מכון טכנולוגי חולון
Holon Institute of Technology

Department of Mathematics
June 2020

Applied Mathematics Project

Deep Learning Prediction of Time Series

Part I – **Artificial Neural Network Theory**

Part II – **Time Series Theory**

Part III – **Time Series and python libraries**

Part IV – **Implementation in python and conclusions**

Advisor:

Dr. Yirmeyahu Kaminski

Students:

Shay Malkin i.d. 036532869

Tal Ladijensky i.d. 322869439

Nir Titelbom i.d. 204708226

Artificial Neural Network

Applied Mathematics

Contents

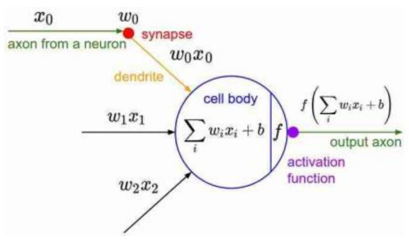
- ❖ Introduction
- ❖ ANN Applications
- ❖ Perceptron
- ❖ Feedforward Neural Network
- ❖ Activation Functions
- ❖ Cost Function
- ❖ Gradient Descent
- ❖ Stochastic Gradient Descent
- ❖ Back Propagation
- ❖ Convolutional Neural Network
- ❖ Recurrent Neural Networks
- ❖ Autoencoder
- ❖ ANN Advantages

Introduction

An ANN is a mathematical model for computation inspired by the structure of a biological neural network (the brain). The model is able to "learn" to perform tasks from given examples (training data), without being "hard coded" with specific instructions. This approach enables us to solve complex problems that otherwise would be impossible to solve (with direct instructions).

A good example is an image classifier, i.e. a model that, given an image from a pool of a finite number of categories (dogs, cats, cars, etc.), classifies the image to its correct category.

The model learns to perform that task by being given a large set of images labeled with their correct category, then allowed to predict their labels (at first, completely arbitrary). Then, we compare the predictions with the correct results and calculate the cost function. That means, how much the prediction was wrong. Via an optimization process (more on that later) we gradually minimize the cost function.



ANN Applications

Artificial neural networks can help solve many problems such as:

- ➢ Pattern classification
- ➢ Categorizations
- ➢ Function approximation
- ➢ Forecasting
- ➢ Optimization
- ➢ Content - addressable memory
- ➢ Control on the inputs and the outputs

We can see artificial neural networks today at many fields like face recognition, character recognition, capital market prediction, text analysis etc.

Perceptron

The Perceptron was introduced by Frank Rosenblatt in 1957. The idea of the Perceptron is to be a type of linear classifier. The Perceptron gets a combination of a feature vector and a set of weights , sums them up and passes the sum forward to produce an output. The Perceptron is a single layer neural network often referred to as a neuron. This artificial neuron is based on our biological neuron.
In the 1950s, the perceptron became the first model that can learn the weights and can define categories of a given examples of inputs from each category.
A multi-layer perceptron is called Neural Networks and it is a mathematical function mapping a set of input values to output values. We can think of each application of a different mathematical function as providing a new representation of the input.

The usage of the idea of the Perceptron is for an algorithm for supervised learning of binary classifiers. This algorithm enables them to study and understand elements in the test data one at a time.

How this algorithm works?

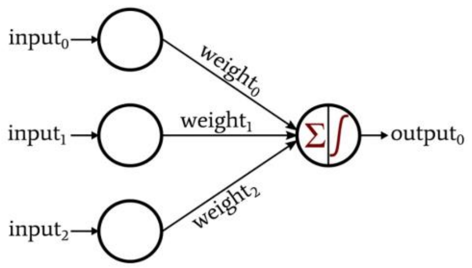
Well, each neuron holds an internal state called activation signal. Each neuron is connected to another neuron via connection link.

We use a function that maps its input ***X***, which is multiplied with the learned bias and weights respectively and an output *f(x)* is generated.

The parameters of the function consist of the learned bias and weights.

The bias ***b*** is a real number and ***W*** is a vector with same size as the inputs. ***X*** is a vector of the input ***n*** variables. For a given input ***X***, the output is either a '1' or a '0'.

$$f(x) = \begin{cases} 1 & if\ W^T X + b > 0 \\ 0 & otherwise \end{cases}$$

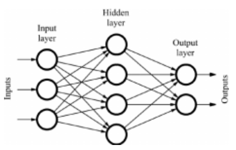


What can we do in the case of an error in the output for a given example?
The error is propagated backward to allow weight adjustment to happen.

To sum up, a perceptron is a mathematical model of the behavior of a single neuron in a biological nervous system. A single neuron can solve some very simple learning tasks, but the power of neural networks comes when many of them are connected in a network architecture. The architecture of an artificial neural network refers to the number of neurons and the connections between them.

Feedforward Neural Network

A feedforward neural network is an artificial neural network where the nodes do not form a cycle. It is a directed acyclic graph. It is the first and simplest type of neural network devised, where the information moves in only one direction, from the input nodes, through the hidden nodes (if they exist), to the output nodes.
The simplest kind of neural network is a single layer perceptron network, which consist of a single layer of output nodes. This network is only capable of learning linearly separable patterns.
A more complex network would be a multilayer perceptron network. It has at least one hidden layer and it can distinguish data that is not linearly separable.



Activation Functions

Activation functions are an extremely important feature of the artificial neural networks. They basically decide whether a neuron should be activated or not. Whether the information that the neuron is receiving is relevant for the given information. The activation function is often a nonlinear transformation that we do over the input signal. This transformed output is then sent to the next layer as input. When we do not have the activation function the weights and bias would simply do a linear transformation.

We would want our neural networks to work on complicated tasks like language translations and image classifications. Without the differentiable nonlinear function, this would not be possible.

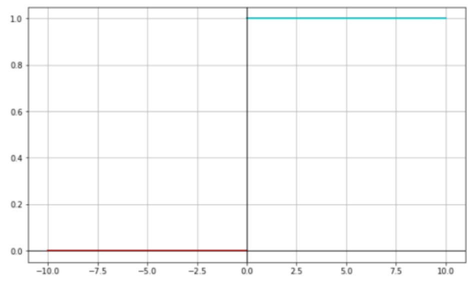
Types of Activation Functions

Binary Step Function:

The binary function is extremely simple. It can be used while creating a binary classifier. When we need to say yes or no for a single class, step function would be the best choice, as it would either activate the neuron or leave it to zero.

Here we define the function as:

$$f(x) = \mathbf{1}_{[0,\infty)}$$

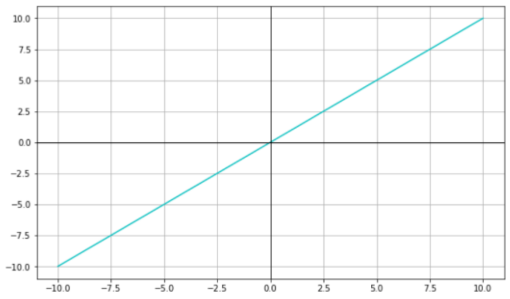


Linear Function:

Here the activation is proportional to the input. The input x, will be transformed to ax. This can be applied to various neurons and multiple neurons can be activated at the same time.

Here we define the function as:

$$f(x) = ax$$

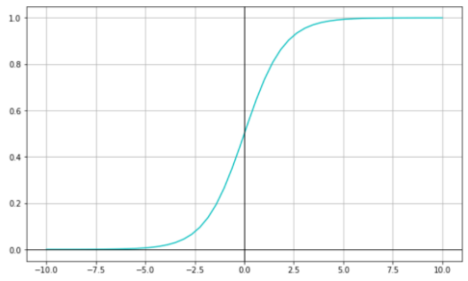


Sigmoid:

This is a smooth function and is continuously differentiable. The biggest advantage that it has over step and linear function is that it is nonlinear. That's means that when we have multiple neurons having sigmoid function as their activation function, the output is nonlinear as well. The function ranges from 0-1.

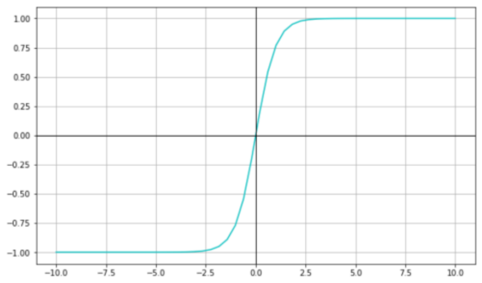
Here we define the function as:

$$f(x) = \frac{1}{1+e^{-x}}$$



Hyperbolic tangent:

The tanh function is very similar to the sigmoid function. It is actually just a scaled version of the sigmoid function. Tanh works similar to the sigmoid function but is symmetric over the origin. it ranges from -1 to 1.It basically solves the problem of the values all being of the same sign.



Here we define the function as:

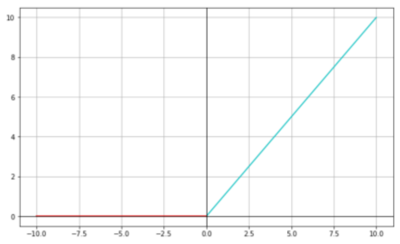
$$f(x) = \tanh(x) = 2 \cdot sigmoid(2x) - 1$$

ReLU:

ReLU is the most widely used activation function. The ReLU function is nonlinear, which means we can easily backpropagate the errors and have multiple layers of neurons being If the input is negative it will convert it to zero and the neuron does not get activated. This means that at a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

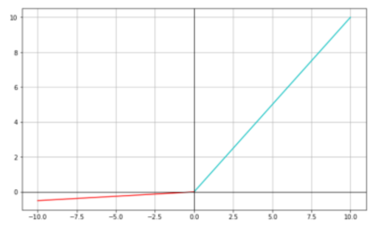
Here we define the function as:

$$f(x) = x \cdot \mathbf{1}_{[0,\infty)}$$



Leaky ReLU:

Leaky ReLU function is an improved version of the ReLU function. Instead of defining the Relu function as 0 for x less than zero, we define it as a small linear component of x (The parameter a is small).



Here we define the function as:

$$f(x) = ax \cdot \mathbf{1}_{(-\infty,0)} + x \cdot \mathbf{1}_{[0,\infty)}$$

Note:

There is Parameterized ReLU function. It is defined similar to the Leaky ReLU. However, in this case, the network also learns the value of ‘a‘ for faster and more optimum convergence.

Softmax:

This is an activation function which convert the input to the neuron in the range of 0 and 1 in such a way that the total sum of the outputs is equal to 1. This function is mostly used where we have to define the multi class classification where it gives the output in the form of the probability of occurrence of a class. It is mostly used in the final layer of the neural network.

Here we define the function as:

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{m=1}^{K} e^{x_m}}$$

where: $j = 1, \ldots, K \quad , \quad x \in \mathbb{R}^K$

Example

$$\begin{bmatrix}1.2\\0.9\\0.4\end{bmatrix} \longrightarrow \boxed{\text{Softmax}} \longrightarrow \begin{bmatrix}0.46\\0.34\\0.20\end{bmatrix}$$

Cost Function

During the training process, we give the model initial weights randomly and try to make him bring us a good prediction on the training set. Cost function calculates how much wrong the model was in his output, by calculating the distance-based error. For example, given an input $x \in \mathbb{R}^{n_1}$ and let $a_j^{[l]}$ denote the output from neuron j at layer l. Accordingly, we define:

$$a^{[0]} = x \in \mathbb{R}^{n_0}$$
$$a^{[l]} = \sigma\left(W^{[l]}a^{[l-1]} + b^{[l]}\right) \quad , \quad l = 1, \ldots, L$$

Now suppose we have N pieces of data (training points) in $\mathbb{R}^{n_1}$, $\{x^{\{i\}}\}_{i=1}^{N}$,for which there are given target outputs $\{y(x^{\{i\}})\}_{i=1}^{N}$ in $\mathbb{R}^{n_l}$. The quadratic cost function is:

$$Cost = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left\|y\left(x^{\{i\}}\right) - a^{[L]}\left(x^{\{i\}}\right)\right\|_2^2$$

Gradient Descent

Gradient Descent is an iterative optimization algorithm for finding the minimum of a function. Our goal is to minimize the cost function. We will denote a superscript ***** to symbol the value that minimizes the function. For example $p^* = arg \ min \ f\left(p\right)$.

In this method , we take steps in the opposite direction to the gradient of the function at a given point. The method works like that if we wish to get to the minimum point, we would want to 'go down', at every step, in the direction with the steepest descent, which is opposite to the gradient. It resembles to the fact that if we wish to go down from a high place we look for the steepest slope.

$$p_{n+1} = p_n - \eta \cdot \nabla f\left(p_n\right)$$

Where p is the input value and η is the step size.

