TOC

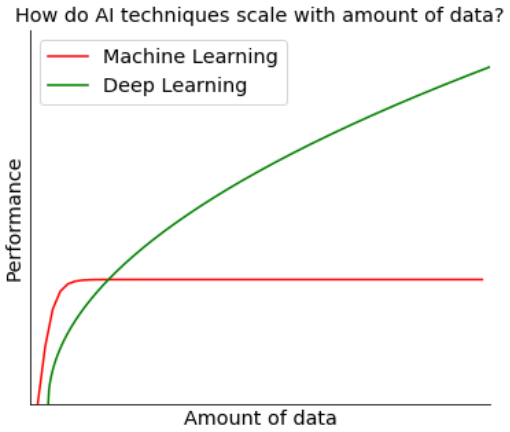
Deep learning basics: Why

The starting of 21st century has introduced us with data as the new fuel to drive today's society. The dependency of beings on electronic devices have been seen to increase at an exploding rate. With the availability of such humongous data generated every second along with the ample computational resources, many complex tasks have been simplified. The creation of deep learning architecture is one such major accomplishment.

The concept of deep learning builds (as a subset) on classical machine learning approaches and proves worthy over them when few constraints are satisfied –

1. Enormous data is available.
2. Enormous computational power is available.
3. Interpretability of model is not desired.

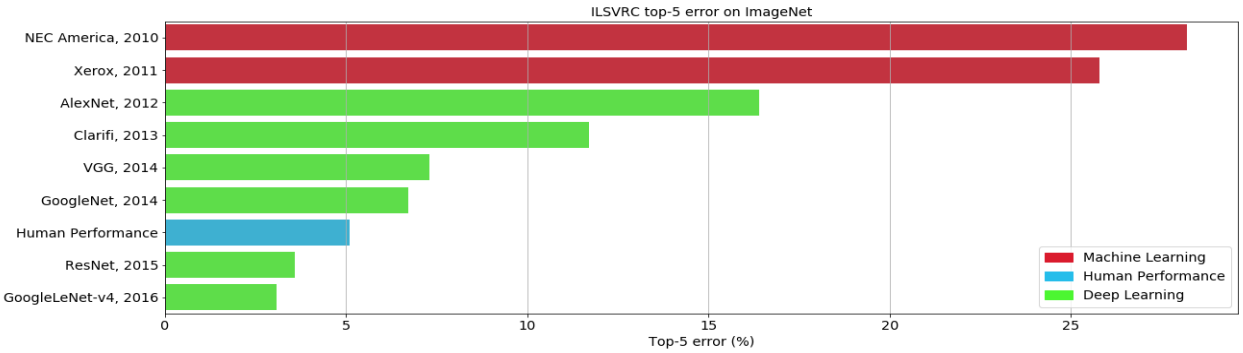
Some of the most advanced domains say, computer vision, natural language processing, speech recognition etc. have grown significantly better using deep learning architectures. Their performance is far better than classical machine learning architectures and in some cases competitive to human experts. With the increase in amount of data ML algorithms tends to follow a plateau structure, however DL algorithms still improves in performance.



[ref] [Graph code provided]

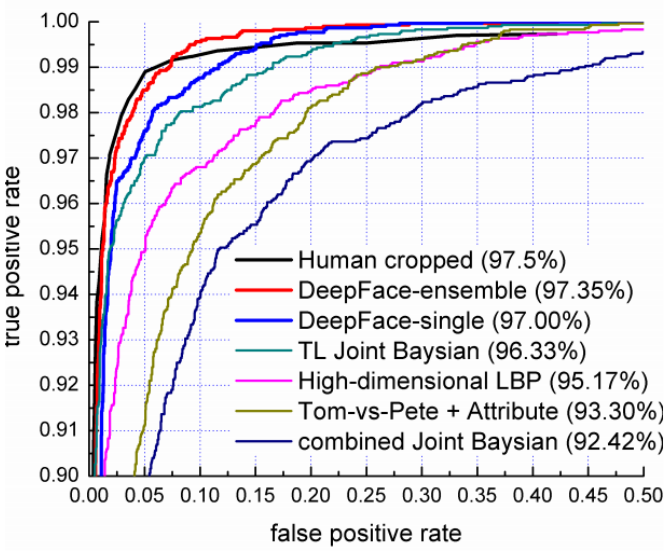
Given below are few of the real-world cases in particular to computer vision and natural language processing in which deep learning architectures have an edge over classical architectures.

1. The ImageNet Large Scale Visual Recognition Challenge (**ILSVRC**) evaluates algorithms for object detection and image classification at large scale. Given below is a comparison plot among classical architectures and deep learning architectures which clearly shows the upper hand of deep learning architectures.



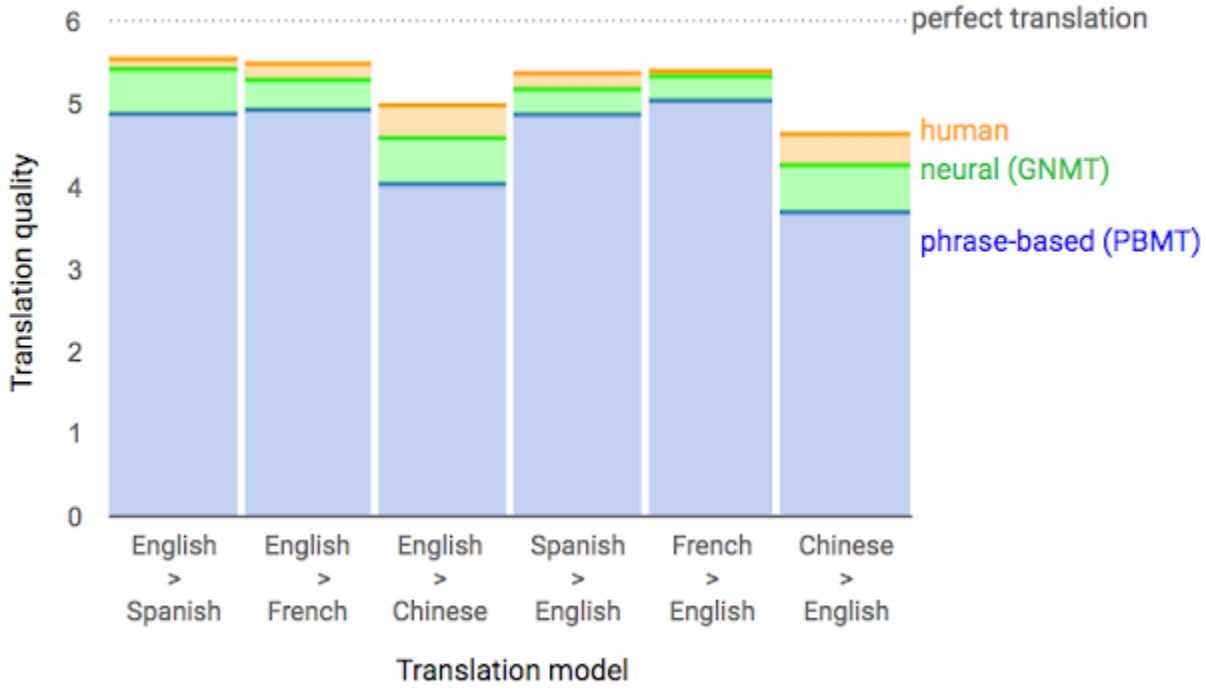
[Graph code provided] [remove grid lines above the bars]

2. The **LFW** dataset consists of 13,323 web photos of 5,749 celebrities which are divided into 6,000 face pairs in 10 splits. DeepFace (a deep learning architecture developed by Facebook) gives an accuracy of 97.35% on LFW dataset closely approaching to human-level performance as seen in given ROC curve.



