SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – IV – DEEP LEARNING – SCSA3015

UNIT 4 OPTIMIZATION AND GENERALIZATION

Optimization in deep learning– Non-convex optimization for deep networks- Stochastic Optimization- Generalization in neural networks- Spatial Transformer Networks- Recurrent networks, LSTM - Recurrent Neural Network Language ModelsWord-Level RNNs & Deep Reinforcement Learning - Computational & Artificial Neuroscience

Optimization in deep learning

Deep learning is the subfield of machine learning which is used to perform complex tasks such as speech recognition, text classification, etc. A deep learning model consists of activation function, input, output, hidden layers, loss function, etc. Any deep learning model tries to generalize the data using an algorithm and tries to make predictions on the unseen data. We need an algorithm that maps the examples of inputs to that of the outputs and an optimization algorithm. An optimization algorithm finds the value of the parameters (weights) that minimize the error when mapping inputs to outputs. These optimization algorithms or optimizers widely affect the accuracy of the deep learning model. They as well as affect the speed training of the model.

While training the deep learning model, we need to modify each epoch‟s weights and minimize the loss function. An optimizer is a function or an algorithm that modifies the attributes of the neural network, such as weights and learning rate. Thus, it helps in reducing the overall loss and improves the accuracy.

Optimization algorithms in deep learning:

1. Gradient Descent
2. Stochastic Gradient Descent
3. Stochastic Gradient descent with momentum
4. Mini-Batch Gradient Descent
5. Adagrad optimizer
6. RMSProp optimizer
7. AdaDelta optimizer
8. Adam optimizer

Hyper parameters of optimization:

Epoch – The number of times the algorithm runs on the whole training dataset.

Sample – A single row of a dataset.

Batch – It denotes the number of samples to be taken to for updating the model parameters.

Learning rate – It is a parameter that provides the model a scale of how much model weights should be updated.

Cost Function/Loss Function – A cost function is used to calculate the cost that is the difference between the predicted value and the actual value.

Weights/ Bias – The learnable parameters in a model that controls the signal between two neurons.

Gradient Descent Deep Learning Optimizer

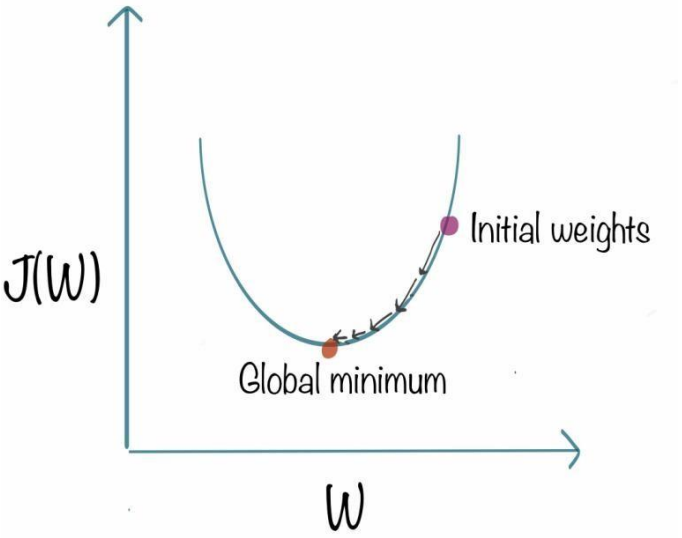
Gradient Descent can be considered as the popular kid among the class of optimizers. This optimization algorithm uses calculus to modify the values consistently and to achieve the local minimum. In simple terms, consider you are holding a ball resting at the top of a bowl. When you lose the ball, it goes along the steepest direction and eventually settles at the bottom of the bowl. A Gradient provides the ball in the steepest direction to reach the local minimum that is the bottom of the bowl.

$$x\_new = x - alpha * f'(x)$$

The above equation means how the gradient is calculated. Here alpha is step size that represents how far to move against each gradient with each iteration.

Gradient descent works as follows:

1. It starts with some coefficients, sees their cost, and searches for cost value lesser than what it is now.
2. It moves towards the lower weight and updates the value of the coefficients.
3. The process repeats until the local minimum is reached. A local minimum is a point beyond which it can not proceed.



Gradient descent works best for most purposes. However, it has some downsides too. It is expensive to calculate the gradients if the size of the data is huge. Gradient descent works well for convex functions but it doesn‟t know how far to travel along the gradient for nonconvex functions.

Mini Batch Gradient Descent Deep Learning Optimizer

In this variant of gradient descent instead of taking all the training data, only a subset of the dataset is used for calculating the loss function. Since we are using a batch of data instead of taking the whole dataset, fewer iterations are needed. That is why the mini-batch gradient descent algorithm is faster than both stochastic gradient descent and batch gradient descent algorithms.

It needs a hyperparameter that is “mini-batch-size”, which needs to be tuned to achieve the required accuracy. Although, the batch size of 32 is considered to be appropriate for almost every case. Also, in some cases, it results in poor final accuracy. Due to this, there needs a rise to look for other alternatives too

Stochastic Gradient Descent Deep Learning Optimizer

The term stochastic means randomness on which the algorithm is based upon. In stochastic gradient descent, instead of taking the whole dataset for each iteration, we randomly select the batches of data. That means we only take few samples from the dataset.

$$w := w - \eta \nabla Q_i(w).$$

The procedure is first to select the initial parameters w and learning rate n. Then randomly

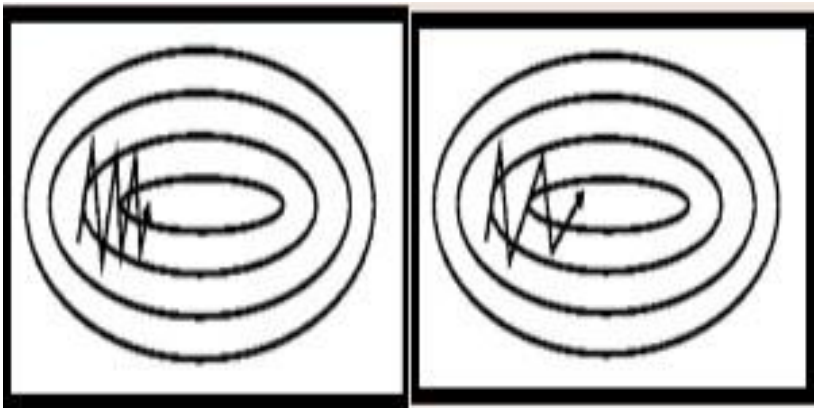
shuffle the data at each iteration to reach an approximate minimum.

Since we are not using the whole dataset but the batches of it for each iteration, the path took by the algorithm is full of noise as compared to the gradient descent algorithm. Thus, SGD uses a higher number of iterations to reach the local minima. Due to an increase in the number of iterations, the overall computation time increases. But even after increasing the number of iterations, the computation cost is still less than that of the gradient descent optimizer. So the conclusion is if the data is enormous and computational time is an essential factor, stochastic gradient descent should be preferred over batch gradient descent algorithm.

Stochastic Gradient Descent with Momentum Deep Learning Optimizer

Stochastic gradient descent takes a much more noisy path than the gradient descent algorithm. Due to this reason, it requires a more significant number of iterations to reach the optimal minimum and hence computation time is very slow. To overcome the problem, we use stochastic gradient descent with a momentum algorithm.

What the momentum does is helps in faster convergence of the loss function. Stochastic gradient descent oscillates between either direction of the gradient and updates the weights accordingly. However, adding a fraction of the previous update to the current update will make the process a bit faster. One thing that should be remembered while using this algorithm is that the learning rate should be decreased with a high momentum term.



In the above image, the left part shows the convergence graph of the stochastic gradient descent algorithm. At the same time, the right side shows SGD with momentum. From the image, you can compare the path chosen by both the algorithms and realize that using momentum helps reach convergence in less time. You might be thinking of using a large momentum and learning rate to make the process even faster. But remember that while

increasing the momentum, the possibility of passing the optimal minimum also increases. This might result in poor accuracy and even more oscillations.