Stochastic Gradient Descent

Another version of Gradient Descent that is based on stochastic evaluation.
The word 'Stochastic' means a system or a process that is linked with a random probability. Hence, in SGD we randomly select one data point from our training set in each iteration in order to reduce the computations in total.
Our goal is to complete the network's task in the most efficient way possible.
Hence, we want to minimize the cost function.
SGD is typically noisier than other ways of Gradient Descent, and it usually takes a higher number of iterations to reach the desired minimum. That is because of the random picking of the training data.
But, it is less expensive than a typical Gradient Descent. That's why in most cases SGD is preferable over other methods.

SGD

We denote $p \in \mathbb{R}^s$ where s is a total of all weights and biases.
As we said, the goal is to minimize the cost function. We will define our cost function to be:

$$Cost(p) : \mathbb{R}^s \mapsto \mathbb{R}$$

In order to find the minimum or in our case $-\nabla Cost(p)$.
We will expand the cost function in to a first order Taylor Series expansion:

$$Cost(p+\Delta p) \approx Cost(p) + \sum_{r=1}^{s} \frac{\partial Cost(p)}{\partial p_r} \Delta p_r$$

We will denote $\nabla Cost(p) \in \mathbb{R}^s$ to be the vector of partial derivatives:

$$Cost(p+\Delta p) \approx Cost(p) + \nabla Cost(p)^T \Delta p$$

We want to choose Δp to lie in our desired direction $-\nabla Cost\left(p\right)$.

This leads us to $p' = p - \eta \cdot \nabla Cost\left(p\right)$ where η is our learning rate (our step size in the gradient descent). Our goal is to iterate until we meet our desired criterion.

Now we let:

$$C_{x^{\{i\}}} = \frac{1}{2}\left\| y\left(x^{\{i\}}\right) - a^{[L]}\left(x^{\{i\}}\right)\right\|_2^2$$

Where $a^{[L]}$ denotes the output at the output level. Then, we can write:

$$\nabla Cost\left(p\right) = \frac{1}{N}\sum_{i=1}^{N} \nabla C_{x^{\{i\}}}\left(p\right)$$

Instead of computing the vector at every iteration, a cheaper method is to replace the mean of the individual gradients over all training points by the gradient at a single randomly chosen training point.

SGD algorithm

The algorithm is simple and it is as follows: For each iteration randomly choose an integer from {1,2,3,.....,N} where each number represent a different training point in the training set and updating p by the formula we presented in Gradient Descent slide:

$$p' = p - \eta \cdot \nabla Cost\left(p\right)$$

Note:

There is a similar method that For $m << N$ we choose m integers, $\{k_1, \ldots, k_m\}$, uniformly at random from {1,2,…,N} and update as follow:

$$p' = p - \eta \cdot \frac{1}{m}\sum_{i=1}^{m} \nabla C_{x^{\{i\}}}\left(p\right)$$

Stochastic Gradient Descent VS. Gradient Descent

In GD we have to go over all of our training data to do a single update for a parameter in every change to P.
On the other hand in SGD we use just a single training point in every change to P.
Hence , if we have a large data set then using GD can be expensive in every iteration when we have to update P because we will have to run through all the samples in our set.
That's why SGD will be faster because we use only one training point and it starts improving itself right away from the first point.
Although the faster conversion SGD won't minimize the cost function as well as GD but the parameters will be close enough and eventually will reach our criterion goal values and remain in the surrounding area.

Backpropagation

A neural network propagates the signal of the input data forward through its parameters towards the moment of decision, and then back-propagates information about the error, in reverse through the network, so that it can change the parameters. Backpropagation takes the error associated with a wrong guess by a neural network, and uses that error to adjust the neural network's parameters in the direction of less error.

A neural network has many parameters, so what we're really measuring are the partial derivatives of each parameter contribution to the total change in error. At the heart of backpropagation is an expression for the partial derivative $\partial C/\partial w$, $\partial C/\partial b$ of the cost function C with respect to any weight w (or bias b) in the network. The expression tells us how quickly the cost changes when we change the weights and biases.

First, we define the following expressions:

w^l_{ij} : weight for node *j* in layer *l* for incoming node *i*

b^l_i : bias for node *i* in layer *l*

a^l_i : product sum plus bias (activation) for node *i* in layer *l*

n_l : number of nodes in layer *l*, where $l \in \{1, ..., L\}$

The output is $a^{[L]}$ and the cost function is $C = \frac{1}{2}\left\| y - a^{[L]} \right\|_2^2$

$z^{[l]} \triangleq W^{[l]}a^{[l-1]} + b^{[l]} \in \mathbb{R}^{n_l}$ for $l \in \{2, ..., L\}$

$a^{[l]} = \sigma\left(z^{[l]}\right)$, $l \in \{2, ..., L\}$

We let $\delta^{[l]} \in \mathbb{R}^{n_l}$ be defined by:

$$\delta^{[l]}_j \triangleq \frac{\partial C}{\partial z^{[l]}_j} \qquad \forall 1 \le j \le n_l \quad , \quad \forall 2 \le l \le L$$

This expression, which is often called the **error** in the *j*th neuron at layer *l*, is an intermediate quantity that is useful both for analysis and computation.

Backpropagation will give us a procedure to compute the error δ, and then will relate δ to the partial derivative $\partial C/\partial w, \partial C/\partial b$.

The cost function can be at least at a minimum if all partial derivatives are zero, that means δ=0 is a useful goal.

We define the Hadamard product as follows:

$$\forall x, y \in \mathbb{R}^d : \quad x \odot y \in \mathbb{R}^d, \quad \left(x \odot y\right)_i \triangleq x_i y_i \quad \forall\ 1 \le i \le d$$

The Hadamard product is formed by pairwise multiplication of the corresponding components.

The four equations behind BP:

$$i. \quad \delta_j^{[L]} = \sigma'\left(z_j^{[L]}\right) \odot \left(a_j^{[L]} - y_j\right)$$

$$ii. \quad \delta^{[l]} = \sigma'\left(z_j^{[l]}\right) \odot \left(W^{[l+1]}\right)^T \delta^{[l+1]} \quad , \ 2 \le l \le L-1$$

$$iii. \quad \delta_j^{[l]} = \frac{\partial C}{\partial b_j^{[l]}} \quad , \ 2 \le l \le L$$

$$iv. \quad \delta_j^{[l]} a_k^{[l-1]} = \frac{\partial C}{\partial w_{jk}^{[l]}} \quad , \ 2 \le l \le L$$

As we mentioned, Backpropagation is a method to update the weights in the neural network by taking into account the actual output and the desired output. We work on each layer separately instead of all weights together.

The derivative with respect to each weight and bias is computed using the chain rule, from the last layer backwards.

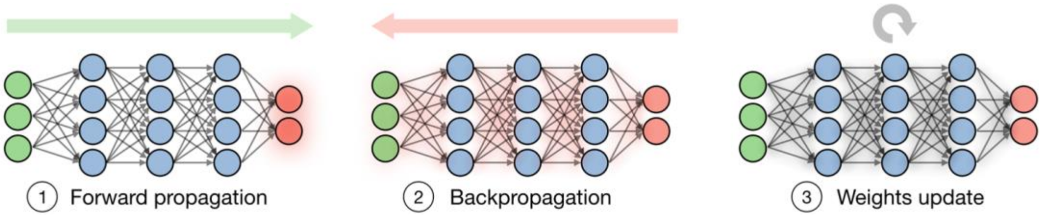
Then, by using this method, we get:

$$\forall l \in \left\{L, L-1, ..., 2\right\} : \quad \begin{aligned} W^{[l]} &\leftarrow W^{[l]} - \eta \cdot \frac{\partial C}{\partial W^{[l]}} \\ b^{[l]} &\leftarrow b^{[l]} - \eta \cdot \frac{\partial C}{\partial b^{[l]}} \end{aligned}$$

η is the step size

By equations (*iii*) and (*iv*) we can see:

$$\forall l \in \left\{L, L-1, ..., 2\right\}: \quad \begin{aligned} W^{[l]} &\leftarrow W^{[l]} - \eta \cdot \delta^{[l]} \left(a^{[l-1]}\right)^T \\ b^{[l]} &\leftarrow b^{[l]} - \eta \cdot \delta^{[l]} \end{aligned}$$



Equation #1: An equation for the error in the output layer

$$\delta_j^{[L]} = \sigma'\left(z_j^{[L]}\right) \odot \left(a_j^{[L]} - y_j\right)$$

The first term on the right, measures how fast the activation function σ is changing at z. The second term on the right, just measures how fast the cost is changing as a function of the output activation. If, for example, the cost function doesn't depend much on a particular output neuron, j, then δ will be small, which is what we expect.

<u>Proof:</u>

From definition of δ *and* a:

$$\delta_j^{[L]} = \frac{\partial C}{\partial z_j^{[L]}} = \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}}$$

So, we can get:

$$a^{[L]} = \sigma\left(z^{[L]}\right) \Rightarrow \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \sigma'\left(z_j^{[L]}\right)$$

$$C = \frac{1}{2}\left\|y - a^{[L]}\right\|_2^2 \Rightarrow \frac{\partial C}{\partial a_j^{[L]}} = \frac{\partial}{\partial a_j^{[L]}} \sum_{k=1}^{n_L} \frac{1}{2}\left(y_k - a_k^{[L]}\right)^2 = a_j^{[L]} - y_k$$

This gives:

$$\delta_j^{[L]} = \left(a_j^{[L]} - y_k\right)\sigma'\left(z_j^{[L]}\right) \blacksquare$$

Equation #2: The error $\delta^{[l]}$ in terms of the error in the next layer

$$\delta^{[l]} = \sigma'\left(z_j^{[l]}\right) \odot \left(W^{[l+1]}\right)^T \delta^{[l+1]} \quad , \quad 2 \le l \le L-1$$

By combining equation 2 with equation 1 we can compute the error δ for any layer in the network. We start by using equation 1 to compute δ in layer L, then apply equation 2 to compute δ in layer L-1, and so on, all the way back through the network.

Proof:

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}}$$

From the definition of z we know:

$$z^{[l+1]} = W^{[l+1]}a^{[l]} + b^{[l+1]} \quad , \quad a^{[l]} = \sigma\left(z^{[l]}\right)$$

So, we get:

$$\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \frac{\partial}{\partial z_j^{[l]}}\left[\sum_{s=1}^{n_l} w_{ks}^{[l+1]}\sigma\left(z_s^{[l]}\right) + b_k^{[l+1]}\right] = w_{kj}^{[l+1]}\sigma'\left(z_j^{[l]}\right)$$

This gives:

$$\delta_j^{[l]} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} w_{kj}^{[l+1]} \sigma'\left(z_j^{[l]}\right)$$

$$\delta^{[l]} = \delta^{[l+1]}\left(W^{[l+1]}\right)^T \sigma'\left(z^{[l]}\right) \quad \blacksquare$$

Equation #3: The rate of change of the cost with respect the bias

$$\delta_j^{[l]} = \frac{\partial C}{\partial b_j^{[l]}} \quad , \quad 2 \leq l \leq L$$

the error δ is exactly equal to the rate of change $\partial C/\partial b$. we have already know how to compute δ by equations 1 and 2.

Proof:

We know that: $z^{[l]} = W^{[l]}\sigma\left(z^{[l-1]}\right) + b^{[l]}$

Since $z^{[l-1]}$ does not depend on $b^{[l]}$, then: $\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1$

$$\frac{\partial C}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}}\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} = \delta_j^{[l]} \quad \blacksquare$$

Equation #4: The rate of change of the cost with respect to the weights

$$\delta_j^{[l]} a_k^{[l-1]} = \frac{\partial C}{\partial w_{jk}^{[l]}} \quad , \; 2 \leq l \leq L$$

The equation can be rewritten in a less index-heavy notation as:

$$\frac{\partial C}{\partial w} = a_{input}\delta_{output}$$

We can see that when the activation a_{input} is small, $a_{input} \approx 0$,the gradient term $\partial C/\partial w$ will also tend to be small. In this case, we'll say the weight ***learns slowly***, meaning that it's not changing much during gradient descent. In other words, one consequence of equation 4 is that weights output from low-activation neurons learn slowly.

Proof:

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \sum_{s=1}^{n_l} \frac{\partial C}{\partial z_s^{[l]}} \frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}}$$

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial}{\partial w_{jk}^{[l]}} \left[\sum_{k=1}^{n_{l-1}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]} \right] = a_k^{[l-1]} \quad , \; \forall s \neq j : \; \frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}} = 0$$

That is, the above two equations follow because the *j*th neuron at layer *l* uses the weights from only the *j*th row of *W* in layer *l*. Hence:

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = \delta_j^{[l]} a_k^{[l-1]} \; \blacksquare$$

BP - Advantages vs. Disadvantages

Most of the advantages of Backpropagation are:

➢ Backpropagation is fast, simple and easy to program.

➢ Simplifies the network structure by elements of the weighted links that have the least effect on the trained network.

➢ It helps to assess the impact that a given input variable has on a network output.

