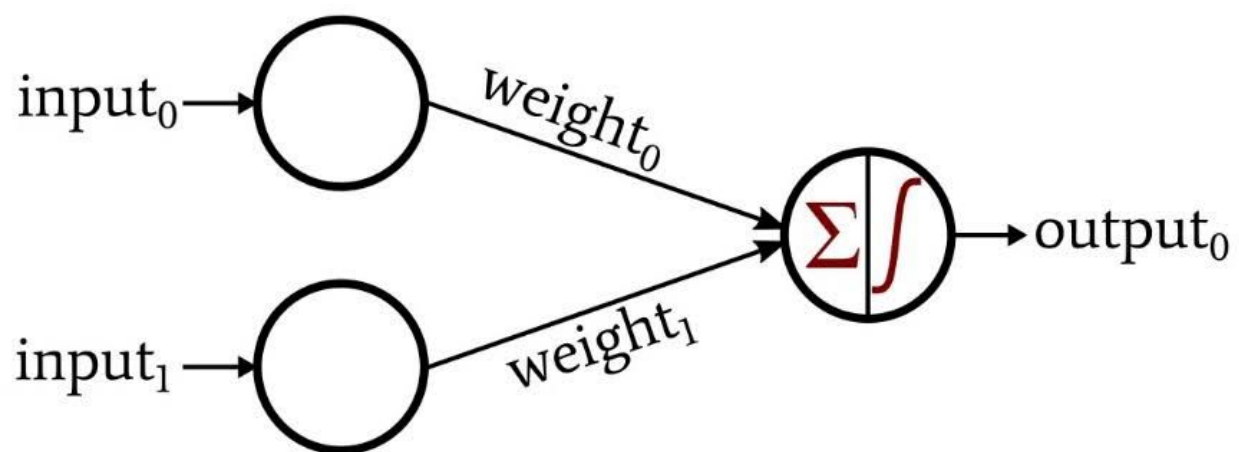







The document is an explanation of the Asimov Neural Network Zoo. Information was learned and explained in a simplified manner using the following link as well as links in the Data Science Informative Articles spreadsheet.








<https://www.asimovinstitute.org/neural-network-zoo/>

Simple Neural Network Example



Different Types Of Cells

1.  **Input Cell** : Point from where input is fed, as it is, into the network.
2.  **Output Cell** : Point from where the final output is computed. Typically uses the $\max()$ activation function to output the highest probabilistic value it has received from previous layers.
3.  **Hidden Cell** : Cells of layers other than the input and output layer. These act as the major computing engines of a NN where values are received, weighted-added, squashed and backfed to tune the network's parameters.
4.  **Recurrent Cell** : Cells that receive and/or store their own output. The storage is typically for one pass.
5.  **Memory Cell** : Typically used in LSTMs, these cells have the tendency to store or discard long series of prior information.

6.  **Gated Memory Cell** : Similar to the aforementioned memory cell but has a different structure.
7.  **Match Input Output Cell** : Used in AutoEncoders to match the input received and the subsequent output.
8.  **Probablistic Hidden Cell** : Applies an Activation Function (typically the Radial Basis Function) to the difference between the cell's mean and the test case.
9.  **Noisy Input Cell** : A point where noise in any form to the data can be added.
10.  **Backfed Input Cell** : Cell that introduces feedback i.e. information from the previous iteration's outcome.
11.  **Kernel** : Cell where a particular kernel is applied to a subset of the data.
12.  **Convolution or Pool** : Aggregates the result of kernel layers and keeps relevant information.

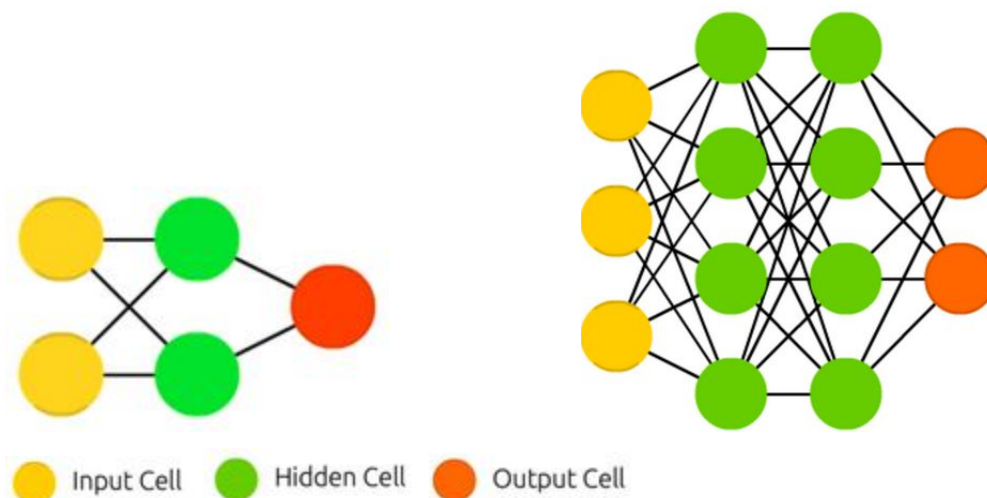
Source of cells: <https://ai.stackexchange.com/questions/5898/neural-network-cell-node-types>

Simple Perceptron (P)



A network, without hidden layers, where 2 or more input cells correspond to an output. A Simple Perceptron is an example of a Feed Forward Neural Network (FFNN) where information flows from the input to the output.

FeedForward & Deep FeedForward Neural Networks (FFNNs & DFFNNs)



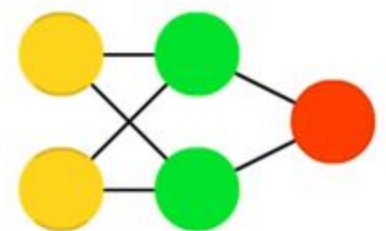
FFNNs are called as such because they involve the flow of information from the start (input) towards the end (output). These networks generally involve an input layer with 1 or more cells, an arbitrary number of hidden layers and an output layer. FFNNs generally make use of backpropagation i.e. a process that updates the weights of the FFNN depending on the error rate from the previous iteration to produce good results. The error rate is always a quantity that quantifies the difference between the input and output and it can be calculated through several ways e.g. Mean Squared Error (MSE), Squared Error or just a simple difference.

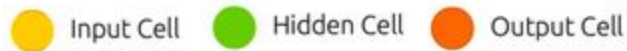
Deep FeedForward Neural Networks (DFFNNs) are like simple FFNNs but with multiple hidden layers. These form the basis of core Deep Learning today as most networks are essentially FFNNs.

Practical Applications:

- Autonomous driving cars.
- Prediction of changes to Alkenes' structure.
- Prediction of cell concentrations depending on certain input features.
- Form the basis of further complex NNs such as RNN, LSTM etc.

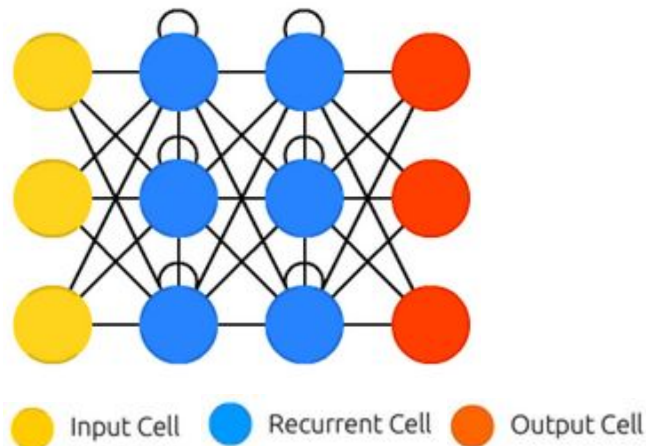
Radial Basis Function Networks (RBFNs)





An implementation of FFNNs with the Radial Basis Function as the activation function used in the network.