.

Adagrad (Adaptive Gradient Descent) Deep Learning Optimizer

The adaptive gradient descent algorithm is slightly different from other gradient descent algorithms. This is because it uses different learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training. The more the parameters get change, the more minor the learning rate changes. This modification is highly beneficial because real-world datasets contain sparse as well as dense features. So it is unfair to have the same value of learning rate for all the features. The Adagrad algorithm uses the below formula to update the weights. Here the alpha(t) denotes the different learning rates at each iteration, n is a constant, and E is a small positive to avoid division by 0.

$$\mathrm{w_t} = \mathrm{w_{t-1}} - \eta'_{\mathrm{t}} \frac{\partial \mathrm{L}}{\partial \mathrm{w(t-1)}}$$

$$\eta'_{\mathrm{t}} = \frac{\eta}{\mathrm{sqrt}(\alpha_{\mathrm{t}} + \epsilon)}$$

The benefit of using Adagrad is that it abolishes the need to modify the learning rate manually. It is more reliable than gradient descent algorithms and their variants, and it reaches convergence at a higher speed.

One downside of AdaGrad optimizer is that it decreases the learning rate aggressively and monotonically. There might be a point when the learning rate becomes extremely small. This is because the squared gradients in the denominator keep accumulating, and thus the denominator part keeps on increasing. Due to small learning rates, the model eventually becomes unable to acquire more knowledge, and hence the accuracy of the model is compromised.

RMS Prop(Root Mean Square) Deep Learning Optimizer

RMS prop is one of the popular optimizers among deep learning enthusiasts. This is maybe because it hasn‟t been published but still very well know in the community. RMS prop can also be considered an advancement in AdaGrad optimizer as it reduces the monotonically

decreasing learning rate.

The algorithm mainly focuses on accelerating the optimization process by decreasing the number of function evaluations to reach the local minima. The algorithm keeps the moving average of squared gradients for every weight and divides the gradient by the square root of the mean square.

$$v(w,t) := \gamma v(w,t-1) + (1-\gamma)(\nabla Q_i(w))^2$$

where gamma is the forgetting factor. Weights are updated by the below formula

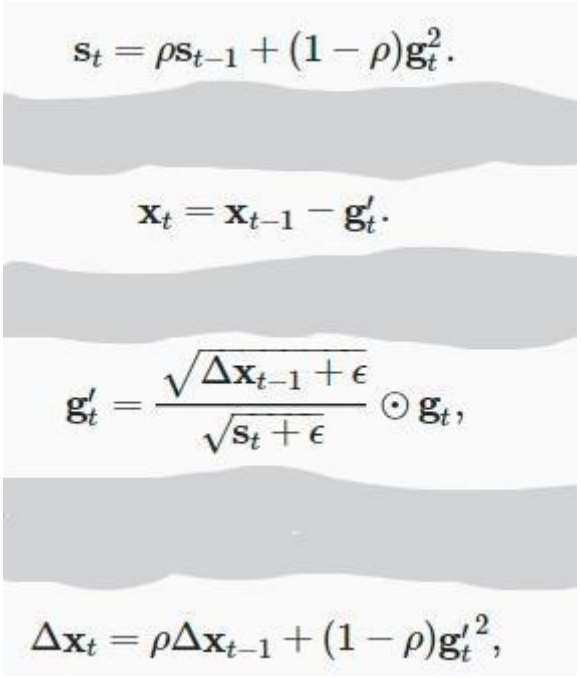
$$w := w - \frac{\eta}{\sqrt{v(w,t)}} \nabla Q_i(w)$$

In simpler terms, if there exists a parameter due to which the cost function oscillates a lot, we want to penalize the update of this parameter. The problem with RMS Prop is that the learning rate has to be defined manually and the suggested value doesn‟t work for every application.

AdaDelta Deep Learning Optimizer

AdaDelta can be seen as a more robust version of AdaGrad optimizer. It is based upon adaptive learning and is designed to deal with significant drawbacks of AdaGrad and RMS prop optimizer. The main problem with the above two optimizers is that the initial learning rate must be defined manually. One other problem is the decaying learning rate which becomes infinitesimally small at some point. Due to which a certain number of iterations later, the model can no longer learn new knowledge.

To deal with these problems, AdaDelta uses two state variables to store the leaky average of the second moment gradient and a leaky average of the second moment of change of parameters in the model.



$$\mathbf{s}_t = \rho \mathbf{s}_{t-1} + (1-\rho)\mathbf{g}_t^2.$$

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \mathbf{g}'_t.$$

$$\mathbf{g}'_t = \frac{\sqrt{\Delta \mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t,$$

$$\Delta \mathbf{x}_t = \rho \Delta \mathbf{x}_{t-1} + (1-\rho)\mathbf{g}'^2_t,$$

Here S_t and delta X_t denotes the state variables, g‟‟$_t$ denotes rescaled gradient, delta X_{t-1} denotes squares rescaled gradients, and epsilon represents a small positive integer to handle division by 0.

Adam Deep Learning Optimizer

The name adam is derived from adaptive moment estimation. This optimization algorithm is a further extension of stochastic gradient descent to update network weights during training. Unlike maintaining a single learning rate through training in SGD, Adam optimizer updates the learning rate for each network weight individually.

The adam optimizer has several benefits, due to which it is used widely. It is adapted as a benchmark for deep learning papers and recommended as a default optimization algorithm. Moreover, the algorithm is straightforward to implement, has faster running time, low memory requirements, and requires less tuning than any other optimization algorithm.

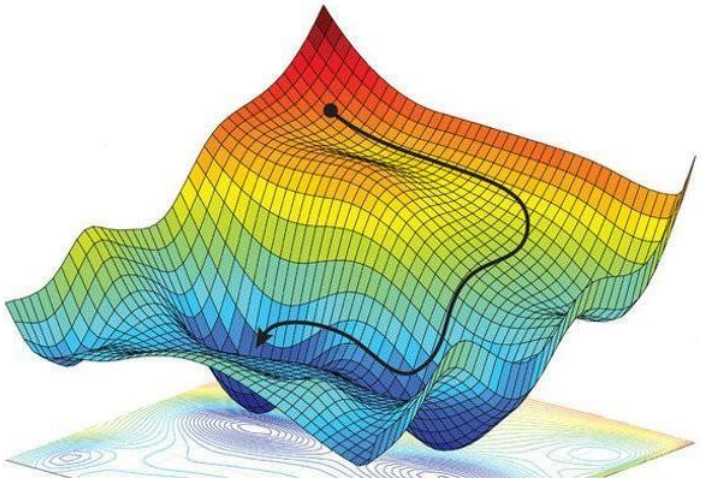
$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\left[\frac{\delta L}{\delta w_t}\right] v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\delta L}{\delta w_t}\right]^2$$

The above formula represents the working of adam optimizer. Here B1 and B2 represent the decay rate of the average of the gradients.

Non-convex optimization for deep networks

Non-Convex Optimization (NCO):

A NCO is a problem where the objective or any of the constraints are non-convex. Even simple looking problems with as few as ten variables can be extremely challenging, while problems with a few hundreds of variables can be intractable. Here, generally, we face the obligation to compromise, i.e. give up seeking the optimal solution, which minimizes the objective over all feasible points. This compromise opens a door for local optimization where intensive research has been conducted and many developments were achieved. As a result, local optimization methods are widely used in applications where there is value in finding a good point, if not the very best. In an engineering design application, for example, local optimization can be used to improve the performance of a design originally obtained by manual, or other, design methods.