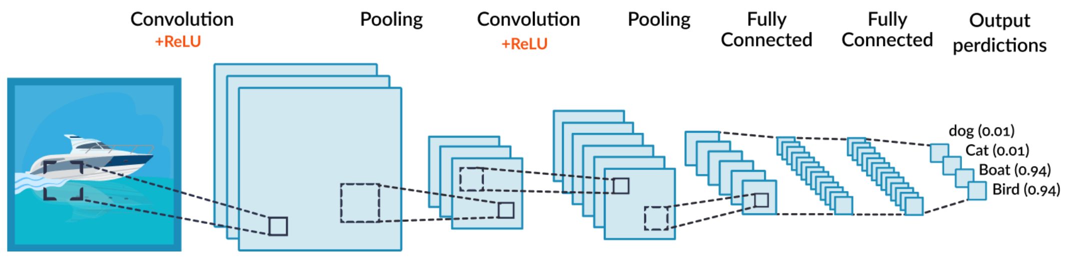
Disadvantages of using Backpropagation are:

➢ The actual performance of backpropagation on a specific problem is dependent on the input data.

➢ Backpropagation can be sensitive to noisy data.

Convolutional Neural Network

CNN's are a specific kind of neural networks useful for processing data that has a known grid-like topology. For example, time series data (represented as a 1D grid) or an image data (2D grid).
CNN's use a convolution operation instead of general matrix multiplication in one or more of it's layers.



The convolution operation

In general, the convolution is an operation between two functions, defined as:

$$S\left(t\right) = \left(x * w\right)\left(t\right) = \int_{\mathbb{R}} x\left(a\right) w\left(t-a\right) da$$

A discrete convolution is defined as:

$$S\left[t\right] = \left(x * w\right)\left[t\right] = \sum_{a \in \mathbb{Z}} x\left[a\right] w\left[t-a\right]$$

In neural net applications, the input is usually a multidimensional array of data, and the kernel is a multidimensional array of parameters. Because these functions are discrete, we usually assume that they are zero everywhere except for a finite set of points, where we store the data. That means that in practice we can sum over a finite number of array elements.

We can also apply the convolution over more than one axis at a time. For example, if we have a 2D image $\mathbf{I}$ as an input and a 2D kernel $\mathbf{K}$, the convolution would be:

$$S\left[i,j\right] = \left(I * K\right)\left[i,j\right] = \sum_m \sum_n I\left[m,n\right] K\left[i-m, j-n\right]$$

Because the convolution is commutative, we can write it as:

$$S\left[i,j\right] = \left(I * K\right)\left[i,j\right] = \sum_m \sum_n I\left[i-m, j-n\right] K\left[m,n\right]$$

In neural networks we usually implement a related function called cross-correlation, which is the same as convolution but without flipping the kernel.

$$S\left[i,j\right] = \left(I * K\right)\left[i,j\right] = \sum_m \sum_n I\left[i+m, j+n\right] K\left[m,n\right]$$

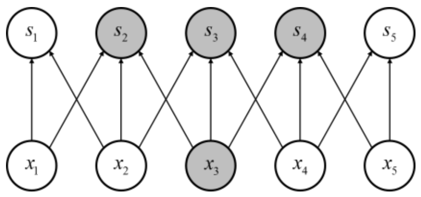
The convolution leverages three important ideas that can help improve our system.

1. Sparse interactions
2. Parameter sharing
3. Equivariant representation

Sparse interactions

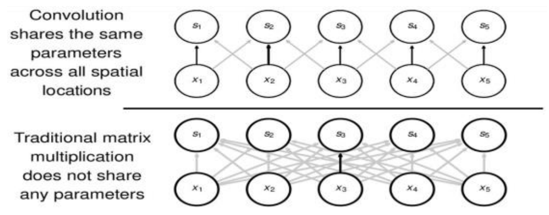
Traditional neural network layers (fully connected layers) use a matrix multiplication to describe the interaction between each input unit and each output unit. Meaning, each output unit interacts with every input unit.

A convolution layer, on the other hand, has sparse interactions. This is the result of having a kernel that is smaller than the input. Each output unit is connected to a small region of the input. For example, if we have a 3x3 kernel, each output unit will be connected to 9 input units.



Parameter sharing

In a fully connected layer, each element of the weight matrix is used only once when computing the output of the layer.
In a convolution layer, each element of the kernel is used at every position of the input (except maybe the boundaries).
Parameter sharing makes convolution much more efficient in terms of memory requirements than matrix multiplication.


Convolution shares the same parameters across all spatial locations
s_1 s_2 s_3 s_4 s_5
x_1 x_2 x_3 x_4 x_5
Traditional matrix multiplication does not share any parameters

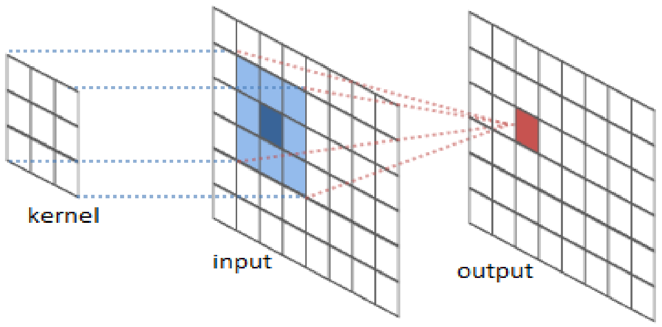
Equivariant representation

A convolution is also equivariant to translation.

For a function to be equivatiant, it means that if the input is shifted, the output is shifted in the same way.

For example, when processing time series data, the convolution produces a sort of timeline that shows where different features appear in the input. If we move an event later in time, the same representation of it will appear in the output, just later in time.

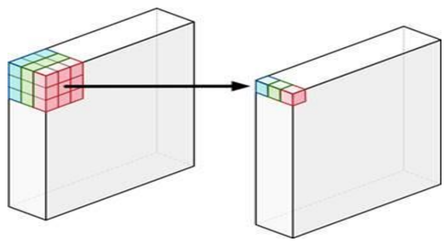
The convolution is achieved by passing a kernel over the input and computing the corresponding dot product at each region.



As an example, we will consider a 32x32x3 image and a 5x5x3 kernel.

For each region, the computation is equivalent to computing a 5x5x3=75 dimensional dot product.

Therefore, if we use only one kernel, the output will have a depth of one. For k kernels, the output will have a depth of k.



Regarding the width and height of the output, because of the dimensions of the kernel, the output will be smaller than the input.

In our example, a 32x32x3 image and a 5x5x3 kernel with stride 1, the output will be 28x28x1

To attend this issue, we usually add zero padding to the image.

If we wish to preserve the original size, we can use zero padding of (F-1)/2 with a stride of 1, where F is the size of the kernel (FxF)

Summary of the dimension aspect of convolution

Input: $W_1 \times H_1 \times D_1$

K - number of filters

F - size of filters (FxF)

S - the stride

P - the amount of padding

Output: $W_2 \times H_2 \times D_2$

$$W_2 = \left(W_1 - F + 2P\right) / S + 1$$
$$H_2 = \left(H_1 - F + 2P\right) / S + 1$$
$$D_2 = K$$

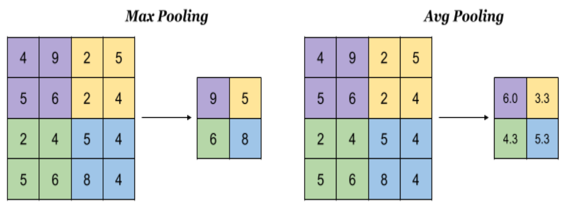
Pooling

A typical convolution layer consists of three stages:

1. Convolution
2. Non - linear activation function
3. Pooling function

A pooling function replaces the output of a certain location with a summary statistic of the nearby outputs. One example is max pooling. It takes the max value of a rectangular neighborhood.

Other examples of pooling are:



- Average pooling
- L_2 norm of neighborhood
- Weighted average based on distance from center.

Pooling helps make the representation become invariant to small translations of the input.

Invariance to local translation is very useful if we care more about whether some feature is present, rather than exactly where it is.

Recurrent neural network

An RNNs is essentially a fully connected neural network that contains a refactoring of some of its layers into a loop. That loop is typically an iteration over the addition or concatenation of two inputs, a matrix multiplication and a nonlinear function.

The following tasks are those RNNs perform well:

- ➢ speech recognition
- ➢ Machine translation
- ➢ Name entity recognition
- ➢ Word prediction
- ➢ Music generation

Other tasks that RNNs are effective at solving are time series predictions or other sequence predictions that aren't image or tabular based.

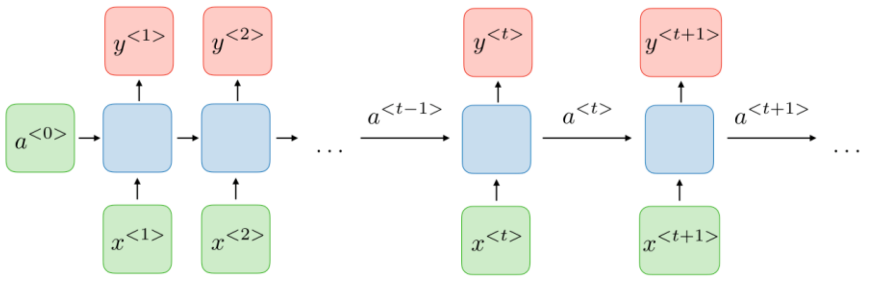
RNNs effectively have an internal memory that allows the previous inputs to affect the subsequent predictions. It's much easier to predict the next word in a sentence with more accuracy, if you know what the previous words were.

For example, the swift key keyboard software uses RNNs to predict what you are typing.

The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit. This hidden state signifies the past knowledge that the network currently holds at a given time step.

This hidden state is updated at every time step to signify the change in the knowledge of the network about the past.

The RNNs are typically as follows:

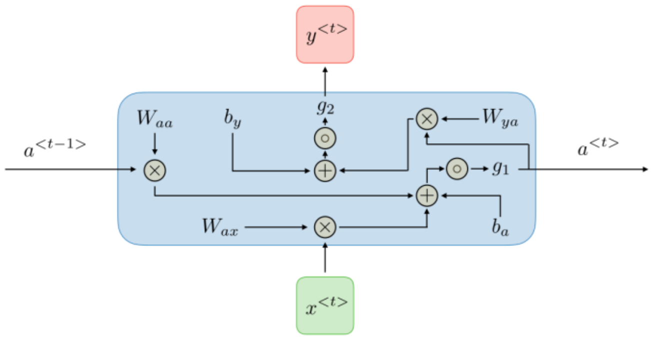


For each timestep *t*, the activation $a^{<t>}$ and the output $y^{<t>}$ are defined as follows:

$$a^{<t>} = g_1\left(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a\right)$$
$$y^{<t>} = g_2\left(W_{ya}a^{<t>} + b_y\right)$$

Where usually we define $a^{<0>}$ to be a vector of zeros, but it can have other values also. $W_{aa}, W_{ax}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and g_1, g_2 are activation functions.
we denote the hidden state and the input at time step *t* respectively as: $a^{<t>}$.

Visualisation of the model:



The most common activation functions used in RNN modules are sigmoid, hyperbolic tangent and ReLU.

Loss Function

The loss function for RNN of all-time steps is based on the loss at every time step. When we begin analyzing our input x to our desired output y throughout all the hidden part h one step at a time, we can define a loss function to explain the errors between output value and our target value as described here:

$$\mathcal{L}\left(\hat{y}, y\right) = \sum_{t=1}^{T} \ell\left(\hat{y}^{<t>}, y^{<t>}\right)$$

where the output state $\hat{y}^{<t>}$ one step at a time.

The function basically sums up every lost item t of each update step so far. This is a generic function ℓ, because this function will be based on the specific problem. (for example: MSE-mean square error, CEL-cross entropy loss etc.)

Vanishing/Exploding gradient

The Back-Propagation process can lead to the following issues.
Vanishing (tend to zero) or Exploding (tend to infinity) gradient is a problem well known in most neural networks as well in RNNs. The reason for this event is that multiplicative gradient could be either growing too much or shrinking too much.

When we have a significantly high number of layers it could cause a problem because it is difficult to capture the dependencies which occur throughout the multiplication level. On the one hand, if the matrix values consists of very large numbers , it will cause the gradient to grow more and more and eventually tends to converge in the broader sense. On the other hand, when there are many multiplications with matrices with very small values, it will cause the opposite effect and will converge to the zero matrix. A solution to this problem is long short-term memory units (referred to as LSTM) which can handle this problem.

Long Short Term Memory

An RNN has short term memory. When used in combination with Long Short Term Memory (LSTM) Gates, the network can have long term memory.
Instead of the recurring section of an RNN, an LTSM is a small neural network consisting of four neural network layers.

These are the recurring layer from the RNN with three networks acting as gates:

1. An Input gate, this controls the information input at each time step.
2. An Output gate, this controls how much information is outputted to the next cell or upward layer.
3. A Forget gate, this controls how much data to lose at each time step.

