 2012 ImageNet LSVRC-2012 competition by a large margin (15.3% VS 26.2% (second place) error rates). The network had a very similar architecture as LeNet by Yann LeCun et al but was deeper, with more filters per layer, and with stacked convolutional layers. It consisted of 11×11, 5×5,3×3, convolutions, max pooling, dropout, data augmentation, ReLU activations, SGD with momentum. It attached ReLU activations after every convolutional and fully-connected layer. AlexNet was trained for 6 days simultaneously on two Nvidia Geforce GTX 580 GPUs which is the reason for why their network is split into two pipelines.

∗ **GoogleNet or Inception Net**

Type Task : It is a supervised task mostly used for Classification. It also acts as a base model for various object detection, segmentation, action recognition, etc tasks.
Definition : Inception Modules are used in Convolutional Neural Networks to allow for more efficient computation and deeper Networks through a dimen-

sionality reduction with stacked 1×1 convolutions. The modules were designed to solve the problem of computational expense, as well as overfitting, among other issues. The solution, in short, is to take multiple kernel filter sizes within the CNN, and rather than stacking them sequentially, ordering them to operate on the same level.

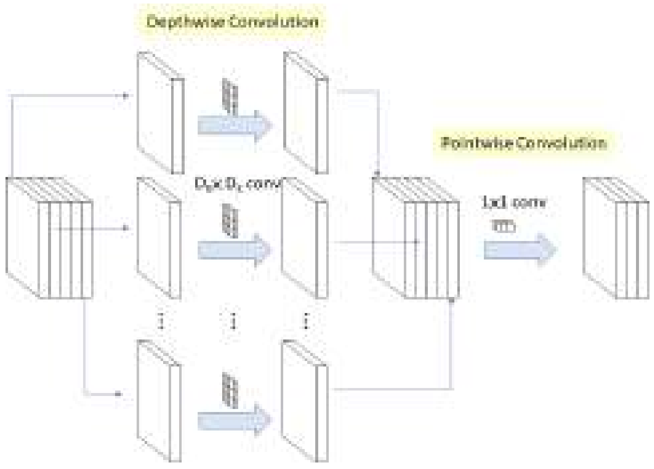


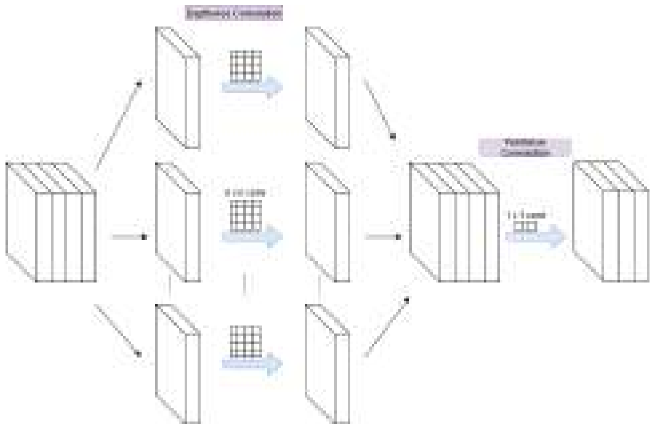
Inception module(Naive version)

* **MobileNet**
  Type Task : It is a supervised task mostly used for Classification. It also acts as a base model for various object detection, segmentation, action recognition, etc tasks.
  Definition : Deep learning has fueled tremendous progress in the field of computer vision in recent years, with neural networks repeatedly pushing the frontier of visual recognition technology. While many of those technologies such as object, landmark, logo and text recognition are provided for internet-connected devices through the Cloud Vision API, we believe that the ever-increasing computational power of mobile devices can enable the delivery of these technologies into the hands of our users, anytime, anywhere, regardless of internet connection. However, visual recognition for on device and embedded applications poses many challenges — models must run quickly with high accuracy in a resource-constrained environment making use of limited computation, power and space.

  MobileNet is a family of mobile-first computer vision models, designed to effectively maximize accuracy while being mindful of the restricted resources for an on-device or embedded application. They are small, low-latency, low-power models parameterized to meet the resource constraints of a variety of use cases. They can be built upon for classification, detection, embeddings and segmentation similar to how other popular large scale models, such as Inception, are used.
* **Xception Net**
  Original Depthwise Separable Convolution

  The original depthwise separable convolution is the depthwise convolution fol-



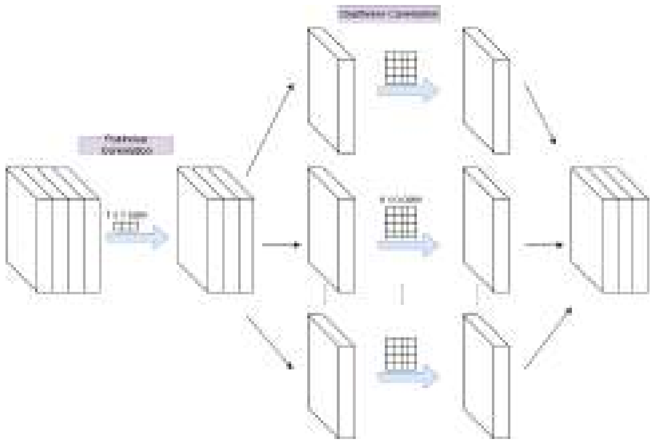


lowed by a pointwise convolution.

- Depthwise convolution is the channel-wise n×n spatial convolution. Suppose in the figure above, we have 5 channels, then we will have 5 n×n spatial convolution.
- Pointwise convolution actually is the 1×1 convolution to change the dimension.

Compared with conventional convolution, we do not need to perform convolution across all channels. That means the number of connections are fewer and the model is lighter.

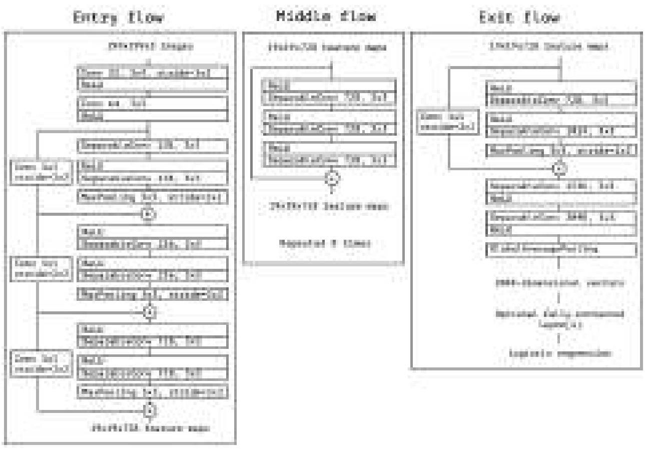
Modified Depthwise Separable Convolution in Xception The modified depth-



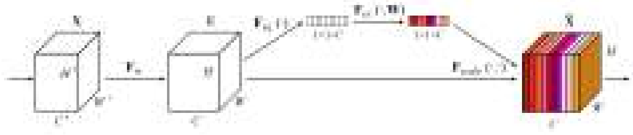
wise separable convolution is the pointwise convolution followed by a depthwise convolution. This modification is motivated by the inception module in Inception-v3 that 1×1 convolution is done first before any n×n spatial convolutions. Thus, it is a bit different from the original one. (n=3 here since 3×3 spatial convolutions are used in Inception-v3.) Two minor differences:

- The order of operations: As mentioned, the original depthwise separable convolutions as usually implemented (e.g. in TensorFlow) perform first channel-wise spatial convolution and then perform 1×1 convolution whereas the modified depthwise separable convolution perform 1×1 convolution first then channel-wise spatial convolution. This is claimed to be unimportant because when it is used in stacked settings, there are only small differences appearing at the beginning and at the end of all the chained inception modules.
- The Presence/Absence of Non-Linearity: In the original Inception Module, there is non-linearity after the first operation. In Xception, the modified depthwise separable convolution, there is NO intermediate ReLU non-linearity.

Overall Architecture



∗ **SEnet**



Squeeze-and-Excitation Networks (SENets) introduce a building block for CNNs that improves channel interdependencies at almost no computational cost.

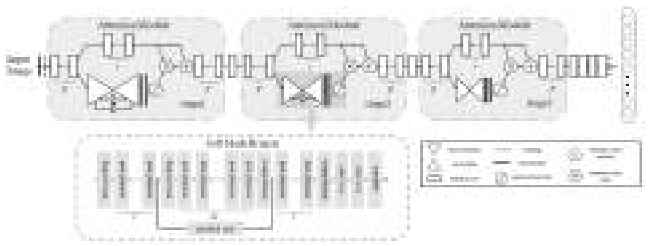
* **Residual Attention Network**
Attention is, to some extent, motivated by how we pay visual attention to different regions of an image or correlate words in one sentence. Human visual attention allows us to focus on a certain region with "high resolution" (i.e. look at the pointy ear in the yellow box) while perceiving the surrounding image in "low resolution" (i.e. now how about the snowy background and the outfit?), and then adjust the focal point or do the inference accordingly. Attention Mechanism is also an attempt to implement the action of selectively concentrating on a few relevant things, while ignoring others in deep neural networks. The idea is to use higher-level semantics to select relevant information.

The Attention Mechanism is introduced in the Resnet using two branches a "trunk" branch T(x) composed of two consecutive residual modules, and an hourglass mask branch M(x).

- Trunk Branch: It is the upper branch in the attention module for feature extraction. They can be Pre-Activation ResNet blocks or other blocks. With input x, it outputs T(x).
- Mask Branch: It uses bottom-up top-down structure to learn the same-size mask M(x). This M(x) is used as control gates similar to Highway Network.

Finally, the output of Attention Module H is:

$$H_{i,c}(x) = M_{i,c}(x) * T_{i,c}(x)$$



The gradient rule of the mask branch is such that it prevents wrong gradients (from noisy labels) to update trunk parameters i.e. it helps trunk branches learn good features and whenever noisy labels are introduced, it makes sure it has no effect on trunk branches.

Also, the incremental nature of stacked network structure can gradually refine attention for complex images.

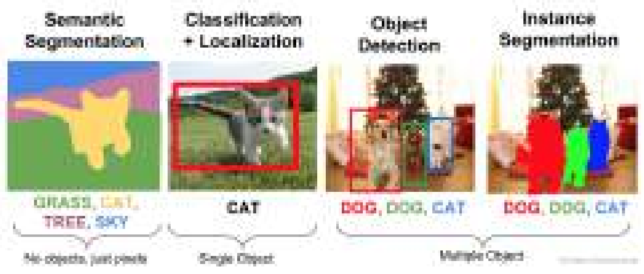
- **Efficient Net**
  Efficient net not only focuses on improving the accuracy but also the efficiency of the models. Recently, the models used for Image Recognition tasks are pretty heavier for a CPU to handle and hence we need GPUs to train these models or even get a prediction from it. Thus there is a need for which can be more efficient in terms of computation along with giving a reasonable accuracy.

  CNNs are commonly developed at a fixed resource cost, and then scaled up in order to achieve better accuracy when more resources are made available.Typically, model scaling is done to arbitrarily increase the CNN depth or width, or to use larger input image resolution for training and evaluation. While these methods do improve accuracy, they usually lead to suboptimal performance.

  In this model, a compound scaling method is proposed which has more control over model performance by having constraints on the scaling coefficients. In other words when to increase or decrease depth, height and resolution of a certain network.

  The first step in the compound scaling method is to perform a grid search to find the relationship between different scaling dimensions of the baseline network under a fixed resource constraints (e.g., 2x more FLOPS).This determines the appropriate scaling coefficient for each of the dimensions mentioned above. We then apply those coefficients to scale up the baseline network to the desired target model size or computational budget.

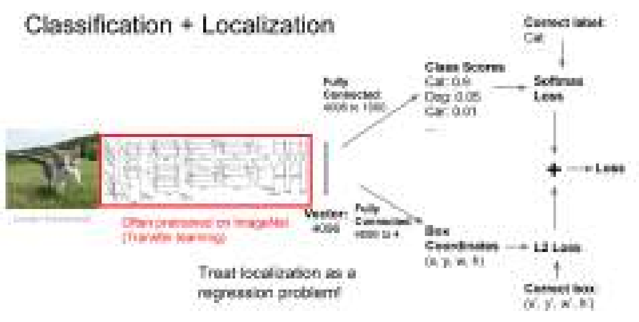
- **Object Detection**
  When working with images we have come across various problems like Classification, Segmentation etc. which is briefed in the image below.



  - Classification+Localization: We were able to classify an image as a cat. Great. Can we also get the location of the said cat in that image by drawing a bounding box around the cat? Here we assume that there is a fixed number(commonly 1) in the image.

- ∗ Object Detection: A More general case of the Classification+Localization problem. In a real-world setting, we don't know how many objects are in the image beforehand. So can we detect all the objects in the image and draw bounding boxes around them?

Instead of jumping directly into Object Detection let's first understand how we can solve this problem if we have a single object in the image i.e. Classification+Localization. We can treat Localization as a Regression problem.
Here is the architecture of the same :


Classification + Localization
Correct label: Cat
Class Scores
Cat: 0.9
Dog: 0.05
Car: 0.01
Fully Connected: 4096 x 1000
Softmax Loss
Loss
Often pretrained on ImageNet (Transfer learning)
Vector: 4096
Fully Connected: 4096 x 4
Box Coordinates (x, y, w, h)
L2 Loss
Correct box: (x', y', w', h')
Treat localization as a regression problem!

The input form in Classification is (X,y) where X is the image and y is the class labels. Now you might wonder what extra do we need to feed as an input. Let me tell you, it's simple as above, it also has a format of (X,y) , where X is still the image and y is an array containing (class_label,x,y,w,h).
Next question that hits the mind is what does x,y,w,h means here,let me help you :
x = bounding box top left corner x-coordinate
y = bounding box top left corner y-coordinate
w = width of bounding box in pixel
h = height of bounding box in pixel

Model: So in this setting we create a multi-output model which takes an image as the input and has (n_labels + 4) output nodes. n_labels nodes for each of the output class and 4 nodes that give the predictions for (x,y,w,h).
Loss: In such a setting setting up the loss is pretty important. Normally the loss is a weighted sum of the Softmax Loss(from the Classification Problem) and the regression L2 loss(from the bounding box coordinates).
Loss = alpha*Softmax_Loss + (1-alpha)*L2_Loss
Since these two losses would be on a different scale, the alpha hyper-parameter needs to be tuned.

Object Detection

Selective Search

Selective Search is a region proposal algorithm used in object detection. It is designed to be fast with a very high recall. It is based on computing hierarchical grouping of similar regions based on color, texture, size and shape compatibility.Here are the steps of Selective Search:

1.Generate initial sub-segmentation, we generate many candidate regions

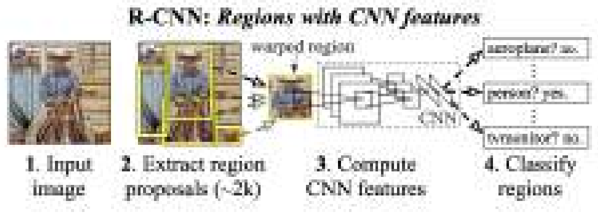
2.Use greedy algorithm to recursively combine similar regions into larger ones

3.Use the generated regions to produce the final candidate region proposals

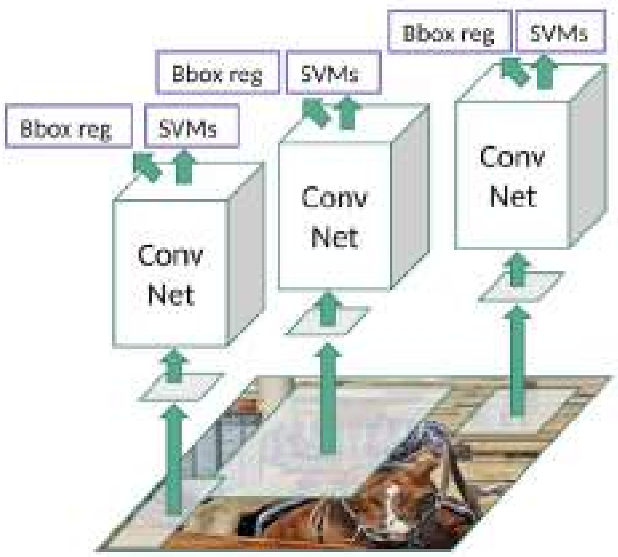
* **RCNN**

  The architecture goes like this :

  The 2000 candidate region proposals extracted from Selective Search Algo-



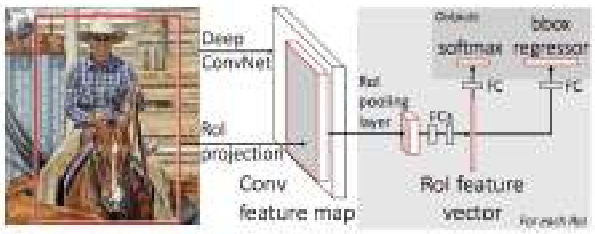
  rithm are warped into a square and fed into a convolutional neural network that produces a 4096-dimensional feature vector as output. The CNN acts as a feature extractor and the output dense layer consists of the features extracted from the image and the extracted features are fed into an SVM to classify the presence of the object within that candidate region proposal. In addition to predicting the presence of an object within the region proposals, the algorithm also predicts four values which are offset values to increase the precision of the bounding box. For example, given a region proposal, the algorithm would have predicted the presence of a person but the face of that person within that region proposal could've been cut in half. Therefore, the offset values help in adjusting the bounding box of the region proposal.