Limitations of non-convex optimization:

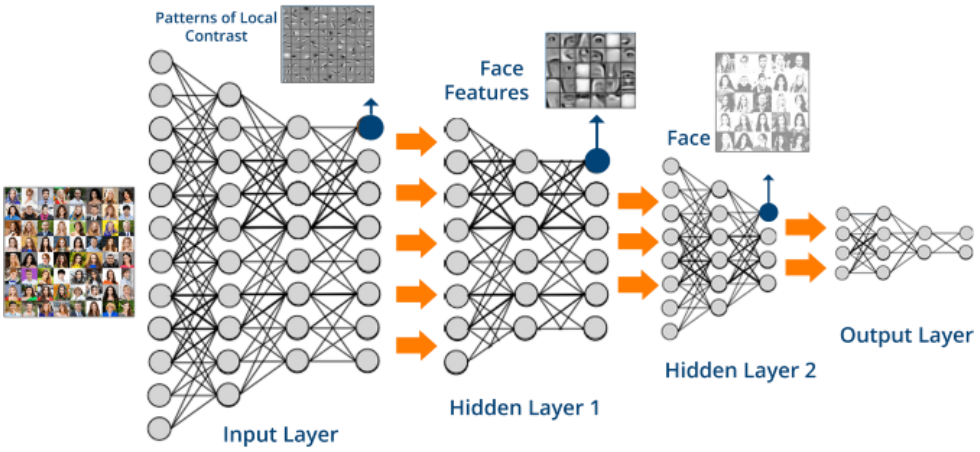
- Optimization problems may have multiple feasible and very flat regions, a widely varying curvature, several saddle points, and multiple local minima within each region.
- It can then take time exponential in the number of variables and constraints to determine that a non-convex problem is infeasible, that the objective function is unbounded, or that an optimal solution is the global optimum across all feasible regions.
- The optimization may also require an initial guess, which is critical and can greatly affect the objective value of the local solution obtained.

Furthermore, little information is provided about how far from the global optimum the local solution is. Beyond this, local optimization methods are often sensitive to algorithm parameter values, which may need to be adjusted for a particular problem, or family of problems. To sum

it up, roughly speaking, local optimization methods, used in non-convex optimization, are more art than technology. In contrast, except some cases on the boundary of what is currently possible, there is little art involved in solving a least-squares problem or a linear program. Consequently, there cannot be a general algorithm to solve efficiently NCO problems in all cases and scenarios. NCP is then at least NP-hard. Weak theoretical guarantees add to the downsides of this type of optimization. However, it has the upside of an endless set of problems that we can try to solve better while learning more theoretical insights and improve the practical implementations.

Necessity of NCO:

NCO existed since the early days of operations research. Still, the topic gained more interest and focus with the emergence of DL in the recent years. In fact, neural networks (NN)are universal function approximators. With enough neurons, they have the ability to approximate any function well. Among the functions to be approximated, non-convex functions are gaining more focus. To approximate them, convex functions cannot be good enough. Hence, the importance of using NCO increased. The freedom to express the learning problem as a non-convex optimization problem gives immense modeling power to the algorithm designer. Despite the guarantees achieved in individual instances, it is still complex to unify a framework of what makes NCO easy to control. On the practical side, conversations between theorists and practitioners can support the identification of what kind of conditions are rational for specific applications, and thus lead to the design of practically motivated algorithms for non-convex optimization with rigorous guarantees.



Non-convex Optimization Convergence:

For NCO, many CO techniques can be used such as stochastic gradient descent (SGD), mini-batching, stochastic variance-reduced gradient (SVRG), and momentum. There are also specialized methods for solving non-convex problems known in operations research such as alternating minimization methods, branch-and-bound methods. But, these methods are not, generally, very popular for ML problems. In the same context, there are varieties of theoretical convergence results including convergence to a stationary point, convergence to a local minimum, local convergence to the global minimum, and global convergence to the global minimum. As an example, let us consider the widely used SGD. Non-convex SGD converges with a slow theoretical rate, but not necessarily to a local minimum, which certainly means it doesn‟t necessarily reach the global optimum. These theoretical insights can be strengthened through the incorporation of stronger conditions and assumptions. With the previous ones, we can prove its convergence to a local minimum with (or without) an explicit rate of convergence. Sometimes, if we start close enough to the global optimum, we can achieve a local convergence to the global optimum. However, it is very expensive in terms of time and applied to specific cases. Beyond that, the global convergence to the global minimum happens when we can converge wherever we initialize. For deep neural networks, it known empirically that it doesn‟t happen.

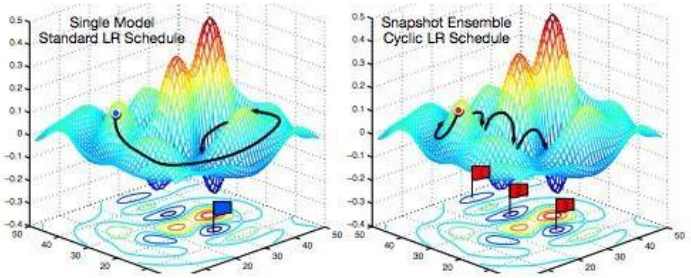
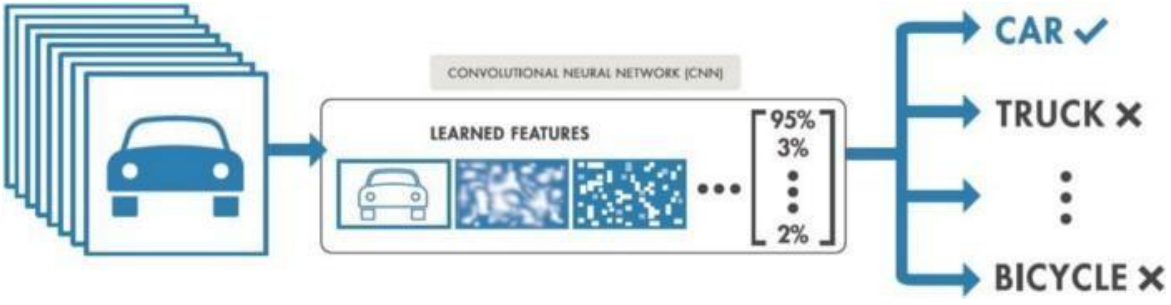


Figure : **Left:** Illustration of SGD optimization with a typical learning rate schedule. The model converges to a minimum at the end of training. **Right:** Illustration of Snapshot Ensembling. The model undergoes several learning rate annealing cycles, converging to and escaping from multiple local minima. We take a snapshot at each minimum for test-time ensembling.

Example:

Because of this, we almost always can‟t prove convergence or anything like that when we run back-propagation SGD on a deep net. In DL as NCO, many things could go wrong. First, convergence to a bad local minimum may happen. I such a case, we can re-optimize the system with different initialization and/or add extra noise to gradient updates. Second, we may face convergence to a saddle point that can be tackled by finding the hessian and computing a descent direction. Third, getting stuck in a region of low gradient magnitude that can be solved using batch norm or designing networks efficiently using a rectified linear unit (ReLU) activation function for instance. Fourth, we may take huge steps and diverging because of the high curvature. In that case, we can use adaptive step size or more intuitively limit the size of the gradient step. Finally, if we end up having a wrong setting of hyper parameters, we can go for hyperparameter optimization methods.



Finally, it seems that the NCO in DL will remain an art unless someone finds a generic framework that can be followed in any problem faced.

Stochastic Gradient Descent (SGD)

SGD tries to solve the main problem in Batch Gradient descent which is the usage of

whole training data to calculate gradients as each step. SGD is stochastic in nature i.e. it picks up a “random” instance of training data at each step and then computes the gradient making it much faster as there is much fewer data to manipulate at a single time, unlike Batch GD.

$$for\ i\ in\ range\ (m):$$

$$\theta_j = \theta_j - \alpha(\widehat{y^i} - y^i)X_j^i$$

The word „stochastic„means a system or process linked with a random probability. Hence, in Stochastic Gradient Descent, a few samples are selected randomly instead of the whole data set for the iteration. In Gradient Descent, there is a term called “batch” which denotes the total number of samples from a dataset that is used for calculating the gradient for the iteration.