As part of the gates computation, the sigmoid function compress the values of these vectors between 0 and 1, and by multiplying them elementwise with another vector you define how much of that other vector you want to "let through".

$$\Gamma_u = \sigma\left(W_{ua} a^{<t-1>} + W_{ux} x^{<t>} + b_u\right)$$
$$\Gamma_f = \sigma\left(W_{fa} a^{<t-1>} + W_{fx} x^{<t>} + b_f\right)$$
$$\Gamma_o = \sigma\left(W_{uo} a^{<t-1>} + W_{uo} x^{<t>} + b_o\right)$$

Next, we need a candidate memory cell $\tilde{c}^{<t>}$ which has a similar computation as the previously mentioned gates but instead uses a tanh activation function to have an output between -1 to 1.

$$\tilde{c}^{<t>} = \tanh\left(W_{ca} a^{<t-1>} + W_{cx} x^{<t>} + b_c\right)$$

We introduce old memory content: $c^{<t-1>}$,which together with the introduced gates controls how much of the old memory content we want to preserve to get to the new memory content.

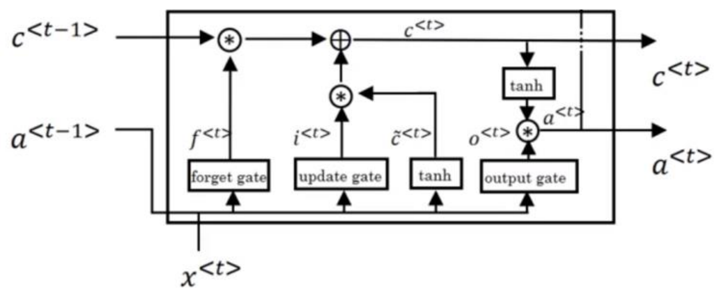
$$c^{<t>} = \Gamma_u \odot \tilde{c}^{<t>} + \Gamma_f \odot c^{<t-1>}$$

Where $\odot$ is the Hadamard product.

The last step is to introduce the computation for the hidden states $a^{<t>}$.

$$a^{<t>} = \Gamma_o \odot \tanh\left(c^{<t>}\right)$$

Visualisation of the model:



Gated Recurrent Unit

One of the lesser known, but equally effective variations, to solve the Vanishing-Exploding gradients problem is the Gated Recurrent Unit Network (GRU).

Unlike LSTM, the update gate acts as a forget and input gate. The coupling of these two gates performs a similar function as the three gates forget, input and output in an LSTM.

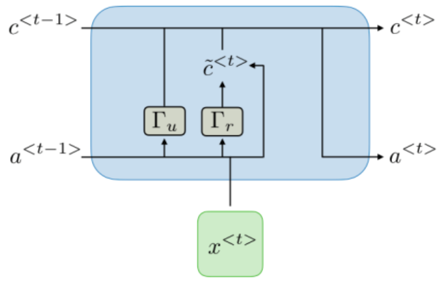
Compared to an LSTM, a GRU has a merged cell state and hidden state, whereas in an LSTM these are separate.

$$\Gamma_u = \sigma\left(W_{ua}a^{<t-1>} + W_{ux}x^{<t>} + b_u\right)$$
$$\Gamma_r = \sigma\left(W_{ra}a^{<t-1>} + W_{rx}x^{<t>} + b_r\right)$$

Now we define:

$$\tilde{c}^{<t>} = \tanh\left(W_c\left[\Gamma_r \odot a^{<t-1>}, x^{<t>}\right] + b_c\right)$$
$$c^{<t>} = \Gamma_u \odot \tilde{c}^{<t>} + \left(1 - \Gamma_u\right) \odot c^{<t-1>}$$
$$a^{<t>} = c^{<t>}$$

Visualisation:

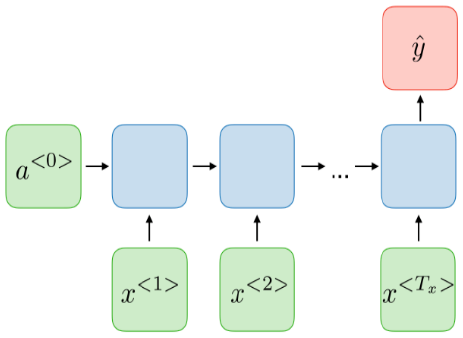


Types of RNNs

We denote T_x, T_y to be the input size and output size, respectively.

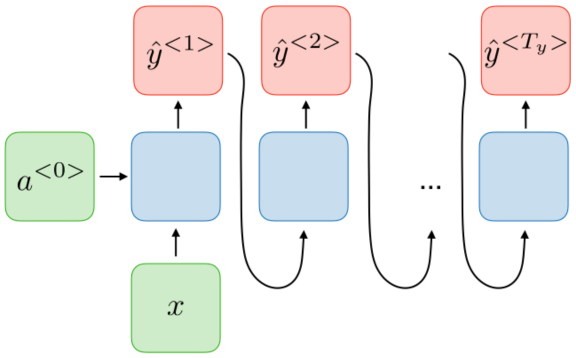
➢ Many to one, $T_x > 1, T_y = 1$

Example: Sentiment classification



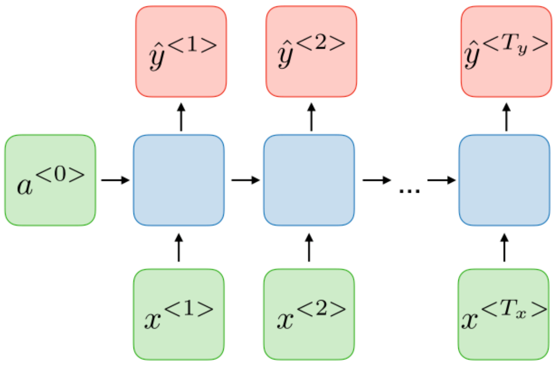
➢ One to many, $T_x = 1, T_y > 1$

Example: Music generation/Word prediction

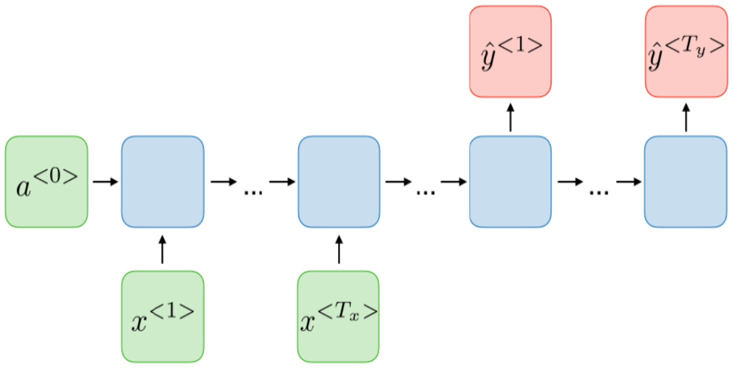


➢ Many to many, $T_x = T_y$

Example: Name entity recognition



➢ Many to many, $T_x \neq T_y$
Example: Machine translation



Deep Recurrent Neural Networks

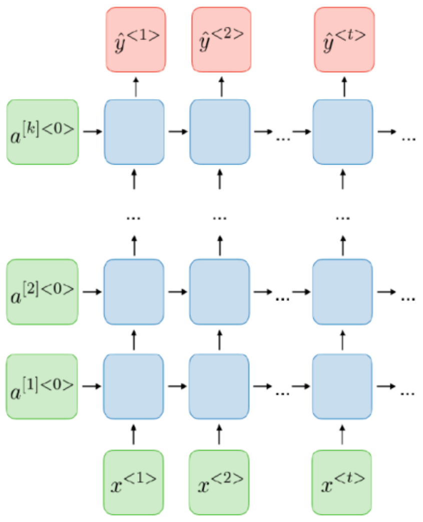
Deep Recurrent Neural Networks (DRNNs) are in theory a really easy concept.

To construct a deep RNN with k hidden layers we simply stack ordinary RNNs of any type on top of each other. Each hidden state $a^{[k]<t>}$ is passed to the next time step of the current layer $a^{[k]<t+1>}$ as well as the current time step of the next layer $a^{[k+1]<t>}$.

For the first layer we compute the hidden state as proposed in the previous models while for the subsequent layer we use the hidden state from the previous layer is treated as input.

The output is then computed where we only use the hidden state of layer k.

Visualisation of the model:



Bidirectional Recurrent Neural Networks

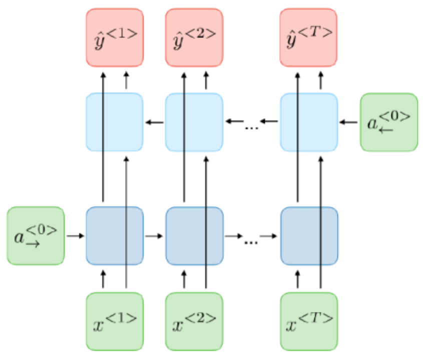
Sometimes it's not just about learning from the past to predict the future, but we also need to look into the future to fix the past.

In speech recognition and handwriting recognition tasks, where there could be considerable ambiguity given just one part of the input, we often need to know what's coming next to better understand the context and detect the present.

This does introduce the obvious challenge of how much into the future we need to look into, because if we have to wait to see all inputs then the entire operation will become costly. In addition, in cases like speech recognition, waiting till an entire sentence is spoken might make for a less compelling use case.

For example:
"He said, *Teddy* bears are on sale"
"He said, *Teddy* Roosevelt was a great president"

Visualisation of the model:



Summary RNNs

Advantages:

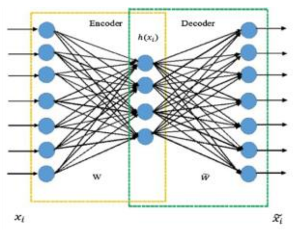
- ➢ An RNN remembers each and every information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
- ➢ Weights are shared across time.
- ➢ Possibility of processing input of any length.
- ➢ It helps to assess the impact that a given input variable has on a network output.

Disadvantages:

- ➢ Gradient vanishing and exploding problems.
- ➢ Computation being slow.
- ➢ Difficulty of accessing information from a long time ago.

Autoencoders

An autoencoder is a unsupervised artificial neural network used for learning a lower-dimensional feature representation of unlabeled data.
The model consists of two parts: an encoder and a decoder.
In between the two, there is a hidden layer sometimes referred to as the code or the latent layer. The input and the output have the same dimension and the latent layer is of a lower dimension. This is called an undercomplete autoencoder.

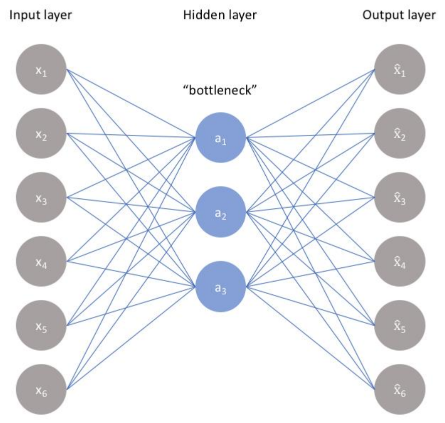


With this design, we impose a bottleneck in the network which forces a compressed knowledge representation of the original input.

If the input features were each independent of one another, this compression and subsequent reconstruction would be a very difficult task. However, if some sort of structure exists in the data (ie. correlations between input features), this structure can be learned and consequently leveraged when forcing the input through the network's bottleneck.

A bottleneck constrains the amount of information that can traverse the full network, forcing a learned compression of the input data.

Visualisation of the model:



The idea is to represent the data with only its most important features, thus reducing its dimension. This is done by the encoder.
The decoder then, reconstructs the data from the compressed code, resulting in an approximation of the original output.
The network is trained by minimizing the reconstruction error, comparing the output to the input.

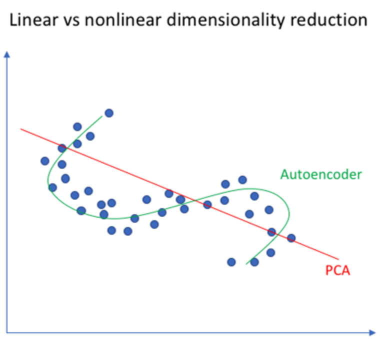
Encoder: $\phi : \mathcal{F} \mapsto \mathcal{H}$

Decoder: $\psi : \mathcal{H} \mapsto \mathcal{F}$

Loss function: $\mathcal{L}\left(x, \hat{x}\right) = \left\|x - \hat{x}\right\|^2 = \left\|x - \psi \circ \phi\left(x\right)\right\|^2$

Because neural networks are capable of learning nonlinear relationships, this can be thought of as a more powerful (nonlinear) generalization of PCA.

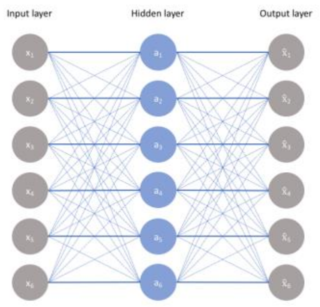
Whereas PCA attempts to discover a lower dimensional hyperplane which describes the original data, autoencoders are capable of learning non-linear manifolds.