Problems with RCNN

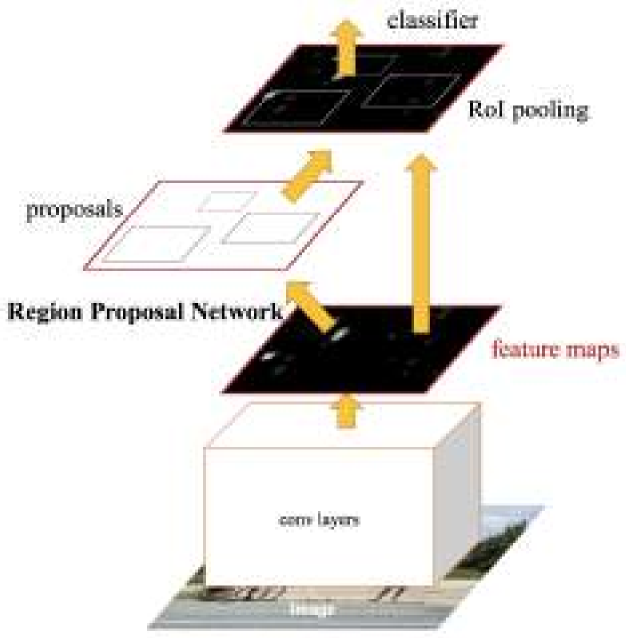
1. It still takes a huge amount of time to train the network as you would have to classify 2000 region proposals per image.
2. It cannot be implemented real time as it takes around 47 seconds for each test image.
3. The selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.

* **Fast RCNN**



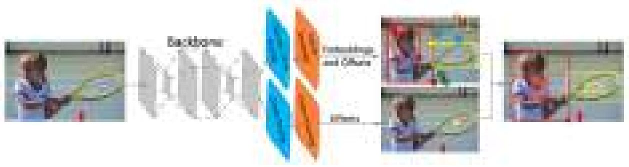
The same author of the previous paper(R-CNN) solved some of the drawbacks of R-CNN to build a faster object detection algorithm and it was called Fast R-CNN. The approach is similar to the R-CNN algorithm. But, instead of feeding the region proposals to the CNN, we feed the input image to the CNN to generate a convolutional feature map. From the convolutional feature map, we identify the region of proposals and warp them into squares and by using a RoI pooling layer we reshape them into a fixed size so that it can be fed into a fully connected layer. From the RoI feature vector, we use a softmax layer to predict the class of the proposed region and also the offset values for the bounding box. The reason "Fast R-CNN" is faster than R-CNN is because you don't have to feed 2000 region proposals to the convolutional neural network every time. Instead, the convolution operation is done only once per image and a feature map is generated from it.

∗ **Faster RCNN**
Similar to Fast R-CNN, the image is provided as an input to a convolutional net-



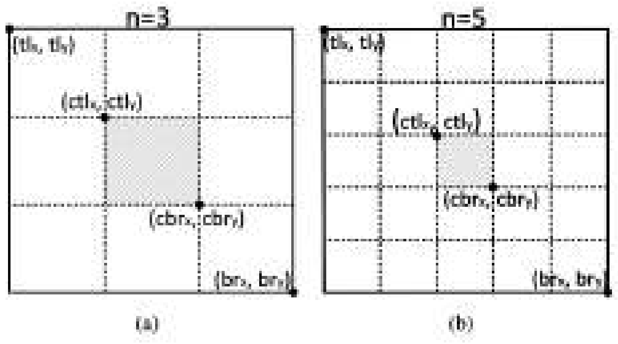
work which provides a convolutional feature map. Instead of using selective search algorithm on the feature map to identify the region proposals, a separate network is used to predict the region proposals. The predicted region proposals are then reshaped using a RoI pooling layer which is then used to classify the image within the proposed region and predict the offset values for the bounding boxes.

∗ **Center Net**
The network uses additional information (centeredness information) to perceive the visual patterns within each proposed region. Instead of using two



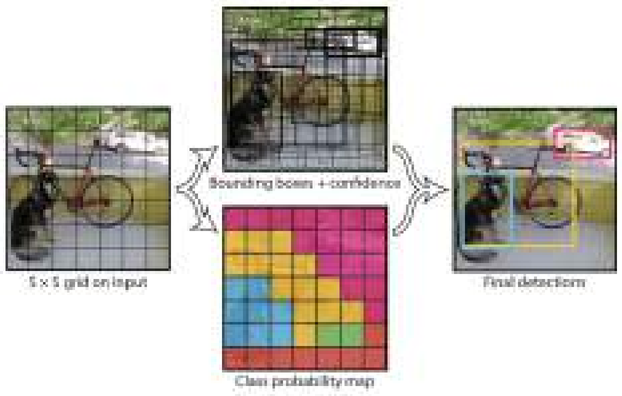
corner information, it uses triplets to localize objects. The work states that if a predicted bounding box has a high IoU with the ground-truth box, then the probability that the center keypoint in its central region is predicted as the same class is high, and vice versa.

It uses new pooling method is proposed to capture richer and more recognizable visual patterns known as Center Pooling. The figure above shows how is



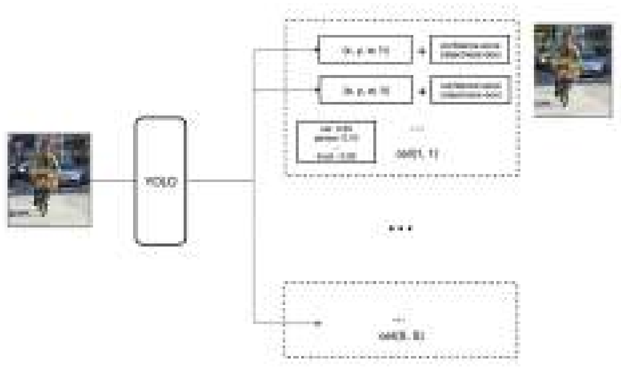
the operation performed.Given the feature map from the backbone layer, we determine if a pixel in the feature map is a center keypoint. The pixel in the feature map itself does not contain enough centeredness information of the object. Therefore, the maximum value of both horizontal and vertical directions are found and added together.By using this it is claimed that better detection of center keypoints are made.

∗ **You look only once(YOLO)**



Let's look step by step how YOLO works :

1. YOLO divides the input image into an S×S grid. Each grid cell predicts only one object.
2. Each grid cell predicts a fixed number of boundary boxes.
3. For each grid cell,
-it predicts B boundary boxes and each box has one box confidence score,
-it detects one object only regardless of the number of boxes B,
-it predicts C conditional class probabilities (one per class for the likeliness of the object class)
4. Example : To evaluate PASCAL VOC, YOLO uses 7×7 grids (S×S), 2 boundary boxes (B) and 20 classes (C) .



5. Each boundary box contains 5 elements: (x, y, w, h) and a box confidence score. The confidence score reflects how likely the box contains an object (objectness) and how accurate is the boundary box.
6. We normalize the bounding box width w and height h by the image width and height. x and y are offsets to the corresponding cell. Hence, x, y, w and h are all between 0 and 1.
7. Each cell has 20 conditional class probabilities. The conditional class probability is the probability that the detected object belongs to a particular class (one probability per category for each cell).
8. So, YOLO's prediction has a shape of (S, S, B×5 + C) = (7, 7, 2×5 + 20) = (7, 7, 30).

Benefits of YOLO
1. Fast. Good for real-time processing.
2. Predictions (object locations and classes) are made from one single network. Can be trained end-to-end to improve accuracy.
3. YOLO is more generalized. It outperforms other methods when generalizing from natural images to other domains like artwork.

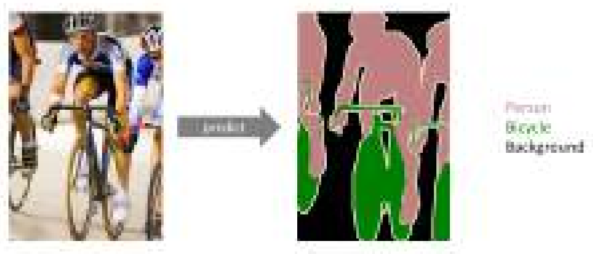
4. Region proposal methods limit the classifier to the specific region. YOLO accesses the whole image in predicting boundaries. With the additional context, YOLO demonstrates fewer false positives in background areas.
5. YOLO detects one object per grid cell. It enforces spatial diversity in making predictions.

- **Semantic Segmentation**
  The goal of semantic image segmentation is to label each pixel of an image with a corresponding class of what is being represented. Because we're predicting every pixel in the image, this task is commonly referred to as dense prediction.

  We can think of semantic segmentation as Image Classification[2.3.1] at a pixel level. For example, in an image that has many cars, segmentation will label all the objects as car objects.However, a separate class of models known as Instance Segmentation is able to label the separate instances where an object appears in an image. This kind of segmentation can be very useful in applications that are used to count the number of objects, such as counting the amount of foot traffic in a mall.

  In a basic sense, Semantic Segmentation does not differentiate between the objects of the same class as it assigns the same color to the output of that class in the output resolution map whereas the task of Instance segmentation is to distinguish between separate objects of the same class.

  Below figure represents how a segmentation map is generated from an Image.

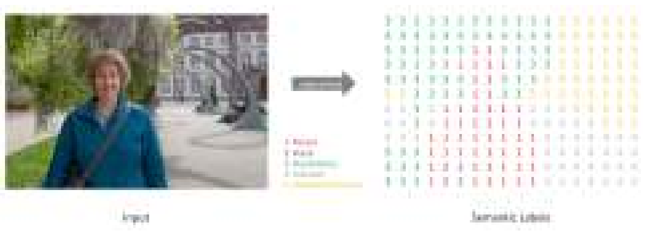


  Formal Definition :

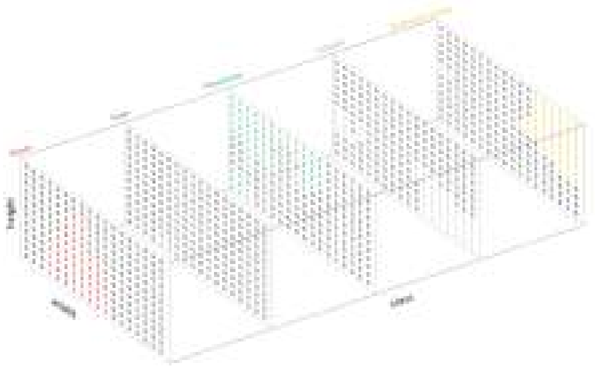
  It is mostly seen as a Supervised problem but there are various research papers working on how to make it semi supervised or unsupervised.

  Our goal is to take either a RGB color image (height×width×3) or a grayscale image (height×width×1) and output a segmentation map where each pixel contains a class label represented as an integer (height×width×1).

Similar to how we treat standard categorical values, we'll create our target by one-



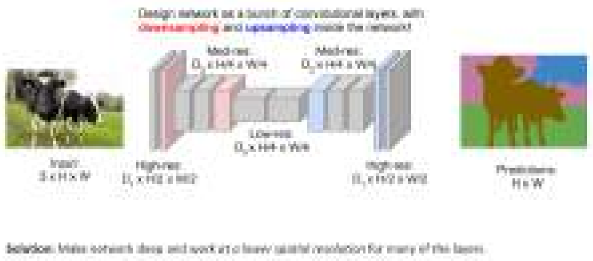
hot encoding the class labels - essentially creating an output channel for each of the possible classes.



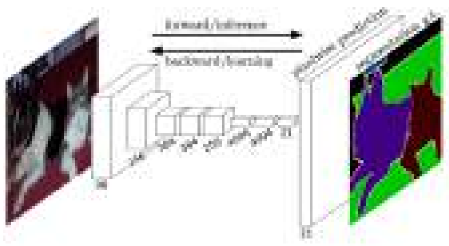
A prediction can be collapsed into a segmentation map (as shown in the first image) by taking the argmax of each depth-wise pixel vector.
We can easily inspect a target by overlaying it onto the observation. When we overlay a single channel of our target (or prediction), we refer to this as a mask which illuminates the regions of an image where a specific class is present.

Constructing a General Architecture :
A naive approach towards constructing a neural network architecture for this task is to simply stack a number of convolutional layers (with same padding to preserve dimensions) and output a final segmentation map. This directly learns a mapping from the input image to its corresponding segmentation through the successive transformation of feature mappings; however, it's quite computationally expensive to preserve the full resolution throughout the network.
Also, as we have developed an idea: earlier layers tend to learn low-level concepts while later layers develop more high-level (and specialized) feature mappings from the Image classification task. However we cannot use an Image classification architecture here because we have to maintain a full resolution of the image as in the former we just want to know what the image contains, not where the objects are located.
One popular approach for image segmentation models is to follow an encoder/decoder structure where we downsample the spatial resolution of the input, developing lower-

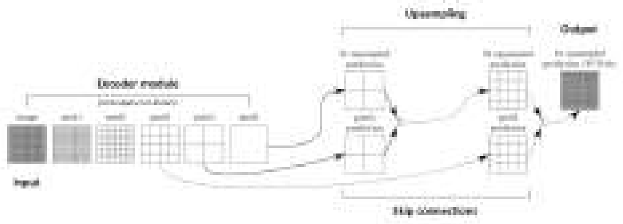
resolution feature mappings which are learned to be highly efficient at discriminating between classes, and the upsample the feature representations into a full-resolution segmentation map. The main idea is to learn representations at different scales or levels so that the learning can happen both at the coarser level and finer level.


Design network as a bunch of convolutional layers, with
downsampling and upsampling inside the network!
Med-res:
Med-res:
Low-res:
Input:
3 x H x W
High-res:
High-res:
Predictions:
H x W

- ∗ **Fully Convolutional Network(FCN)**
  This approach generally trains a fully connected network end to end. It uses alexnet as an encoder module and appends a decoder module with transposed convolutional layers to upsample the coarse feature maps into a full-resolution segmentation map. However, since in decoding from low resolution image to



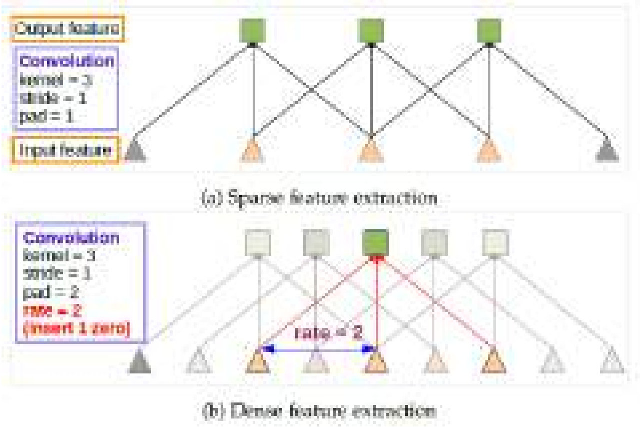
  high resolution, we lose various details an idea of skip connection was introduced. A skip connection is a connection that bypasses at least one layer. Here it is used to pass information from the down sampling step to the up sampling step. Merging features from various resolution levels helps combining context information with spatial information. Below is a figure demonstrating how skip connection was introduced. **Drawbacks :**



  In decoding and merging skip connections, we tend to lose various information around the corners and sometimes to train it fukky we need to take help of various data augmentation techniques.

* **Deep Lab**
  This architecture introduced the concept of Atrous Convolutions or Dilated Convolutions. The main idea behind this architecture are:
  1. Reduced feature resolution caused by a repeated combination of max-pooling and downsampling.
  2. Existence of objects at multiple scales.
  3. Reduced localization accuracy caused by DCNN's invariance since an object-centric classifier requires invariance to spatial transformations.

  To capture the essence of objects at multiple scale, concept of Atrous Convolutions was introduced. Atrous convolution allows us to enlarge the field of view of filters to incorporate larger context and thus offers an efficient mechanism to control the field-of-view and finds the best trade-off between accurate localization (small field-of-view) and context assimilation (large field-of-view). Below figure explains the concept of Atrous convolution where it is just convolving the input at a distance of r(r=2).



  With the help of changing the rate , the architecture exploits multi scale features with the help of multiple parallel filters at different rates. Below is the architecture:

  There are many versions of Deep Lab which is based on these 3 points and reengineering the Atrous spatial pyramid pooling (ASPP):
  1. Convolutions with upsampled filters for dense prediction tasks
  2. Atrous spatial pyramid pooling (ASPP) for segmenting objects at multiple scales
  3. Improving localization of object boundaries by using DCNNs.
  Below is the architecture:

* **ICNet**
  It is mostly used for real time semantic segmentation. It focuses on predicting

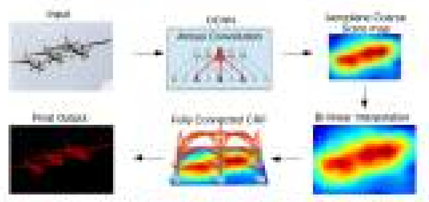


Fig. 1: Model Illustration. A Deep Convolutional Neural Network such as VGG-16 or ResNet-101 is employed in a fully convolutional fashion, using atrous convolution to reduce the degree of signal downsampling (from 32x down 8x). A bilinear interpolation stage enlarges the feature maps to the original image resolution. A fully connected CRF is then applied to refine the segmentation result and better capture the object boundaries.

the segmentation map faster than the above networks but at the cost of accuracy.