[Ref] [Copied, need to be remade]

3. Google's Neural Machine Translation System (an addition to Google Translate) utilizes deep LSTM networks with 8 encoder and 8 decoder layers to provide the largest improvement over Phrase Based Machine Translation (key starter algorithm in Google Translate) as depicted in graph below.

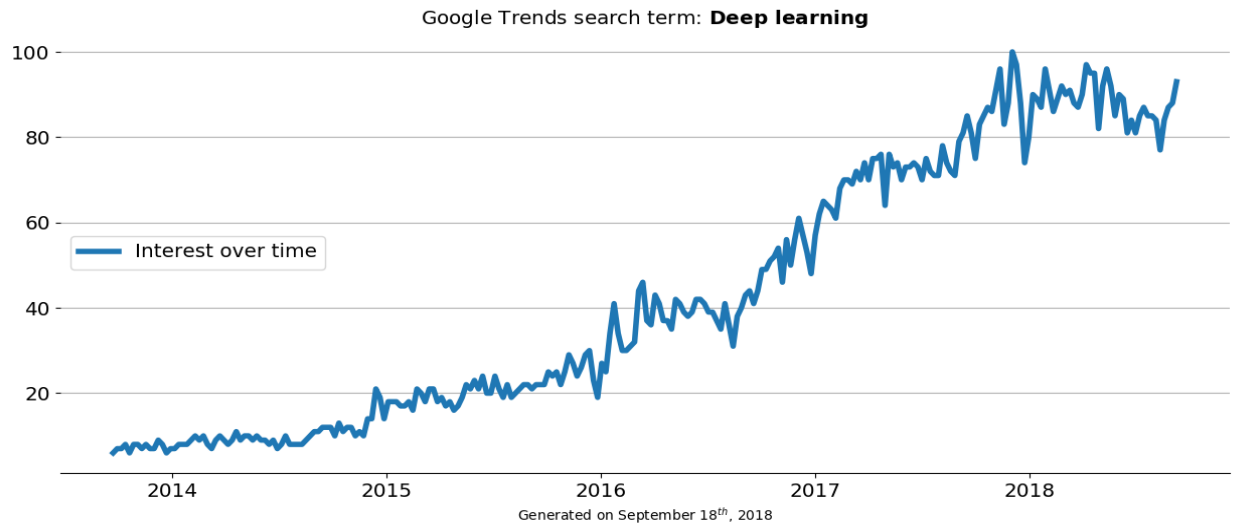


[Ref] [Copied, need to be remade]

As per above major examples, we have seen that deep learning has an edge over machine learning and even competes to human-level performance. So, let us now understand the basics of deep learning.

Deep learning: What

The term "Deep Learning" has gained popularity quite recently due to the availability of higher computational resources and ample open-source data. The same can be confirmed by the trend in the search term "deep learning" over the past 4 years as shown by Google Trends.



[Graph code provided]

Quoting the definition of deep learning from a paper published in Nature titled simply "Deep Learning" by Yann LeCun, Yoshua Bengio and Geoffrey Hinton –

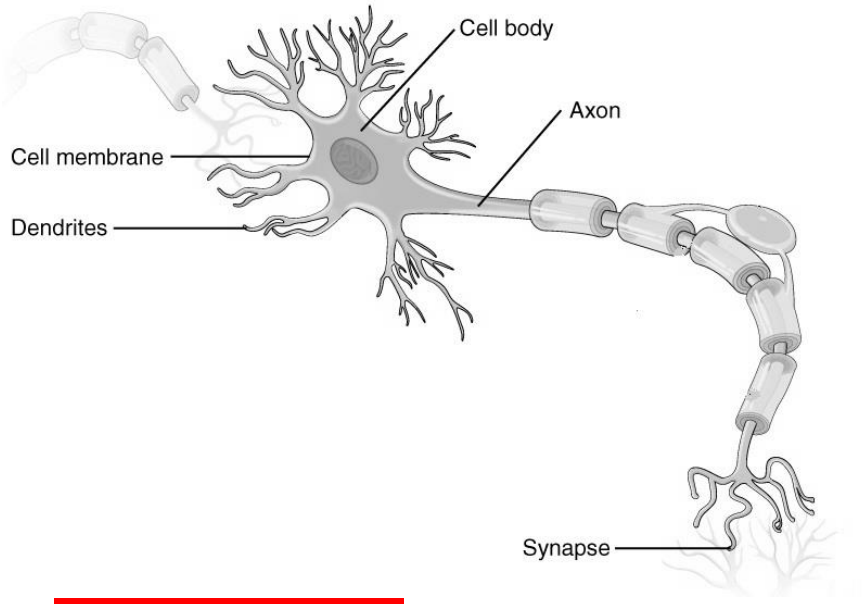
"*Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction.*"[ref]

Given below is a brief history of deep learning [ref] –

Year	Milestone achieved	Word done
1960s	Shallow neural networks	McCulloch-Pitts Neuron, The Perceptron
1960-70s	Backpropagation emerges	Backpropagation, Deep feedforward multilayer perceptron
1979-80	Convolution emerges	Neocognitron [ref]
1982	Recurrent emerges	Hopfield networks
1990s-2000s	Supervised deep learning	LSTM, Gradient-based learning
2006s-present	Modern deep learning includes unsupervised learning	AlexNet, Auto-encoders, GAN

Deep Learning: Feed-forward networks

To develop a computational model one requires to store, transmit and process data. These minimalistic requirements can be fulfilled with a biological neuron. A biological neuron can have various components like mitochondria, nucleus, myelin sheath, dendrites, etc. However, to build a computational model we adopt only three neuron components viz. dendrites, cell body and axon.

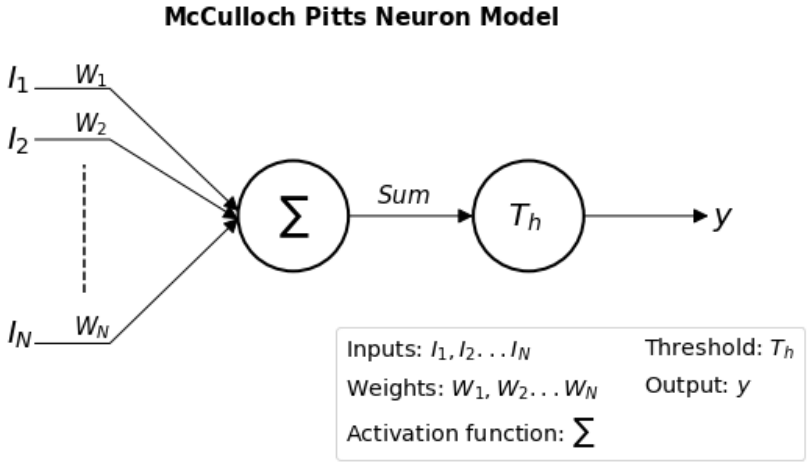


[ref] [Copied, need to be remade]

A biological neuron acts as a computational model in following manner –

- Each dendrite is connected to a distinct neuron to receive data and pass on to its cell body.
- Cell body does the necessary data processing and pass the information to a single cylindrical shaped component named as axon.
- At the end of axon lies synapses which stores information and transmits information via synaptic gap. Synapses also determines the direction for the transmission of information.
- Each synapse is associated with a weight, when all the synapses are activated at same time the total information will be the summation of all the weights which if crosses a particular threshold fires a signal else hinder the signal.