Recurrent Neural Networks (RNNs)



RNNs are a special form of FFNNs. These networks contain hidden layers with cells that temporarily maintain state (generally for one or very few epochs). Maintaining state means that the cells not only receive input from the previous layer but they also receive the value that they themselves outputted in the previous iteration. This can be attributed to the cell being able to ***temporarily remember its prior activity*** hence RNNs are called stateful networks.

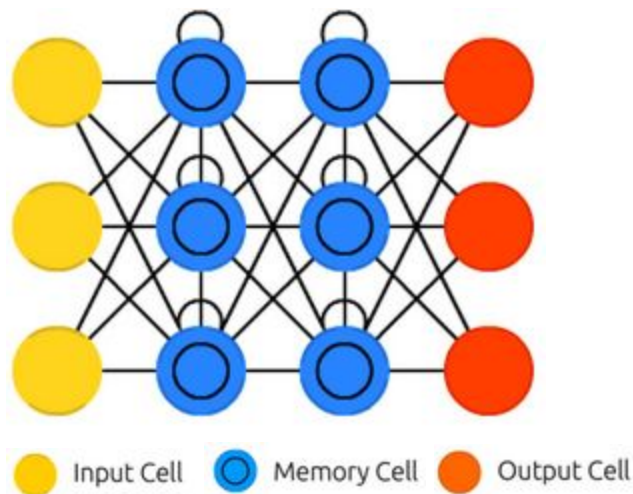
This allows RNNs to greatly reduce the error-rate per epoch. However, an important consideration that needs to be made with RNNs is that the order of data inputted to the network matters. Arbitrarily feeding the RNNs with data can yield to varying results.

RNNs can suffer from the Vanishing Gradient Problem. This problem generally tends to occur because of activation functions squashing values into a small range. This causes even a huge change in the parameter values of the initial layers to have a very small impact on the output of the network. The Vanishing Gradient Problem is generally faced by gradient-based learning methods.

Practical Applications:

- Autocorrection and autocompletion in text.
- Image recognition and characterization when used in conjunction with a Convolutional Neural Network (CNN).
- Machine Translation.
- Used to build more complex networks such as LSTMs.

Long / Short Term Memory (LSTM)



LSTMs were introduced to combat the Vanishing Gradient Problem in RNNs. This is done with the help of gates, present at the memory cells to regulate the flow of information and updating of network parameters. The three types of gates are as follows:

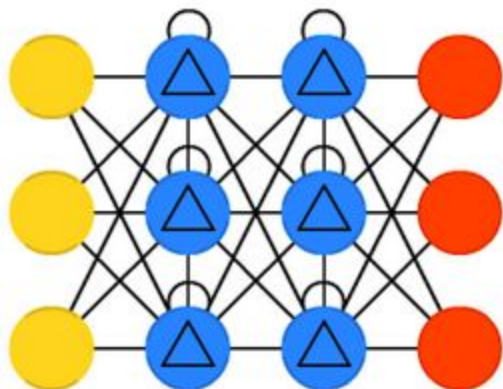
- **Forget Gate:** This gate is responsible for regulating how much old information should be *forgotten* by a cell to allow it to learn new information. This is usually done by element-wise multiplying the input with a vector whose values decide how much of the information should be remembered or discarded. It outputs a vector that's calculated by passing input from the previous state (or previous cell) and the current input through a sigmoid function that squashes values between 0 to 1.
 - Vector with values close to 1 indicate that all of the old memory must be remembered and contribute to the new state of the cell.
 - Vector with values close to 0 indicate that most of the old memory must be forgotten and contribute less to not at all to the new state of the cell.
- **Input Gate:** The input gate determines how the new state of the cell will be determined. In other words, this gate determines what information will be kept by the cell as part of its updated state. This involves the following steps:
 - Passing of previous state's information and the current input to a sigmoid function that determines what values are important to remember and what are unimportant. Close to 1 corresponds to being important and close to 0 corresponds to being unimportant.
 - Passing of previous state's information and the current input to a tanh function to help regulate the flow of the network. The purpose of this step is to simply normalize the values or simply to perform squashing.
 - The output of both the previous steps is then multiplied. The sigmoid output determines the values that will be kept and forgotten from the tanh output.
 - The new cell state is then calculated through:

- The multiplication of previous cell state information with the output of the forget gate (forget vector). This ensures dropping of unimportant values.
 - Element-wise addition of the output of the aforementioned step with the output of the input gate.
- The result of all the aforementioned steps gives us our new cell state.
- Output Gate: This gate determines what information will be passed onto the next neuron as the hidden state of the current neuron. This information contains updated values because of all the aforementioned steps and it determines the state of the next neuron. Henceforth, this recurring procedure occurs until the output layer which determines the final LSTM output. The output gate works as follows:
 - The information from the previous hidden state (previous cell) and the current input are passed to a sigmoid function.
 - The new cell state from the input gate is then passed to a tanh function.
 - The output of both the sigmoid and tanh function is then multiplied with each other. This determines what information will be passed to the next cell as the current cell's hidden state.
- In a nutshell:
 - Forget gate determines what information to keep or discard from previous states.
 - Input gate determines what information to keep or discard from the current input to modify the current state of the cell to an updated state.
 - The output gate determines what information related to the new state of the cell should be passed-on to the next cell.

Practical Uses:

- Autocompletion of texts.
- Generating music or speech.
- All types of forecasting/time-series problems such as stock price prediction, expected sale prediction etc..

Gated Recurrent Units (GRUs)





GRUs are a variation of LSTMs. They were introduced to reduce some computational overhead and complexity of LSTMs but one cannot claim one to be a clear winner over the other in most use-cases. GRUs have one gate lesser than LSTMs and the flow of information through them is a tad different. They also tend to combine the previous state and current cell state of LSTMs.

- **Update Gate:** The update gate's function is to determine how much information from the previous states shall be allowed to enter the current cell and how much will be kept in the current cell. This gives GRUs, like LSTMs, the ability to remember or discard long series of previous values. It's equivalent to the forget and update gates of LSTMs. Its working is as follows:
 - The information from previous states and the current input are multiplied by their corresponding weights.
 - Both the resulting vectors are then added and passed to a sigmoid function that squashes these values between 0 and 1.
 - Again, the closer values to 0 means they are to be forgotten while closer to 1 means that they are to be remembered. This step ensures tackling of the Vanishing Gradient Problem.
- **Reset Gate:** Acts like the forget gate of an LSTM which also helps in determining how much of the past information can be discarded.

Practical Applications:

- Same as LSTMs but are faster.

LSTMs vs GRUs:

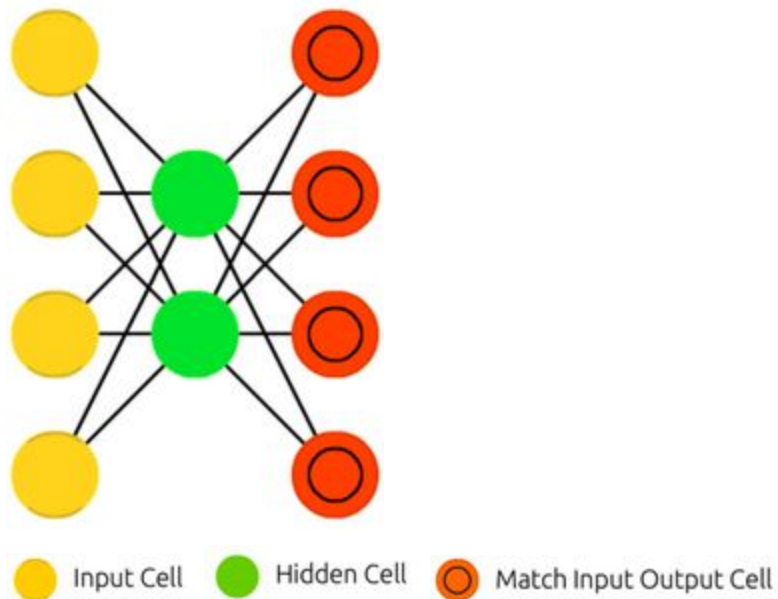
GRUs are said to be computationally efficient and less complex compared to LSTMs. LSTMs on the other hand are said to perform better when it comes to tackling problems that involve taking into account very long sequences of data. GRUs are a relatively newer concept compared to LSTMs. As mentioned earlier, there is no clear winner between the two and ML/DL Engineers generally tend to use both to determine which works better.