It uses deep supervision and runs the input image at different scales, each scale through their own subnetwork and progressively combining the results. The idea is to let low-resolution images go through the heavy CNN computation to predict a coarser prediction map and combine them with other resolution prediction maps where resolution increases gradually and the CNN computation gets low with increase in resolution. Below is the architecture:

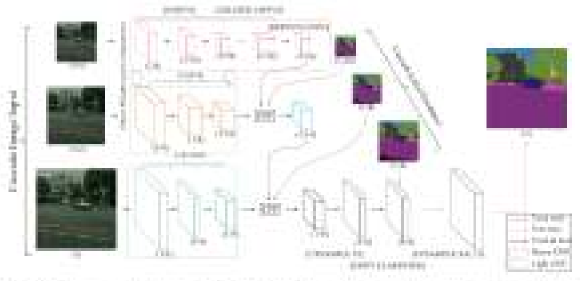


Fig. 2. Network architecture of ICNet. 'CFF' stands for cascade feature fusion detailed in Sec. 3.3. Numbers in parentheses are feature map size ratios to the full-resolution input. Operations are highlighted in brackets. The final ×4 upsampling in the bottom branch is only used during testing.

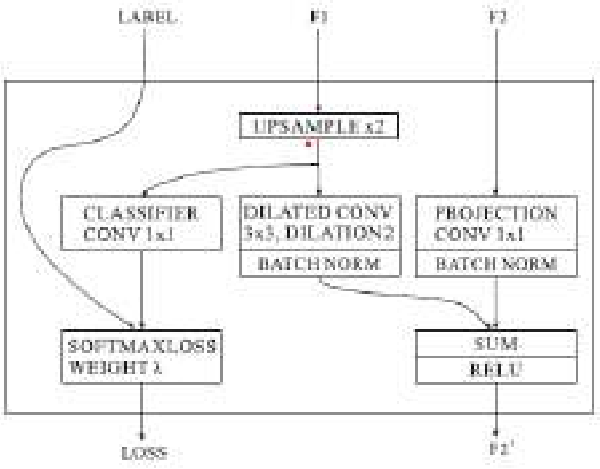


Fig. 3. Cascade feature fusion.

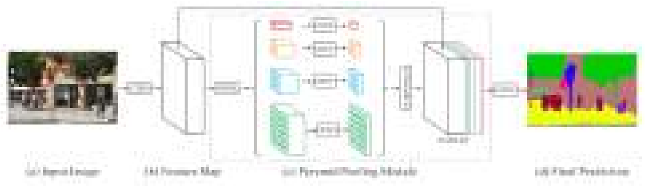
The combination of all the prediction maps at different resolution levels is done through Cascade Feature Fusion which is a method of using a coarser ground truth segmentation map to train the respective CNN of that resolution. It also uses upsampling operations like bilinear interpolations to combine the prediction maps of different resolutions.

Since, the low resolution images are trained heavily and high resolution images are using low dimensional CNN, the network takes very less time to train and predict enabling it to use in real time video segmentation.

∗ **PSPNet**
Performing multi scale analysis at different levels is computationally very expensive and hence PSPnet was introduced which gave a clever way to get around this by using multiple scales of pooling. It is also computationally very efficient.

It starts off with a standard feature extraction network (ResNet, DenseNet etc) and takes the features of the third downsampling for further processing. To get the multi-scale information, PSPNet applies 4 different max pooling operations with 4 different window sizes and strides. This effectively captures feature information from 4 different scales without the need for heavy individual processing of each one! It simply does a lightweight convolution on each one after, upsample so each feature map is at the same resolution, and concatenates them all.

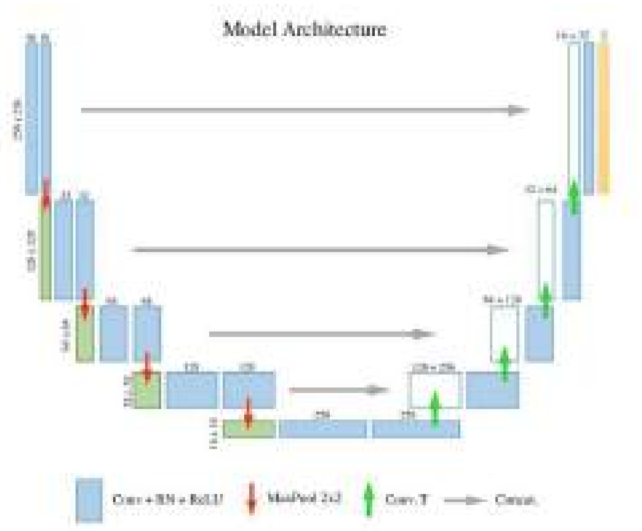
All of this is done on the lower resolution feature maps for high-speed. At the end, we upscale the output segmentation map to the desired size using bilinear interpolation . This technique of upscaling only after all processing is done is present in many state-of-the-art works. Below is the architecture.



∗ **U-Net**
The U-Net architecture is built upon the Fully Convolutional Network (FCN) and modified in a way that it yields better segmentation in medical imaging.

Compared to FCN-8, the two main differences are:
1. U-net is symmetric and
2. The skip connections between the downsampling path and the upsampling path apply a concatenation operator instead of a sum.

These skip connections intend to provide local information to the global information while upsampling. Because of its symmetry, the network has a large number of feature maps in the upsampling path, which allows it to transfer information. Information from larger scales (upper layers) can help the model classify better. Information from smaller scales (deeper layers) can help the model segment/localize better. Below is the architecture: U-Net architecture



is separated in 3 parts:
1. The contracting/downsampling path
2. Bottleneck
3. The expanding/upsampling path

3 QUANTUM COMPUTING MODULE

3.1 WHAT IS COMPUTATION?

Let us start this journey of playing with qubits by observing the broader landscape of computation. Usually, our view of computation is limited to an abstract 'input-processing-output' procedure, which is sufficient to represent most of the practical information processing channels. We know that an algorithm will compute the correct output for a given set of inputs in finite time by executing some well-defined operations on the inputs. It is a valid operational definition, but does not capture the essence of what computation is!

We want to present to you a physical view of computation. Every physical process evolves an initial state to a final state under influence of certain interactions. Does it sound something like an 'input-process-output' channel? Yes! When we develop computational systems, we are effectively encoding meaningful information in the states of physical systems and process this information by controlling the interactions. Only those operations can be carried out which can be mapped to physical processes. The task of building different kinds of computers boils down to identifying the right physical system and the right interactions which can solve our problems.

Let us consider a physical system consisting of three light bulbs, identified as B1, B2, and B3. We designate the 'switched-on' state of a bulb as 1 and the 'switched-off' state of a bulb as 0. In a sense, we are encoding the state-space information of a bulb in the symbols 1 or 0 and mapping it to the actual state of 'switched-on' and 'switched-off'. Now, we define a rule as follows: if both the bulbs B1 and B2 are alight, only then the bulb B3 will be switched-on; otherwise it will be switched-off. An electrical circuit corresponding to this rule is:

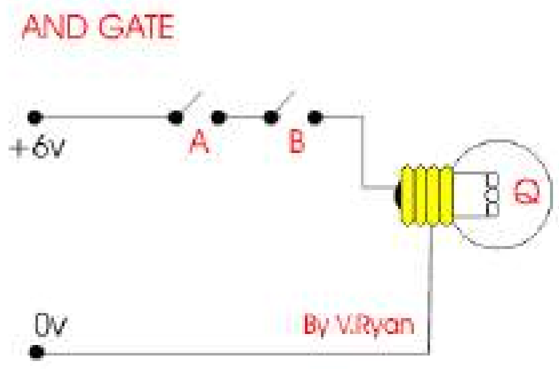


FIGURE 1: Illustration of 'AND' gate using switches,
Remark: Note that the bulb can either be in a switched-on state or the switched-off state at a particular instant of time. These two states are strictly mutually-exclusive.

Observe that we have introduced 'logic' and 'information processing' in this two-level system. In fact, this simple electrical circuit implements the AND logic gate of classical Boolean algebra! To implement more complex logic, we need to create more sophisticated physical systems. Now that we understand the relevance of information processing in physical systems, we will create a mathematical framework to model these states and interactions.

3.1.1 CLASSICAL COMPUTATION

Bit: We refer to this simple two-level system as a bit, which can exist in state 0 or state 1 at a particular instant of time. A single bit has the representation capacity of 2 states, while two bits have the representation capacity of 4 states: 00, 01, 10, 11. Consequently, 2 bits can process information on these 4 states. It is simple to extend this idea: an n-bit system has a representation capacity of 2^n states, and information can be processed on these 2n states. Since the bit-states 0 and 1 are mutually-exclusive, an n-bit system exists in exactly one of the 2^n states.

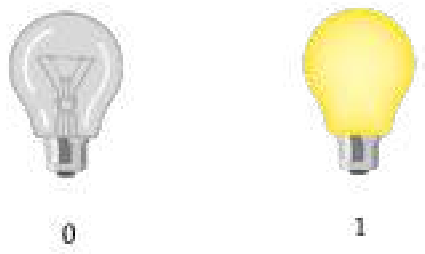
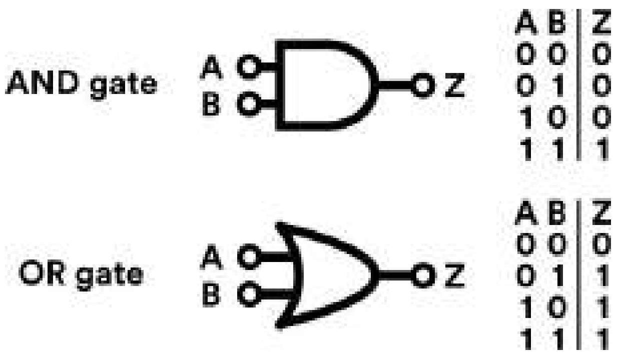


FIGURE 2: A Bit as possible states of a bulb

3.1.2 CLASSICAL 'AND' LOGIC GATE

Below is the **circuit-representation** of an AND gate. Given two input bits x_1 and x_2, the output bit x_3 is 1 if and only if x_1 is 1 and x_2 is 1. More precisely, $f(x_1, x_2) = x_1 \wedge x_2$



A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

FIGURE 3: Remarks:The AND Gate is implementing an irreversible computation. Given the result $x_1 \wedge x_2$ we can not retrieve the exact value of bits x_1 and x_2 , except when $x_1 x_2$ is 1. This idea is intimately related to the energy considerations of computations, which we will revisit when we discuss quantum computation.

The availability of different physical interactions makes it possible to design different types of computers. For example, classical computers operate on logical actions mapped to two-level electronic systems in transistors. The bit register is an abstraction of these states of transistors. Can we create information processors by manipulating quantum systems like electrons and photons? It turns out that we can, and that is what precisely what we are building at QpiAI.

Before we look at quantum computation as a physical process, let us define the fundamental unit of quantum information as an abstract entity: a qubit is the probabilistic generalization of a bit, and the state of a qubit is a ray in a complex vector space(Hilbert Space $\mathcal{H}$) with unit length.

3.1.3 QUBITS

The quantum bit is the fundamental unit of information in quantum mechanical information processing systems. Quantum information is represented by the states of qubits and quantum computation is performed by changing these states. We represent the state of a qubit by two complex numbers α and β, which determine the probability of a qubit being in the state 0 or 1. Let's take a canonical example: assume a bit being abstracted by a coin placed on a table, and the state is the H or T depending on the visible face.

Now, we toss this coin and observe it when it is spinning: we can not determine if its state is exactly 0 or 1, but can model its state by a probability distribution over all the possible combinations of states 0 and 1. More precisely, a qubit is the probabilistic generalization of a bit, and the complex numbers α and β are the probability amplitudes corresponding to states 0 and 1. We represent the qubit as $\alpha\,|0\rangle + \beta\,|1\rangle$,

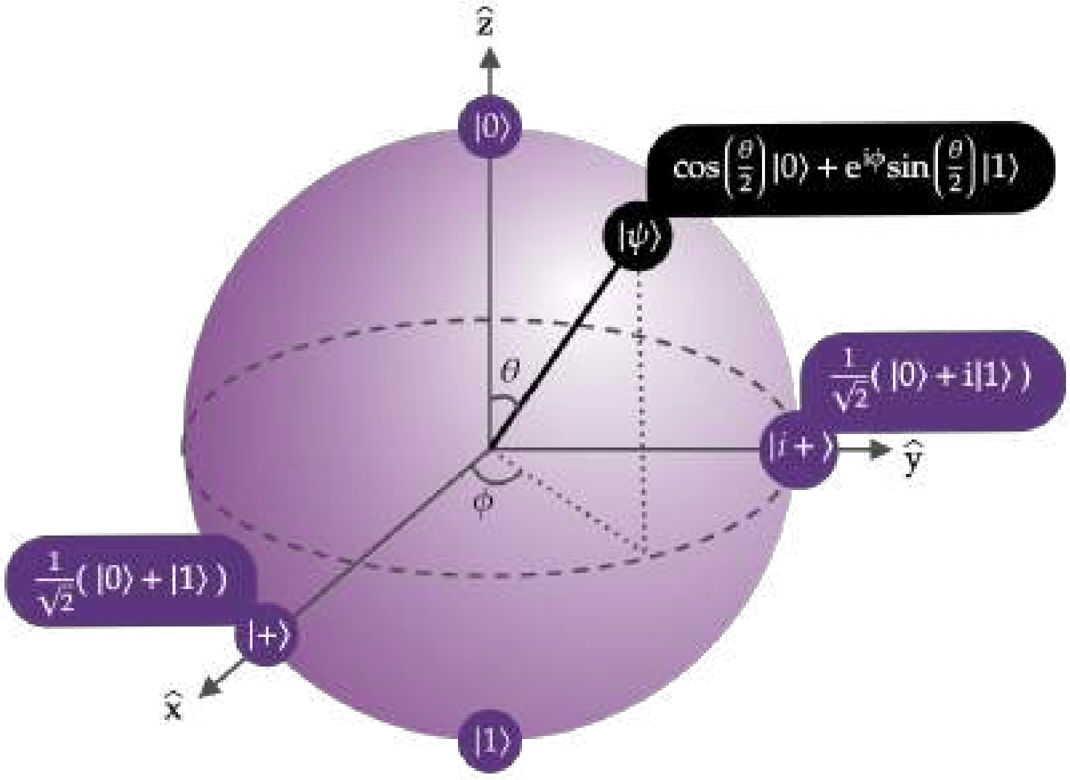
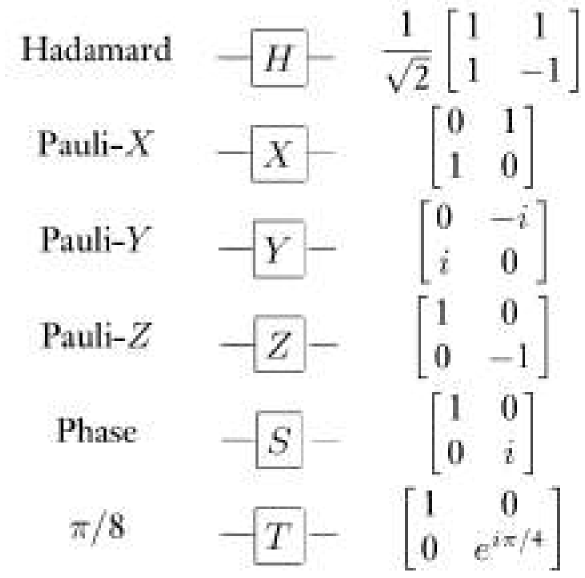

$\hat{z}$
$|0\rangle$
$\cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle$
$|\psi\rangle$
$\frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$
θ
$|i+\rangle$
$\hat{y}$
φ
$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$
$|+\rangle$
$\hat{x}$
$|1\rangle$

FIGURE 4: The states of the qubit can be visualised as points on the **Bloch Sphere**

3.1.4 QUANTUM GATES

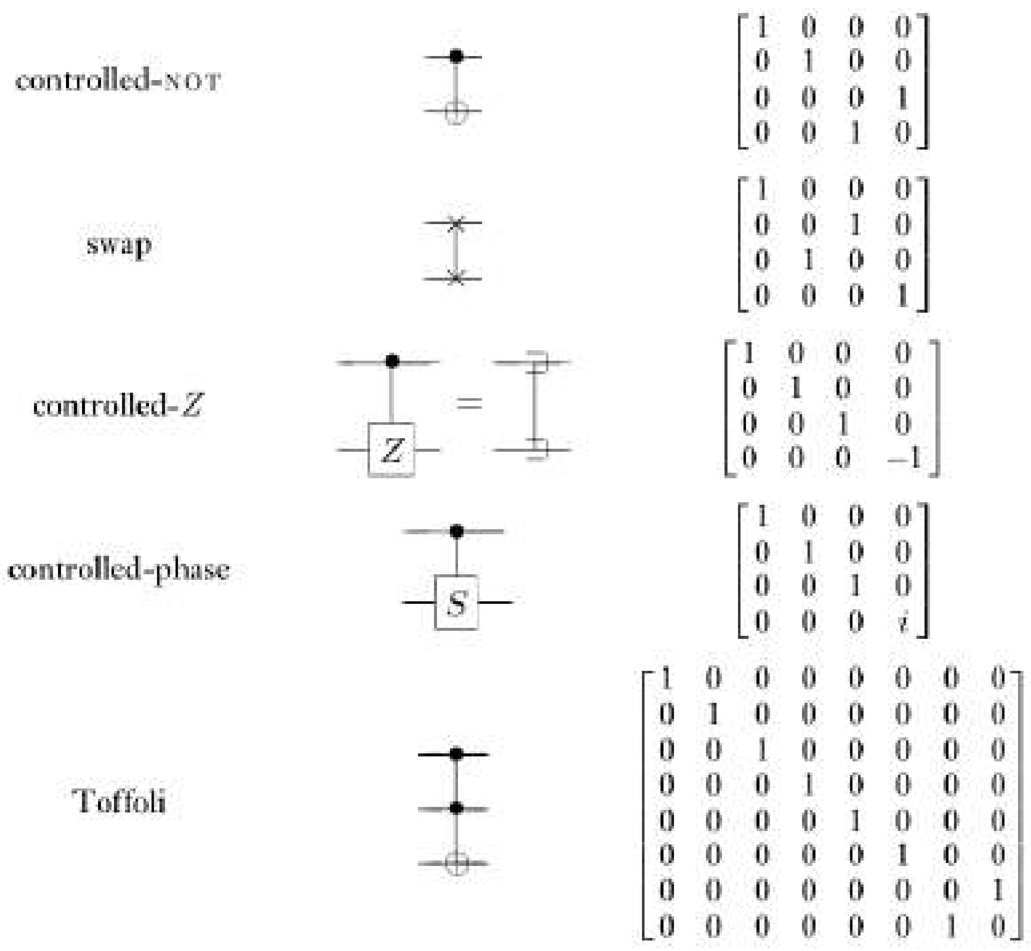
Quantum Gates are the basic operations on a set of Qubits. Since Qubits are complex vector spaces, operations on Qubits can be represented as complex matrices. A complex matrix(M) have to be **Unitary**(i.e $M^\dagger M = 1$) to represent a Quantum Gate. all the eigenvalues of such a matrix will be a pure complex number($e^{i\phi}$). This makes sure that the length of the vector remains the same even after the Quantum Gate. A Unitary matrix is a rotation in the complex vector space. Some of the commonly used Gates are shown below.