The first implementation of artificial neuron model was given by McCulloch and Pitts. It consists of a set of inputs I1, I2 … IN with associated similar normalized positive weights W1, W2 … WN. The activation function generates a weighted sum $\sum_{i=1}^{N} I_i W_i$ which is forwarded to a binary threshold gate. This structure has limitations like fixed weights and threshold with only binary output. However, it was the first one to mimic a biological neuron as shown in diagram below.



[Graph code provided]

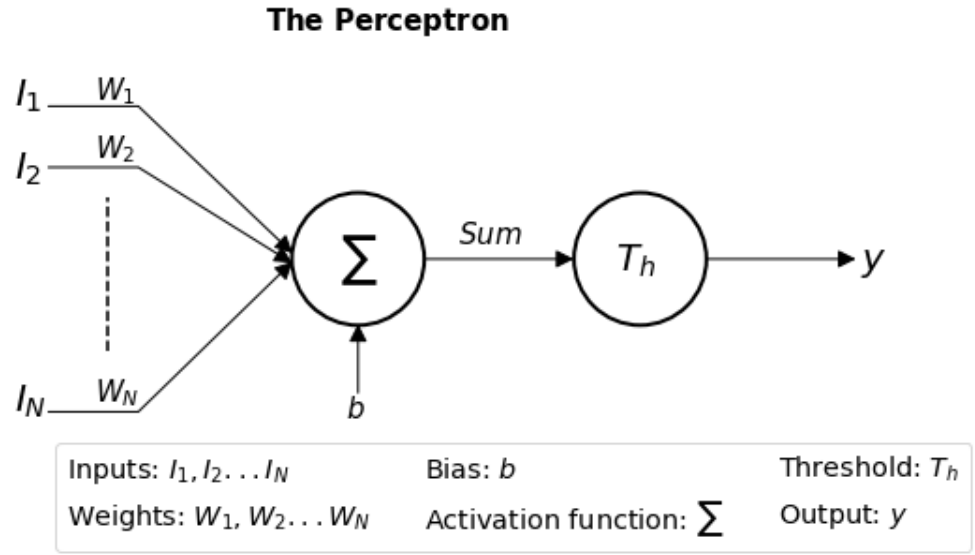
To improve upon McCulloch-Pitts neuron model, Frank Rosenblatt introduced the concept of a *Perceptron* which has variable normalized weights (both negative and positive) as well as holds a new component named as *bias*. Additionally, now model follows a learning algorithm by updating previous values of weights and bias using error between the original and calculated output as shown [ref].

$$error_N = target\ output_N - resultant\ output_N$$
$$\Delta weight_N = \epsilon * (error_N * resultant\ output_N) * I_N$$
$$\Delta bias = \epsilon * error_N * old\ bias$$

Where $0 \leq \epsilon < 1$is the learning rate.
Sigmoidal function is chosen as a part of threshold function due to its continuous range of output.

The structure of new model with modified equation is given below –

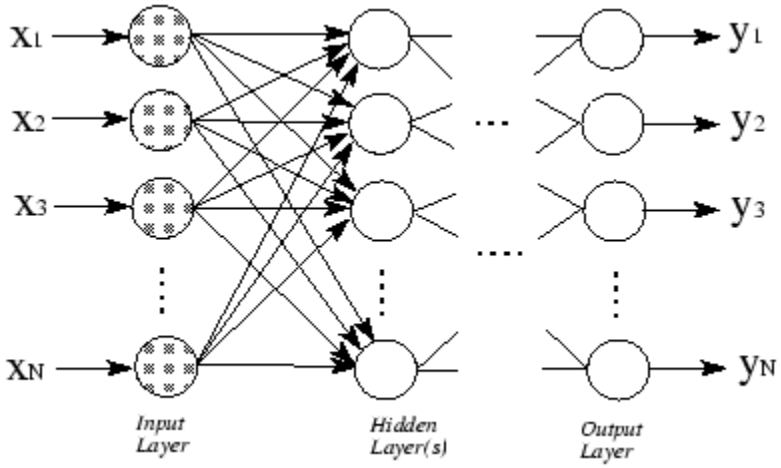
$$Sum = \sum_{i=1}^{N} I_i W_i + b$$



[Graph code provided]

[**Go to**: Python implementation of Perceptron] [rosenblatt_perceptron.ipynb]

To achieve higher level of computational capabilities a cascaded layer network of multiple neurons is formed which is coined as Artificial Neural Network (ANN) as shown below. The first layer is termed as input layer which receives respective inputs, followed with multiple optional hidden layers, ending with output layer.



Deep learning: Activation function

An advantage of perceptron model over McCulloch-Pitts model is sigmoid acting as a threshold function. Following are the reasons why sigmoid is chosen over dis-continuous linear threshold function –

- Sigmoid is a smooth/**continuous** function which means that error rate will also follow a continuous function adjusting the weights in an optimal manner and hence minimizing the loss function.
- To use the backpropagation learning method (to be discussed later), the activation function should be continuously **differentiable** so that the range of the activation function is finite.
- An activation function needs to be **monotonic** e.g. sigmoid, using a non-monotonic activation function may cause that increasing the neuron's weight affect less on the neurons in next layer and model might not converge to the solution.

However, researchers (LeCun et al., 1998b) found that logistic sigmoid function slows down learning due to its non-zero mean and hence suggested to use an anti-symmetric activation function like *tanh* given as:

$$f(x) = \text{atanh}(bx)$$

in a range of $[-a, a]$, where$a = 1.7159$ and $b = 2/3$, $\tanh(x) = 2sigmoid(2x) - 1$

Recently, researchers (Glorot et al., 2011) demonstrated the use of rectified linear unit (ReLU) over sigmoidal and tanh activation function due to following reasons –

- ReLU is quite faster and much more efficient in learning for high-dimensional data.
- It doesn't require intensive computation as it computes the function $f(x) = \max(0, x)$ which simply thresholds input matrix to zero.
- It doesn't saturate i.e. no vanishing or exploding gradient and hence suitable for deep networks (multiple cascaded layers).

On the other side, ReLU is also responsible for creating dying neurons (a dying neuron is the one which never fires). Dying neuron are generated when no gradient flows backward through ReLU. There are two solutions to mitigate this problem –

- Assign small learning rate.
- Use leaky ReLU which allows a small negative slope when the unit is not active.