However, GRUs are said to be faster and easier to train. LSTMs on the other hand take on a little extra computational overhead but perform better with very long sequences.

BiRNN / BiLSTM / BiGRU:

The Bi corresponds to bidirectional, which means that the networks have access to future information. Access to future information means that they're also fed in the next information that helps them to contextually process the new information and then make decisions accordingly. The Bi versions of these networks have exactly similar architectures to the simpler or unidirectional counterparts.

AutoEncoders (AEs)



Auto Encoders are an implementation of FFNNs with a known structure that follows certain characteristics. They're used to compress input data, layer by layer up until the middle layer and can be trained using gradient-based methods such as backpropagation. AEs are referred to as a feature extraction algorithm that uses existing features to create new ones. Since the AEs' main function is to try to recreate the input by using the most relevant features, it is very similar to the Principle Component Analysis (PCA) algorithm. Structure and some other information related to AEs is as follows:

- The shape of the network is always like an hourglass.
- The middle layer(s) (can be two if the total number of layers is an even number) has the smallest number of neurons of all layers making it the chokepoint of the network.
- The network is symmetric around the middle layer i.e. it has an equal number of layers before and after it.
- The number of neurons in each layer decreases up to the middle layer and then starts to increase up to the output layer.
- The decreasing of neurons to the middle layer corresponds to the AE removing unwanted features of the data to reconstruct the output.
- The part of the network just before the middle layer is called the encoder; this part encodes/compresses the data to extract only the relevant features.
- The middle layer(s) are called the bottlenecks or chokepoints.
- The part after the middle layer(s) is called the decoder; this is the reverse of encoding. This part uses the extracted features to reconstruct the original input in the best possible manner. It basically performs decompression of the encoder part.

Since the middle layer(s) are chokepoints with lesser number of neurons, AEs have to formulate a way to represent the input using lesser number of features compared to the original input. This ensures that AEs remove unwanted information throughout the encoding part of the network and extract only that information which is most relevant to the reconstruction of the input.

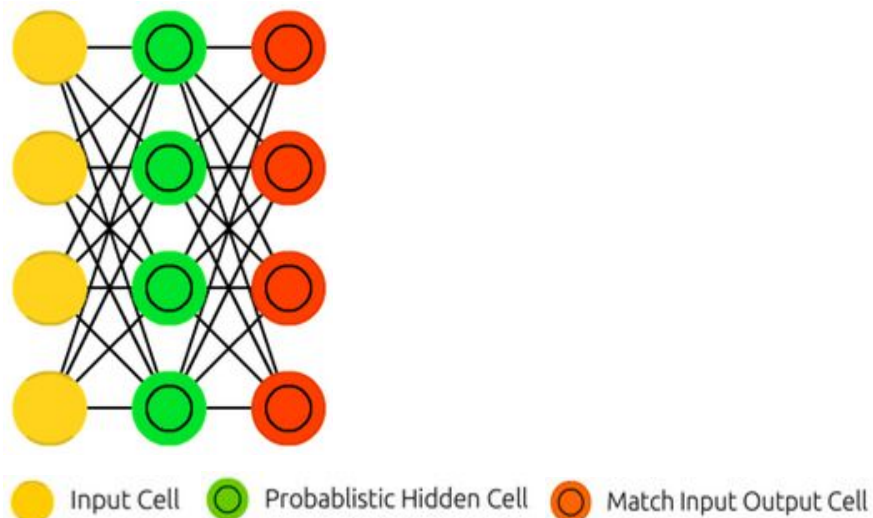
The chokepoint(s) is the place where data is in its lowest representation. This compressed data is then used by the decoding part to recreate the input the network received in the first place using lesser number of features.

Since we can use backpropagation to train the network, the training is aided by a distance function which quantifies the difference between the input and the output during each compression. This is called the loss function.

Practical Applications:

- Denoising data.
- Dimensionality reduction.
- Used in conjunction with other algorithms as an input source.

Variational AutoEncoders (VAEs)



Variational AutoEncoders are different from AEs because they follow probability theory rather than compression and decompression of information. The main difference between VAEs and AEs is that AEs are used to **reconstruct the input** they've received using a lower dimensional representation through the process of feature extraction and lossy compression of data.

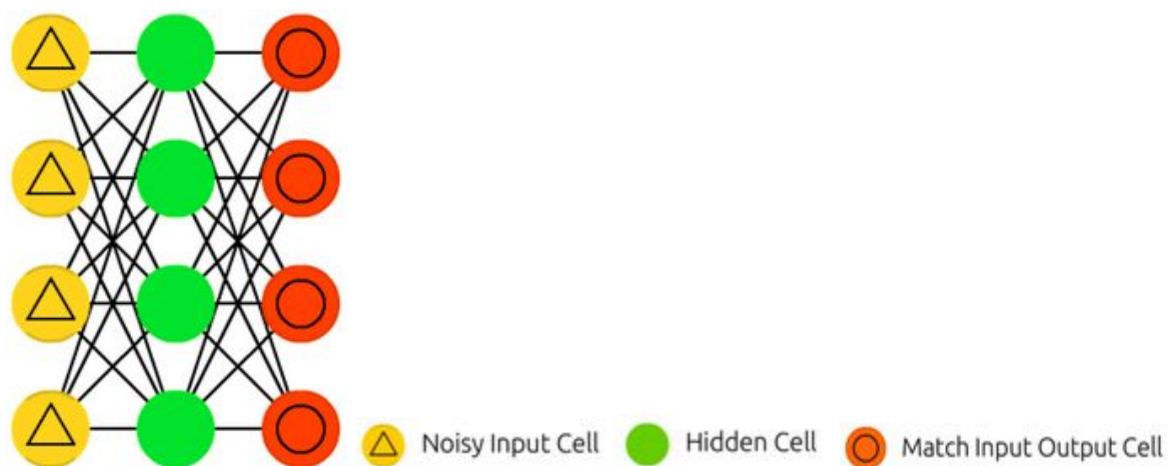
VAEs on the other hand are able to model the data like a generative algorithm. They learn a probability distribution of the input data and based on this, are able to **recreate as well as generate new data** by sampling from the distribution. Their training is regularized and because of using some complex Bayesian mathematical procedures under the hood, they can use the

concept of independence of events to regulate error propagation; the influence of those cells (or data) is reduced that are independent during backpropagation which results in better learning of network parameters.

Practical Applications:

- Used for constructing images.
- Used for image compression.
- Interpolating between two sentences.
- Aforementioned applications of AEs also apply to VAEs.