Measurement is a different operation on can do on a Qubit. A Measurement with a basis$\{|0\rangle, |1\rangle\}$ probabilistically projects the state($\alpha|0\rangle + \beta|1\rangle$) on to $|0\rangle$ with probability $|\alpha|^2$ and on to $|1\rangle$ with probability $|\beta|^2$.



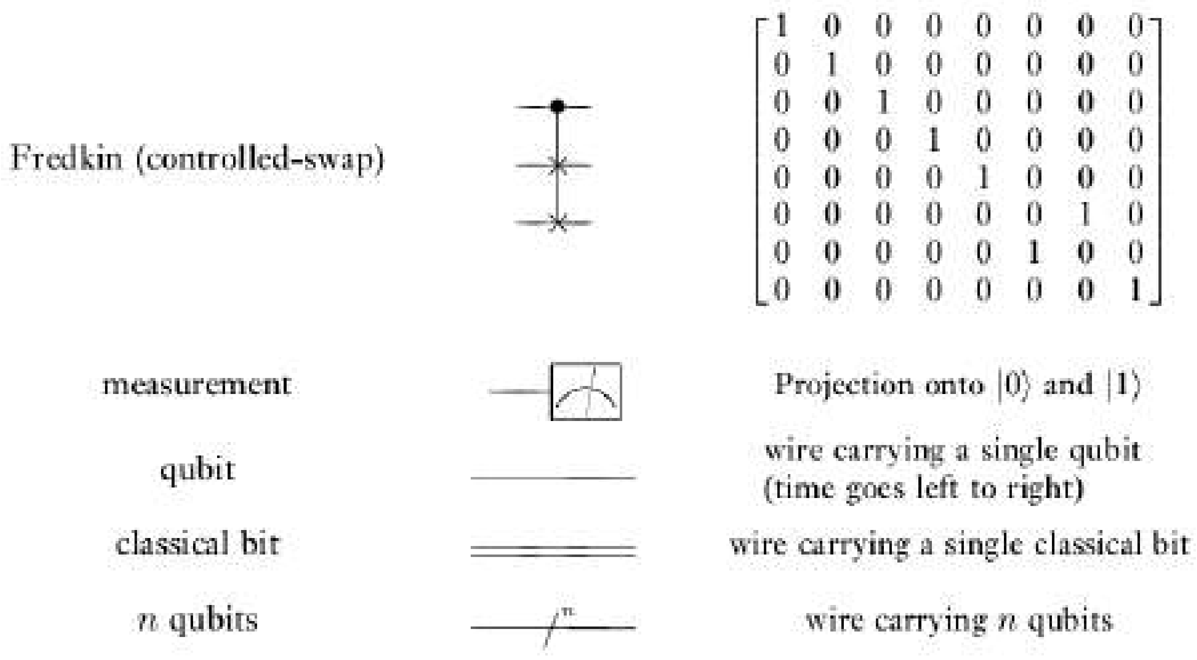
Hadamard	$\boxed{H}$	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$
Pauli-X	$\boxed{X}$	$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$
Pauli-Y	$\boxed{Y}$	$\begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}$
Pauli-Z	$\boxed{Z}$	$\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$
Phase	$\boxed{S}$	$\begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix}$
$\pi/8$	$\boxed{T}$	$\begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix}$

FIGURE 5: Some Common Gates and their Matrix Representation



controlled-NOT	$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$
swap	$\begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix}$
controlled-Z	$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&-1\end{bmatrix}$
controlled-phase	$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&i\end{bmatrix}$
Toffoli	$\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{bmatrix}$

FIGURE 6: Some Common Gates and their Matrix Representation



Fredkin (controlled-swap)	$\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\end{bmatrix}$
measurement	Projection onto $\lvert 0\rangle$ and $\lvert 1\rangle$
qubit	wire carrying a single qubit (time goes left to right)
classical bit	wire carrying a single classical bit
n qubits	wire carrying n qubits

FIGURE 7: Some Common Gates and their Matrix Representation

3.1.5 QUANTUM CIRCUITS

A Quantum Computation manipulates states of qubits by acting with 'unitary' operators.A **unitary operator** is a generalisation of rotations to complex vector spaces.Essentially unitary operators conserves probabilities. i.e the sum of probabilities for the qubit to be in state $|0\rangle , |1\rangle$ always adds up to one even after a Unitary transformation.

There are an infinite number of unitary operators that we can imagine. But it turns out that any such operation can be realised by combinations of a finite number of universal operators(Gates) (Solovay–Kitaev theorem) . If we can realise these universal gate sets by some physical phenomenon ,we can do universal Quantum Computation! This is analogous to how one can do arbitrary classical computations with a bunch of universal gates(NAND or NOR). This fact is the basis of the Universal Gate model Quantum Computation. A Quantum Circuit is such a sequence of Gates applied to a set of qubits to perform a well defined computation.

We have seen how quantum computers leverage Quantum Parallelism and Interference to gain a computational advantage. In essence a Quantum Computation is an orchestra of Interference effects and A quantum circuit is the notation that one can use to describe this orchestra!

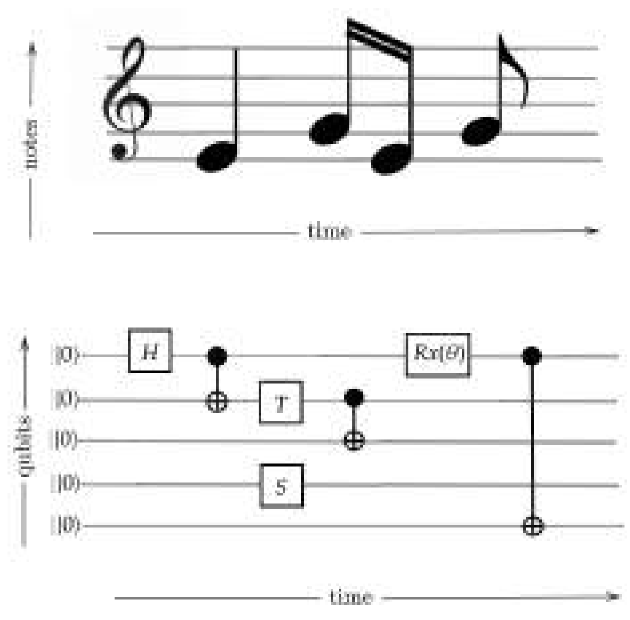


FIGURE 8: Sheet Music vs Quantum Ciruits

3.1.6 PREPARE SUPER POSITIONS

One qubit Complete Superposition states are,

- $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$
- $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$
- $|i+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle)$
- $|i-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle)$

n- Qubit superpositions,

$$|\psi\rangle = \frac{1}{\sqrt{2}} \sum_{i\in\{0,1\}^n} |i\rangle$$

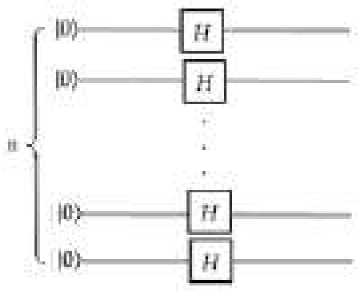


FIGURE 9: Circuit for preparing n-qubit superpositions

3.1.7 CIRCUIT FOR PREPARING BELL STATES

Entanglement is one of the most useful resources for doing Quantum Computation. Most Quantum algorithms use entanglement in some form or the other. There can be varied amounts of entanglement in states. Entanglement measures " How much uncertain are we about the state of subsystems even if we know the state of the whole".The simplest maximally entangled states are the four Bell States.

Bell States

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A|0\rangle_B + |1\rangle_A|1\rangle_B) \qquad |\Phi^-\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A|0\rangle_B - |1\rangle_A|1\rangle_B)$$

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A|1\rangle_B + |1\rangle_A|0\rangle_B) \qquad |\Psi^-\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A|1\rangle_B - |1\rangle_A|0\rangle_B)$$

FIGURE 10: Bell States

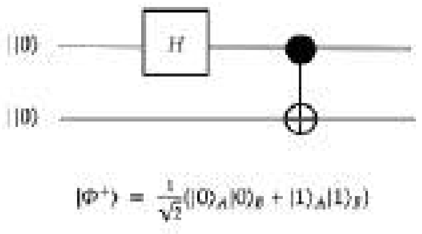


FIGURE 11: circuit for $|\Phi^{+}\rangle$

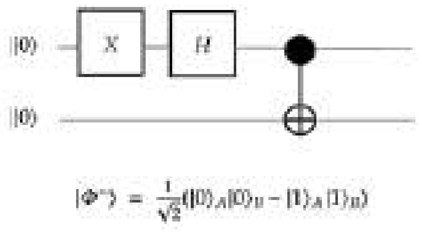


FIGURE 12: circuit for $|\Phi^{-}\rangle$

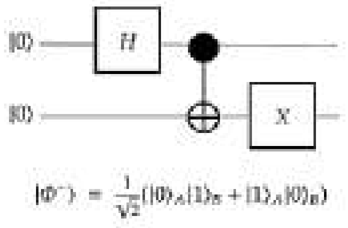


FIGURE 13: circuit for $|\Psi^{+}\rangle$

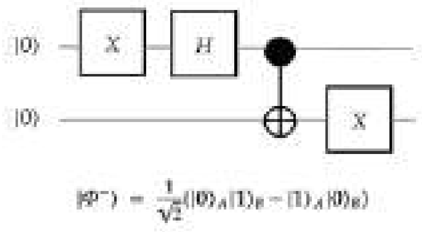


FIGURE 14: circuit for $|\Psi^{-}\rangle$

3.1.8 CREATE GHZ STATE

GHZ state is another widely used entangled state. It is an essential tool for various Quantum Cryptographic Algorithms.

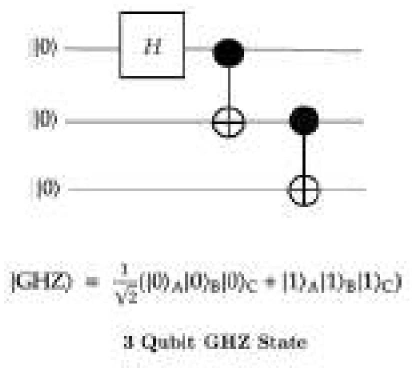


FIGURE 15: circuit for $|GHZ_3\rangle$

3.1.9 QUANTUM TELEPORTATION

Quantum Teleportation is an algorithm that can transmit 'unknown' Quantum states between two parties(Alice and Bob). This algorithm requires a shared **entangled state** between Alice and Bob. Unlike Classical Information, Quantum States cannot be just copied and Transmitted because of the **No - Cloning Theorem**. Teleportation destroys Alice's State and recreates it at Bob's end.
Some key steps of the protocol are

1. Alice and Bob creates and share a pair of Entangled qubits($\frac{1}{\sqrt{2}}(|0\rangle_A |0\rangle_B + |1\rangle_A |1\rangle_B)$)
2. Alice does a **bell measurement** on her qubit
   (**remark**:A Bell Measurement is nothing but changing the usual orthogonal basis to the bell basis and then measuring.$|00\rangle , |01\rangle , |11\rangle \rightarrow |\Phi^+\rangle , |\Phi^-\rangle , |\Psi^+\rangle , |\Psi^-\rangle$)
3. Alice sends the results(c_1, c_2) to Bob through a classical channel.
4. Bob does a suitable gate on his qubit according to the bits he receives from Alice.

   – $c_1 = c_2 = 0 \implies$ do nothing.
   – $c_1 = 0; c_2 = 1 \implies$ apply $\hat{X}$.
   – $c_1 = 1; c_2 = 0 \implies$ apply $\hat{Y}$.
   – $c_1 = 1; c_2 = 1 \implies$ apply $\hat{X}\hat{Y}$.

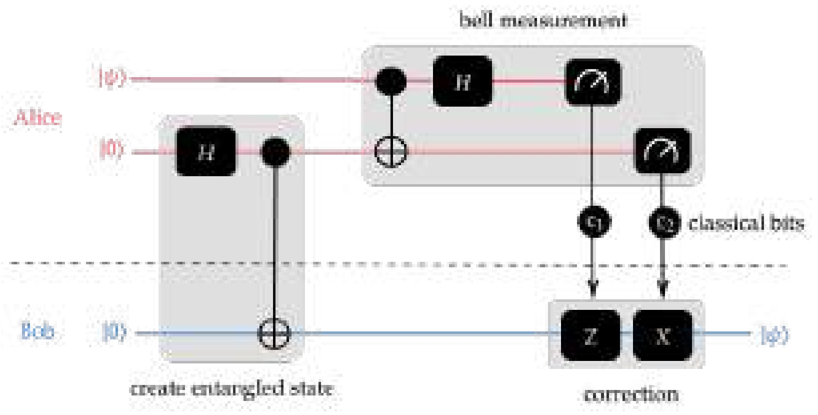


FIGURE 16: Quantum Teleportation Protocol

3.1.10 SUPER DENSE CODING

Using this protocol we can transmit two classical bits by sending just a single Qubit. This is opposite of what we did with teleportation, where we send two bits to transmit a Qubit.

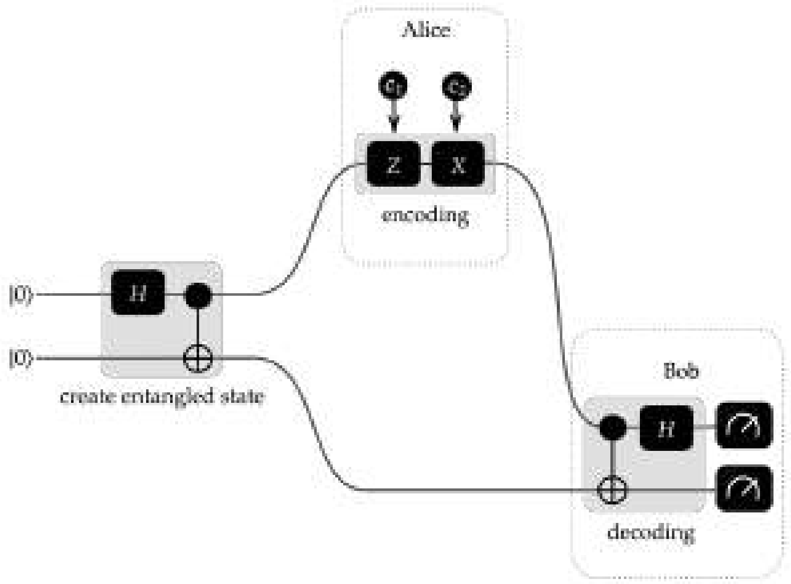


FIGURE 17: Super Dense Coding Protocol

3.1.11 QUANTUM ERROR CORRECTION

One of the main challenges in making a Quantum Computer is Errors. Errors are unavoidable even on our digital computers. But we have built robust systems in place to keep these errors in check. Error correction is essential for making any computational device work. Quantum Errors are more intricate and harder to account for , which made many scientists doubt the feasibility of a useful Quantum Computer but many theoretical breakthroughs over the last couple of decades have proved that Quantum Error Correction is possible.

Classical error correction works by leveraging redundancy. We can easily make copies of bits and use them even if some of the bits gets corrupted.

For example,

Encode:$0 \rightarrow 000, 1 \rightarrow 111$ Suppose a bit flips with probability 'p'. If a single such flip occurs, we can recover the original bit by looking at the majority.

Decode : $\{000, 001, 010, 100\} \rightarrow 0, \{111, 110, 101, 011\} \rightarrow 1$

If more than 1 bit flips this method will not work. For sufficiently small 'p' the probability of two or more bits flipping together is small. So this code works well for most cases. Unfortunately we cannot directly make copies of quantum states in a similar way because of the no-cloning theorem. Instead **quantum error correction** works by spreading information between qubits using entanglement.