In some models, the latent layer has the same (or a higher) number of nodes as the input and the output, meaning, there is no bottleneck. In this case, the network could easily learn the "Identity function" and thus become useless.

In the image below, we can see that the network simply memorizes the input and passes it forward.

There are different ways for dealing with this problem.



Sparse autoencoders

Sparse autoencoders offer us an alternative method for introducing an information bottleneck without requiring a reduction in the number of nodes at our hidden layers. Rather, we'll construct our loss function such that we penalize *activations* within a layer.

There are different ways by which we can impose this sparsity constraint. They involve measuring the hidden layer activations for each training batch and adding some term to the loss function in order to penalize excessive activations.

For example, L1 regularization:

$$\mathcal{L}\left(x, \hat{x}\right) + \lambda \cdot \sum_i \left|a_i^{(h)}\right|$$

Denoising Autoencoders

Differently from sparse autoencoders or undercomplete autoencoders that constrain representation, Denoising autoencoders try to achieve a good representation by changing the reconstruction criterion.

The idea is to slightly corrupt the input data but still maintain the uncorrupted data as our target output. Then, we feed forward the information normally, but calculate the loss function by comparing the output to the uncorrupted data.

This way, we create a latent representation of the uncorrupted data.
This restrict the network from learning the identity function (because we compare the output to the uncorrupted data).

This model is useful for cleaning and restoring noisy data.

Contractive Autoencoder

Contractive autoencoder adds an explicit regularizer to their loss function that forces the model to learn a function that is robust to slight variations of input values.

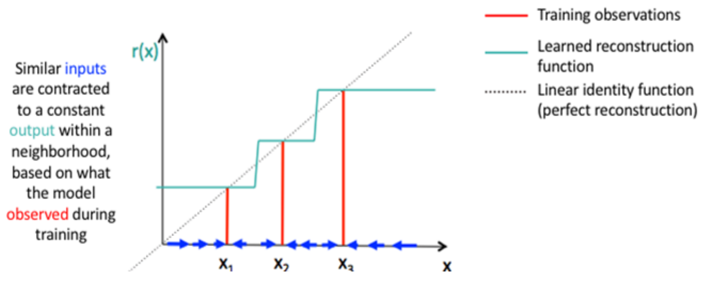
$$\mathcal{L}\left(x, \hat{x}\right) + \lambda \cdot \sum_{i=1}^{m} \left\| \nabla_{x} a_{i}^{(h)}\left(x\right) \right\|^{2}$$

For m observations and n hidden layer nodes, we can calculate these values as follows.

$$\left\| A \right\|_{F} = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} \left| a_{ij} \right|^{2}}$$

$$\nabla_{x} a_{i}^{(h)}\left(x\right) = \begin{bmatrix} \frac{\partial a_{1}^{(h)}\left(x\right)}{\partial x_{1}} & \cdots & \frac{\partial a_{1}^{(h)}\left(x\right)}{\partial x_{m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_{n}^{(h)}\left(x\right)}{\partial x_{1}} & \cdots & \frac{\partial a_{n}^{(h)}\left(x\right)}{\partial x_{m}} \end{bmatrix}$$

We can explicitly train our model, requiring that the derivative of the hidden layer activations are small with respect to the input. In other words, for small changes to the input, we should still maintain a very similar encoded state.
The name contractive comes from the fact that the Contractive autoencoder is encouraged to map a neighborhood of input points to a smaller neighborhood of output points.


Similar inputs are contracted to a constant output within a neighborhood, based on what the model observed during training
r(x)
x₁
x₂
x₃
x
Training observations
Learned reconstruction function
Linear identity function (perfect reconstruction)

ANN Advantages

ANN has the ability to work with incomplete knowledge. After we finish the training process, our data may predict a 'good' output despite the incomplete information. The total loss of performance depends on the significance of the missing data.

The network can to be fault tolerant. If one or more cells of the network are corrupted, it will not prevent it from generating an output.

In addition, ANN has a distributed memory. The network's success is a derivative of the examples that are given to the network. Furthermore, ANN has the capability of parallel processing. The network has enough numerical strength, so that he can perform more than one process at the same time.

Finally, ANN has the ability to make machine learning. The network can learn from examples and make a decision based on similar data.

Time Series

Applied Mathematics

Contents

- ❖ Introduction
- ❖ Components of Time Series
- ❖ Properties
- ❖ Time Series Forecasting Using Statistical Models
- ❖ Time series forecasting with Neural Networks
- ❖ Time Lagged Neural Networks (TLNN)
- ❖ Seasonal Artificial Neural Networks (SANN)
- ❖ Long Short Time Series (LSTM)
- ❖ Time series classification with Convolutional Neural Network (CNN)
- ❖ Denoising Autoencoders (DAE)

Introduction

Time series is a sequence of data points in chronological sequence, most often gathered in regular intervals.
Time series adds an explicit order dependence between observations: a time dimension.
In addition, time series forecasting is an important area of machine learning.
Forecasting involves making models fit on historical data and using them to predict future observations.
An important distinction in forecasting is that the future is completely unavailable and must only be estimated from what has already happened.
Examples: average daily temperatures, stock value at the end of each business day, hourly electricity demand.

Components of Time Series

Time series analysis provides a body of techniques to better understand a dataset. Perhaps the most useful of these is the decomposition of a time series into 4 constituent parts: trend, seasonality, noise or randomness, and the cyclic movements. not all time series data will include every one of these time series components. For instance, audio files that are taken in sequence are examples of time series data, however they won't contain a seasonal component. On the other hand, most business data will likely contain seasonality.

❖ <u>Cyclic Movements:</u>
  These are long term oscillations occurring in a time series. The duration of a cycle depends on the type of business or industry being analyzed.

❖ Noise:

All time series data will have noise or randomness in the data points. The optional variability in the observations that cannot be explained by the model.

❖ Seasonality:

If there are regular and predictable fluctuations in the series that are correlated with the calendar, then the series includes a seasonality component. It's important to note that seasonality is domain specific, for example real estate sales are usually higher in the summer months.

❖ Trend:

The optional and often linear increasing or decreasing behavior of the series over time. An example of a trend would be a long term increase in a companies sales data.

Properties

❖ Stationarity:

We will define a time series to be stationary if statistical attributes such as means and covariances do not change with time. For a time series $\{x_t, t \in \mathbb{Z}\}$ to be called stationary, it needs to have the following attributes:

1. $var(x_t) < \infty,\ \forall t \in \mathbb{Z}$
2. $\mathbb{E}(x_t) = \mu,\ \forall t \in \mathbb{Z}$
3. $cov(x(\alpha), x(\beta)) = cov(x(\alpha + \gamma), x(\beta + \gamma)),\ \forall \alpha, \beta, \gamma \in \mathbb{Z}$

A time series is called strictly stationary if the distributions of $(x_{t_1}, \dots, x_{t_k})$ and $(x_{t_{1+h}}, \dots, x_{t_{k+h}})$ are all the same $\forall k, t_1, \dots, t_k, h \in \mathbb{Z}$.

❖ Multiplicative and Additive models:

Multiplicative: $Y(t) = T(t) \times S(t) \times C(t) \times N(t)$

Additive models: $Y(t) = T(t) + S(t) + C(t) + N(t)$

Here $Y(t)$ is the observation and $T(t), S(t), C(t), N(t)$ are respectively the trend, seasonal, cyclical and noise at time t.

❖ Gaussian time series:

The time series is said to be a Gaussian time series if all finite-dimensional distributions are normal.

❖ Integrated time series models:

A time series is called integrated if it satisfies $y_{t+1} - y_t = \varepsilon_t$ where ε_t is a stationary series.

❖ White Noise series:

The time series is said to be a white noise with mean μ and variance σ^2, written $\{x_t\} \sim WN(\mu, \sigma^2)$ if $\mathbb{E}(x_t) = \mu, var(x_t) < \infty$ and $cov(x_t, x_s)$ exist. White Noise time series does not have to be normal, it can be other distributions.

❖ IID Noise series:

The time series is said to be a white noise with mean μ and variance σ^2, written $\{x_t\} \sim IID(\mu, \sigma^2)$. IID Noise time series has independent and identically distributed samples, so each sample must come independently from the same probability density function which is not a requirement for WN.

Time series forecasting

Predicting the future values of the time series using a current information set.

Current information set: current and past values of the series and other “exogenous” series.

Information set definition:

x_t - exogenous series values at time t

y_t - series value at time t

$$\hat{y}_{t+K|t} = f(y_t, y_{t-1}, \dots, y_1, x_t, x_{t-1}, \dots, x_1)^T$$

Forecasting models

Naive, rule based models:

Constant value: $\hat{y}_{t+1} = a$

Last value: $\hat{y}_{t+1} = y_t$

Rolling average: $\hat{y}_{t+1} = \frac{1}{M}\sum_{s=t-M}^{t} y_s$

Statistical models:

Auto regressive model:

$$\hat{y}_{t+K} = (y_t, y_{t-1}, \dots, y_{t-M})^T \beta$$

M is memory length, β predictor weights, $\hat{y}_{t+K}$ is linear function of inputs window $y_{t-M:t}$.

Time Series Forecasting Using Statistical Models

The Autoregressive Moving Average (ARMA) Models

An ARMA(p, q) model is a combination of AR(p) and MA(q) models and is suitable for univariate time series modeling. In an AR(p) model the future value of a variable is assumed to be a linear combination of p past observations and a random error together with a constant term. Mathematically the AR(p) model can be expressed as:

Here y_t and ε_t are respectively the actual value and random error at time period t, $\varphi_i (i = 1,2, \dots, p)$ are model parameters and c is a constant. The integer constant p is known as the order of the model. Sometimes the constant term is omitted for simplicity.

Just as an AR(p) model regress against past values of the series, an MA(q) model uses past errors as the explanatory variables. The MA(q) model is given by:

$$y_t = \mu + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j} + \varepsilon_t = \mu + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + \cdots + \theta_q \varepsilon_{t-q} + \varepsilon_t$$

Here μ is the mean of the series, $\theta_j (j = 1,2, \dots, q)$ are the model parameters and q is the order of the model. The random shocks are assumed to be a sequence of independent and identically distributed (i.i.d) random variables that follow the normal distribution: $\varepsilon_t \sim N(0, \sigma^2)$. Fitting an MA model to a time series is more complicated than fitting an AR model because in the former one the random error terms are not fore-seeable.

Autoregressive (AR) and moving average (MA) models can be effectively combined together to form a general and useful class of time series models, known as the ARMA models where p is the order of the AR part and q is the order of the MA part.

Mathematically an ARMA(p, q) model is represented as:

$$y_t = \sum_{i=1}^{p} \varphi_i y_{t-i} + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j} + \varepsilon_t$$

Usually ARMA models are manipulated using the lag operator notation.

The **lag or backshift operator**, L, is defined as $Ly_t = y_{t-1}$ for a time series $\{y_t\}$.

Furthermore, we will define our lag operator for linear combinations by:

$$\forall c_1, c_2 \in \mathbb{R}: L\left(c_1 y_{t_1} + c_2\, y_{t_1}\right) = c_1 y_{t_1-1} + c_2 y_{t_2-1}.$$

In addition we will define powers of L to be:

$$L^k y_t = L^{k-1}(Ly_t) = L^{k-1} y_{t-1} = L^{k-2} y_{t-2} = \cdots = y_{t-k}$$

and for linear combinations to be: $\left(\alpha L^k + \beta L^m\right) y_t = \alpha y_{t-k} + \beta y_{t-m}.$

<u>Differencing operator</u>

Operators can be defined on the operators. We will define the Differencing operator in terms of L as follows:

$$\nabla y_t = (1 - L) y_t = y_t - y_{t-1}$$

This operator deals with linear combinations like the lag operator.

Additionally we will define again powers of this operator as well:

$$\nabla^2 y_t = \nabla(\nabla y_t) = \nabla(y_t - y_{t-1}) = y_t - y_{t-1} - (y_{t-1} - y_{t-2}) = y_t - 2y_{t-1} + y_{t-2}$$

This leads to the following conclusion:

$$\nabla^k y_t = W(y_t, y_{t-1}, \dots, y_{t-k})^T$$

where W is the k^{th} row in Pascal triangle where each even index changes sign.

Polynomials of lag operator or lag polynomials are used to represent ARMA models as follows:

AR(p): $\varepsilon_t = \varphi(L)y_t$

MA(q): $y_t = \theta(L)\varepsilon_t$

ARMA(p, q): $\varphi(L)y_t = \theta(L)\varepsilon_t$

Where, $\varphi(L) = 1 - \sum_{i=1}^{p} \varphi_i L^i$ and $\theta(L) = 1 + \sum_{j=1}^{q} \theta_j L_j$.