Denoising AutoEncoders (DAEs)

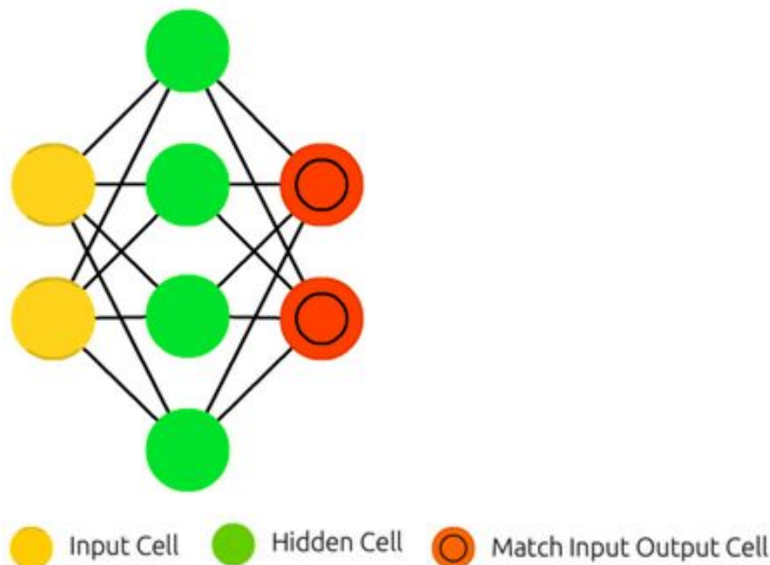


DAEs are another special form of AEs that learn to differentiate between noise and the more relevant features of the input data. This is achieved by feeding the network both the original input data as well as a more noisier form of it. The network, using backpropagation as its learning method, is able to make sense of what features are noise and what contribute more towards producing a better output. This in-turn allows the DAE to learn relevant features of data and make them *immune* to noise.

Practical Applications:

- Sharpening filters.
- Increasing training data when data is less; we can add varying levels of noise to the existing data to create more.

Sparse AutoEncoders (SAEs)



SAEs and previous AEs have one thing in common; all are feature extraction algorithms. However, the way SAEs behave and their architecture is completely opposite to the previously mentioned AEs. The input layer of SAEs has the lowest number of cells. The number of cells tends to increase until the middle layer(s) of the network and decrease again to the output layer. This is in stark contrast to the Simple AutoEncoder(AE) whose input layer has the highest number of cells and they decrease towards the middle layer while increasing again from there-on.

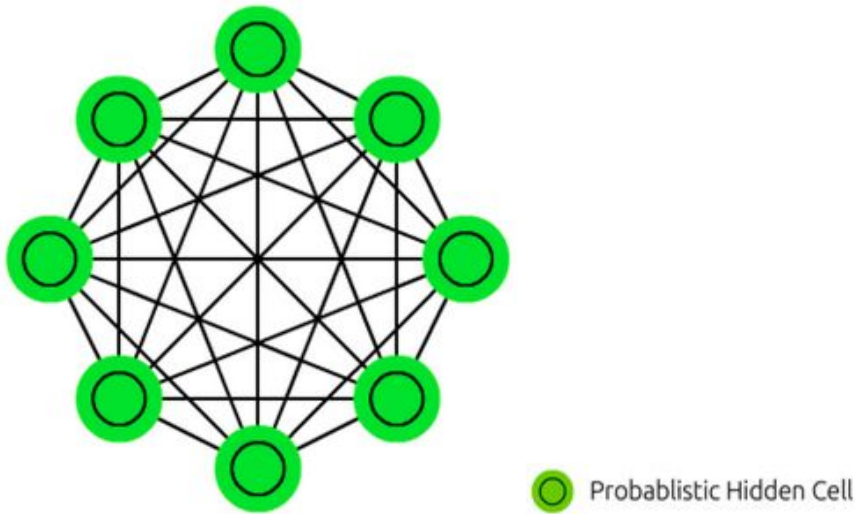
This architecture of SAEs allows them to extract numerous features from the input. However, this can also lead to overfitting as the algorithm is in-fact trying to learn extra things compared to what it is being fed. To avoid that, L1 or Lasso Regularization (and sometimes KL-Divergence or Kullback-Leibler Divergence) is applied to the cost function of the network. A third penalty called the Sparsity Penalty is added to the cost function to control the activation of cells for a given pass.

This cost function ensures that the SAE is not using all the cells in its middle layers all the time and is hence not overfitting the data. This is why SAEs are referred to as sparse because not all of their cells are used to compute values during each epoch. This ensures that the SAE is still compressing the input (despite having a greater number of neurons in the middle layers compared to the input layer).

Practical Applications:

- Information Retrieval
- Medical Image Processing
- Anomaly Detection

Markov Chains (MCs)



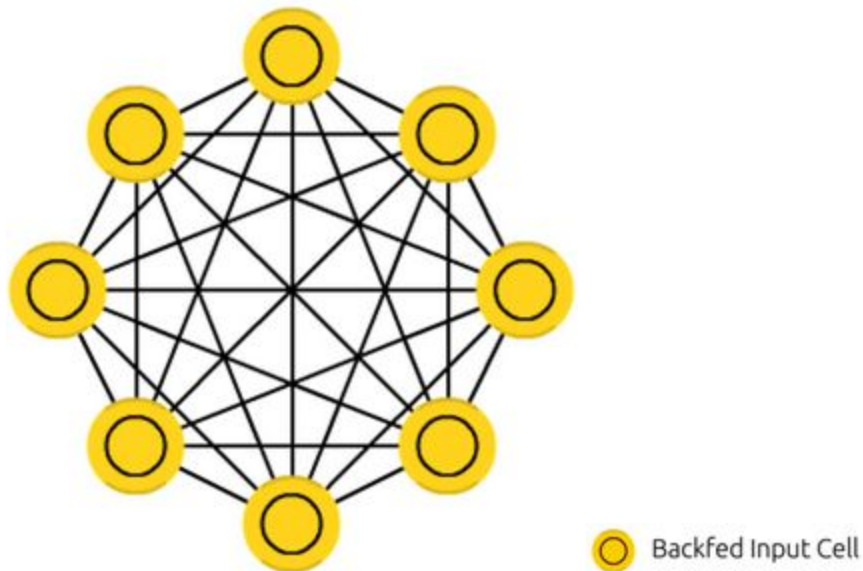
Whilst not being your traditional Neural Network, Markov Chains are nevertheless a very interesting algorithm based on probability theory. Markov Chains are used to predict the occurrence of a series of events in a given process. The process follows the Markov Property which states that if we're aware of the current state of a given process, its past and future states are completely independent of one another. In simpler words, if we're somehow aware of the current state of a given process, its past states will have no effect on the prediction of the next state.

Moving from one state to the other is completely dependent on the current state. The probability of moving from one state to the other is referred to as the Transition Probability. Markov Chains need not to be fully connected i.e. every neuron is connected to every other neuron.

Practical Applications:

- In finance to predict market trends e.g. bearish, bullish or stagnant.
- Predicting asset prices.
- Speech recognition.
- Studying animal population trends.
- Sequence completion.

Hopfield Networks (HNs)