Generalization in neural networks

One of the major advantages of neural nets is their ability to generalize. This means that a trained net could classify data from the same class as the learning data that it has never seen before. In real world applications developers normally have only a small part of all possible patterns for the generation of a neural net. To reach the best generalization, the dataset should be split into three parts:

- The **training set** is used to train a neural net. The error of this dataset is minimized during training.
- The **validation set** is used to determine the performance of a neural network on patterns that are not trained during learning.
- A **test set** for finally checking the over all performance of a neural net.

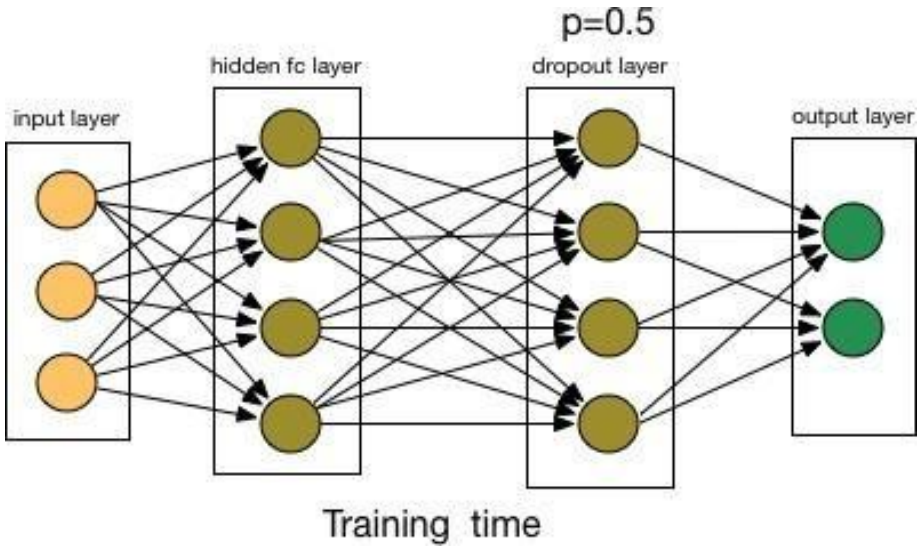
Dropout

This is the one of the most interesting types of generalization technique. It also produces very good results and is consequently the most frequently used regularization technique in the fieldof deep learning.

Dropout is a technique where randomly selected neurons are ignored during training. They are“dropped-out” randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

Dropout may be implemented on any or all hidden layers in the network as well as the visible or input layer. It is not used on the output layer. Dropout regularization is a generic approach. The default interpretation of the dropout hyperparameter is the probability of training a given node in a layer, where 1.0 means no dropout, and 0.0 means no outputs from the layer. A goodvalue for dropout in a hidden layer is between 0.5 and 0.8. Input layers use a larger dropout rate, such as of 0.8.

Dropout is a regularization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data. It is a very efficient way of performing model averaging with neural networks. The term "dropout" refers to dropping out units (both hidden and visible) in a neural network.



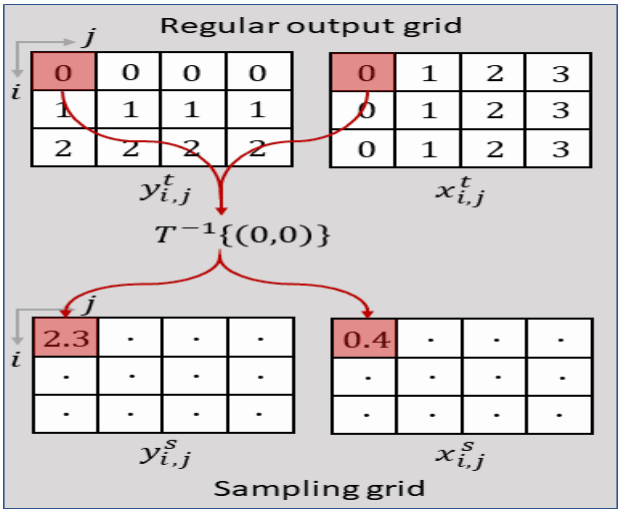
Spatial Transformer Networks

Spatial Transformer modules, introduced by Max Jaderberg et al., are a popular way to increase spatial invariance of a model against spatial transformations such as translation, scaling, rotation, cropping, as well as non-rigid deformations. They can be inserted into existing convolutional architectures: either immediately following the input or in deeper layers.

The main benefit of this approach is that we now get two components with separate responsibilities: grid generator and sampler. The grid generator has the exclusive task of performing the inverse transformation and the sampler has the exclusive task of performing bilinear interpolation.

Grid Generator

The grid generator iterates over the regular grid of the output/target image and uses the inverse transformation T-1{…} to calculate the corresponding (usually non-integer) sample positions in the input/source image:



the superscripts t and s are taken from the original paper and denote “target image” and “source image”. The row and column indexes of the sampling grid are denoted as i and j, respectively. Please also note, that in the original paper the inverse transformation T-1{…} over the regular output grid is denoted as $\nearrow\theta(G)$.

Whereas in the above illustration the coordinates are calculated in a sequential manner for the sake of clarity, real world implementations of the grid generator will try to transform as many points as possible in parallel for reasons of computational efficiency.

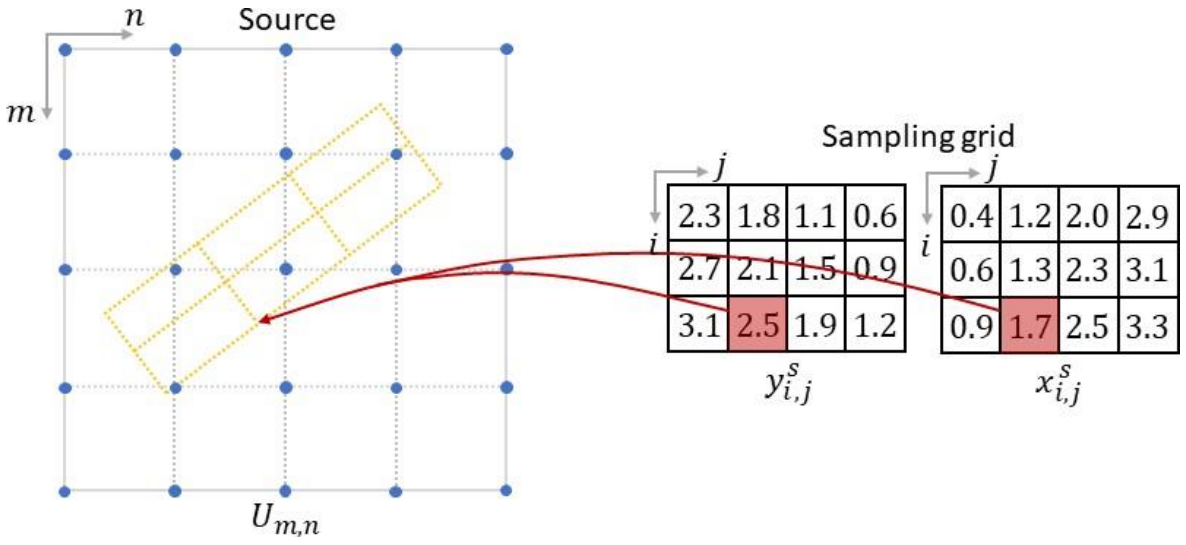
The output of the grid generator is the so called sampling grid, which is a set of points where the input map will be sampled to produce the spatially transformed output:

$$(y_{i,j}^s, x_{i,j}^s)$$

where

$$0 \leq i < H' \quad \text{and} \quad 0 \leq j < W'$$

Please note, that the size of the sampling grid, determines the size of the target image.