Encode: $|0\rangle \rightarrow |000\rangle, |1\rangle \rightarrow |111\rangle : \alpha|0\rangle + \beta|1\rangle \rightarrow \alpha|000\rangle + \beta|111\rangle$

Possible bit flip($\hat{X}$) errors are listed below with their respective probabilities,

error	state	probability
E_0=I⊗I⊗I	$\alpha|000\rangle + \beta|111\rangle$	(1-p)^3
E_1=X⊗I⊗I	$\alpha|100\rangle + \beta|011\rangle$	p(1-p)^2
E_2=I⊗X⊗I	$\alpha|010\rangle + \beta|101\rangle$	p(1-p)^2
E_3=I⊗I⊗X	$\alpha|001\rangle + \beta|110\rangle$	p(1-p)^2
E_4=X⊗X⊗I	$\alpha|110\rangle + \beta|001\rangle$	p$^2(1-p)$
E_5=I⊗X⊗X	$\alpha|011\rangle + \beta|100\rangle$	p$^2(1-p)$
E_6=X⊗I⊗X	$\alpha|101\rangle + \beta|010\rangle$	p$^2(1-p)$
E_7=X⊗X⊗X	$\alpha|111\rangle + \beta|000\rangle$	p^3

Here $X \otimes I \otimes I$ means a bit flip(X) on the first qubit and identity on the others.

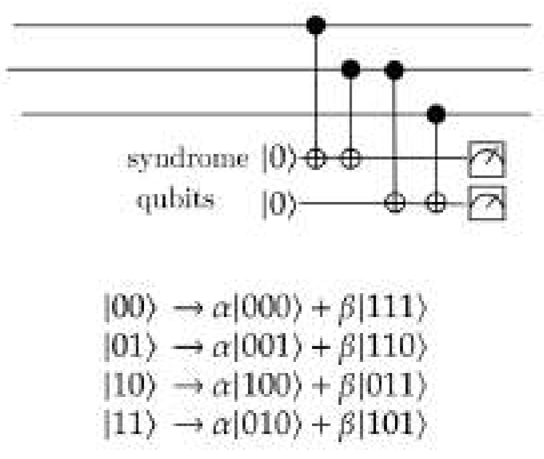
Decode:

- $|000\rangle \rightarrow |0\rangle$
- $|001\rangle \rightarrow |0\rangle$
- $|010\rangle \rightarrow |0\rangle$
- $|100\rangle \rightarrow |0\rangle$
- $|011\rangle \rightarrow |1\rangle$
- $|101\rangle \rightarrow |1\rangle$
- $|110\rangle \rightarrow |1\rangle$
- $|111\rangle \rightarrow |1\rangle$

We can do a decoding similar to the classical case by a majority vote. With sufficiently small 'p' we can successfully correct most of the errors. But in quantum mechanics we cannot directly count how many ones and zeros are present because that will collapse the state.

$\alpha\,|000\rangle + \beta\,|111\rangle \xrightarrow{X\otimes I\otimes I} \alpha\,|100\rangle + \beta\,|011\rangle \xrightarrow{\text{Measure}} |100\rangle$ or $|011\rangle$

Therefore we have to measure the majority information indirectly without collapsing the state.This can be done with the below circuit.



$$\begin{aligned}
|00\rangle &\rightarrow \alpha|000\rangle + \beta|111\rangle\\
|01\rangle &\rightarrow \alpha|001\rangle + \beta|110\rangle\\
|10\rangle &\rightarrow \alpha|100\rangle + \beta|011\rangle\\
|11\rangle &\rightarrow \alpha|010\rangle + \beta|101\rangle
\end{aligned}$$

FIGURE 18: sub circuit for measuring syndromes

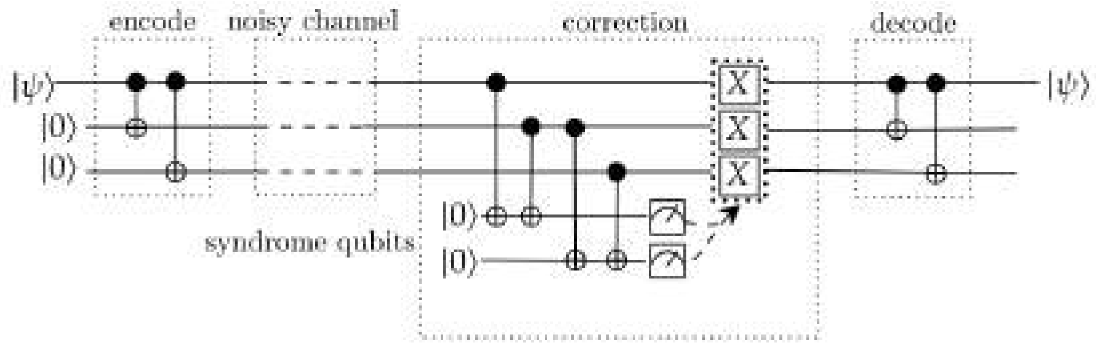


FIGURE 19: 3 qubit repetition Code

In quantum computers Bit flip is not the only Error that can occur. In fact there is a continuum of possible errors. But it turns out that if we can correct bitflips(X) and phase flips(Z), it is possible to correct any arbitrary error on a single qubit.

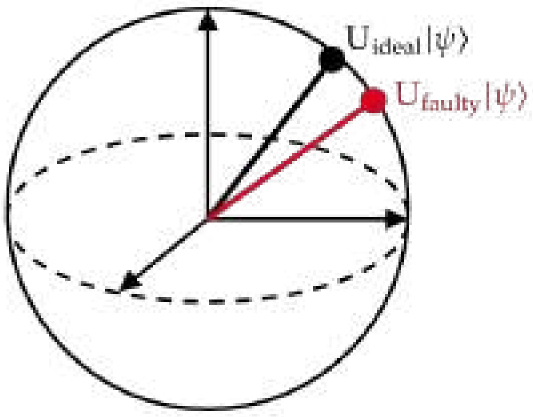


FIGURE 20: Continuous error

3.2 NISQ DEVICES

Despite the recent advancements in quantum technologies, it's still unclear what is the best way towards developing a fault-tolerant quantum computer. Research labs and companies all over the world are placing their bets on different technology candidates such as silicon quantum dots, superconducting circuits, neutral atoms, impurity spins, trapped ions, photons, NV-centered diamonds etc., for developing qubits. Most of these candidates have been already being implemented in labs to develop hardware which have their own pros and cons, namely, coherence times, scalability, control circuitry, noise-resilience, error-correction, etc. Therefore, a consensus on which technology is the way to go forward is a long term goal, however, our immediate challenge is to see what all could be done with the currently available quantum hardware. These current-generation hardware are part of the NISQ era, a term coined by John Preskill. This term NISQ expands to Noisy Intermediate-Scale Quantum. Here, "noisy" refers to the imperfect qubit controls, low coherence time along with limited qubit connectivity. Whereas, "Intermediate-Scale " refers to the size of the hardware i.e. the number of qubits available for performing useful computations. For the hardware in this era, this number is in the range of 100s. This is still far from the ideal qubit count that would be required to run quantum algorithms like Shor or HHL which is in the range of millions.

Hence, a new range of algorithms is being developed that can utilize NISQ hardware to solve the computational task. These algorithms are more specifically referred to as hybrid heuristics because their computational advantage is still uncertain and they make use of both classical and quantum hardware for execution. However, recently an energy-based benchmark for computational-advantage developed by researchers at NASA was used to show that NISQ devices used several orders of magnitude less energy than classical supercomputers on certain tasks. Moreover, in comparison to annealers, the computations performed on NISQ devices are realized using the gate-model paradigm, which makes them more suitable candidates for performing the universal computation.

In the coming sections, first, we will be discussing some algorithms such as QPE and HHL, that have a proven-exponential advantage but require fault-tolerant quantum computers for execution. Next, we will be introducing hybrid heuristics such as VQE, QAOA, etc., which can be executed on the NISQ hardware.

3.3 TRADITIONAL QUANTUM ALGORITHMS

Algorithm design in classical computing makes use of various algorithmic paradigms such as dynamic programming, constrained optimization, local search, etc. In a similar way, algorithm design for quantum computing also makes use of some such algorithmic paradigms, for example, quantum Fourier transform (QFT), the Harrow-Hassidim-Lloyd (HHL), Amplitude Amplification (AA) etc. The number of known quantum algorithmic paradigms is much smaller compared to the number of known classical paradigms, mostly due to the unitarity and reversible conditions attached with the No-Go theorems in quantum paradigm.

In this section, we majorly focus on algorithms based on the quantum Fourier transform (QFT), and the Harrow-Hassidim-Lloyd (HHL) paradigm.

3.3.1 QUANTUM FOURIER TRANSFORM (QFT)

The quantum Fourier transform is the quantum implementation of the discrete Fourier transform over the amplitudes of a wavefunction. Hence, it has the same form and properties such as linear transformation and invertibility. If mathematically Fourier transform is defined as fol-

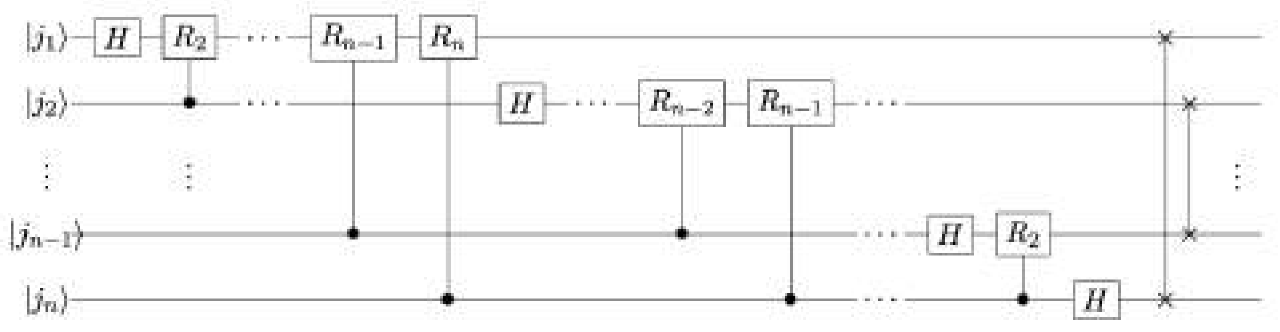


FIGURE 21: Efficient circuit for quantum Fourier transform. [1]

lows:

$$y_k = \sum_{j}^{N-1} e^{\frac{2\pi ijk}{N}} x_j$$

Then, in a similar spirit, we can define quantum Fourier transform as:

$$|j\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi ijk/N} |k\rangle$$

$$\sum_{j=0}^{N-1} x_j |j\rangle \rightarrow \sum_{k} \sum_{j=0}^{N-1} y_k |k\rangle$$

By careful inspection of the above two definitions of quantum Fourier transform, it becomes natural that the above transformation will require the usage of phase-shifting gates. Hence, by using the high school algebra we obtain the following product formulation [1]:

$$|j\rangle \equiv |j_1, \ldots j_{log_2(N)}\rangle \rightarrow$$

$$\frac{(|0\rangle + e^{2\pi i j_{log_2(N)}/2}|1\rangle)(|0\rangle + e^{2\pi i (j_{log_2(N)-1}/2 + j_{log_2(N)}/4)}|1\rangle)\ldots(|0\rangle + e^{2\pi i (j_1/2+\ldots+j_{log_2(N)}/2^{log_2(N)})}|1\rangle)}{\sqrt{N}}$$

Using the equivalence of product formulation and our initial definition, an efficient circuit for the quantum Fourier transform can be obtained that utilizes the following phase gate R_k:

$$R_k \equiv \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix}$$

The classical discrete Fourier transform is used for finding time-based sine waves, but quantum Fourier transform is used in phase estimations problems, period finding, etc. Even though NISQ hardware cannot efficiently run such QFT, however, the simulator provided as part of $QpiAI^{TM}$ Explorer is capable of running it. We can implement a QFT circuit using an advanced version of our circuit generator module. As an example circuit, a curious user can try to implement the following circuit for $n = 5$ qubits, i.e. $N = 2^n = 32$:

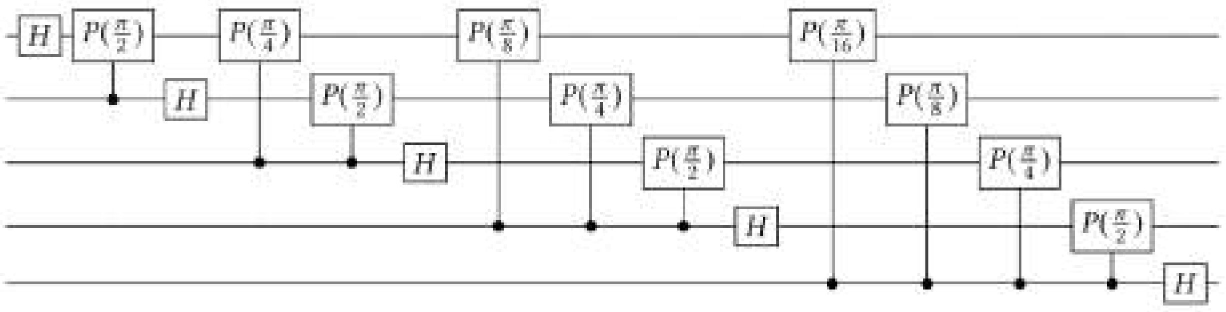


FIGURE 22: A Quantum Fourier Transform circuit for five qubits [2]. Notice that final swap gates are not shown here, which reverse the order of the qubits.

3.3.2 QUANTUM PHASE ESTIMATION (QPE)

We use Quantum Phase estimation (QPE) to find the eigenvalues of a unitary matrix U. To do so, we define the following equation:

$$U|u\rangle = e^{2\pi i\lambda_u}|u\rangle$$

Here, $|u\rangle$ is the eigenvector of U, and $e^{2\pi i\lambda_u}$ is the corresponding eigenvalue. The phase estimation procedure uses the QFT operator to estimate the eigenphases of the operator U by performing the following transformation:

$$|0\rangle|u\rangle = |\lambda_u^*\rangle|u\rangle$$

Here, λ_u^* is an estimate for λ_u. This estimate will get more accurate with the increase in the number of qubits used to perform the QFT subroutine. Now to find this λ_u^* using a given U and $|u\rangle$, we use the following:

1. Given $U|u\rangle = e^{2\pi i\lambda_u}|u\rangle$, one can find $U^k|u\rangle = e^{2\pi ik\lambda_u}|u\rangle$.

2. By applying controlled U^{2^j}, where j is an integer, on $|u\rangle$ such that the control qubit is in the state $1/\sqrt{2}(|0\rangle + |1\rangle)$ due to application of Hadamard gate, we get:
$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|u\rangle \to \frac{1}{\sqrt{2}}(|0\rangle|u\rangle + |1\rangle U^{2^j}|u\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i2^j\lambda_u}|1\rangle)|u\rangle$$

3. Now, let N be the dimension of eigenvector $|u\rangle$, i.e. $N = 2^n$, where n is the number of qubits required to represent $|u\rangle$. Now using m ancilla bits, to apply controlled U^{2^j}, gives us the following state:
$$\frac{1}{\sqrt{2^m}}(|0\rangle + e^{2\pi i2^{m-1}\lambda_u}|1\rangle)\dots(|0\rangle + e^{2\pi i2\lambda_u}|1\rangle)(|0\rangle + e^{2\pi i\lambda_u}|1\rangle)|u\rangle = \frac{1}{2^m}\sum_{k=0}^{2^m-1} e^{2\pi ik\lambda_u}|k\rangle|u\rangle$$

4. From the above equation, to solve for λ_u, we need to apply QFT^{-1} or $QFT^{\dagger}$ for the first 2^m qubits which can be done by the circuit given in the figure 23. Since the accuracy of the solution will depend on m, we need to use an optimal value which maximizes accuracy to complexity ratio.

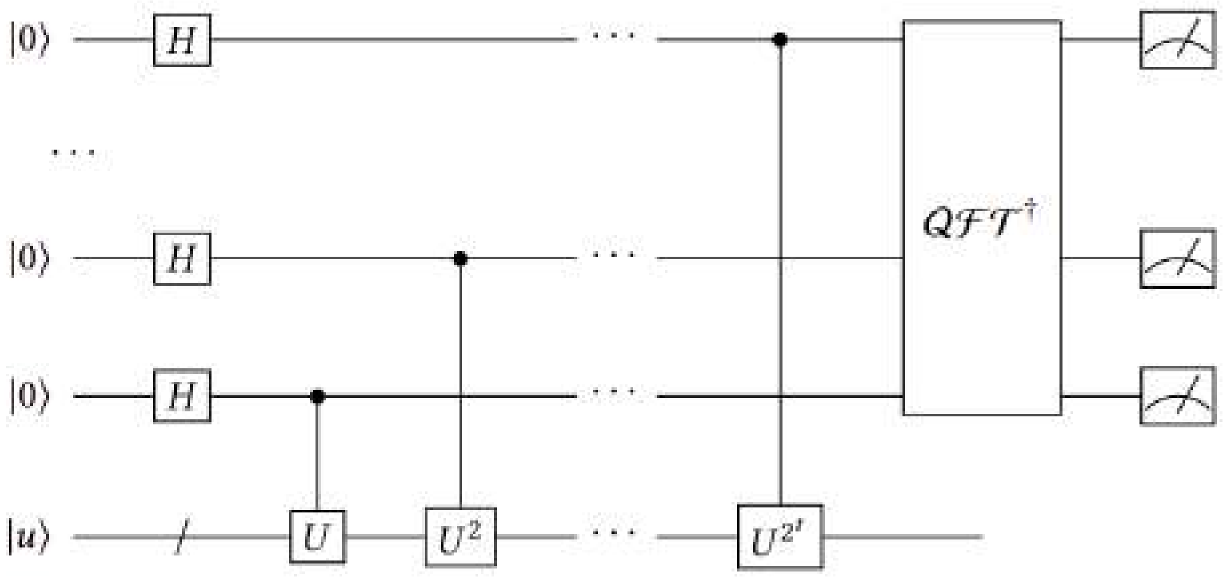


FIGURE 23: Quantum circuit for phase estimation. Here, we use $t+1$ ancilla qubits and encoding $|u\rangle$ require n qubits. [2]

3.3.3 HARROW-HASSIDIM-LLOYD (HHL)

In computational mathematics and sciences, the majority of problems can be visualized as the problem of solving linear equations. We define it as follows: Given a system $A\vec{x} = \vec{b}$, find $\vec{x}$ for a given Hermitian matrix A and normalized vector $\vec{b}$. We can then formulate a quantum formulation of the above problem with A being a hermitian operator, and $|x\rangle$, $|b\rangle$ as quantum states.

$$A|x\rangle = |b\rangle$$

Now, let us introduce the idea behind the HHL algorithm.