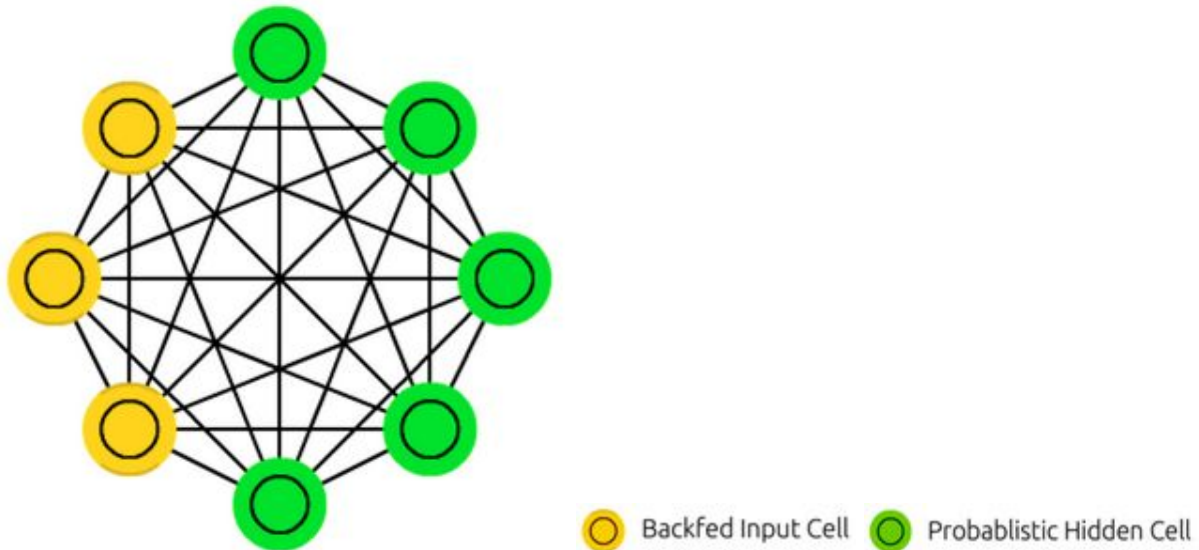
Hopfield Networks are trained on a limited amount of preselected data and are fully connected. Their limited training allows them to respond to different kinds of unknown data (that is close to the data they've been trained on) with what they've already learned.

Each cell acts as an input cell before training, a hidden cell during training and an output cell after training is complete. The tuning of their weights according to the specific data they receive, makes them respond to different unseen data in a similar manner.

Practical Applications:

- Denoising of data.
- Image completion.
- Text completion.

Boltzmann Machines (BMs)



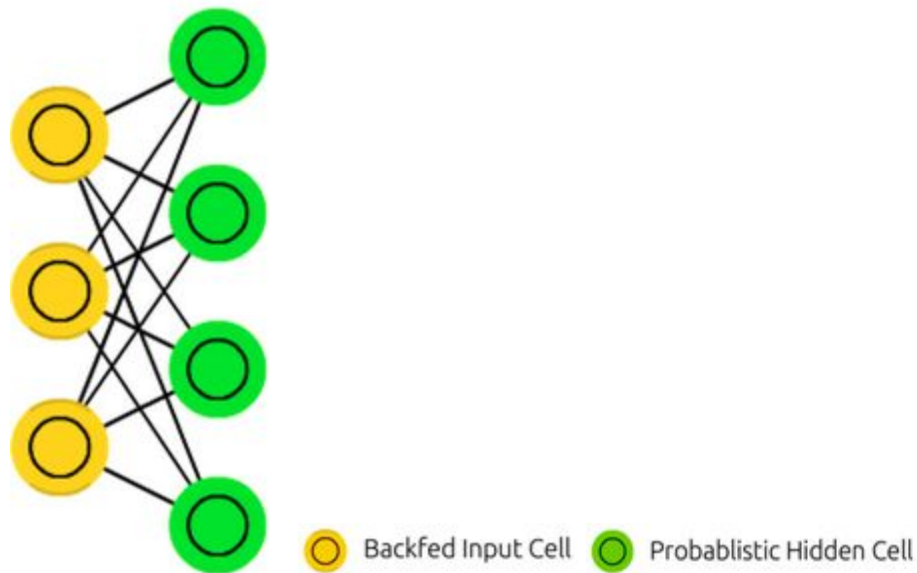
BMs have a similar structure to Markov Chains with the fundamental difference being the introduction of backfed input cells. These input cells remain as points of input but convert themselves into output cells once all hidden cells change their state. They're interconnected which allows them to share information among one another to learn better.

BMs are generative algorithms i.e. they try to build model(s) for each of the different data and make new classifications accordingly. The sharing of information among one another also allows them to generate subsequent data which allows them to solve combinatorial problems i.e. problems that involve finding an optimal solution from a finite set of solutions.

Practical Applications:

- Feature extraction.
- Search problems.
- Combinatorial problems such as shortest path estimation.

Restricted Boltzmann Machines (RBMs)

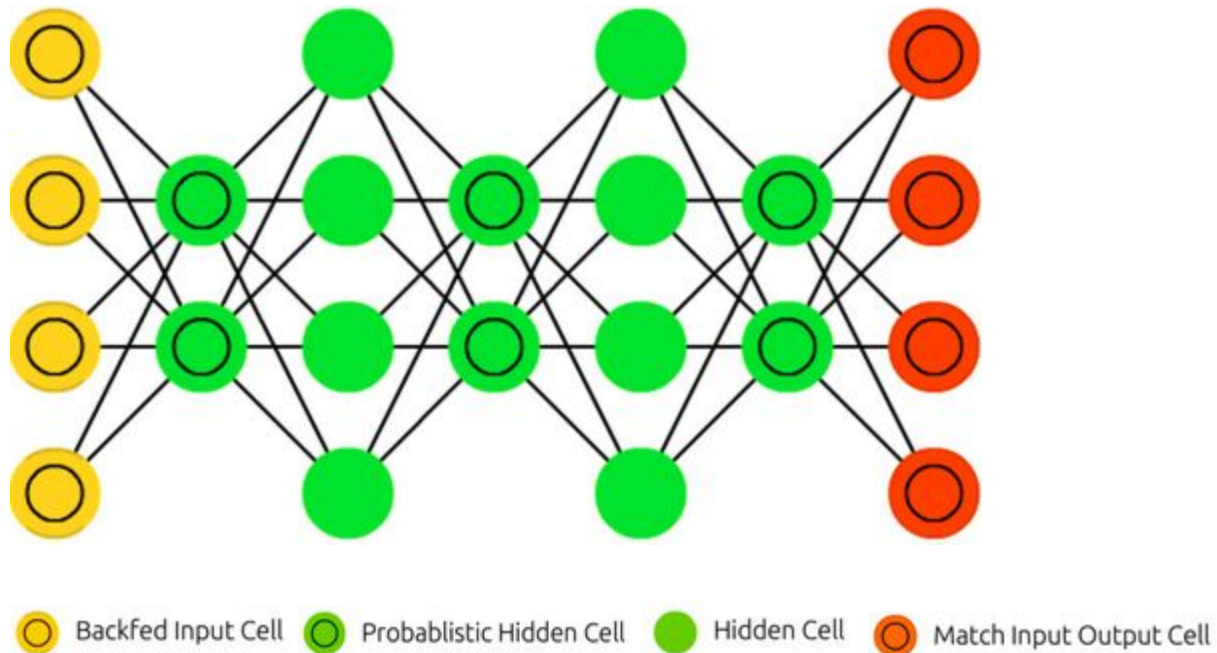


RBMs are similar to BMs but have a fundamental difference in their structure; all the cells are not fully connected to one another and they are two-layered networks. This is the reason why RBMs are referred to as *restricted*. No two cells in a single layer are connected to one another. This property makes them less complex compared to BMs and they're generative algorithms because they learn a probability distribution over their inputs.

Practical Applications:

- Stacked to form Deep Belief Neural Networks (DBNNs).
- Feature extraction/elimination.
- Optimization of speech recognition algorithms.

Deep Belief Neural Networks (DBNNs)



DBNNs are generally a stacked version of multiple RBMs or VAEs on top of one another. Each preceding RBM/VAE is used to train the superseding RBM/VAE by using the output of the previous as an input to the next. DBNNs learning is based on a greedy approach; each layer learns its parameters one by one and get trained and produce an output accordingly (local learning). This allows all stacked RBMs/VAEs to receive data in different representations that promotes better learning.