Sampler

The sampler iterates over the entries of the sampling grid and extracts the corresponding pixel values from the input map using bilinear interpolation:

The extraction of a pixel value consists of three operations:

1. find the four neighboring points (upper left, upper right, lower left and lower right)
2. for each neighboring point calculate its corresponding weight
3. take the weighted average to produce the output

Localisation Net

The task of the localisation network is to find parameters θ of the inverse transformation T-1{…}, which puts the input feature map to a canonical pose, thus simplify recognition in the following layers. The localisation network can take any form, such as a fully-connected network or a convolutional network, but should include a final regression layer to produce the transformation parameters θ:

The size of θ can vary depending on the transformation that is parameterized, e.g. for an affine transformation θ is 6-dimensional:

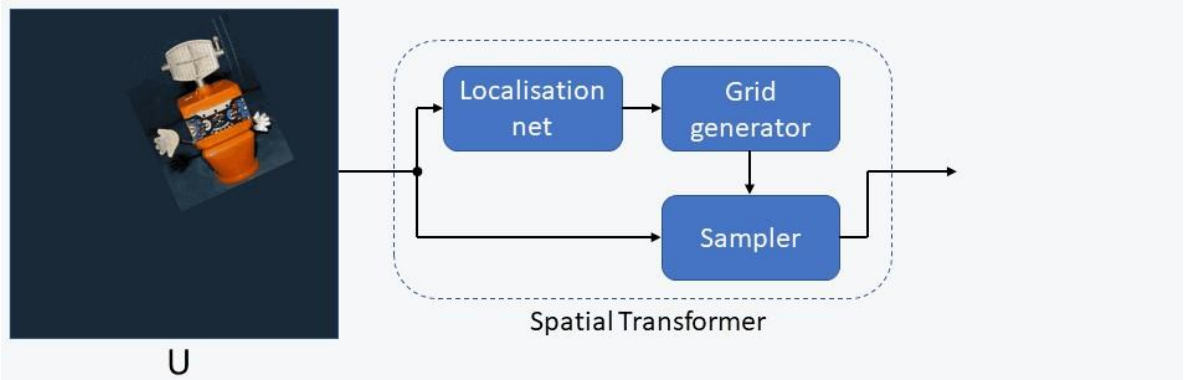
$$\begin{bmatrix} x_{i,j}^s \\ y_{i,j}^s \end{bmatrix} = T^{-1}\left\{\begin{bmatrix} x_{i,j}^t \\ y_{i,j}^t \end{bmatrix}\right\} = \begin{bmatrix} \theta_1 & \theta_2 \\ \theta_3 & \theta_4 \end{bmatrix} \cdot \begin{bmatrix} x_{i,j}^t \\ y_{i,j}^t \end{bmatrix} + \begin{bmatrix} \theta_5 \\ \theta_6 \end{bmatrix}$$

The affine transform is quite powerful and contains translation, scaling, rotation and shearing as special cases. For many tasks however a simpler transformation may be sufficient, e.g. a pure translation is implemented using only 2 parameters:

$$\begin{bmatrix} x_{i,j}^s \\ y_{i,j}^s \end{bmatrix} = T^{-1}\left\{\begin{bmatrix} x_{i,j}^t \\ y_{i,j}^t \end{bmatrix}\right\} = \begin{bmatrix} x_{i,j}^t \\ y_{i,j}^t \end{bmatrix} + \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

Both the grid generator and the sampler are parameter less operations, i.e. they don‟t have any trainable parameters. In this regard they are comparable to a max-pooling layer. The brainpower of a spatial transformer module hence comes from the localisation net, which must learn to detect the pose of the input feature map (such as its orientation, scale etc.) in order to produce an appropriate transformation.

The input feature map U is first passed to the localisation network, which regresses the appropriate transformation parameters θ. The grid generator then uses the transformation parameters θ to produce the sampling grid, which is a set of points where the input feature map shall be sampled. Finally, the sampler takes both the input feature map and the sampling grid and using e.g. bilinear interpolation outputs the transformed feature map.



Recurrent networks (RNN)

RNN works on the principle of saving the output of a particular layer and feeding this back to the input in order to predict the output of the layer.

Below is how you can convert a Feed-Forward Neural Network into a Recurrent Neural Network:

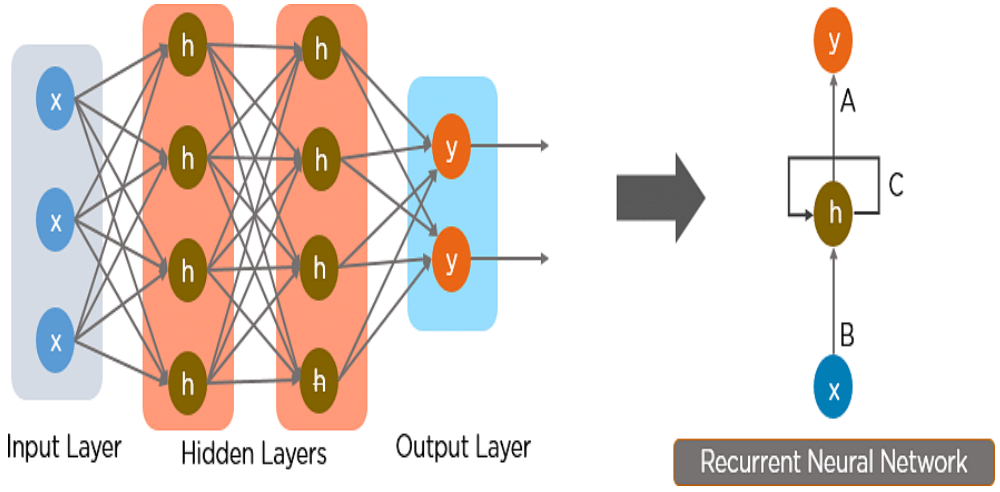


Fig: Simple Recurrent Neural Network

The nodes in different layers of the neural network are compressed to form a single layer of recurrent neural networks. A, B, and C are the parameters of the network.

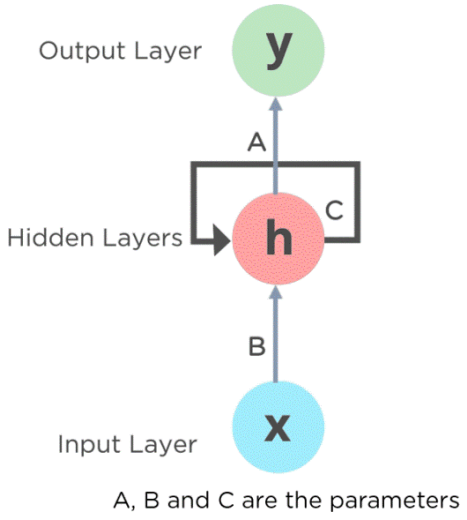


Fig: Fully connected Recurrent Neural Network

Here, "x" is the input layer, "h" is the hidden layer, and "y" is the output layer. A, B, and C are the network parameters used to improve the output of the model. At any given time t, the current input is a combination of input at x(t) and x(t-1). The output at any given time is fetched back to the network to improve on the output.