We can see:

$$\varphi(L)y_t = \theta(L)\varepsilon_t \Leftrightarrow \left(1 - \sum_{i=1}^{p} \varphi_i L^i\right) y_t = \left(1 + \sum_{j=1}^{q} \theta_j L_j\right) \varepsilon_t \Leftrightarrow y_t - \sum_{i=1}^{p} \varphi_i L^i y_t = \varepsilon_t + \sum_{j=1}^{q} \theta_j L_j \varepsilon_t$$

$$\Leftrightarrow y_t = \sum_{i=1}^{p} \varphi_i y_{t-i} + \sum_{j=1}^{q} \theta_j \varepsilon_{t-j} + \varepsilon_t \ \blacksquare$$

Autoregressive Integrated Moving Average (ARIMA) Models

The ARMA models, described above can only be used for stationary time series data. However, in practice many time series such as those related to socio-economic and business show non-stationary behavior. Time series, which contain trend and seasonal patterns, are also non-stationary in nature. For this reason, the ARIMA model is proposed, which is a generalization of an ARMA model to include the case of non-stationarity as well. In ARIMA models a non-stationary time series is made stationary by applying finite differencing of the data points. The mathematical formulation of the ARIMA(p, d, q) model using lag polynomials is given by:

$$\varphi(L)(1 - L)^d y_t = \theta(L)\varepsilon_t$$

❖ Here, p, d and q are integers greater than or equal to zero and refer to the order of the autoregressive, integrated, and moving average parts of the model respectively.

❖ The integer d controls the level of differencing. Generally $d = 1$ is enough in most cases. When $d = 0$, then it reduces to an ARMA(p, q) model.

❖ ARIMA(0,1,0), i.e. $1(1-L)y_t = 1\varepsilon_t \Leftrightarrow y_t - Ly_t = \varepsilon_t \Leftrightarrow y_t = y_{t-1} + \varepsilon_t$ is a special one and known as the Random Walk model.
It is widely used for non-stationary data, like economic and stock price series.

Seasonal Autoregressive Integrated Moving Average (SARIMA) Models

The ARIMA model is for non-seasonal non-stationary data. A generalization of this model is the Seasonal ARIMA (SARIMA) model. In this model seasonal differencing of appropriate order is used to remove non-stationarity from the series. As most seasonal time series display increasing trend and/or seasonal variations, both seasonal and nonseasonal differencing are often used to stabilize the time series.
Seasonal ARIMA model building requires the specification of differencing orders (d, D) and the orders of both nonseasonal and seasonal AR and MA operators (p, q, P, Q) as well as the estimation of model parameters in the AR and MA operator polynomials.

The SARIMA $(p, d, q) \times (P, D, Q)_s$ model in terms of lag polynomials can be expressed as:

$$\phi_p(L)\boldsymbol{\Phi}_P(L^s)(1-L)^d(1-L^s)^D y_t = \theta_q(L)\boldsymbol{\Theta}_Q(L^s)\varepsilon_t$$

where: $\phi_p(L) = 1 - \phi_1 L - \phi_2 L^2 - \cdots - \phi_p L^p$

$\boldsymbol{\Phi}_P(L^S) = 1 - \boldsymbol{\Phi}_s L^s - \boldsymbol{\Phi}_{2s} L^{2s} - \cdots - \boldsymbol{\Phi}_{Ps} L^{Ps}$

$\theta_q(L) = 1 - \theta_1 L - \theta_2 L^2 - \cdots - \theta_q L^p$

$\boldsymbol{\Theta}_Q(L^S) = 1 - \boldsymbol{\Theta}_s L^s - \boldsymbol{\Theta}_{2s} L^{2s} - \cdots - \boldsymbol{\Theta}_{Qs} L^{Qs}$

and s is the seasonal length ($s = 4$ for quarterly data and $s = 12$ for monthly data), L is the lag operator defined by $L^k y_t = y_{t-k}$ and $\varepsilon_t \sim N(0, \sigma^2)$. $(1-L)^d$ and $(1-L^s)^D$ are the nonseasonal and seasonal differencing operators respectively.

Time series forecasting with Neural networks

In many cases, a time series is quite complex and cannot be accurately fitted with a linear model. We'll need a more complex, non-linear model.

Feed forward networks are a good place to start.
The problem is that if we simply feed the network with the past values of the series as the input vector, the model doesn't account for time order.
One way to deal with that is to use a convolutional layer (or more than one) before the fully connected layers. The convolution takes a temporally coherent input and is able to extracts meaningful features. Those features are then passed on to the fully connected layers for classification.

Time Lagged Neural Networks (TLNN)

ANNs can be truly referred as model free structures because they do not need any prior knowledge about the intrinsic data generating process. ANNs are also favored due to their distinctive ability of nonlinear modeling with remarkable accuracies.

In the feed forward network (FNN), the input nodes are the successive observations of the time series, i.e. the target y_t is a function of the values $y_{t-i}, i = 1,2,\dots,p$ where p is the number of input nodes. Another variation of FNN is the TLNN architecture that also widely used. In TLNN, the input nodes are the time series values at some particular lags. For example, a typical TLNN for a time series, with seasonal period $s = 12$ can contain the input nodes as the lagged values at time $t-1$, $t-2$ and $t-12$. The value at time t is to be forecasted using the values at lags 1, 2 and 12.

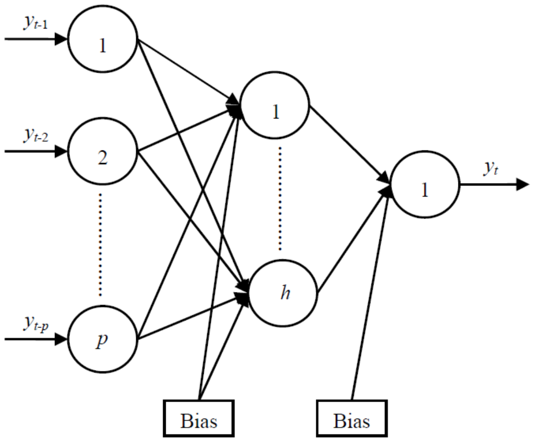
In addition, there is a constant input term, the bias, and this is connected to every neuron in the hidden and output layer. In a fully connected TLNN model with p input, h hidden and a single output node, the general prediction equation for computing a forecast may be written as:

$$\hat{y}_t = \Psi\left(\alpha_0 + \sum_{j=1}^{h} \alpha_j \Phi\left(\beta_{0j} + \sum_{i=1}^{p} \beta_{ij} y_{t-i}\right)\right)$$

Here, the selected past observations y_{t-i} are the input terms, $\alpha_j, \beta_{ij}, i = 1,2,\dots,p; j = 1,2,\dots,h$ are the connection weights, α_0, β_{0j} are the bias terms. Φ and Ψ are the hidden and output layer activation functions respectively.

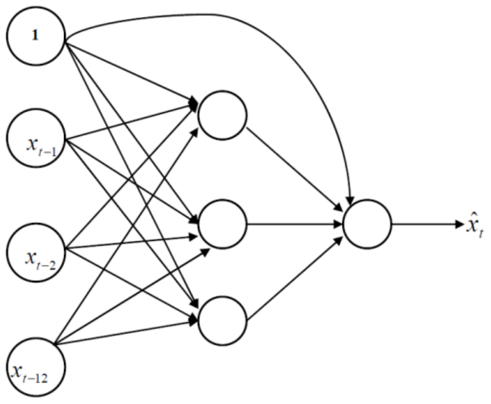
We use the notation $NN(1,2,\dots,p;h)$ to denote the TLNN with inputs at lags $1,2,\dots,p$ and h hidden neurons.

A $p \times h \times 1$ TLNN model with the arrows as the network weights:



For example, TLNN architecture for monthly data, $NN(1,2,3;3)$ model:

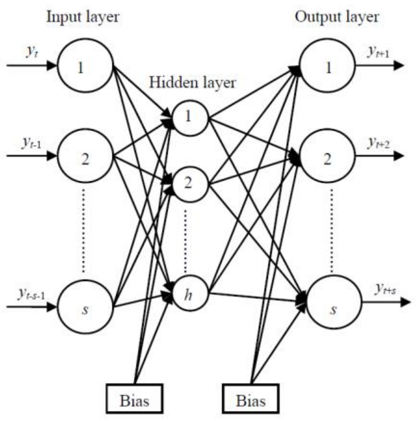
Here the bias is taken as one.



Seasonal Artificial Neural Networks (SANN)

The traditional statistical methods for seasonal time series forecasting suffer from various drawbacks. These include the assumption of linearity, fixed model form or seasonal differencing, etc. The SANN structure is proposed in order to improve the forecasting performance of ANNs for seasonal time series data. The proposed SANN model does not require any preprocessing of raw data. Also SANN can learn the seasonal pattern in the series, without removing them, contrary to some other traditional approaches, such as SARIMA. It is also simple to design and implement because the network structure is already specified and only the number of appropriate hidden nodes is to be determined.
In this model, the seasonal parameter s is used to determine the number of input and output neurons.

The i^{th} and $(i+1)^{th}$ seasonal period observations are respectively used as the values of input and output neurons in this network structure. Each seasonal period is composed of a number of observations.
An SANN structure $s \times h \times s$ can be shown as:



Mathematical expression for the output of the model is:

$$y_{t+l} = \alpha_l + \sum_{j=1}^{h} w_{jl} \Phi \left(\theta_j + \sum_{i=0}^{s-1} v_{ij} y_{t-i} \right) \quad , \forall\, t$$

Here $l = 1,2, \dots, s;\ j = 1,2, \dots, h;\ i = 0,1, \dots, s-1$

y_{t+l} are the predictions for the future s periods and y_{t-i} are the observations of the previous s periods. v_{ij} are weights of connections from input nodes to hidden nodes and w_{jl} are weights of connections from hidden nodes to output nodes. Also α_l and θ_j are weights of bias connection and Φ is the activation function.

Thus while forecasting with SANN, the number of input and output neurons should be taken as 12 for monthly and 4 for quarterly time series. The appropriate number of hidden nodes can be determined by performing suitable experiments on the training data.

The successful forecasting through an ANN largely depends on the appropriate model designing which is however not a trivial task. The prime benefit of using SANN is that with this model, the architecture selection actually boils down to the selection of the optimal number of hidden nodes only.

Usually, the number of hidden nodes is selected through the widely popular Bayesian Information Criterion (BIC).

The BIC effectively controls the network size by penalizing for each increase in the number of network parameters. Out of several feedforward SANN structures, the one which minimizes the BIC is chosen to be the optimal one. It should however be noted that the use of BIC is popular in feedforward neural networks only.

For the $s \times h \times s$ SANN model, the BIC is mathematically given by the formula:

$$BIC \stackrel{\text{def}}{=} N_{s,h} + N_{s,h}\log(n) + n\log\left(\frac{S(\boldsymbol{W})}{n}\right)$$

where, $N_{s,h} = s + h(2s+1)$ is the number of total network parameters, $n = N - s$ is the number of effective observations, N being the size of the training set, $\boldsymbol{W}$ is the space of all connection weights and biases and $S(\boldsymbol{W})$ is the network misfit function which is commonly taken as the SSE (Sum Squared Error).
Here the log is a natural base logarithm.

Long Short Time Series (LSTM)

LSTM - Long Short Time Series is a RNN architecture to process more than a single data point but the entire data.
An LSTM unit is assembled by three networks acting as gates- input gate, output gate and a forget gate.
In our goal of predicting time series sequences we will use LSTM models because of their advantage in handling the long-term dependencies within the time series data.
LSTM deals with the exploding and vanishing gradient problems.
LSTMs help us preserve the error that can be backpropagated through time and layers. It allows recurrent neural networks to learn over many time steps. LSTMs contain data alongside the recurrent network in a gated cell.

These gates help us make informed choices on what data to read and which data to return via analog gates that open and close. Most information which we receive is not perfect. It is usually insufficient and delayed and the gates act on the signals they receive, limiting the data getting in, based on its score. After that, they filter the passed data with their own set of weights which are fixed throughout the RNN learning process. Additionally, the forget cell helps use less storage by deleting information we no longer need. For example, if we work on several files with no connection between them and we reach the end of a file, it is no longer useful data to store.

For conclusion the LSTM learns when to allow data to enter, leave or be deleted through the iterative process of making guesses, backpropagating error, and adjusting weights via gradient descent.

In our goal of time series prediction we will use the LSTM model for two roles in our network, The LSTM model will be used for an encoder network and for a decoder network.

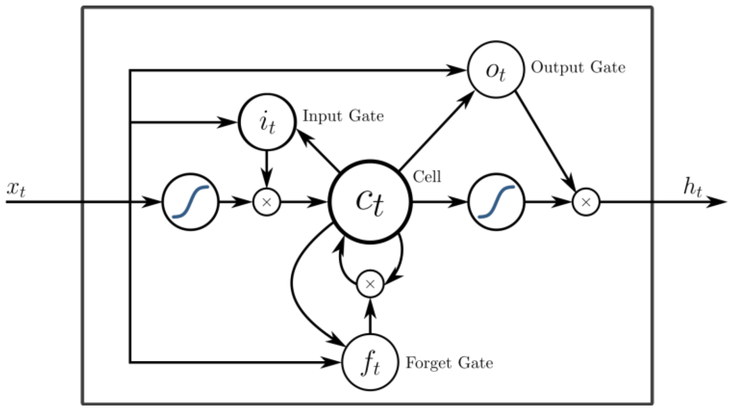
The update step to the LSTM networks will be computed as previously presented in the last presentation (Artificial Neural Network):

$$\begin{aligned}\Gamma_u &= \sigma\left(W_{ua}a^{<t-1>} + W_{ux}x^{<t>} + b_u\right)\\ \Gamma_f &= \sigma\left(W_{fa}a^{<t-1>} + W_{fx}x^{<t>} + b_f\right)\\ \Gamma_o &= \sigma\left(W_{uo}a^{<t-1>} + W_{uo}x^{<t>} + b_o\right)\end{aligned}$$

The encoder networks embeds the historical data $\{y_1, y_2, y_3, \dots, y_n\}$ into our desired embedded space.