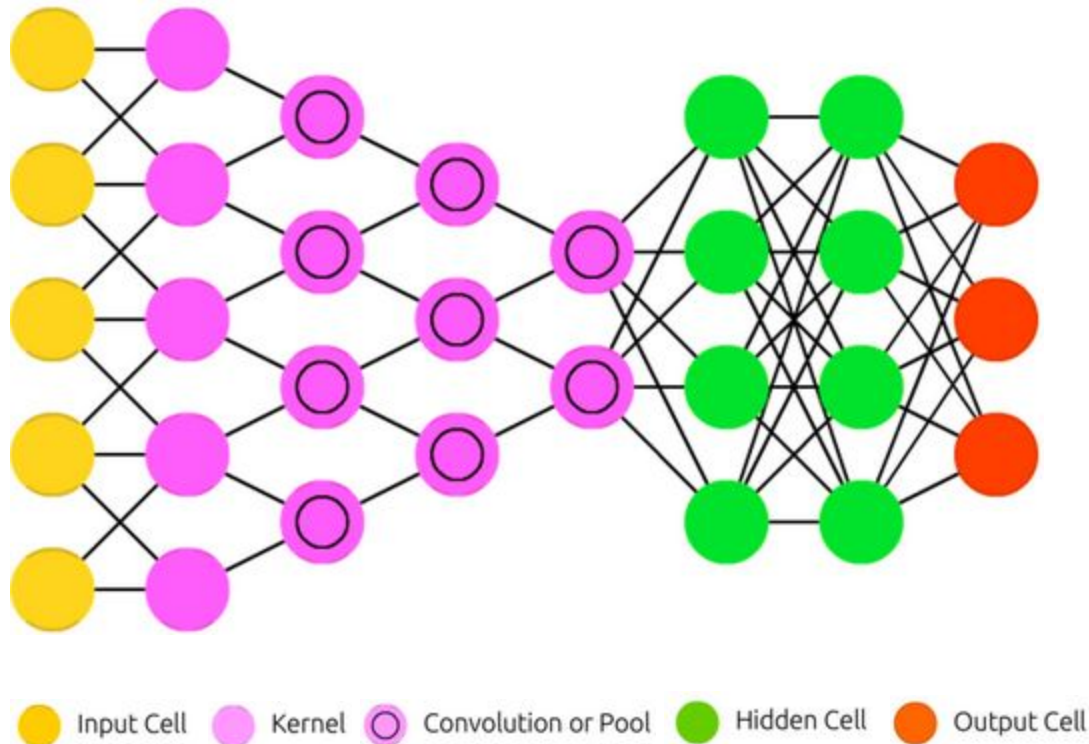
Greedy approaches are based on locally learning the best solutions and combining them to form a global solution. Local learning is typically independent of other layers which makes this process not very optimal. Henceforth, the culmination of multiple local learnings may not always give you the optimal global solution. To cater for that, weights of a DBNN are fine-tuned through different algorithms like backpropagation.

Despite greedy approaches not being the most optimal, they still have their own benefits and this iterative learning procedure allows lower layers to learn different representations of data.

Practical Applications:

- Can be used for both classification and generating new data.
- Image and video recognition.
- Object tracking.

Convolutional Neural Networks (CNNs)



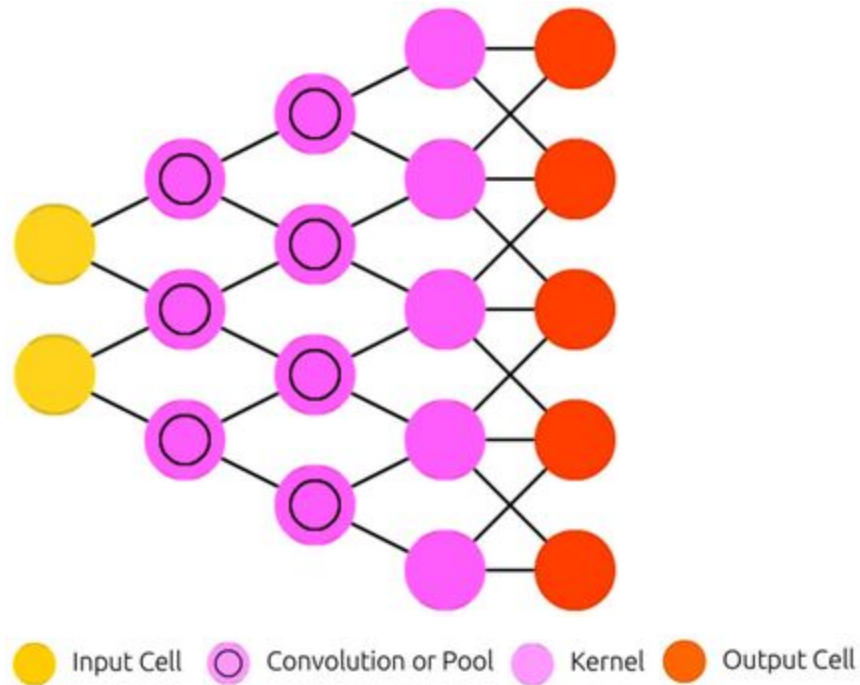
CNNs are one of the most Deep Learning algorithms. They allowed breakthrough Image Processing applications and later were used to work with other types of data as well such as audio and even text!

A typical CNN architecture is as follows:

- **Input layer:** point of entry for the data into the CNN.
- **Convolutional layer(s):** This layer(s) is responsible for applying a kernel to the input data. A kernel is a $N \times N$ dimension filter that is applied to certain pixels of an image (typically starting from the top-left corner) and then passed forward. One after the other, the entire image is convolved-over by the kernel.
- **Pooling layer(s):** This layer(s) is responsible for aggregating the results of the convolutional layers generally either through max pooling or average pooling.
- **Fully connected layers:** These layers receive output from the Pooling layers and then make sense of what those outputs mean.

Working of a CNN is as follows: TBAAdded.

Deconvolutional Neural Networks (DNNs)



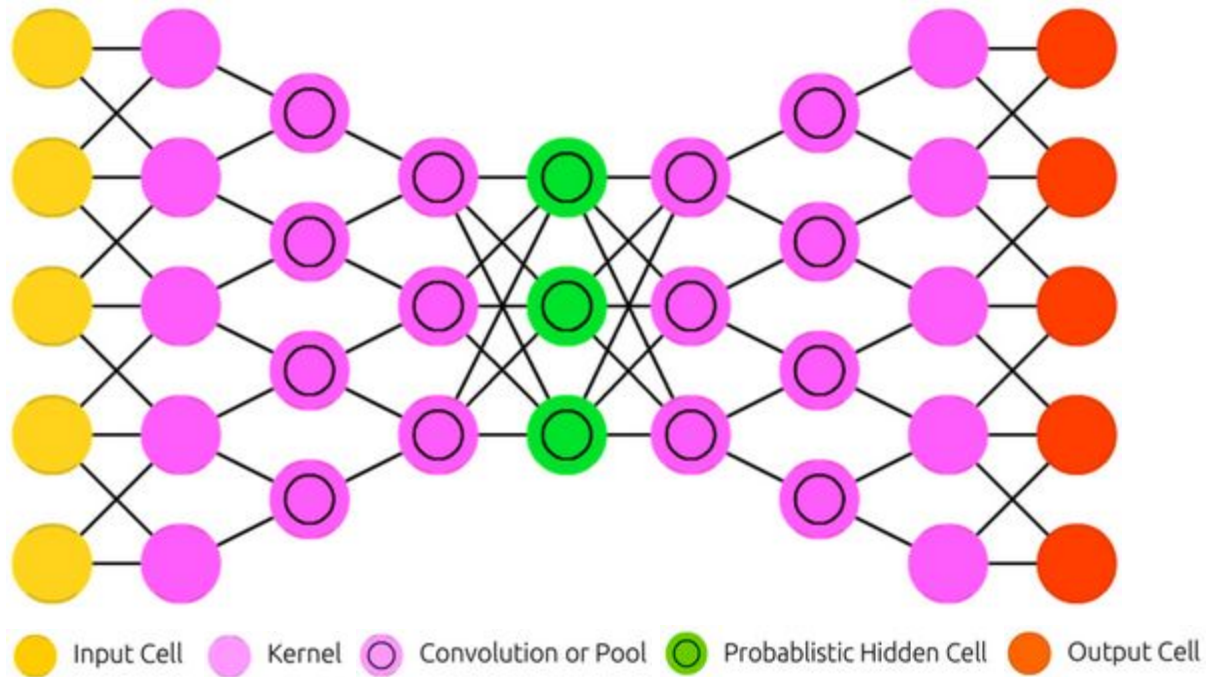
DNNs are an example of CNNs but with a completely opposite architecture. Instead of convolving and pooling the original image into smaller parts and then making a classification, DNNs take an image and try to learn more features about it and hence reconstruct it.