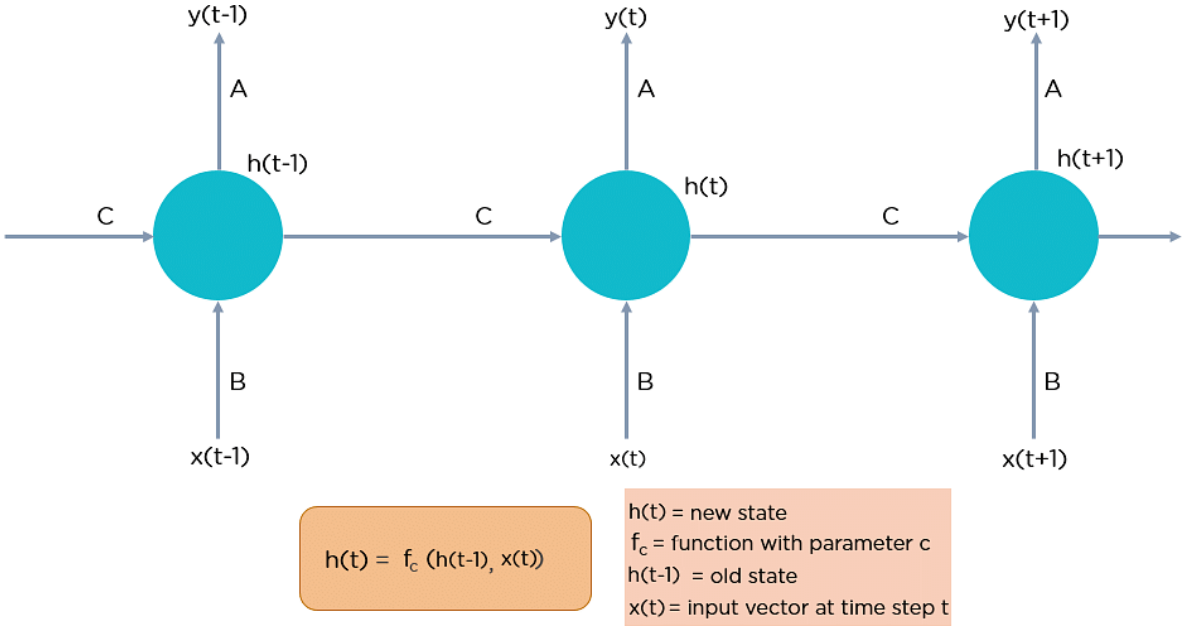


Fig: Fully connected Recurrent Neural Network

Now that you understand what a recurrent neural network is let"s look at the different types of

recurrent neural networks.

RNN were created because there were a few issues in the feed-forward neural network:

- Cannot handle sequential data
- Considers only the current input
- Cannot memorize previous inputs

The solution to these issues is the RNN. An RNN can handle sequential data, accepting the current input data, and previously received inputs. RNNs can memorize previous inputs due to their internal memory.

How Does Recurrent Neural Networks Work?

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

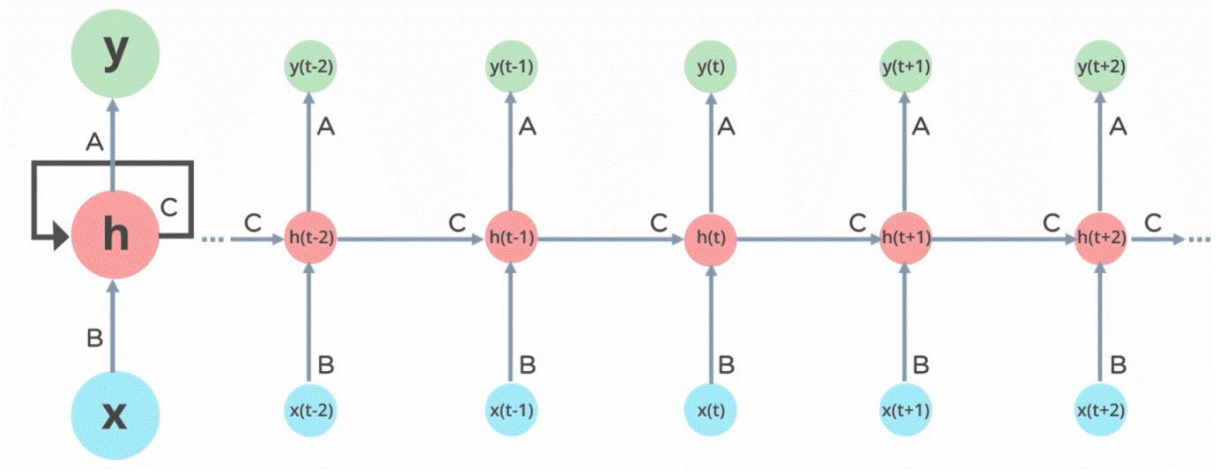


Fig: Working of Recurrent Neural Network

The input layer „x" takes in the input to the neural network and processes it and passes it onto the middle layer.

The middle layer „h" can consist of multiple hidden layers, each with its own activation functions and weights and biases. If you have a neural network where the various parameters

of different hidden layers are not affected by the previous layer, ie: the neural network does not have memory, then you can use a recurrent neural network.

The Recurrent Neural Network will standardize the different activation functions and weights and biases so that each hidden layer has the same parameters. Then, instead of creating multiple hidden layers, it will create one and loop over it as many times as required.

Feed-Forward Neural Networks vs Recurrent Neural Networks

A feed-forward neural network allows information to flow only in the forward direction, from the input nodes, through the hidden layers, and to the output nodes. There are no cycles or loops in the network.

Below is how a simplified presentation of a feed-forward neural network looks like:

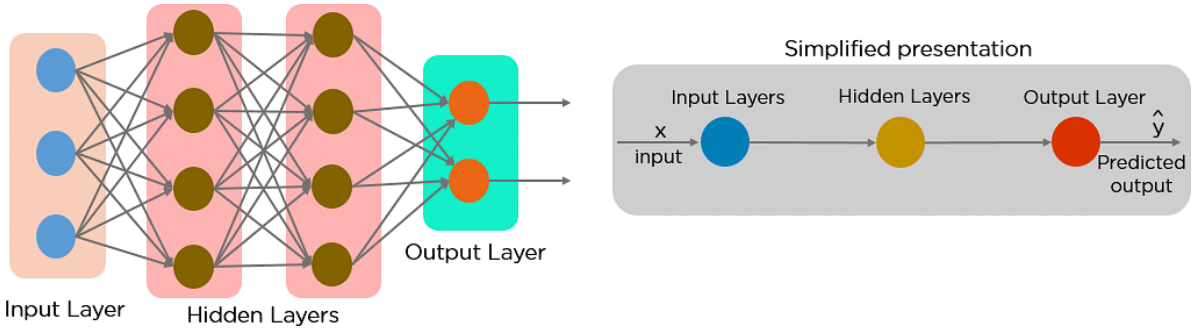


Fig: Feed-forward Neural Network

In a feed-forward neural network, the decisions are based on the current input. It doesn‟t memorize the past data, and there‟s no future scope. Feed-forward neural networks are used in general regression and classification problems.

Applications of Recurrent Neural Networks

- Image Captioning
- Time Series Prediction
- Any time series problem, like predicting the prices of stocks in a particular month, can be solved using an RNN.

- Natural Language Processing
- Text mining and Sentiment analysis can be carried out using an RNN for Natural Language Processing (NLP).
- Machine Translation

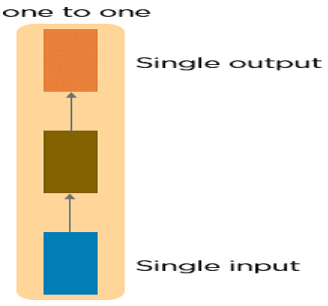
Types of Recurrent Neural Networks

There are four types of Recurrent Neural Networks:

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

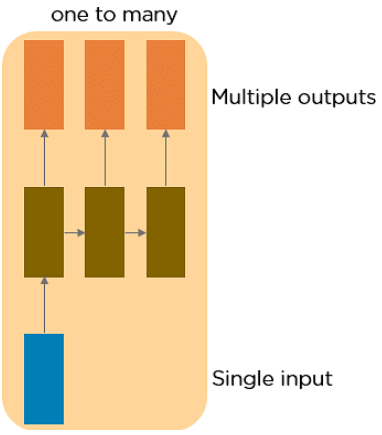
One to One RNN

This type of neural network is known as the Vanilla Neural Network. It's used for general machine learning problems, which has a single input and a single output.



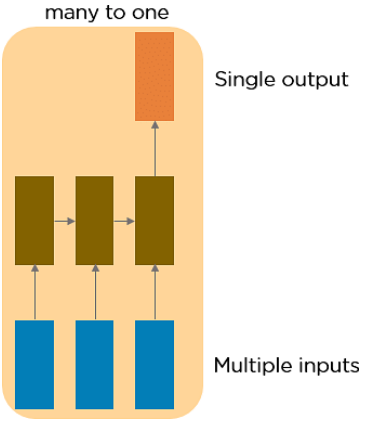
One to Many RNN

This type of neural network has a single input and multiple outputs. An example of this is the image caption.