The decoder network ‘translates’ the embedded representation into the predictions of our entire time series prediction network.

We can see here an example of LSTM architecture:



We implement the input data x_t in a regular feed-forward and then we feed it forward to our gates or compute on it an activation function (sigmoid, ReLU, etc.) and feed it to our memory cell c_t.
The arrows from the memory cell represent the contributions of the activation of the memory cell c_{t-1} and not c_t. In other words, the gates calculate their functions at time step t considering the memory cell at time $t-1$ and in that we achieve better handling of the long-term dependencies within the time series data.

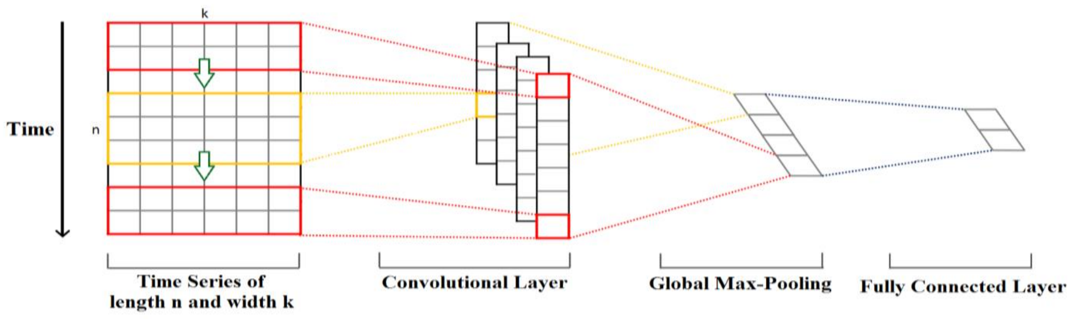
Time Series classification with CNN

CNN's can be a powerful tool for time series classification. There are many ways to design a model for this task.

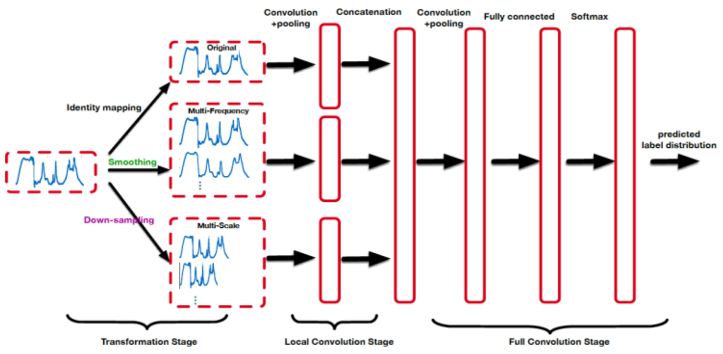
Let's look at a simple example. Consider a multivariate time series of length n and width k. N is the number of time steps and k is the number variables (for example: temperature, wind, humidity etc.) The convolution kernels always have the same width as the time series, while their length can vary. This way, the kernel moves in one direction from the beginning of a time series towards its end, performing convolution.

We use various kernels with different sizes and we get a new "filtered" time series for every kernel. Depending on the length of the kernel, different aspects, properties, "features" of the initial time series get captured in each of the new filtered series.

Next, we apply global max pooling for each vector, resulting in one vector composed of the maximum value of each vector. This new feature vector is fed to a fully connected layer for classification.



Now, let's see a more complex, multiscale CNN architecture.
The framework of this network consists of 3 consecutive stages: transformation, local convolution, and full convolution.



<u>**Transformation**</u>

On this stage different transformations are applied to the original time series on 3 separate branches. The first branch transformation is identity mapping.
The second branch transformation is smoothing the original time series with a moving average with various window sizes. This way, several new time series with different degrees of smoothness are created. The idea behind this is that each new time series consolidates information from different frequencies of the original data.
Finally, the third branch transformation is down-sampling the original time series with various down-sampling coefficients. The smaller the coefficient, the more detailed the new time series is, and, therefore, it consolidates information about the time series features on a smaller time scale.

Down-sampling with larger coefficients results in less detailed new time series which capture and emphasize those features of the original data that exhibit themselves on larger time scales.

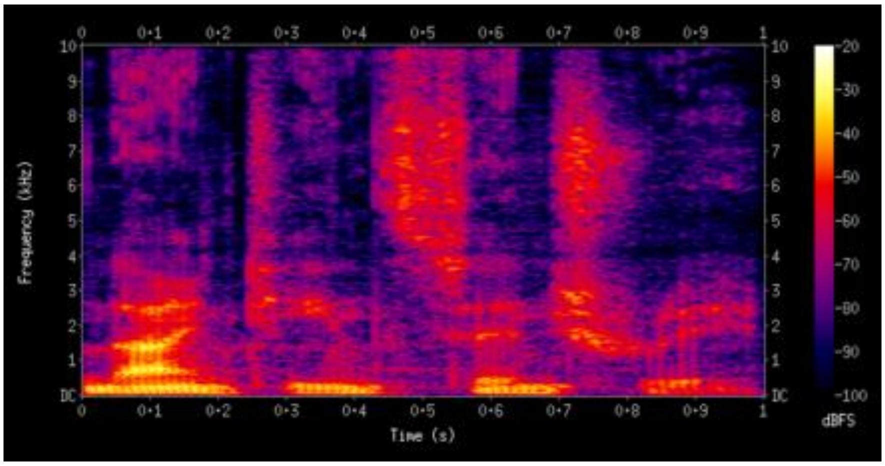
Local Convolution

On this stage the 1-D convolution with different filter sizes that we discussed earlier is applied to the time series. Each convolutional layer is followed by a max-pooling layer. This stage is called local convolution because each branch is processed independently.

Full Convolution

On this stage all the outputs of the local convolution stage from all 3 branches are concatenated. Then several more convolutional and max-pooling layers are added. After all the transformations and convolutions, you are left with a flat vector of deep, complex features that capture information about the original time series in a wide range of frequency and time scale domains. This vector is then used as an input to fully connected layers with Softmax function on the last layer.

Another way of using CNN's for time series is by representing them in the form of images (for example, by using their spectrograms) and applying a normal 2D convolution on them.



Denoising Autoencoders (DAE)

DAE - Denoising Autoencoders is a way to handle slightly corrupt data.
Time series is often inconstient and in forecasting models, we do not want to use training data which is problematic, it makes our goal to minimize the harm that the noise is creating to our data.
If we use a data which is corrupt or a small percentage is missing, DAE will provide us with a 'fixed' data to input into our NN preinitialization, for example: Salt-and-pepper noise or failed data recovery.
The idea is to train our DAE with parts of the data that we know are trustworthy. This way, we can later use it to fix the corrupted data before feeding it to our model.
(For more information on DAE go to the last presentation - Artificial Neural Network))

Time Series – part II
Implementation

Applied Mathematics

Common methods frequently used in the field of time series forecasting

There are 12 different classical time series forecasting methods commonly used in forecasting time series:

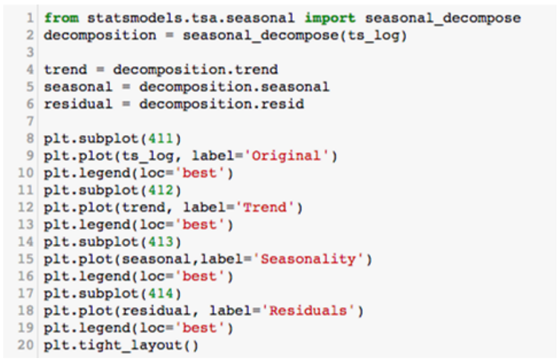
1. Autoregression (AR)
2. Moving Average (MA)
3. Autoregressive Moving Average (ARMA)
4. Autoregressive Integrated Moving Average (ARIMA)
5. Seasonal Autoregressive Integrated Moving-Average (SARIMA)
6. Seasonal Autoregressive Integrated Moving-Average with Exogenous Regressors (SARIMAX)
7. Vector Autoregression (VAR)
8. Vector Autoregression Moving-Average (VARMA)
9. Vector Autoregression Moving-Average with Exogenous Regressors (VARMAX)
10. Simple Exponential Smoothing (SES)
11. Holt Winter's Exponential Smoothing (HWES)
12. Time-Delay Neural Network (TDNN)

How can we model our time series?

Prophet is a forecasting tool designed for analyzing time-series and display patterns on different time scales such as yearly, weekly and daily.
The TDNN forecasting works in most cases for stationary data for forecasting short time periods. Prophet is a way to crack the cases which TDNN can't handle by building a better tool to deal with these cases. Prophet handled these cases better by using knowledge from Bayesian statistics, including seasonality, the inclusion of domain knowledge, and confidence intervals to forecast data by better estimating the risk with the given data.

Visualization of the time-series data

We can use a tool called time-series decomposition that allows us to decompose our time series into the three components we talked about in the last presentation: trend, seasonality, and noise. An example in python:

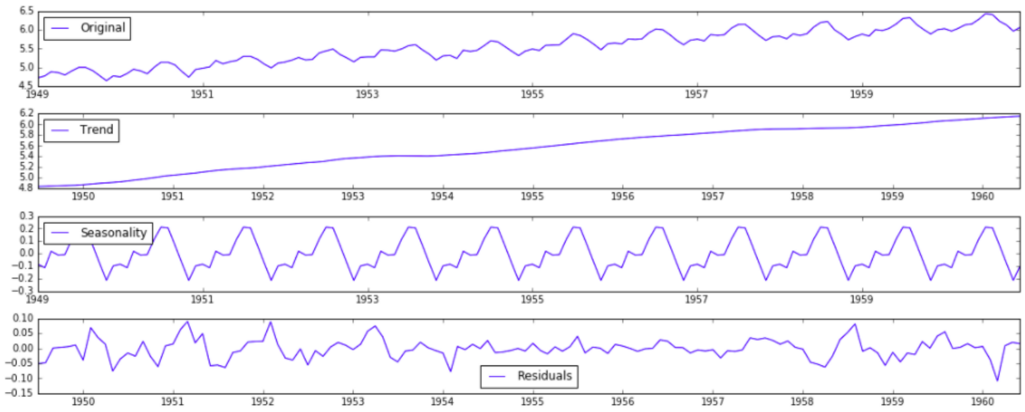


```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(ts_log)

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(ts_log, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```

The result of the previous example:



Stationary

Most Time-Series forecasting models assume that the series is stationary. How can we make our Time-Series Stationary?

There are several methods that work well in some cases and other methods that work well in different cases while the other still don't achieve the wanted results. We will explain the 2 methods used in most cases.

1. Differencing

In short, we are given n samples: $\mathrm{X} = \{x_1, x_2, x_3, x_4, \dots, x_n\}$. We assume this time series is not stationary. We create a new time series by taking the difference between consecutive data points. The time series with difference of degree 1 becomes:

$\tilde{X} = \{x_2 - x_1, x_3 - x_2, \dots., x_n - x_{n-1}\}$.

We will then check if there is any improvement in X autocorrelation.

If not, we can try a second, a third or even higher order differencing.

We need to be careful with this method because the more we difference, the analysis and all of the calculations and conclusions are going to be harder to execute.

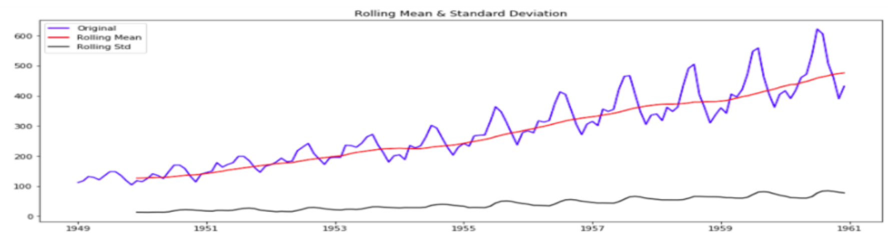
2. Transformation

In many time series the variance is not constant and it is not stable. Transformation is a way normally suggested only when Differencing doesn't work.

The most common transformation is the log transform. Other known transformations include power transform and square root transformation.

An example in python:

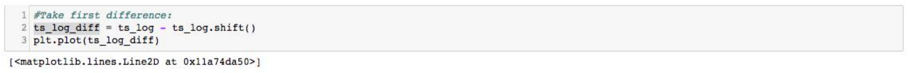
Let's assume that out series is not stationary.