1. Let $\{u_j\}$ and $\{\lambda_j\}$ be the normalized eigenvectors and rescaled eigenvalues of A, such that $||u_j||_2 = 1$ and $0 < \lambda_j < 1$.
2. Expand the state $|b\rangle$ as the linear combinations of eigenvectors $\{u_j\}$, $|b\rangle = \sum_{j=1}^{N} \beta_j |u_j\rangle$.
3. Now, the goal of the algorithm is to get $|x\rangle$ in the form: $|x\rangle = \sum_{j=1}^{N} \beta_j/\lambda_j |u_j\rangle$

The HHL algorithm requires three sets of qubits: a single ancilla qubit, a register of n qubits used to store the eigenvalues of A in binary format with precision up to n bits, and a memory of $O(log(N))$ that initially stores $|b\rangle$ and eventually stores $|x\rangle$. Start with a state $|0\rangle_a|0\rangle_r|b\rangle_m$, where the subscripts a, r, m, denote the sets of ancilla, register and memory qubits, respectively and then run the phase estimation procedure on the unitary operator e^{iA}. In figure 24, we visualize and explain the above ideas as a three-step process defined as follows:

$$A|x\rangle \equiv V^\dagger UV|x\rangle = |b\rangle \xrightarrow{\text{Step-1}} UV|x\rangle = V|b\rangle \xrightarrow{\text{Step-2}} V|x\rangle = U^{-1}V|b\rangle \xrightarrow{\text{Step-3}} |x\rangle = V^\dagger UV|b\rangle$$

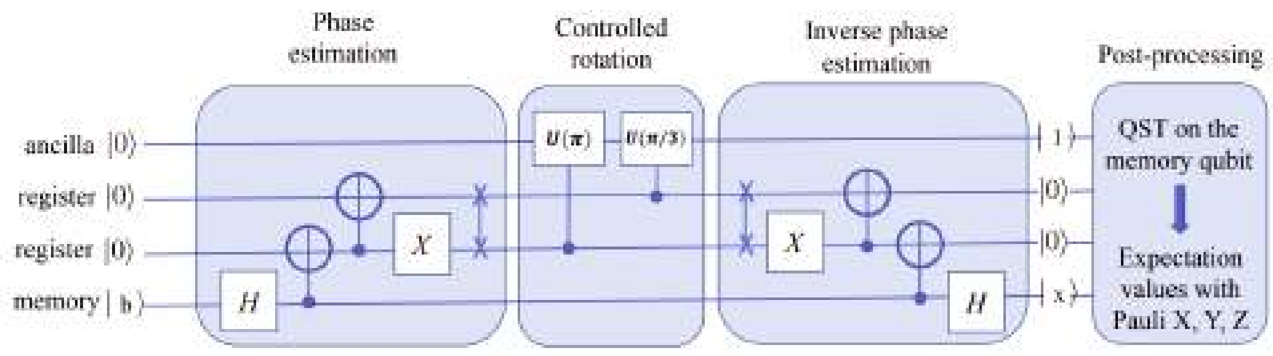


FIGURE 24: Schematic of the circuit for the quantum algorithm for solving a 2×2 linear system. The first step involves phase estimation, which maps the eigenvalues λ_j of A into the register in the binary form. The second step involves controlled rotation of the ancilla qubit so that the inverse of the eigenvalues $1/\lambda_j$ show up in the state. The third step is the inverse phase estimation to disentangle the system and restores the registers to $|0\rangle$. The memory qubit now stores $|x\rangle$, which is then post-processed to get the expectation values with respect to the Pauli operators X, Y and Z. [2]

3.4 NISQ-CENTRIC HYBRID QUANTUM ALGORITHMS

Hybrid quantum algorithms are the paradigm of quantum algorithms that can be executed on NISQ devices. These use both classical and quantum computing resources to solve potentially useful tasks. Recent proposals of the algorithms of this paradigm include Variational Quantum Eigensolver (VQE) for quantum simulations and related applications, Quantum Approximate Optimization Algorithm (QAOA) for combinatorial optimization problems, and Quantum Neural Networks (QNN) for quantum machine/deep learning, generative modelling, etc. These algorithms make use of Parameterized Quantum Circuits (PQCs) which are quantum circuits dependent on parameters which can be trained via a classical or quantum routine. A PQC can be realized as $U(\theta)$ where $\vec{\theta}$ is a vector of parameters θ_i. In general depending upon whether the PQC's structure i.e. the way gates generating $U(\theta)$ changes with the progress of the algorithm or not, we define adaptive and non-adaptive hybrid algorithms. Currently, our platform, $QpiAI^{TM}$ Explorer, supports only the non-adaptive algorithms due to their easier and straightforward implementation. In general, they will be sufficient for the needs of a beginner or intermediate user. However, for the expert users, we will be adding support for adaptive hybrid algorithms such as Adapt-VQE, QSL, etc., in our upcoming updates to the platform.

In this section, we will be focusing on the standard non-adaptive hybrid algorithm such as VQE, QAOA and QNN.

3.4.1 VARIATIONAL QUANTUM EIGENSOLVER (VQE)

The algorithm, Variational Quantum Eigensolver (VQE), as the name suggests is used to find eigenvalues of a Hermitian operator such as Hamiltonian by training a PQC, specifically referred to as ansatz, using the variational principle. Hence, if we are given a Hamiltonian H, then the task we require to solve is the following:

$$H|\psi\rangle = E_g|\psi\rangle$$

Here, E_g refers to the ground state of the Hamiltonian $\hat{H}$. Various problems in computational sciences require us to solve such equations.

In principle, we could use the quantum phase estimation algorithm which was described in section II, to get the eigenvalues of the Hamiltonian $\hat{H}$. To do so we would require hardware that could perform controlled operations with the unitary $U = exp(-iH\delta t/\hbar)$, where δt is the time step. Then, by preparing different initial states $|\psi_i\rangle$ and repeating the phase estimation routine several times, one can obtain many times one can obtain the whole spectrum of the eigenvalues and the corresponding eigenvector of the Hamiltonian $\hat{H}$. However, implementation of these controlled U operations is not straightforward, and leads to large circuit depth which makes NISQ hardware incapable of implementing it.

VQE was developed to overcome this limitation. The basic idea behind VQE is to take the advantages of both quantum and classical computers, as shown in Fig. 5. The problem in hand is separated into two parts, one which can be solved easily on a classical processor and the other where we could make use of the quantum advantage. Such a partitioning of the problem ensures that the circuit depth remains minimal. We can understand a step-wise approach for the algorithm as follows:

1. Prepare the initial state $|\psi_i\rangle$ state preparation unitary. Then use the ansatz to evolve this state to prepare a variational state $|\psi(\vec{\theta})\rangle$ with parameters $\{\vec{\theta}\}$. In principle, it is optimal that the number of parameters is polynomial in the size of the problem.

2. If you've a Hamiltonian $\hat{H}$ of a physical system, we need to convert it to a qubit Hamiltonian first. This transformation to qubit representation can be done via many transformations such as Bravyi-Kitaev, Jordan-Wigner, Parity, etc. Once, we do this, we can calculate the expectation value of qubitized Hamiltonian as $E = \langle\psi(\vec{\theta})|\hat{H}|\psi(\vec{\theta})\rangle$. We can do this in parallel by splitting Hamiltonian into commutating terms which can be measured together.

3. Our aim is to minimize the expectation value calculated in the previous step. This is done by finding an optimal $\{\vec{\theta^*}\}$, for which the $E \approx E_g$. We use classical optimization routines like nlopt or mlpack to update these parameters. Note that, while doing so, if one uses a gradient-based optimizer, it can get stuck with the problem of a barren plateau i.e.

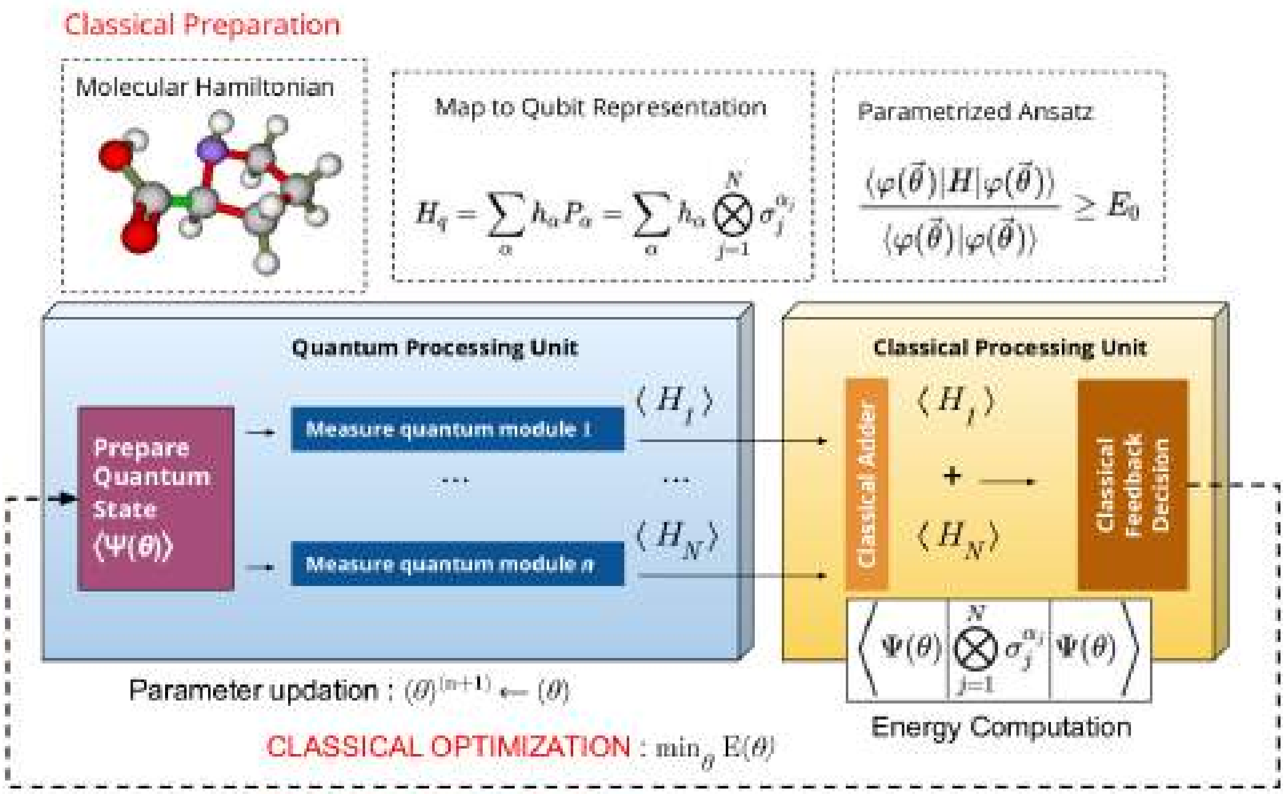

Classical Preparation
Molecular Hamiltonian
Map to Qubit Representation
$H_q = \sum_\alpha h_\alpha P_\alpha = \sum_\alpha h_\alpha \bigotimes_{j=1}^{N} \sigma_j^{\alpha_j}$
Parametrized Ansatz
$\frac{\langle \varphi(\vec{\theta})|H|\varphi(\vec{\theta})\rangle}{\langle \varphi(\vec{\theta})|\varphi(\vec{\theta})\rangle} \geq E_0$
Quantum Processing Unit
Prepare Quantum State $\langle\Psi(\theta)\rangle$
Measure quantum module 1
Measure quantum module n
$\langle H_1\rangle$
$\langle H_N\rangle$
Classical Processing Unit
Classical Adder
Classical Feedback Decision
$\left\langle \Psi(\theta) \middle| \bigotimes_{j=1}^{N} \sigma_j^{\alpha_j} \middle| \Psi(\theta) \right\rangle$
Energy Computation
Parameter updation : $(\theta)^{(n+1)} \leftarrow (\theta)$
CLASSICAL OPTIMIZATION : $\min_\theta E(\theta)$

FIGURE 25: Schematic of VQE performed for a chemical system

negligible gradient in a non-optimal region. This procedure is repeated iteratively until convergence is reached.

4. For VQE, the final result is dependent on various factors such as expressibility and entanglibility of the ansatz, power of the optimizer, and the number of measurements performed to approximate E. $QpiAI^{TM}$ Explorer platform provides you the VQE module for you to play around. We allow users to change ansatz (custom, hardware efficient, etc.), optimizers (nlopt based or mlpack based), Hamiltonian mapping transformations (Bravyi-Kitaev, Jordan-Wigner, or Parity), etc. We support all ranges of systems like fermionic, Pauli, Ising-spin, etc. Moreover, for working with chemical systems, in particular, we support major computational chemistry libraries such as pyscf, psi4, and openfermion. The figure 25, gives a more graphical illustration of all of these steps.

3.4.2 QUANTUM APPROXIMATE OPTIMIZATION ALGORITHM (QAOA)

The algorithm QAOA is more specifically used for solving combinatorial optimizations problems such as Travelling Salesman, MaxCut, etc. A large variety of combinatorial problems could be realized a generic unconstrained optimization problem with n binary decision variables, z, and m binary functions of those variables $C(z)$:

$$maximize: \quad \sum_{\alpha=1}^{m} C_\alpha(z) \qquad z_i \in \{0,1\} \forall i \in \{1,\dots,n\}$$

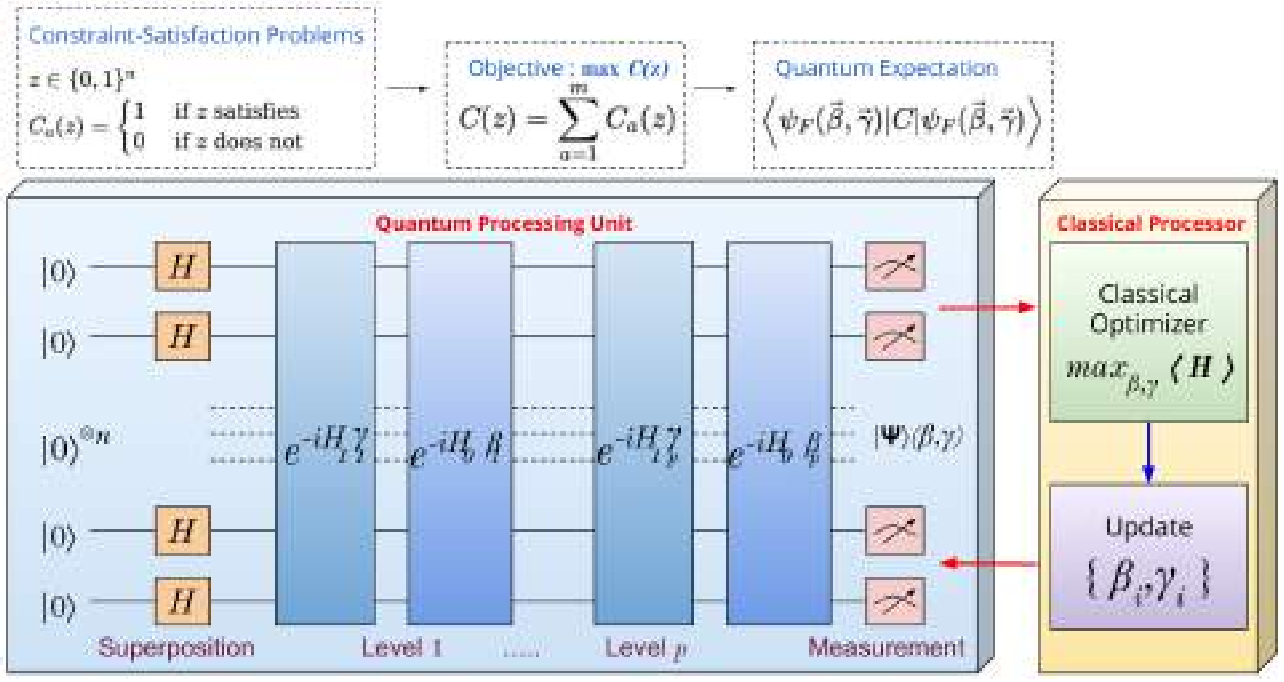


FIGURE 26: Schematic of QAOA performed for a MaxSat.

In general, such a formulation can be rewritten with spins, $\sigma \in \{-1, 1\}$ by using the mapping $z = (\sigma + 1)/2$. Doing this we find an equivalent mapping of the above formulation as an Ising Hamiltonian i.e. equivalent quantum clause Hamiltonians C_α. Hence, finding the optimal solutions becomes equivalent to finding the ground state of the equivalent Ising formulation. However, despite this transformation, the NP-Completeness of the problem is retained which means we might not be able to solve it completely but rather have an advantage in solving using a quantum computer.