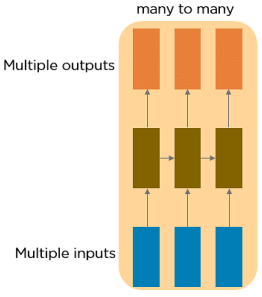
Many to One RNN

This RNN takes a sequence of inputs and generates a single output. Sentiment analysis is a good example of this kind of network where a given sentence can be classified as expressing positive or negative sentiments.



Many to Many RNN

This RNN takes a sequence of inputs and generates a sequence of outputs. Machine translation is one of the examples.

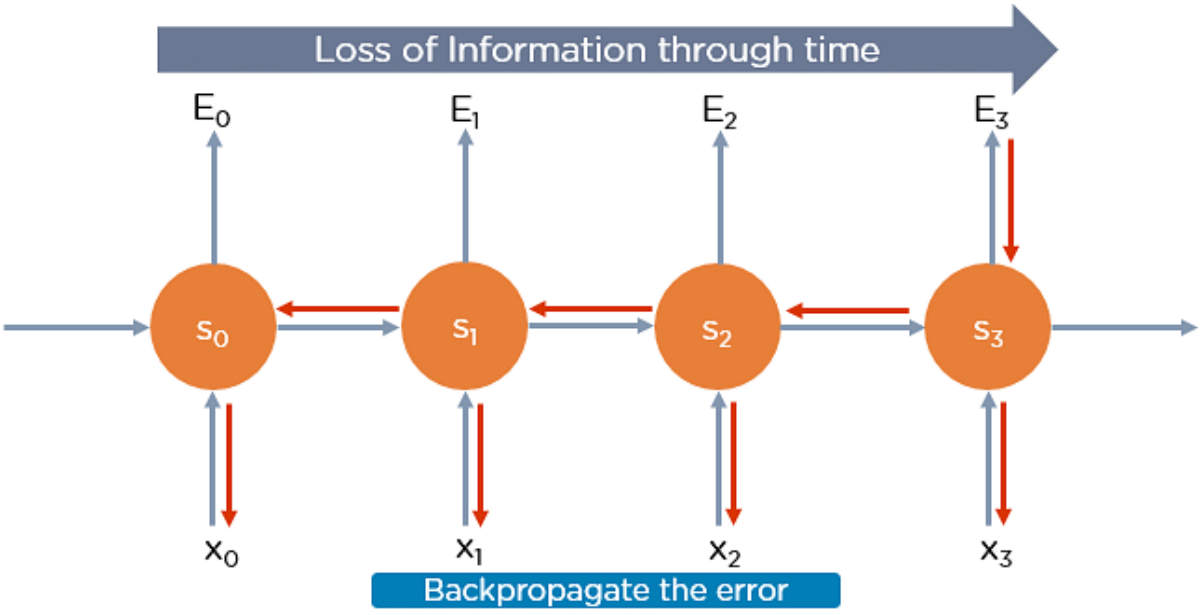


Two Issues of Standard RNNs

1. Vanishing Gradient Problem

Recurrent Neural Networks enable you to model time-dependent and sequential data problems, such as stock market prediction, machine translation, and text generation. You will find, however, RNN is hard to train because of the gradient problem.

RNNs suffer from the problem of vanishing gradients. The gradients carry information used in the RNN, and when the gradient becomes too small, the parameter updates become insignificant. This makes the learning of long data sequences difficult.



2. Exploding Gradient Problem

While training a neural network, if the slope tends to grow exponentially instead of decaying, this is called an Exploding Gradient. This problem arises when large error gradients accumulate, resulting in very large updates to the neural network model weights during the training process.

Long training time, poor performance, and bad accuracy are the major issues in gradient problems

LSTM

Long Short Term Memory Network is an advanced RNN, a sequential network, that allows information to persist. It is capable of handling the vanishing gradient problem faced by RNN. A recurrent neural network is also known as RNN is used for persistent memory.

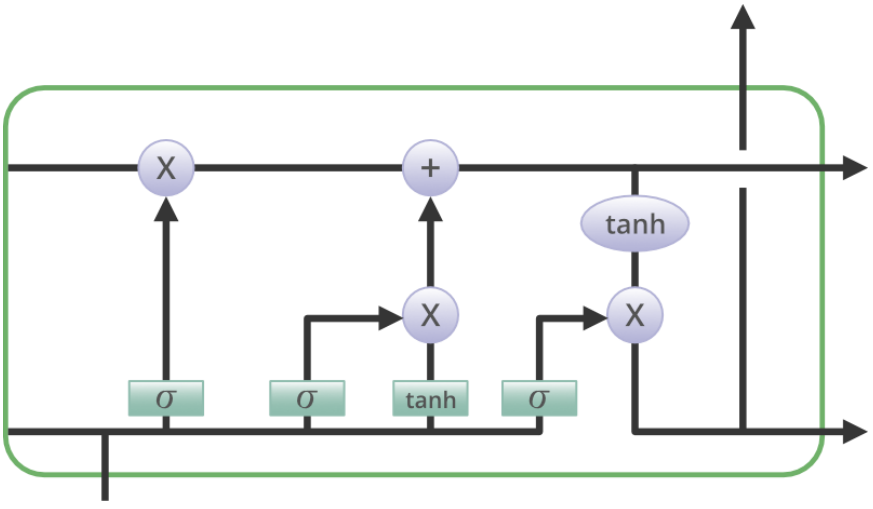
- LSTM is a special kind of recurrent neural network capable of handling long-term dependencies.
- Understand the architecture and working of an LSTM network

Long Short Term Memory Network is an advanced RNN, a sequential network that allows information to persist. It is capable of handling the vanishing gradient problem faced by RNN. A recurrent neural network is also known as RNN is used for persistent memory.

At a high-level LSTM works very much like an RNN cell. Here is the internal functioning of the LSTM network. The LSTM consists of three parts, as shown in the image below and each part performs an individual function.

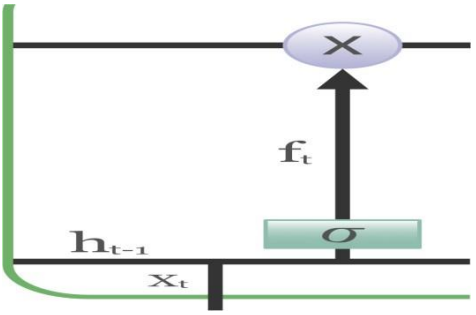
Structure of LSTM:

LSTM has a chain structure that contains four neural networks and different memory blocks called **cells**.

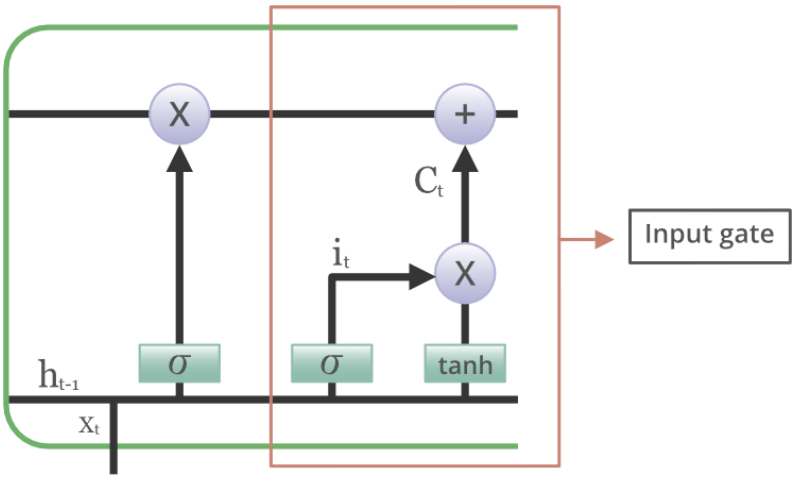


Information is retained by the cells and the memory manipulations are done by the **gates.** There are three gates –

1. Forget Gate: The information that is no longer useful in the cell state is removed with the forget gate. Two inputs *x_t* (input at the particular time) and *h_t-1* (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through an activation function which gives a binary output. If for a particular cell state the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use.