[**Go to**: Python implementation of MLP] [MLP_classification_keras_fashion_mnist.ipynb]
[**Go to**: Python implementation of T-SNE] [T-SNE Visualization.ipynb]

Deep learning: Backpropagation and optimization methods

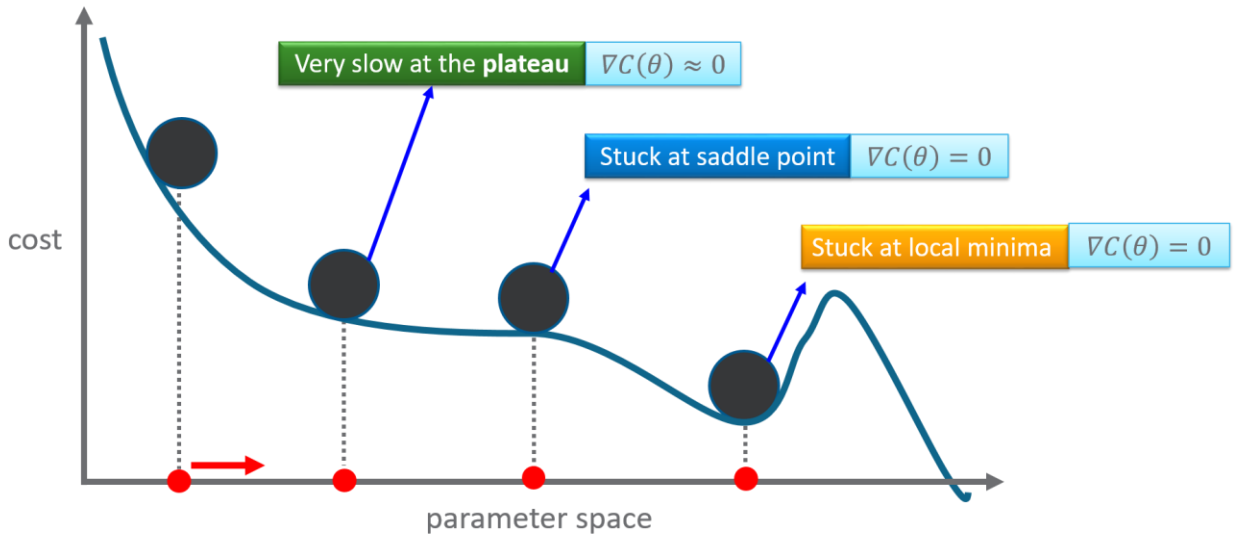
For a feed-forward network with inputs x_i, weights w_i and target output y_i, the information flows from input layer to output layer via hidden layers to produce an output $\hat{y}_i$. During forward propagation, the dot product is calculated between inputs and their corresponding weights followed with the introduction of non-linearity via activation function. This process generates a loss function which needs to be minimized to bring actual output as closer to target output. Therefore, this information is propagated backward through the network in order to compute the gradient to update weights and biases (i.e. model's parameters). This whole process is termed as backpropagation.

For a gradient, arriving at optimal model parameters' value leads to following challenges –

- Slowing down on plateau.
- Getting stuck on a saddle point[1].
- Getting stuck on a local minimum.

[1] A saddle point is a point on the surface where the derivative in orthogonal directions are both zero.
Note: A step by step backpropagation example, Visual introduction to neural network using backpropagation.

These problems can be visualized through given graph –



To tackle challenges for a gradient, various optimization algorithms are prescribed.

Optimization algorithms –

Optimization algorithms are divided into two categories –

1. **First order** – The objective here is to minimize/ maximize the loss function by calculating the first order gradient/derivative with respect to the model's parameters. It provides a tangential line to its error surface marking if the gradient is increasing/decreasing.
2. **Second order** – The objective here is to minimize/maximize the loss function by calculating the second order derivative (also called as Hessian) with respect to the model's parameters. It provides a quadratic surface which touches the curvature of error surface. It represents the increase/decrease of first gradient. Second order optimization algorithms provide better performance over first order on the cost of higher computation. Example, L-BFGS.

The traditional gradient descent algorithm falls under the category of first order optimization algorithm. Under first order optimization algorithms, there are two sub-categories –

1. Algorithms with gradient descent variants.
    a. Stochastic Gradient Descent (SGD)

    b. Mini-batch
    c. Momentum
    d. Nesterov Accelerated Gradient (NAG)
2. Algorithms with adaptive learning rates.
    a. Adagrad
    b. Adadelta
    c. RMSprop
    d. Adaptive Movement estimation (Adam)

Given below is a short summary of advantages and challenges faced by each of the above listed algorithms:

Algorithms with gradient descent variants

For a very large dataset, the performance of traditional batch gradient descent algorithm degrades due to a single update for complete dataset, plus computing redundant updates.

Stochastic Gradient Descent (SGD) performs parameter updates for each training instant one at a time, thus reducing the limitation of redundant updates by batch gradient descent. Also, the computation time per update is not dependent on training data size. Hence, convergence can take place quite early relative to gradient descent even with large dataset. SGD performs frequent parameter updates with high variance and thus fluctuating loss function. This property aids with a possibility of finding better local minimum but also makes convergence more difficult to achieve.

Mini-batch Gradient Descent takes advantage from both the parent gradient variants (i.e. batch gradient descent and SGD) by forming mini-batches, calculating error and updating parameters on each mini-batches. Depending upon the implementation (sum the gradient of mini-batches or average of gradient of mini-batches), it helps in reducing the variance for parameter updates. Due to batches, large datasets can be processed more efficiently. However, it suffers from challenges like getting stuck on saddle points and choosing right learning rate.

Momentum is built to accelerate the progress of SGD when stuck in a local minimum by introducing the concept of velocity. With the introduction of velocity, SGD progresses in a relevant direction by reducing the oscillation from irrelevant directions. Moreover, the step size is now governed by number of successive gradients pointing in exactly the same direction. With updates taking place for only relevant instances, the convergence happens quickly. The challenge for momentum arrives when it reaches a minimum and skips it by moving upward due to high velocity. Also, it introduced a new hyper-parameter, momentum.

Nesterov Accelerated Gradient (NAG) overcomes the challenge of momentum by linking a foresight for next update. It starts with a high step in the direction of the previously accumulated gradient, measuring the gradient and then making a correction, resulting in an update. This foresight update prevents from going too fast and results in increased responsiveness.

Algorithms with adaptive learning rates.

One of the most difficult hyper-parameter to set in neural networks is the learning rate. Out of all above discussed gradient descent variants, momentum algorithm is found to mitigate this problem but with an introduction of another hyperparameter, momentum. There are various algorithms which have been introduced recently to adapt the learning rate for the model parameters.

Adagrad individually adapts the learning rates of all model parameters. The learning rate decreases rapidly for parameters with largest partial derivative of the loss and vice-versa. The challenge for Adagrad is that it's not suitable for non-convex setting and also leads to excessive decrease in learning rate due to accumulation of squared gradients from the beginning of training.

Adadelta overcomes the challenge of Adagrad's squared gradients by limiting the window of accumulated squared gradients to some fixed size.

RMSProp overcomes the challenge of Adagrad's non-convex setting by changing the gradient accumulation into an exponentially weighted moving average. The moving average also introduces a hyper-parameter to control the length of moving average. RMSProp may have high bias in early training.