Generally, the DNNs are trained by giving them a set of data. This generates class vectors i.e. vectors with information that determine what the input actually is. Once trained, the DNNs are then fed variations of these which they in-turn use to reconstruct the image.

Practical Applications:

- Simple image reconstruction and reconstruction from different angles.
- Image synthesis and analysis.
- Images of objects that no longer exist.

Deep Convolutional Inverse Graphics Networks (DCIGNs)



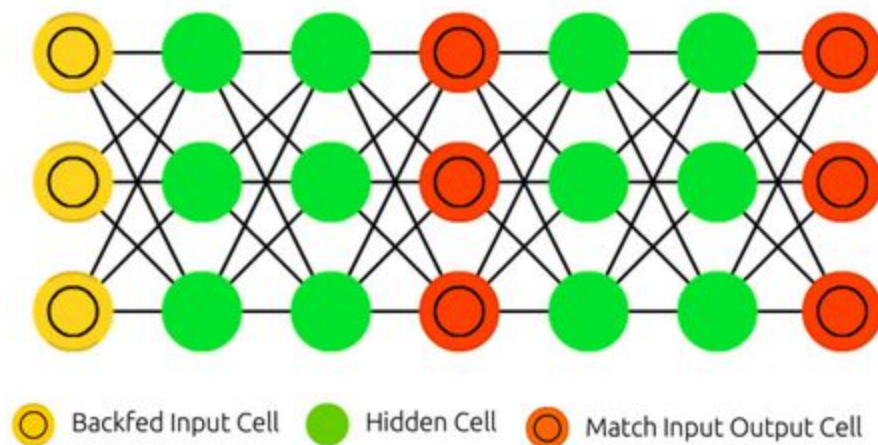
DCIGNs are comprised of Convolutional Neural Networks (CNNs) and Deconvolutional Neural Networks (DNNs) working in tandem. The beginning part is a CNN and the part after the middle is a DNN which makes the DCIGN behave like a Variational AutoEncoder (VAE); the first part compresses/encodes the input image while the second part decompresses/decodes the input received from the CNN.

Due to its VAE-like structure, DCIGNs are also generative algorithms that work by learning probability distributions of their input. They can also work with images on which they haven't been trained previously.

Practical Applications:

- Removing unwanted objects from images.
- Adding objects to a single image using different images.
- Producing new images with different representations of the original image e.g. image with object rotated or translated or with different lighting conditions.

Generative Adversarial Networks (GANs)



Generative Adversarial Networks or GANs were introduced in 2014 and have since taken the ML world by storm. These networks operate on the principle of Adversarial Learning where both the networks try and confuse one another thereby improving each other in return.

The first part of GANs is the generative part. This part is responsible for learning to generate images based on the discriminative part, which is the second part. The discriminative part discriminates whether how close the output of the generative part is to the actual data. This back and forth generation and discrimination eventually leads to network parameters from which the discriminator can no longer discriminate between the generated and the original images. This is where GANs are ready to make predictions.

Working of GANs is as follows:

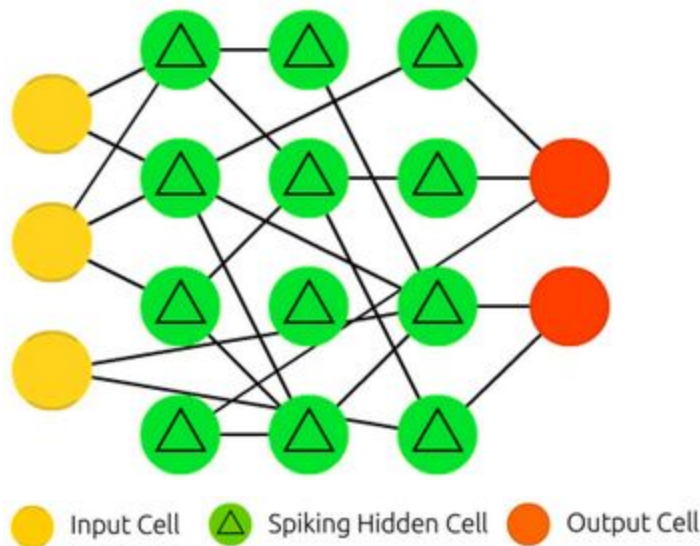
- The discriminative part of the network is fed training data through which it learns on what basis will it discriminate the generator's output - Supervised Learning.
- The generator is then fed random noise or different variations of the original data.
- The generator uses that noise to generate fake images which are then passed to the discriminator.
- The discriminator then determines whether that fake image was close enough to the original images.
- Based on that, it updates the error function of the network that tells the generator how badly it did and also points its towards the direction in which it should update its weights.
- The generator then uses this information to tune its weights and then reproduce another image that is then fed again to the discriminator.
- The same process is then repeated back and forth until the the error function of the network is minimized and no longer updates by the discriminator; this basically corresponds to that state of the discriminator when it's unable to tell the difference between generated and original images.

- When this state is reached, the network parameters are optimally set and the generator is then ready to generate new images.

Practical Applications:

- Image manipulation e.g. face swapping, adding sunglasses to a person in an image.
- Image generation of things that do not exist.
- Text to image generation.
- Face aging or anti-aging.

Liquid State Machines (LSMs)



Liquid State Machines or LSMs are special types of networks in which all the neurons are connected in a random manner with each other in a sparse manner. These networks use special types of hidden cells called Spiking Hidden Cells because unlike other hidden cells which are activated for any value greater than 0 (computed by their activation functions), these cells only activate once a specific threshold is reached.

That is why, they can be called as remaining dormant and then spiking instantaneously once a threshold value is reached or crossed. They also have the tendency to remember information from previous iterations (have memory)

Input cells are the part from where input is fed to the spiking cells. Initially thresholds of the spiking cells are set at random. During the training phase, these cells tend to accumulate information with them for a certain number of timesteps until their threshold is reached. As soon as that happens, certain neurons activate and produce outputs.

Since not all neurons are connected to one another, a certain pattern of hidden neurons is activated as a result of the aforementioned step and this happens recurrently during the training

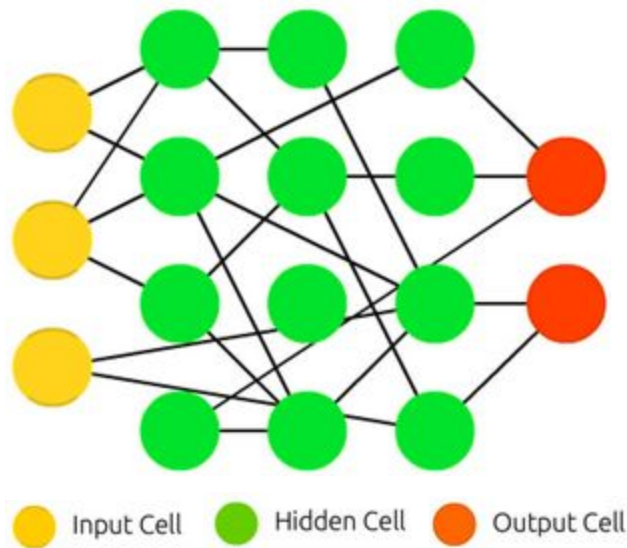
phase. These patterns of neurons firing for a specific sequence of input is what's used to train the LSMs and make predictions.