The general idea behind QAOA is to find discretize an adiabatic pathway from an easily preparable ground state of Hamiltonian H_0 such as $\sum_i \sigma_i^x$ to the ground state of the cost Hamiltonian, H_c, which will be the Ising Hamiltonian formulation of our problem instance. This is equivalent to trotterizing the unitary we require to transform the initial product state $|\psi_{gs}^{H_0}\rangle$ to the final state $|\psi_{gs}^{H_c}\rangle$ into p steps using the parameters $\{\beta, \gamma\}$ as follows:

$$U = U(H_0, \beta_0)U(H_c, \gamma_0)\ldots U(H_0, \beta_{p-1})U(H_c, \gamma_{p-1})$$

We optimize these parameters $\{\beta, \gamma\}$ to obtain the optimal set $\{\beta^*, \gamma^*\}$ to reach $|\psi_{gs}^{H_c}\rangle$ using a classical processor by using the results of the expectation measurement $\langle\psi_F(\beta, \gamma)|H_c|\psi_F(\beta, \gamma)\rangle$. This process is illustrated in figure 26 for the problem of MaxSat. Our platform, $QpiAI^{TM}$ Explorer, currently makes use of the QAOA routine to solve the problem of MaxCut and Portfolio optimization. We are currently working on adding additional problems soon in our coming releases.

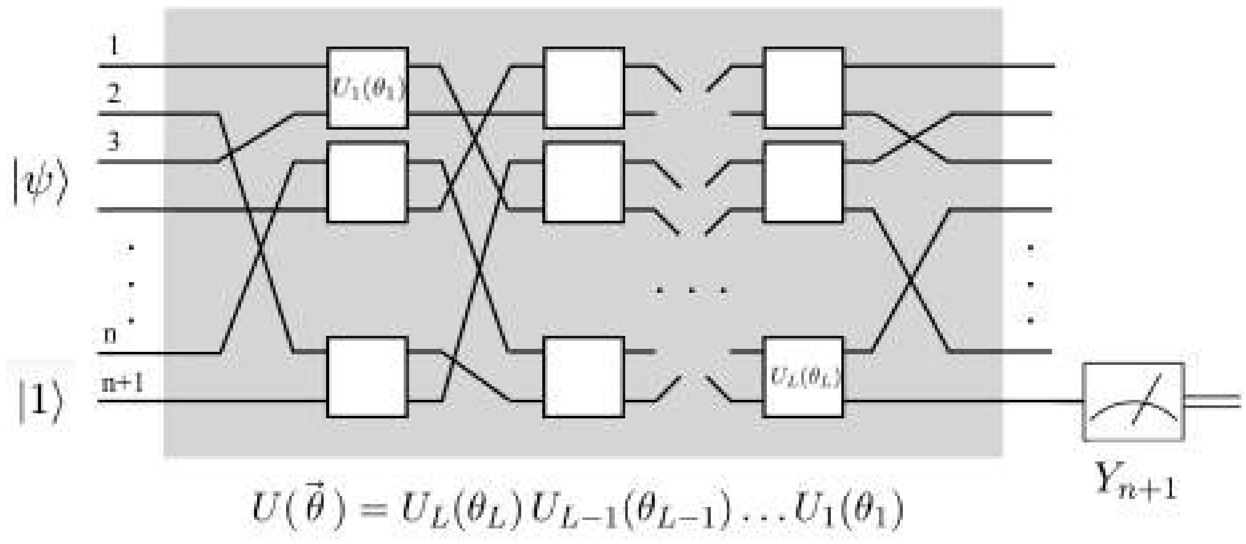

1
2
3
$|\psi\rangle$
n
n+1
$|1\rangle$
$U_1(\theta_1)$
$U_L(\theta_L)$
Y_{n+1}
$U(\vec{\theta}) = U_L(\theta_L)U_{L-1}(\theta_{L-1})\ldots U_1(\theta_1)$

FIGURE 27: Schematic of QNN used for a classification task

3.4.3 QUANTUM NEURAL NETWORKS (QNNS)

A quantum neural network (QNN) could be realized using PQCs. This means it consists of a sequence of parameter-dependent unitary transformations which acts on an input quantum state. It has use cases in various machine learning tasks such as generative modelling and classification. Figure 27, illustrates the schematic of a quantum neural network that could be utilized for the classification task. The input state $|\psi 1\rangle$ is prepared and then transformed via a series of parameterized qubit unitaries $U(\theta_i)$. These parameterized unitaries are trained using some Pauli measurement of in the routine. In this case, it is Y_{n+1}, which produce the desired label for $|\psi\rangle$.

In $QpiAI^{TM}$ Explorer we use QNNs to perform data-driven circuit learning (DDCL). In the context of quantum machine learning, it is a generative modelling task in which a PQC i.e. an ansatz, is trained to prepare a target distribution. To do this we make available two loss functions, Jensen-Shannon divergence (JSD) and maximum mean discrepancy (MMD), and their corresponding gradient-based/gradient-free optimizers. Though, we allow users to provide a suitable ansatz on their own, apart from a few templates whose basic structure is illustrated in Figure 28. We provide a more detailed explanation of how QNNs work and can be used to implement hybrid neural network layers in the next section titled Quantum Machine Learning (QML).

3.5 QUANTUM MACHINE LEARNING

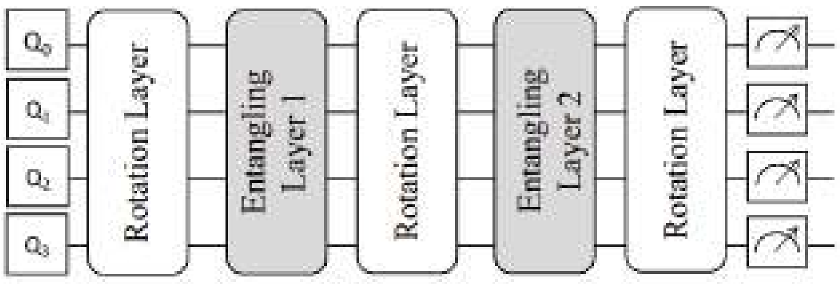


FIGURE 28: Schematic of Quantum Neural Network for DDCL. Here, the rotation layer consist of single-qubit rotation gates and Entangling layers consists of the pairwise application of controlled gates such as CZ, CX, CR etc.

3.5.1 INTRODUCTION

Models, simulations, and experiments have been at the forefront of research in both technology and natural sciences because of the large amount of data they produce. Extraction and processing of this large amount of data have been possible due to our advances towards more complex and powerful computational capabilities. These include implementation of linear algebraic techniques such as regression, principal component analysis, support vector machines, development of statistical frameworks such as Markov chains, and the emergence of novel machine learning and deep learning methods such as artificial neural networks (perceptrons), convolutional neural network (CNN), and Boltzmann machines.

In the past decade, these techniques have been extensively used in large datasets in both industries and academic research. In fact, a large number of HPCs (with GPUs, TPUs, etc.) have used their billions of clock cycles to train deep learning networks on useful datasets to identify complex and subtle patterns in data. These have resulted in the development of computational programs useful in disease classification in clinical diagnosis, new materials and compounds discovery, path planning in autonomous robots, etc. However, training these networks to do something useful takes a lot of computational resources depending on the complexity of the task at hand. This is because classical/deep machine learning methods either require algebraic routines (SVM, PCA, etc), recognizing statistical patterns in data (deep neural networks) and often both. As we will be studying below, this observation leads to a hope that quantum computing could be a way forward in improvement, but before jumping there, let us first briefly revisit quantum computers.

A quantum computational device uses quantum mechanical resources such as superposition, entanglement, and tunnelling to gain a possible computational advantage over classical processors, also known as quantum speedup. In the last couple of years, there has been successful development and demonstration of quantum processors by IBM, Rigetti, Google, etc. However, the computational capabilities of these current generation quantum processors also known as Noisy Intermediate-Scale Quantum (NISQ) devices, are considerably restricted

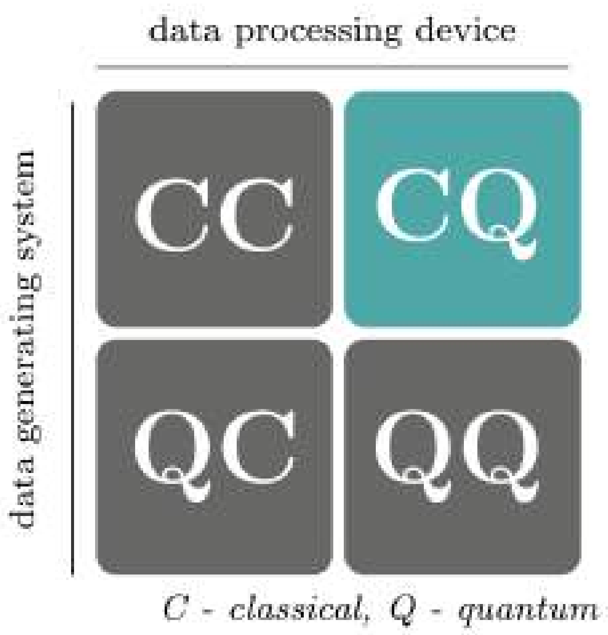


FIGURE 29: Four approaches that combine quantum computing and machine learning. The case CC refers to classical data being processed classically, and the case QC refers to using classical machine learning in quantum computing. Here, we focus on CQ and QQ approaches in which we use quantum computing devices to process both quantum and classical data. [4]

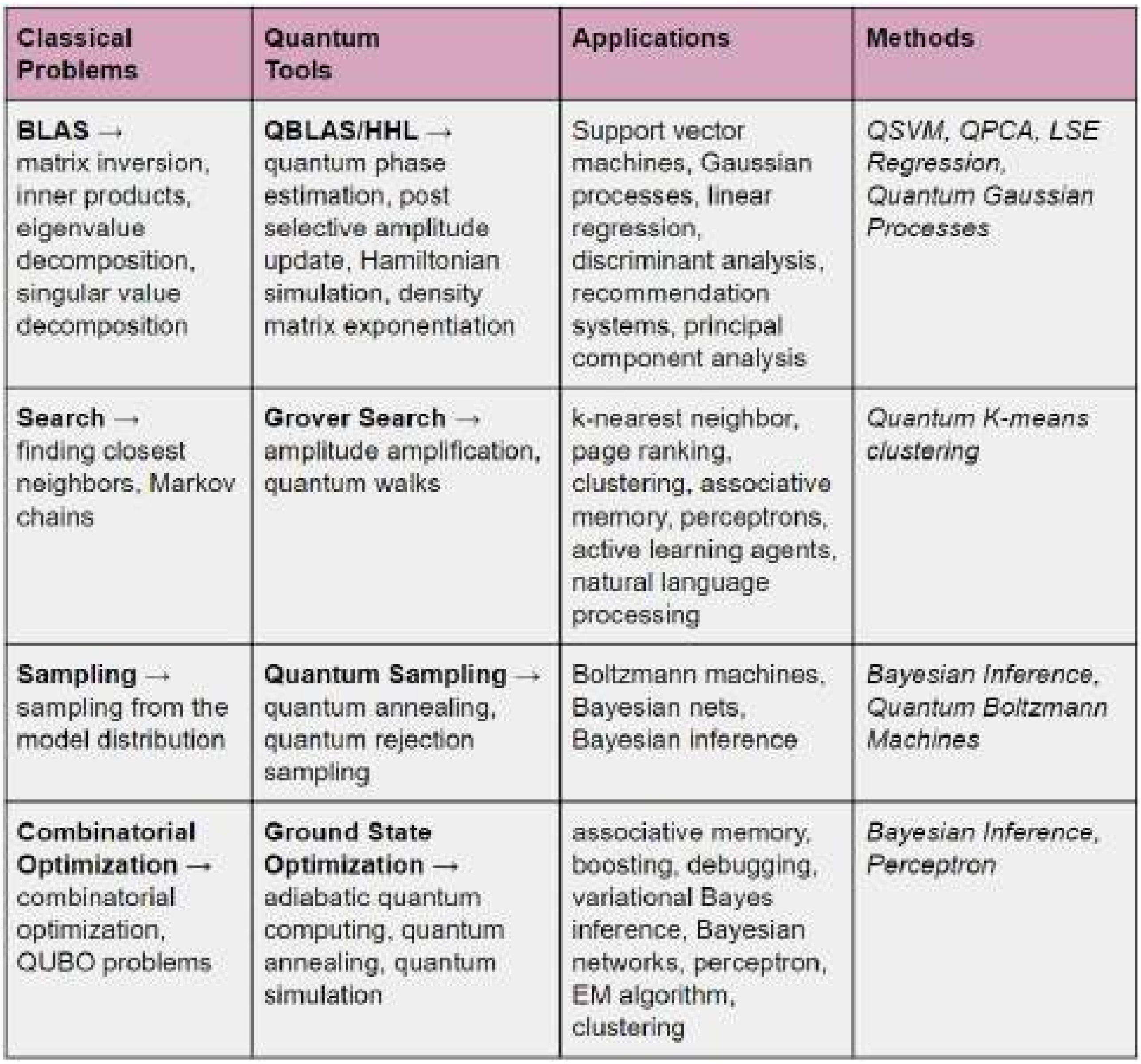
due to their intermediate size (in terms of qubits count), limited connectivity, imperfect qubit-control, short coherence time and minimal error-correction. Nevertheless, they can still be used with a classical computer as an accelerator, to solve those parts of the problem which are intractable for classical computers. This will be discussed further in Section-2.

Therefore, even though fault-tolerant quantum computers are still years away, the field has already left the purely academic sphere and is on the agenda of industries like us. Now, coming back to using the current generation quantum computers in machine learning, it must be noted that their success depends on whether an efficient quantum algorithm (such as QBoost) can be found for this task. This idea is being explored under a collaborative framework of Quantum Machine Learning (QML), a term popularised by Lloyd, Mohseni and Rebentrost (2013), and Peter Wittek (2014). In general, there are multiple definitions of this term based on the approach followed to integrate both quantum computing and machine learning, depending upon data generation and data processing systems - Quantum (Q) and Classical (C). These approaches are represented neatly in Figure 29.

In this article, we will be mainly focusing on the CQ and QQ approach, where quantum computing is used on both classical and quantum (i.e. data generated from quantum experiments, simulations, sensors, etc.) data respectively. Moreover, depending upon how quantum computers are used to attain a quantum speedup in a classical/deep learning algorithm we have defined two waves in QML. Algorithms proposed in the first wave required fault-tolerance quantum devices and qRAM, whereas, the second wave algorithms can be run on NISQ devices and don't require qRAM.

3.5.2 THE FIRST WAVE OF QUANTUM MACHINE LEARNING

The first wave in QML started in 2010 when theoretical quantum algorithms like QSVM, QPCA, QBoost, Q-Means, etc. were proposed. These algorithms were aimed towards speeding ML, and hence required the existence of both fault-tolerant quantum devices and QRAM to attain a quantum advantage. We have listed down the motivation and promises of the algorithms put forward in the first wave in Table 1 and Table 2 respectively.

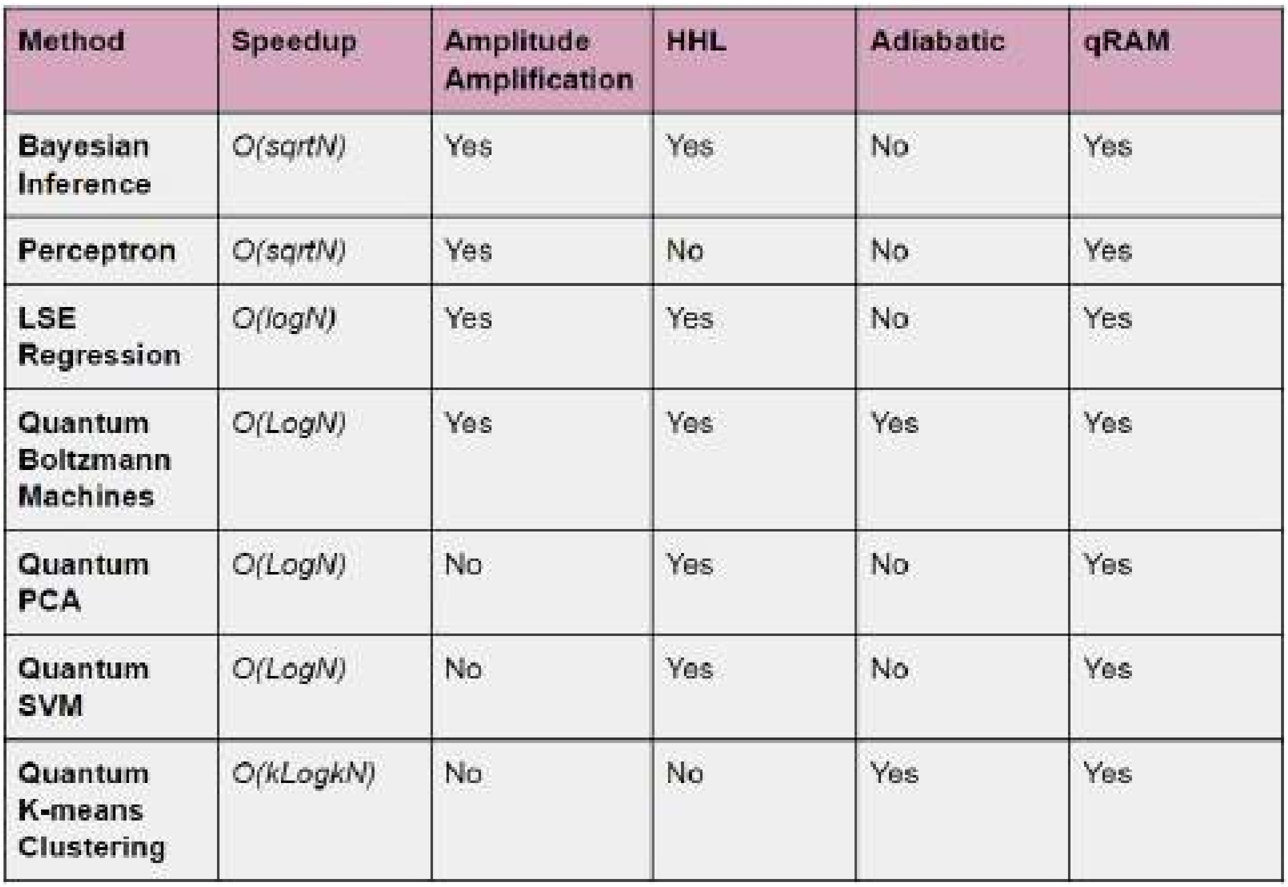


Classical Problems	Quantum Tools	Applications	Methods
BLAS → matrix inversion, inner products, eigenvalue decomposition, singular value decomposition	**QBLAS/HHL** → quantum phase estimation, post selective amplitude update, Hamiltonian simulation, density matrix exponentiation	Support vector machines, Gaussian processes, linear regression, discriminant analysis, recommendation systems, principal component analysis	*QSVM, QPCA, LSE Regression, Quantum Gaussian Processes*
Search → finding closest neighbors, Markov chains	**Grover Search** → amplitude amplification, quantum walks	k-nearest neighbor, page ranking, clustering, associative memory, perceptrons, active learning agents, natural language processing	*Quantum K-means clustering*
Sampling → sampling from the model distribution	**Quantum Sampling** → quantum annealing, quantum rejection sampling	Boltzmann machines, Bayesian nets, Bayesian inference	*Bayesian Inference, Quantum Boltzmann Machines*
Combinatorial Optimization → combinatorial optimization, QUBO problems	**Ground State Optimization** → adiabatic quantum computing, quantum annealing, quantum simulation	associative memory, boosting, debugging, variational Bayes inference, Bayesian networks, perceptron, EM algorithm, clustering	*Bayesian Inference, Perceptron*