Adaptive Movement estimation (Adam) takes advantage of both RMSProp and Momentum algorithm. It adds bias-correction and momentum to RMSProp. It stores exponentially decaying average of past squared gradients as well as past gradients. It rectifies most of the optimization problems such as vanishing learning rate, slow convergence, and high variance.

In practice, adaptive learning algorithm like Adam provides fast convergence and better results (especially in the optimization end when gradient becomes sparser). It also mitigates the need of learning rate tuning. On the flip side, gradient variant like SGD tries to find the best minimum but consequently takes more time relative to other algorithms.

[**Go to**: Python implementation of optimization techq] [Various optimization methods.ipynb]

Deep Learning: Hyperparameter optimization

Just like common machine learning algorithms, even deep learning algorithms constitute of various hyperparameters and thus comes a need of hyperparameter optimization. For a multi-layer perceptron (MLP) hyperparameters are learning rate, weights, biases, number of hidden layers, number of hidden units in each layer and activation function type. In precise, any parameter which can be initialized is considered as a hyperparameter.

Learning rate: If the learning rate is low then algorithm will take too long to converge due to small steps, if it is high then algorithm may simply not converge/ may even diverge.

To arrive at an optimum learning rate for initialization, Leslie N. Smith proposed an approach to train a network starting from a low learning rate and increasing it exponentially for every batch. This gives a graph between the learning rate and training loss for every batch from which the value of optimum learning rate can be chosen which lies in the range of fastest decrease in the loss.

Once an initial learning rate value is chosen, it is not recommended to let the learning rate remain constant or monotonically decrease throughout training but rather cyclically vary it between reasonable boundary values as proposed here.

[**Go to**: Python implementation of optimizing learning rate] [HP_optimization_MLP_learning rate.ipynb]

Biases: Bias can be set to zero if weights are initialized as non-zero or if weights are small enough to break the symmetry. However, biases for a ReLU hidden unit can be initialized with a value more than 0 like 0.01 [ref] or 0.1 [ref] to avoid large saturation during initialization.

[**Go to**: Python implementation of optimizing learning rate] [HP_optimization_MLP_biases.ipynb]

Weights: Weights must not be initialized to zero as it doesn't break the symmetry among the hidden units of the same layer. There are many proposed solution to initialize weights including but not limited to –

- Random initialization.
- Xavier (also referenced by Glorot) initialization built on uniform distribution.
- He initialization built on Gaussian distribution.
- Random orthogonal matrix initialization

Above methods doesn't guarantee an optimal performance because imposed property set for initialized weights may not persist throughout the learning. Also, for scaling rules like Glorot and He, with increase in number of layers every weight becomes small enough due to same standard deviation set for all initialized weights. Sparse initialization introduced by Martens helps to introduce more diversity among the units at initialization time.

[**Go to**: Python implementation of optimizing learning rate] [HP_optimization_MLP_weights.ipynb]

Weight decay: To avoid overfitting, the concept of regularization (same as machine learning) is applied on deep neural networks. Regularization parameter penalizes big weights within in the loss function. Depending upon the use-case, any one of the ridge, lasso and elastic-net regularization can be applied in the network.

[**Go to**: Python implementation of optimizing learning rate] [HP_optimization_MLP_regularization.ipynb]

Dropout regularization: Dropout provides an inexpensive but powerful approximation to training and evaluating a bagged ensemble of exponentially many neural networks. It does so by forming subnetworks out of original network by removing units randomly from hidden layers.

[**Go to**: Python implementation of optimizing learning rate] [HP_optimization_MLP_dropout_regularization.ipynb]

Early stopping: With an increasing number of epochs, it can be seen that a consistently decreasing validation set error can rise again. In such cases, training needs to be stopped at the point in time with the lowest validation set error. A hyper-parameter named as *patience* is provided in early stopping which determines a threshold to the number of epochs after which if significant improvement is not seen in the monitored metric then training needs to be stopped.

[**Go to**: Python implementation of optimizing learning rate] [HP_optimization_MLP_early_stopping.ipynb]

Note – For more information on neural network hyperparameters, refer here.

[**Go to**: Exercises]

1. **Classification** – To classify a Bollywood actor/actress as young, middle or old age from his/her facial attributes. The data has a total of 19906 images.
2. **Regression** – House price
3. **Regression** – Google GStore price prediction

Convolutional Neural Network [ref]

Convolution Neural Network are quite similar to ordinary neural networks as –

- They are made up of neurons which have learnable weights and biases.
- Each neuron receives some input, performs dot product and optionally follows a non-linearity function.
- It consists of a loss function (softmax/SVM) in the last fully-connected layer.

So what does change?

CNN architecture makes an explicit assumption that **inputs are images**.
Thereby, many certain properties can be encoded; making the network more efficient by reducing vast number of parameters in the network.

ConvNet vs ordinary Neural Net

Scenario – Processing images of different scales.
Ordinary neural network –

- Considering an image of shape 16x16x3. A single fully-connected neuron in a first hidden layer would have 16*16*3 = 768 weights.
- However, with increased number of neurons and image size (say 300x300x3 with 270000 weights) the structure with vast number of parameters quickly leads to overfitting.

ConvNet –

- ConvNet pre-assumes the input to be images and hence arranges its neurons in 3 dimensions: width, height and depth; thus forming a 3D volumes of neurons.
- So, an image of shape 16x16x3 will form a volume of similar shape i.e. 16x16x3.
- Also, the neurons in a layer will only be connected to a small region to its preceding layer, instead of all the neurons.
- The final output will be reduced to a single vector of class score with dimension 1x1xh (where, h denotes number of classes), arranged in the dimension of depth layer.

ConvNet layers – in brief

A ConvNet is made up of layers. Every layer accepts a 3D volume and results into a 3D volume with some differentiable function which may or may not have parameters.
ConvNet stacks up three major types of layers to build ConvNet architecture –

1. Convolutional layer
2. Pooling layer
3. Fully-connected layer

Dataset
Consider CIFAR-10 dataset which consists of 60,000 images of shape 32x32x3 each categorized into one of the 10 categories (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks). Each class have 6000 images. [link]

ConvNet on CIFAR-10

A typical ConvNet architecture could be outlined having layers in given order –

$$[INPUT - CONV - RELU - POOL - FC]$$

In detail –
INPUT layer: It will hold the raw pixel values of the image, in our case it will have width = 32, height = 32 and depth distributed among three color channels (RGB).
CONV layer: It will compute the output of the neurons which are connected to the *local regions* of the input layer, each calculating the dot product between their weights and the small region they are connected to in the input volume.

CONV layer: This may result in the volume of shape [32x32x15], if we decided to use 15 filters.
RELU layer: This layer applies an elementwise *activation function* (say $\max(0, x)$), thresholding at zero. This doesn't change the shape of volume.
POOL layer: This layer performs *downsampling operation* along the two dimensions (width and height), leaving us with the volume shape equals to [16x16x15]
FC layer: This layer computes the class score resulting in a vector space of shape [1x1x10] (since we have 10 classes in CIFAR-10 dataset). As its name (Fully-Connected) suggest, each neuron in this layer will be connected to all the previous layer volumes.
In summary,

- CONV/FC layers perform transformations which are not only the function of input volume activations but also of the parameters (weights and biases; trained by Gradient Descent)
- RELU/POOL layers will implement a fixed function.
- Each layer may or may not have parameters (CONV/FC do, RELU/POOL doesn't)
- Each layer may or may not have additional hyperparameters (CONV/FC/POOL do, RELU doesn't)

ConvNet layers – in detail

Input layer
The input layer (width and height of image) should be divisible by 2 many number of times for the efficient use of memory blocks. Possible numbers are 16, 32, 64, 224, 256, etc.