LSMs are a particular implementation of Reservoir Computing.

Practical Applications:

- Video activity recognition.
- Speech recognition.

Extreme Learning Machines (ELMs)



Structure of ELMs is a mix between your typical Feed Forward Neural Network (FFNN) and the Liquid State Machine (LSM). The major difference between ELMs and LSMs is that ELMs do not have spiking cells and their training (learning of network parameters) is done in a single step using the least squares fit technique. Also, they're generally an implementation of what are referred to as Single Hidden Layer Feedforward Neural Networks (SLFNNs); they have one hidden layer.

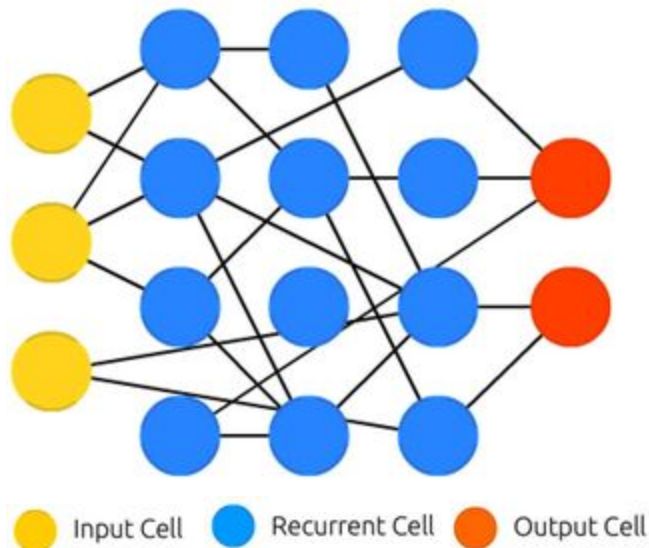
The weights and biases for the connections between the input and hidden layer(s) are randomly assigned. For the connections between the hidden and output layer, their weights are calculated using a least squares fit technique in one single go. This is contrary to the incremental learning and tuning of weights for other Neural Networks which makes the ELMs very fast in to train.

The least squares fit technique is aimed to find a model that minimizes the sum of squared differences between the actual and predicted values. ELMs use this to fit a model and hence one of the major advantages said to be of ELMs is that they're extremely fast to train and tend to produce good generalization results. Albeit, not the most accurate it's said.

Practical Applications:

- Classification problems such as diagnosis of diseases.
- Pattern Recognition.
- Forecasting Problems.

Echo State Networks (ESNs)



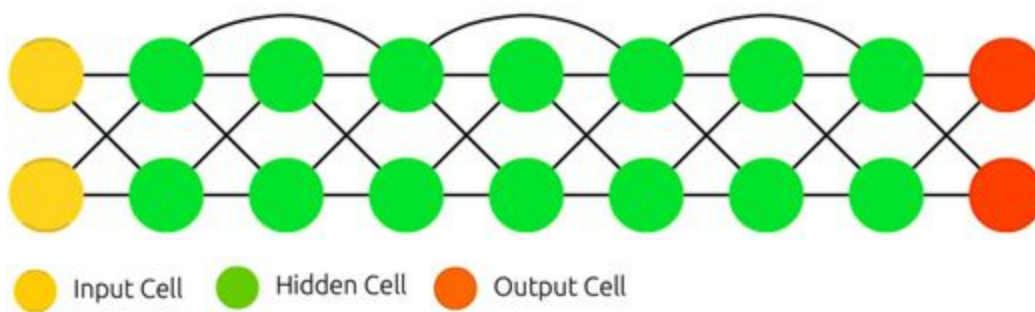
Echo State Networks (ESNs) have a similar structure to LSMs and ELMs in a sense that their cells are sparsely connected. Only the weights connecting from the last hidden layer to the output layer (output weights) get trained while all the remaining weights are sampled from a random distribution.

This kind of training allows ESNs to learn specific patterns which they can then use to make predictions. The sparsity of these networks is typically very low, sometimes less than 10 or even 1%.

Practical Applications:

- Used for chaotic time series problems e.g. stock price prediction.

Deep Residual Networks (DRNs or DResNets)



Deep Residual Networks or ResNets as they're often referred to are an implementation of FFNNs with many more layers. Not only does input get feed forwarded in the traditional manner but input is also fed forward by cells to layers other than the exactly next layer referred to as *skip or skipping connections*.

This allows the network to learn better in a way that it learns from a variation of the input on different layers i.e. certain cells are supplied both the input from the previous neuron as well as the input from the previous layers. This property is said to allow these networks to learn very well despite going in excess of 150 layers.

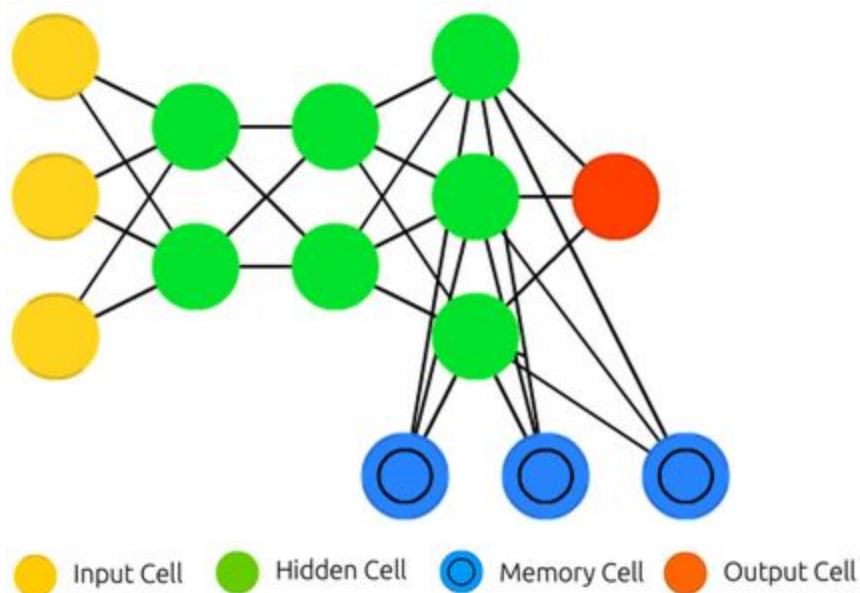
Generally, increasing the number of hidden layers tends to overfit the data as the network tries to learn a more complex function. The use of skipping connections not only prevents ResNets from overfitting the data but they're also said to combat the Vanishing Gradient Problem. This is considered as a major breakthrough when it comes to Deep Learning.

However, for now, one can only add layers up to a certain extent as adding more and more layers has been proven to increase both training and validation error.

Practical Applications:

- Image classification.
- Object detection.
- Facial recognition.

Neural Turing Machines (NTMs)



Neural Turing Machines are Neural Networks that mimic the functioning of a normal computer. A simple computer has a memory/storage to whom it can write and read from allowing it to perform different tasks.

NTMs can also be called another form of LSTMs with memory cells (gated cells of LSTMs) segregated from the hidden cells. This gives these the separated memory to which the hidden cells of the network can read and write from. These allows NTMs to have LSTM-like capabilities of processing large sequences of inputs.

A major part of their architecture is the controller. The controller is the part of the network the takes the input, stores/updates information into the memory cells and depending on the output, does this process again during the training phase. This allows NTMs to perform like computers in essence that they can learn to do different tasks like computer algorithms allow them to perform different things.

Practical Applications:

- Text Completion
- Language Modelling
- Perform simple algorithmic tasks such as sorting, copying.