TABLE 1: Quantum alternatives to solve classical problems

Proposals of these algorithms did heat the quantum research community. However, there were certain caveats in their success:

1. Fault-tolerance - Since most of these algorithms require methods like HHL, they were



Method	Speedup	Amplitude Amplification	HHL	Adiabatic	qRAM
Bayesian Inference	O(sqrtN)	Yes	Yes	No	Yes
Perceptron	O(sqrtN)	Yes	No	No	Yes
LSE Regression	O(logN)	Yes	Yes	No	Yes
Quantum Boltzmann Machines	O(LogN)	Yes	Yes	Yes	Yes
Quantum PCA	O(LogN)	No	Yes	No	Yes
Quantum SVM	O(LogN)	No	Yes	No	Yes
Quantum K-means Clustering	O(kLogkN)	No	No	Yes	Yes

TABLE 2: Quantum speedup achieved in different methods. [3]

not feasible on near-term quantum processors.

2. QRAM - It might be possible that an advantageous QRAM might not be possible. Hence, for most of these algorithms, the advantage gained during computation on quantum devices could be nullified by the cost incurred during the data encoding/loading procedure.

3. Dequantization - Ewin Tang presented quantum-inspired classical algorithms for recommendation systems, low-rank HHL, PCA, etc. that essentially gave the same performance as the quantum algorithms.

3.5.3 THE SECOND WAVE OF QUANTUM MACHINE LEARNING

The successful development and demonstration of quantum hardware in 2016 initiated the second wave quantum machine learning, which is currently in progress. Till now, the proposed candidates are hybrid quantum-classical heuristics. These involve using limited-depth parameterized quantum circuits on NISQ processors to prepare highly entangled quantum states, and then application of classical optimization routine for tuning parameters to converge to the target quantum state. Such a hybrid quantum-classical approach is represented in Figure 30.

The first important quantum method of learning used the quantum Ising model to train Boltzmann machines and Hopfield models. This integration leads to better convergence in

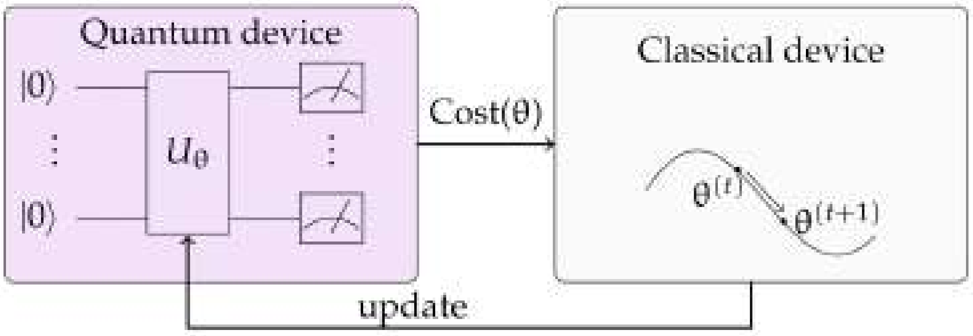


FIGURE 30: In the hybrid quantum-classical paradigm, the quantum computer prepares quantum states according to a set of parameters. Measurement outcomes from this state are used to calculate cost on the classical device. It then makes use of learning algorithms and adjusts the parameters in order to minimize the cost. The updated parameters are then fed back to the quantum hardware completing the feedback loop. [10]

such models. On the other hand, the current most popular methods of learning are hybrid algorithms. One basic example of these could be the variational algorithms, which are based on the variational principle of quantum mechanics. In this algorithm, the problem is encoded in a hermitian operator M such as Hamiltonian. Then a trial wavefunction is prepared using the state preparation circuit, which acts as our guess answer state. Next, we use a parameterized circuit known as an ansatz to evolve this trial wavefunction. A classical processor then performs measurements to evaluate the expectation value of M, and tune the ansatz parameters. These parameters are optimized such that they evolve our initially prepared state to the target wavefunction which minimized the expectation value of M. A variational paradigm of training ML is given in Figure 30, and a few examples of the ansatz used in the variational circuits are given in figure 31.

These hybrid approaches based on parameterized quantum circuits (PQCs) have been demonstrated to perform incredibly well on machine learning tasks such as classification, regression, and generative modelling. This success is easier to explain if we notice the similarities between PQCs and classical models, such as neural networks and kernel methods. The similarity with both of these methods could be by the representation given in Figure 32.

Moreover, in the recent research literature, there has been a lot of work regarding computing the partial derivative of quantum expectation values with respect to the gate parameters through procedures such as parameter shift rule. The development of these results has allowed some important advancement in the field of QML, such as gradient-based optimization strategies. This ability to compute quantum gradients means that quantum computations are now compatible with techniques such as backpropagation, which is an important procedure in deep learning. Moreover, this allows the development of automatically differentiable hybrid networks, which involves both quantum and classical nodes. Such a pipeline has been

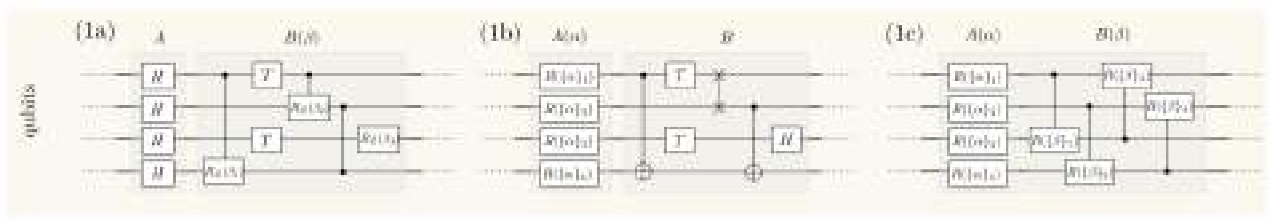


FIGURE 31: Ansatz in the variational paradigm can be segregated into two parts: A - state evolver and B - state entangler. The first part A consists of single unitary gates and evolves the state fed to it via state preparatory circuit. The second part B consists of both single and two-qubit gates. This introduces the entanglement between the qubits. In short, for any ansatz by parameterizing these two parts either individually or together let us tune its expressibility i.e. the states it could cover in Hilbert space and its entangling capability i.e. the measure of entanglement in the evolved state. Therefore, in order to choose a good ansatz, its template is decided based on the problems in hand (problem-based ansatz) and hardware being used (hardware efficient ansatz). [6]

represented in figure 33 with calculations of gradients using parameter shift rule.

Apart from such classically-parametrized quantum routines, there also has been considerable work done in developing quantumly-parametrized networks. To train these types of networks we make use of quantum mechanical phenomena such as quantum tunneling, phase kickbacks to get information regarding gradients. Thus, this allows for quantum-enhanced optimization techniques such as Quantum Dynamical Descent (QDD) and Momentum Measurement Gradient Descent (MoMGrad) to train parameters of these hybrid/quantum networks. Similarly, for meta-training, i.e., the quantum hyper-parameter training of such a hybrid/quantum network, variants of these algorithms such as Meta-QDD and Meta-MoMGrad have been proposed as quantum meta-learning methods. These methods rely on producing entanglement between the quantum hyper-parameters in superposition. Detailed discussion on these techniques is given in [10].

3.6 APPLICATIONS

Till now, we have learned that being in the middle of the second wave of QML, our best bet is hybrid approaches. These involve training parameterized quantum circuit models for a variety of supervised and unsupervised machine learning tasks on both classical and quantum data. In general, a few examples of these tasks could be:

- Recognizing patterns to classify classical data.
- Learning the probability distribution of the training data and generating new data following similar probability distributions.
- Finding short-depth hardware-efficient circuits for high-level algorithms.

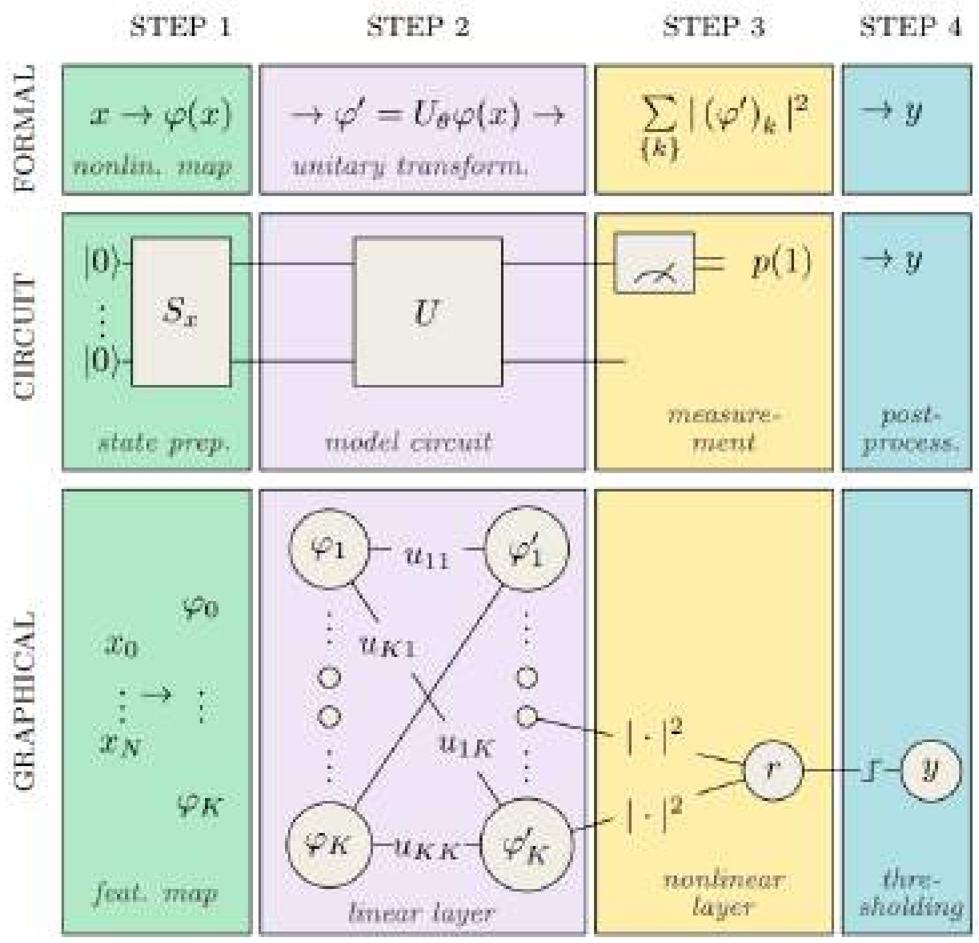


FIGURE 32: The quantum circuit utilized in the variational paradigm can be realized in the three ways: a formal mathematical framework, a quantum circuit framework, and a graphical neural network framework. In the first step, we encode our data into the quantum state i.e. our feature vector, using a state preparation scheme. The quantum circuit then evolves this state using unitary operations. This could be thought of as the linear layers of a neural network. Measurements present in the next step effectively implements the weightless nonlinear layer. Finally, at the post-processing step, we evaluate the thresholding function to produce a binary output. [6]

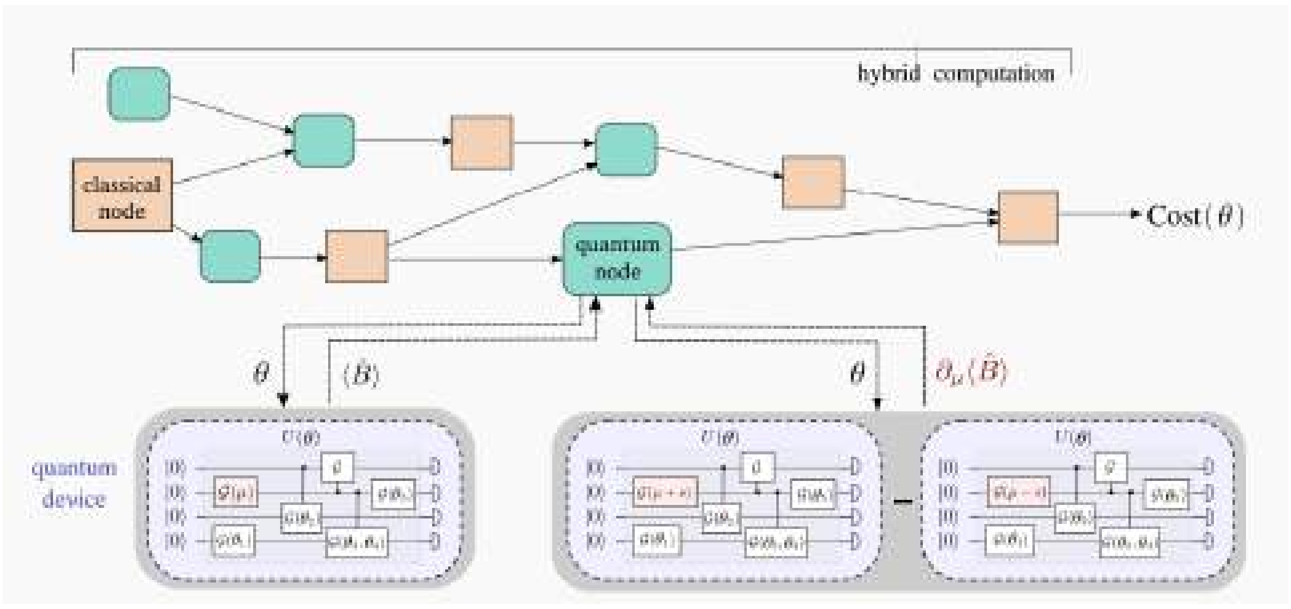


FIGURE 33: In the hybrid computation paradigm we can build networks that consist of both quantum (orange) and classic (blue) nodes. In such networks, an output from a classical node is fed to the quantum node on which it executes a variational quantum algorithm and finds the expectation value of the operator B depending on parameters θ. Here, we show the parameter shift rule, which allows us to compute the derivative of the quantum node by executing the circuit twice, but with a shift $(\mu \pm s)$ in the parameter with respect to which the derivative is calculated. [11]

- Compression of the quantum states.

These tasks enable using quantum machine learning in a variety of fields ranging from chemical simulations, combinatorial optimizations, finance, quantum many-body simulations, the energy sector, healthcare sector, etc

References

[1] Jurgen Van Gael, The Role of Interference and Entanglement in Quantum ComputingThe Role of Interference and Entanglement in Quantum Computing (2005), http://pages.cs.wisc.edu/ dieter/Papers/vangael-thesis.pdf

[2] Andrey Y. Lokhov, et al., Quantum Algorithm Implementations for Beginners, arXiv:1804.03719.

[3] Jacob Biamonte, Peter Wittek, et al., Quantum Machine Learning, 1611.09347

[4] Brassard, et al., Machine learning in a quantum world. In: Advances in Artificial Intelligence, Springer 2006.

[5] Farhi, Neven, Classification with Quantum Neural Networks on Near Term Processors, 1802.06002.

[6] Schuld et al., Circuit-centric quantum classifiers, 1804.00633

[7] Benedetti et al., Parameterized quantum circuits as machine learning models, 1906.07682

[8] Schuld, Petruccione, Supervised Learning with Quantum Computers, Springer 2018

[9] Guerreschi, Smelyanskiy, Practical optimization for hybrid quantum-classical algorithms, 1701.01450

[10] Mitarai et al., Quantum Circuit Learning, 1803.00745

[11] Schuld et al., Evaluating analytic gradients on quantum hardware, 1811.11184

[12] Guillaume Verdon et al., A Universal Training Algorithm for Quantum Deep Learning, 1806.09729