Convolution (CONV) layer
When dealing with an image, each neuron is connected to a local region in the input volume. The spatial extent of this local region is named as receptive field a.k.a filter size (a hyperparameter).
A CONV layer consists of a set of *learnable* filters whose width and height can vary from that of an input image but its depth is equal to the image depth.
For instance, if an image has dimension [32x32x3], then the receptive field can be of dimension [5x5x3], i.e. each neuron in the CONV layer will have 5*5*3 = 75 weights (75 connections to input volume + 1 bias parameter)

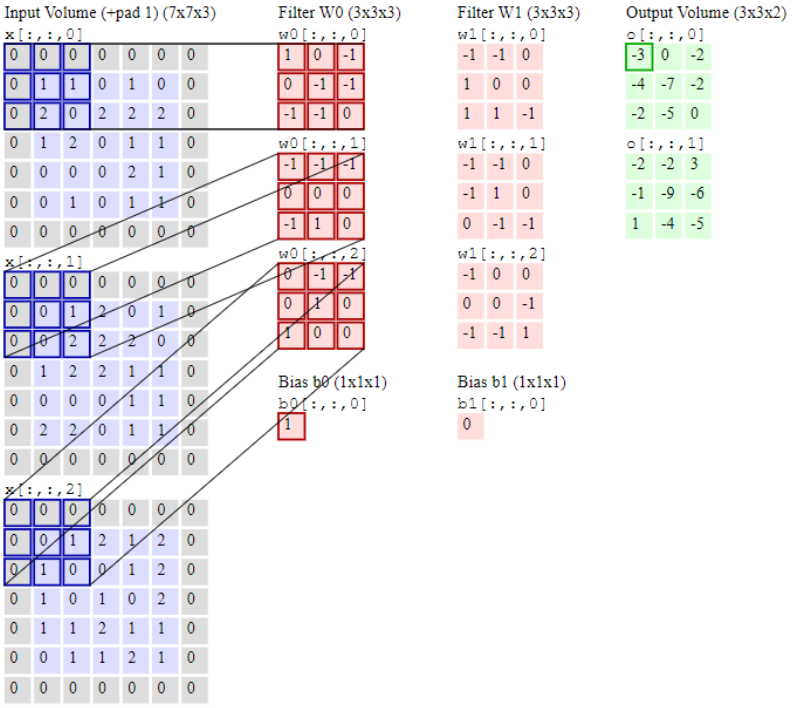
Moving ahead, there are three hyperparameter which governs the size of output volume namely depth, stride and zero-padding.

1. **Depth** – It refers to the number of filters where each filter look for distinct features likes edges, blobs etc.
2. **Stride** – Our filter slides over the input volume convolving with the local region at a time. The number of pixels to jump for next convolution is governed by the stride. Stride with value 1 makes the filter slide over the input volume with 1 pixel, a value of 2 makes the filter slide with 2 pixels and so on. Larger the stride, lesser the spatial extent of output volume.
3. **Zero-padding** – Zero-padding across the input volume border provides us two benefits: first, it helps retain the border information of the image. Since with each convolution the size of the image keeps reducing and hence without padding the border information may simply be removed. Second, it helps keep the shape of input and output volume equal. Since filter convolution may change the output volume spatial extent, hence padding helps to avoid such cases.

CONV layer conventions
Few conventions for CONV layer terminologies –

- Inputs a volume of size $W_1 * H_1 * D_1$
- Requires four hyperparameters:
  –> Number of filters K
  –> Receptive field F
  –> The stride S
  –> The amount of zero-padding P
- Produces a volume of size $W_2 * H_2 * D_2$ where
  –> $W_2 = \frac{(W_1 - F + 2P)}{(S+1)}$
  –> $H_2 = \frac{(H_1 - F + 2P)}{(S+1)}$
  –> $D_2 = K$

CONV layer illustration
The AlexNet (winner of ImageNet, 2012) accepted images of size $[227\ x\ 227\ x\ 3]$
Let us try to find the number of neurons connected to the input volume with given parameters.
Filter size, $F = 11$
Stride, $S = 4$
Zero-padding, $P = 0$
Number of filters, $K = 96$
Number of neurons which fit, $\frac{(W_1 - F + 2P)}{(S+1)} = \frac{227-11+0}{4+1} = 55$
Number of neurons N $= [55\ x\ 55\ x\ 96] = 290{,}400$
Each of these N neurons are connected to a region of size $[11\ x\ 11\ x\ 3]$ in input volume.



[ref] [Copied, need to be remade]

ReLU layer

ReLU layer applies an elementwise *activation function* (say $\max(0, x)$), thresholding at zero. This doesn't change the shape of volume.

Pooling layer

Pooling layers can be inserted periodically in-between successive consecutive CONV layers. Its function is to *progressively* reduce the spatial size of the resultant volume to reduce the number of parameters, computation and control overfitting.

It uses the MAX function and requires two hyperparameters namely F and S. Padding is generally not used with pooling layer. Also, it doesn't introduce any new parameter as it works on a fixed function.

Alternative to pooling layer: A study suggests that using a higher stride once and using only CONV layers can completely remove the need of having a pooling layer in the architecture.

Fully-Connected layer

Neurons in the FC layers are connected to all the activations in previous layers as in ordinary neural networks. It uses softmax activation function for classifying input images into various classes.

[**Go to**: Python implementation of CNN upon CIFAR10] [CNN_CIFAR_10.ipynb]

[**Go to**: Exercises]

1. Fashion MNIST (build a CNN and compare the results with ANN results)
2. **Street view house numbers**
3. **ImageNet** (You can download the repo. and then use them)

Note – For further study, refer here.