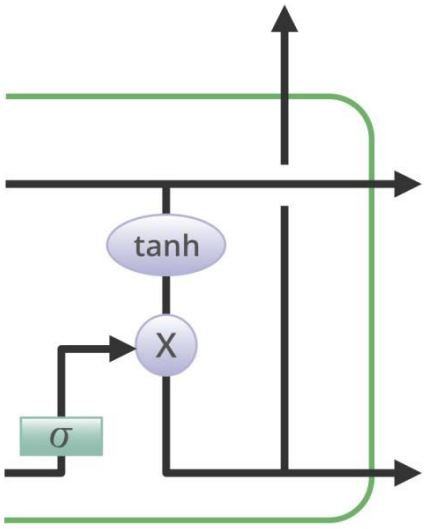


2. Input gate: The addition of useful information to the cell state is done by the input gate. First, the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs *h_t-1* and *x_t*. Then, a vector is created using *tanh* function that gives an output from -1 to +1, which contains all the possible values from h_t-1 and *x_t*. At last, the values of the vector and the regulated values are multiplied to obtain the useful information



3. Output gate: The task of extracting useful information from the current cell state to be presented as output is done by the output gate. First, a vector is generated by applying tanh function on the cell. Then, the information is regulated using the sigmoid function and filter

by the values to be remembered using inputs *h_t-1* and *x_t*. At last, the values of the vector and the regulated values are multiplied to be sent as an output and input to the next cell.



Some of the famous applications of LSTM includes:

1. Language Modelling
2. Machine Translation
3. Image Captioning
4. Handwriting generation
5. Question Answering Chatbots

Deep Reinforcement Learning

- Reinforcement Learning directly takes inspiration from how human beings learn from data in their lives. It features an algorithm that improves upon itself and learns from new situations using a trial-and-error method. Favourable outputs are encouraged or „reinforced", and non-favourable outputs are discouraged or „punished".
- In every iteration of the algorithm, the output result is given to the interpreter, which decides whether the outcome is favourable or not.
- In case of the program finding the correct solution, the interpreter reinforces the solution by providing a reward to the algorithm. If the outcome is not favourable, the algorithm is forced to reiterate until it finds a better result. In most cases, the reward system is directly tied to the effectiveness of the result.
- In typical reinforcement learning use-cases, such as finding the shortest route between two points on a map, the solution is not an absolute value. Instead, it takes on a score

of effectiveness, expressed in a percentage value. The higher this percentage value is, the more reward is given to the algorithm.

- Thus, the program is trained to give the best possible solution for the best possible reward.

Types of Reinforcement learning

There are mainly two types of reinforcement learning, which are:

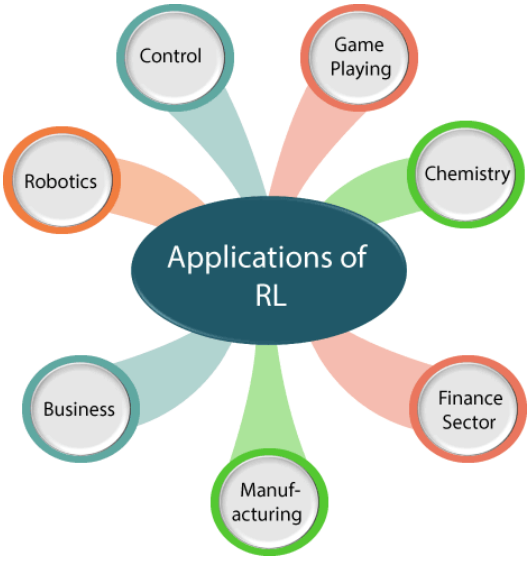
- **Positive Reinforcement**

The positive reinforcement learning means adding something to increase the tendency that expected behaviour would occur again. It impacts positively on the behaviour of the agent and increases the strength of the behaviour. This type of reinforcement can sustain the changes for a long time, but too much positive reinforcement may lead to an overload of states that can reduce the consequences.

- **Negative Reinforcement:**

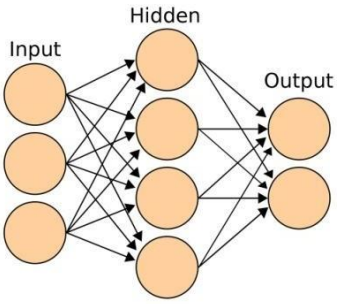
The negative reinforcement learning is opposite to the positive reinforcement as it increases the tendency that the specific behaviour will occur again by avoiding the negative condition.It can be more effective than the positive reinforcement depending on situation and behaviour, but it provides reinforcement only to meet minimum behaviour.

Reinforcement Learning Applications

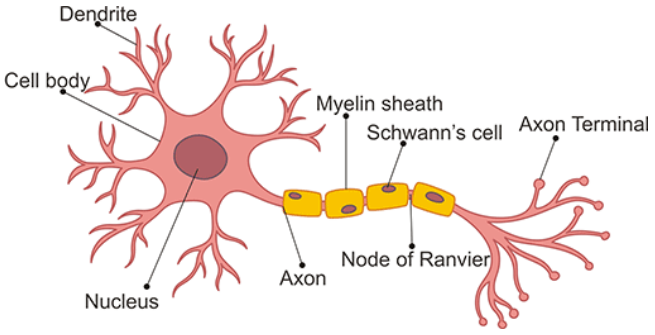


Computational & Artificial Neuroscience

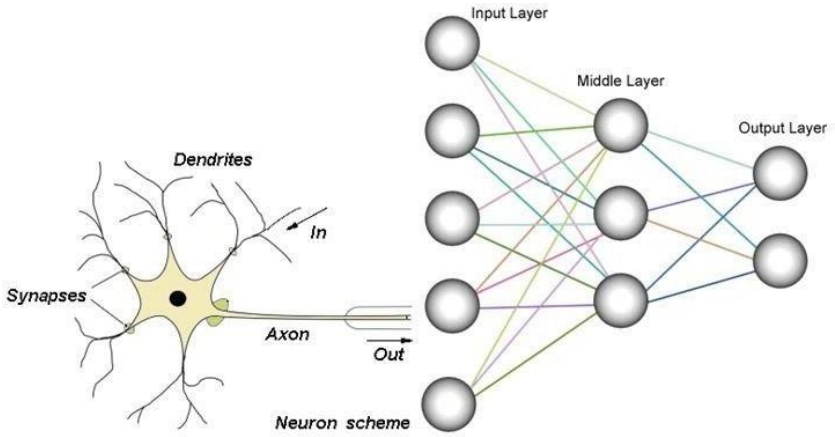
A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain.



Neuroscience examines the structure and function of the human brain and nervous system. Neuroscientists use cellular and molecular biology, anatomy and physiology, human behavior and cognition, and other disciplines, to map the brain at a mechanistic level.



The collaboration between artificial intelligence and neuroscience can produce an understanding of the mechanisms in the brain that generate human cognition. This article reviews multidisciplinary research lines that could achieve this understanding. Artificial intelligence has an important role to play in research, because artificial intelligence focuses on the mechanisms that generate intelligence and cognition.



Questions to revise:

Part A:

1. Establish the difference between LSTM and RNN?
2. Define Non-convex optimization.
3. Illustrate the architecture of RNN?
4. What are the gates and the uses of optimization problem?
5. Define Spatial Transformer Networks?
6. Define Reinforcement Neural Network?
7. Identify the Recurrent Neural Network Language Models?
8. Define Artificial Neuroscience and its application?

Part B:

1. Sketch and explain in detail about Recurrent Neural Network / RNN Language models?
2. Sketch and explain in detail about the Architecture of LSTM?
3. Summarize the methods of Optimization in deep learning.
4. Discover the applications of Deep Reinforcement Learning with a real time example.