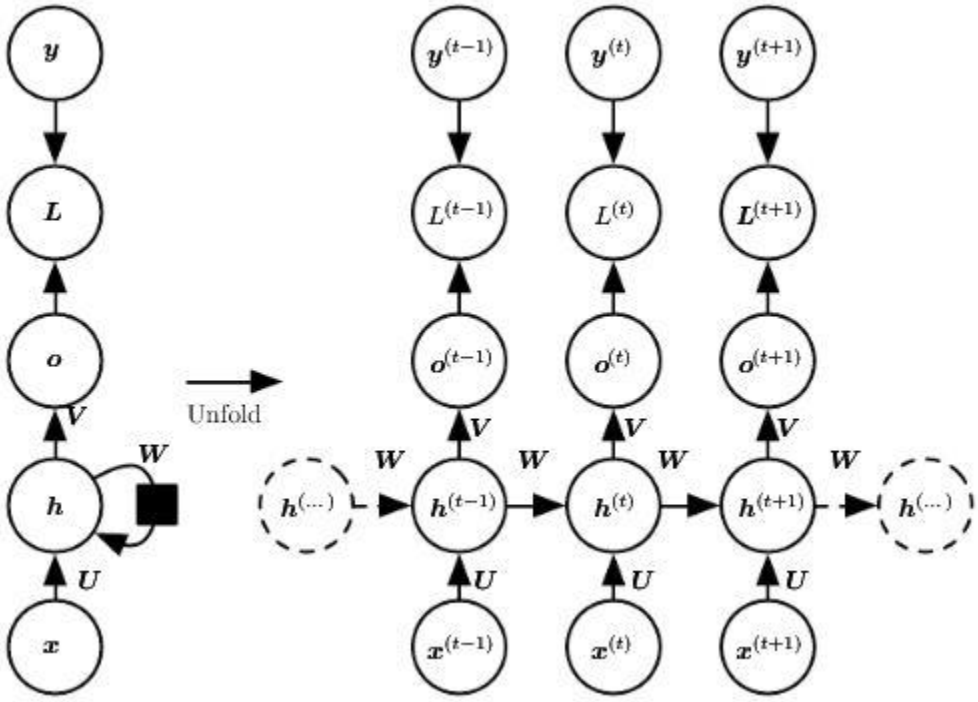
Recurrent Neural Network

Recurrent Neural Network, introduced in 1980s are meant to process sequential data like $x_1 x_2 \dots x_\tau$ rather than a grid of values. The connections between the units of RNN form a directed graph and thus makes them applicable to process sequential data with applications like word prediction, time series forecasting, etc. Just as CNNs are best suitable to process images and can readily scale onto large images, RNN can scale onto larger sequential data which will not be possible by non-sequential models.

A simple recurrent equation can be written as below; it is recurrent as current state with time t is dependent upon previous state in time $t-1$.

$$y_t = f(y_{t-1}; \theta)$$

RNNs have many variants, one of the simplistic variant graph has been shown below –

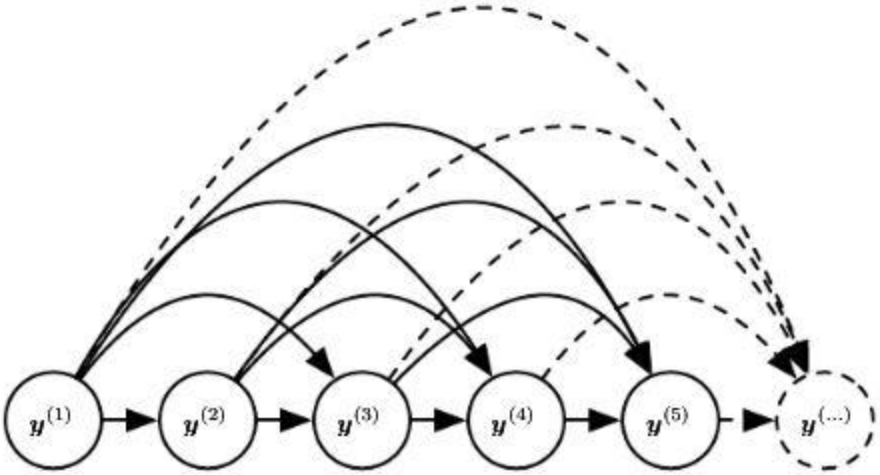


[ref] [Copied, need to be remade] Page 369

The above network produces an output at each time step and has recurrent connections between hidden units. It maps the input sequences x to corresponding output sequences o. A loss L measures how far each output value o is from the target value y. The gradient computation involves forward propagation (left to right) followed by a backward propagation (right to left). Since the all the steps follow a sequence, hence parallelization can be implemented. The backpropagation applied to RNN is termed as backpropagation through time (BPTT).

Let us understand the difference between RNN connections and a fully-connected network.

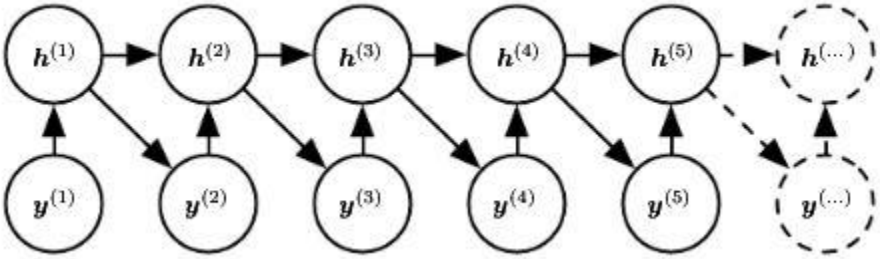
Fully-connected graphical model – It represents a joint probability distribution over some vector y; it can be decomposed into several distinct conditional distribution where we begin with predicting y_2 given y_1 and predicting y_3 given y_2 and y_1.



[ref] [Copied, need to be remade] Page 382

The model complexity grows exponentially as more number of nodes comes into picture.

RNN graphical model – It organizes variables according to time and follows one update rule which moves from one time stamp to another i.e. it defines a correlation between y_1 and y_2 but it doesn't define a direct timestamp between y_1 and y_{10}. y_1 is related to y_2 which is connected to y_3 and so on till y_{10}. Hence, it can traverse a long distance in time without making all those relationships as in fully connected model. It can of course miss out some interesting relationship between two values like y_1 and y_{100} as they are too far apart but in most of the cases it assumes that only nearby timestamps are more important.



[ref] [Copied, need to be remade] Page 383

To determine the length of sequence there're few proposed approaches –

- Add a special symbol to the end of a sequence.
- Add an extra Bernoulli output to the model at each time step which decides to either continue generation or halt it. This approach is more general to the one mentioned above as it is not specific to the sequence of symbols.
- Add an extra output to the model that predicts the integer τ itself where τ is the sequence length.

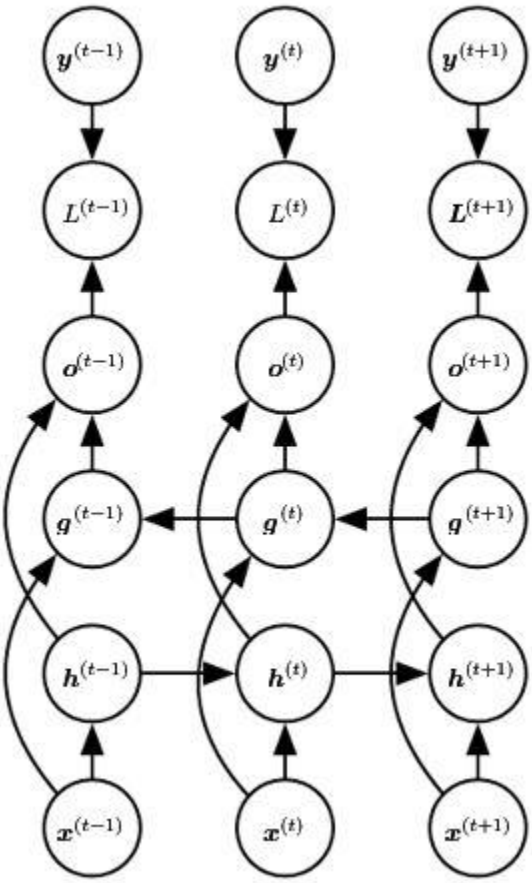
RNN types

Two of the major variants of RNN are –

1. Bidirectional RNN
2. Recursive NN

Bidirectional RNN – BRNN combine an RNN that moves forward through time beginning from the start of the sequence with another RNN that moves backward through time beginning from the end of the sequence. The output sequence o_t is computed by both the past h_t and the future g_t but is more sensitive to the input values x and time t.

These networks can be applied only on local files and not on real time data because the network doesn't know the future value in real data. Application of bi-directional RNN includes video annotation.



[ref] [Copied, need to be remade] Page 389

Recursive Neural Network – Recursive NN are structured as deep tree rather than chain like structure and forms a generalization of recurrent neural networks. For a sequence of same length τ, the depth can be reduced from τ to $O(log\tau)$ which can help in dealing with long term dependencies. The applications include processing data structures as input to neural nets, NLP and computer vision.

Long-term dependencies

The gradient propagated over many stages tend to either vanish (most frequently) or explode (rarely, but with much damage to optimization). Assuming network is stable, such problems still arises due to exponentially smaller weights given to long-term interactions compared to short-term ones.

RNN involve the composition of the same function multiple times, one per time step. These compositions result into non-linear behavior. The gradient vanishes when eigenvalues with magnitude less than one decay to zero and it explodes when eigenvalues' magnitude becomes greater than one.

To tackle such problem, a model needs to be designed that operates at multiple time scales, so that some parts of the model operate at fine grained time scales and can handle small details; while other parts operate at coarse time scales and transfer information from the distant past to the present more efficiently.

There are three ways to achieve it –

1. **Adding skip connections through time** – Adding direct connections from variables in the distant past to variables in the present. RNN are introduced with a time-delay of d to mitigate this problem. Now, gradients diminish exponentially as a function of $\frac{\tau}{d}$ rather than τ. Since there are still both single step connections and delayed connections, gradients may still explode exponentially in τ.
2. **Leaky units** – Leaky units are hidden units with linear self-connections and a weight near to one. Mathematically, a leaky unit h_t with a value v_t can be written as –
   $$h_t \leftarrow \alpha h_{t-1} + (1-\alpha)v_t$$
   Here, α is an example of a linear self-connection from h_{t-1} to h_t whose values can be varied in a larger scale as compared to adjusting the integer-valued skip length d.
   If $\alpha \approx 1$, the h_t remembers information about the past for a long time.
   If $\alpha \approx 0$, the h_t discards the information about the past rapidly.
3. **Removing connections** – It involves actively removing length-one connections and replacing them with longer connections. Units receiving such new connections may learn to operate on a long time scale but may also choose to focus on their other short-term connections.

Gated RNN

Leaky units as mentioned earlier provide an efficient solution to avoid gradient problems. However, leaky units have few limitations which are filled by gated RNN. Let us consider what does Gated RNN improves over leaky units.

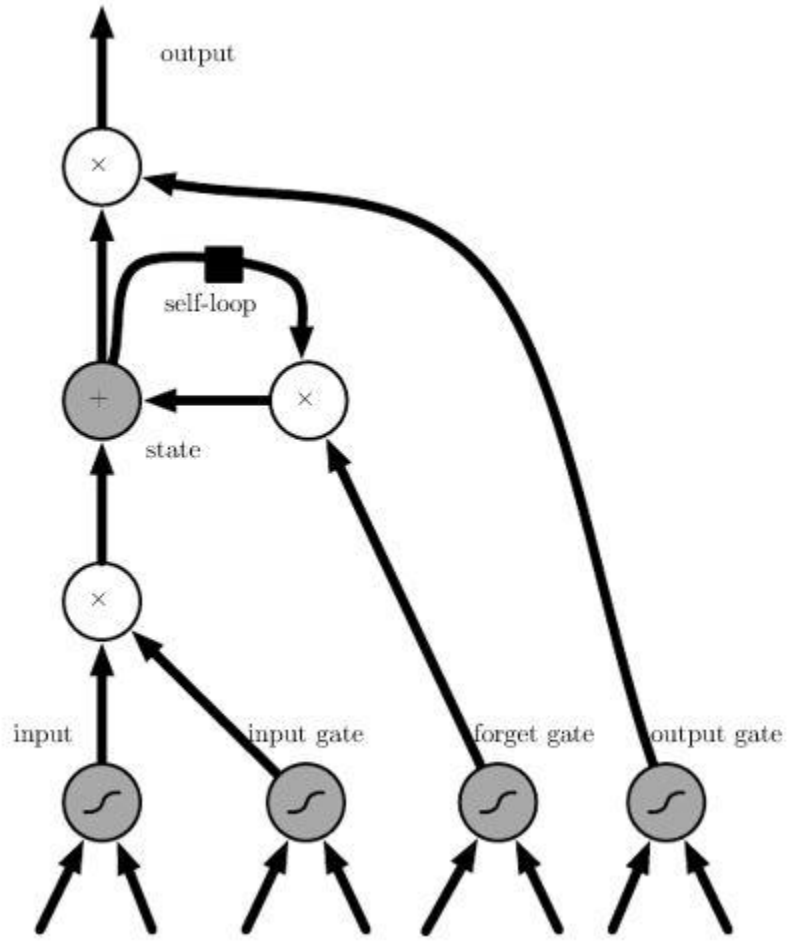
- Leaky units assigned connection weights that are either manually chosen constants or parameters. Gated RNNs generalizes this to connection weights that may change at each time step.
- Leaky units allow the network to accumulate information over a long duration and once used forget the old state. Gated RNN clears this state by themselves rather than requiring any manual efforts.

Long short-term memory (LSTM) and RNN with Gated recurrent unit (GRU) are two divisions inside Gated RNN.

Long short-term memory (LSTM)

LSTM also have a chain like structure but the repeating module has a different structure. Instead of a unit that simply applies an elementwise non-linearity to the output of recurrent units, LSTM consists of "LSTM cells" which have an internal recurrence (a self-loop) in addition to the outer recurrence. Each cell has more parameters and a system of gating units that control the flow of information.

The self-loops allow the gradient to flow for long durations. Also, by making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically.



[ref] [Copied, need to be remade] Page 405

Key points in LSTM –

- The cells are connected recurrently to each other, replacing the usual hidden units.
- An input feature is computed with a regular ANN.
- The state unit has a linear self-loop whose weight is controlled by the forget gate.
- The output of the cell can be shut-off by the output gate.
- All the gating units have a sigmoidal non-linearity while input unit can have any non-linearity.

The applications of RNN ranges includes handwriting recognition, speech recognition, handwriting generation, image captioning, machine translation etc.

RNN with Gated recurrent unit (GRU)

Gated recurrent unit is built upon LSTM. It introduces two new gates, reset gate and update gate (combination of the forget and input gates). In GRU, a single gating unit simultaneously controls the forgetting gate and the decision to update the state unit.

The update gates act like some conditional leaky integrators that can linearly gate any dimension, thus choosing to copy it or completely ignore it by replacing it with new 'target state' value.
The reset gates control which parts of the state get used to complete the next target state. Thus, introducing an additional non-linear effect in the relationship between past and future states.

Additional resources to refer for LSTM and GRU –

1. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
2. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21?gi=dc9949d72bd1