```
Results of Dickey-Fuller Test:
Test Statistic                   0.815369
p-value                          0.991880
#Lags Used                      13.000000
Number of Observations Used    130.000000
Critical Value (5%)             -2.884042
Critical Value (1%)             -3.481682
Critical Value (10%)            -2.578770
dtype: float64
```

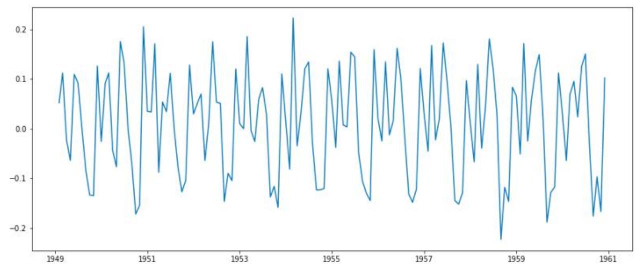
This series is not stationary because the mean values are increasing while the standard deviation is small throughout the time series data.

We will transform the series with log transformation and we will take the difference of the value at a particular time with that of the previous time by 1.



```
#Take first difference:
ts_log_diff = ts_log - ts_log.shift()
plt.plot(ts_log_diff)
```

```
[<matplotlib.lines.Line2D at 0x11a74da50>]
```

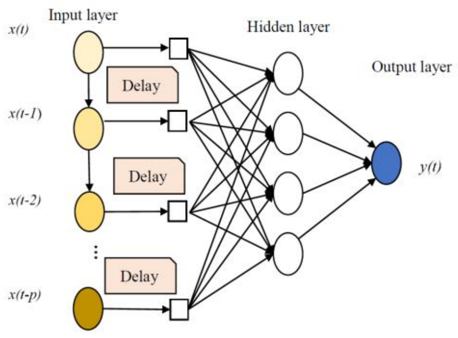


We can see here that we achieved a stationary time series to represent the model, in order to get the wanted result we will simply apply the Inverse function.

Time-Delay Neural Network (TDNN)

TDNNs can be referred to as feedforward neural networks, except that the input weights have a delay element associated with them. The time series data is often used in the input and the finite responses of the network can be captured. Accordingly, a TDNN can be considered as an ANN architecture whose main purpose is to work on sequential data.
As we know, In the feed forward network (FNN), the input nodes are the successive observations of the time series, i.e. the target $x(t+1)$ is a function of the values $x(t-i)$, $i = 0,1,2,\dots,p$ where p is the number of input nodes.
In TDNN, the input nodes are the time series values at some particular lag. A TDNN has multiple layers and inter-connection between units in each layer to ensure the ability to learn complex nonlinear decision surfaces.

The next figure shows the architecture of a TDNN. The structure of the TDNN includes an input layer, one or more hidden layers, and an output layer. Each layer contains one or more nodes determined through a trial and error process of the given data, as there is no theoretical basis. In here, the figure present one hidden layer in the TDNN's structure.



In addition, as shown, the network input layer utilized the delay components embedded between the amounts of input-units to attain the time-delay. Each node had its own values and through the network computation, achieves the output results. Under the input–output relationship function of the network, if the output result is the next time prediction of the input x, there must be a certain relationship between present and future.
This is given by $y = x(t+1) = H[x(t), x(t-1), \dots, x(t-p)]$. Consequently, the TDNN is to seek the activation function H of the input–output in the network. This is given by $\Phi_j = \sum_{i=0}^{p} w_{ij} x(t-i) + \theta_j$ and $y(t) = \sum_j w_{jk} \mathbf{\Psi}\left(\Phi_j\right)$, where Φ_j and $y(t)$ are the function at the input and output layers, respectively; p is the number of tapped delay nodes. w_{ij} is the weight of the i^{th} neurons in the input layer into the j^{th} neurons in the hidden layer; and θ_j is the bias weight of the j^{th} neurons.

The activation function $\boldsymbol{\Psi}$ represent ReLU or a nonlinear sigmoid.

We define the function as: $sigmoid \overset{\text{def}}{=} \frac{1}{1+e^{-x}}$; $ReLU \overset{\text{def}}{=} \max(0, x)$.

The precise architecture of TDNNs (time-delays, number of layers) is mostly determined depending on the classification problem and the most useful context sizes. The delays or context windows are chosen specific to each application.

In time delay networks , the connections have time delays of different length. Such a time delay postpones the forwarding of a unit's activation to another unit. When using multiple time delays, these networks can be trained to deal with a sequence of past time series elements. At each time step, a single sequence element is fed into the input, but the network's forecast takes preceding sequence elements into account.

One of the common uses is in detecting temporal patterns for stock market prediction tasks.

Another model - CNN

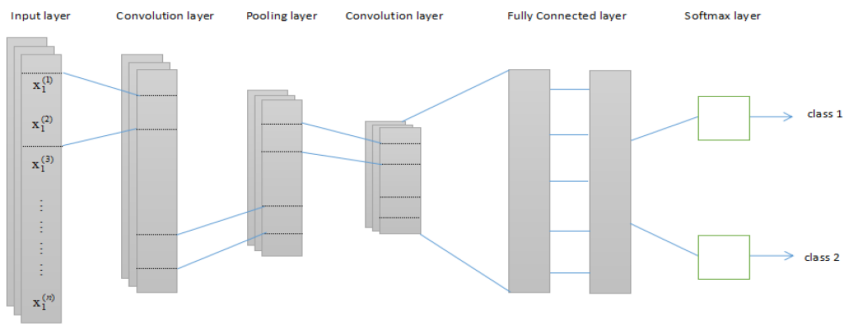
Convolutional Neural Network is a feed-forward neural network. Like the traditional architecture of a neural network including input layers, hidden layers and output layers, convolutional neural network also contains these features and the input of the layer of convolution are the output of the previous layer of convolution or pooling. Of course, they still have some unique features such as pooling layers, full connection layers, etc.

The computation of convolution in a CNN model usually accepts a 2D image as its input. But, if our goal is predict the stock price movement, that belongs to1D time series data, it is necessary to change the function to help us do the computation. The new function we adopt in the model accepts a number of stock data, and other properties such as the number of the filters, width of the filters and stride as its input.

The outputs of our function are a number of matrices, and the specific number is decided by the number of the filters. CNN model will extract some features from them and they also will be the input of the next pooling layer after the computation of activation function.

The input form can be defined as following matrix: $X = \left(x^{(1)}, x^{(2)}, \dots, x^{(m)}\right)$
Here m indicates the number of samples as inputs. Each sample consist of k features, so the input of the neural network is a $m \times k$ matrix.
For getting the final classification values, which will tell us the stock will be up or down in future, we will use "teaching" number to train the model.
The output form is basically similar as the input form: $Y = \left(y^{(1)}, y^{(2)}, \dots, y^{(m)}\right)$
Here each y is a Boolean value: 1 or 0. Additionally, the input data should be normalized because there is too much difference between each feature.

Traditionally, the activation function that people always use can be concluded as followings: Tanh, Relu, Sigmoid and Lrelu. To solve the problem of gradient vanishing, we choose the Relu and Lrelu function to do the computation of activation.
An example the architecture of CNN model:



Deep learning libraries

TensorFlow

TensorFlow is the most widely used framework for deep learning.
Created by the Google Brain team, TensorFlow is an open source library for numerical computation and large-scale machine learning. It uses Python to provide a convenient front-end API for building applications with the framework, while executing those applications in high-performance C++. TensorFlow allows the user to build static computational graphs, that is, a graph structure is built once, and then execute the computation with different values.
TensorFlow also computes the gradients of the graph, making backpropagation and thus, training a model, very easy.
TensorFlow is optimized to run efficiently on CPU, GPU and TPU (Tensor Processing Unit, a piece of hardware produced by google to fit this purpose exactly).

Keras

Although TensorFlow is a flexible and powerful tool, it can be somewhat difficult to use. The user needs to hard code a lot of details in the model.
It takes time, effort and knowledge to use TensorFlow.
That where Keras comes in!
Keras is an API built on top of TensorFlow designed specifically to enable the user to build and modify neural networks. It is very useful for research, when one wishes to try many different model variations, Keras makes it very easy.
In general, Keras makes deep learning applications very accessible to everyone.
Nowadays, TensorFlow comes with Keras built in.

PyTorch

Another widely used library for deep learning is PyTorch.
PyTorch was developed by Facebook and is open sourced and based on the Torch library.
It has a very friendly syntax, very similar to Numpy.
The main difference between PyTorch and TensorFlow is that PyTorch uses dynamic computational graphs as opposed to static ones.
If static graphs are built once, a dynamic graph is built over and over in every iteration of the training. On one hand, this is less efficient, having to build the same structure multiple times. On the other hand, this allows changing the graph during runtime (if some criteria is met, for example).
Other libraries for deep learning exist. For example, Caffe2 (Facebook), MXNet (Amazon), CNTK (Microsoft) and more.

Time Lagged Neural Networks (TLNN) - Implementation in python

The following code uses the code created by Abhishekmamidi.

Link: https://github.com/Abhishekmamidi123/Time-Series-Forecasting

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import keras
from keras.models import Sequential
from keras.layers import Dense
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error

import itertools
import warnings
warnings.filterwarnings('ignore')
```

```
Using TensorFlow backend.
```

In [0]:

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error

def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

def root_mean_squared_error(y_true, y_pred):
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    return rmse
```

In [0]:

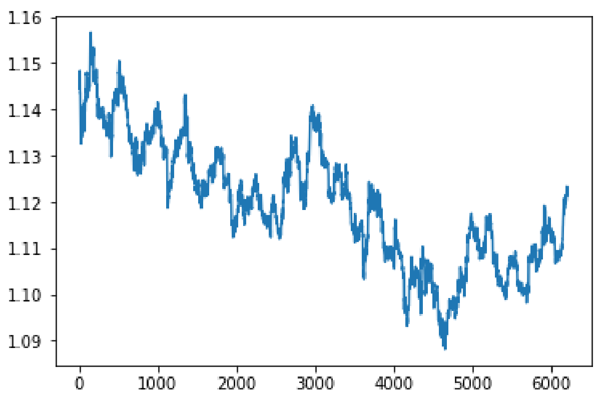
```
# We take samples every 60 minutes from the original data
original_data = np.genfromtxt('DAT_MT_EURUSD_M1_2019.csv', delimiter=',')
original_data = original_data[1::60,2]
length_data = len(original_data)
```

In [4]:

```
plt.plot(original_data)
```

Out[4]:

```
[<matplotlib.lines.Line2D at 0x7f823e7e4c88>]
```



In [0]:

```
def get_combinations(parameters):
    return list(itertools.product(*parameters))
```

In [0]:

```
def create_NN(input_nodes, hidden_nodes, output_nodes):
    model = Sequential()
    for k in range(4):
      model.add(Dense(int(hidden_nodes), input_dim=int(input_nodes)))
    model.add(Dense(int(output_nodes)))

    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

In [0]:

```
def train_model(model, X_train, y_train, epochs, batch_size):
    model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, verbose=0, shuffle=False)
    return model
```

In [0]:

```
def reshape_arrays(X_train, y_train):
    X_train = np.array(X_train)
    y_train = np.reshape(y_train, (len(y_train), 1))
    return X_train, y_train
```

In [0]:

```
def preprocess_TLNN(data, time_lagged_points):
    data = np.array(data)[:, 0]
    X_train = []
    y_train = []
    for i in range(max(time_lagged_points), data.shape[0]):
        x = [data[i-p] for p in time_lagged_points]
        y = data[i]
        X_train.append(list(x))
        y_train.append(y)
    return X_train, y_train
```

In [0]:

```
def forecast_TLNN(model, time_lagged_points, last_sequence, future_steps):
    forecasted_values = []
    max_lag = max(time_lagged_points)
    for i in range(future_steps):
        input_sequence = [last_sequence[max_lag - p] for p in time_lagged_points]
        forecasted_value = model.predict(np.reshape(input_sequence, (1, len(input_sequence))))
        forecasted_values.append(forecasted_value[0][0])
        last_sequence = last_sequence[1:] + [forecasted_value[0][0]]
    return forecasted_values
```

In [0]:

```
def TLNN(data, time_lagged_points, hidden_nodes, output_nodes, epochs, batch_size, future_steps):
    X_train, y_train = preprocess_TLNN(data, time_lagged_points)
    X_train, y_train = reshape_arrays(X_train, y_train)
    model_TLNN = create_NN(input_nodes=len(time_lagged_points), hidden_nodes=hidden_nodes,
                           output_nodes=output_nodes)
    model_TLNN = train_model(model_TLNN, X_train, y_train, epochs, batch_size)

    max_lag = max(time_lagged_points)
    forecasted_values_TLNN = forecast_TLNN(model_TLNN, time_lagged_points,
                                           list(data[-max_lag:]), future_steps=future_steps)

    return model_TLNN, forecasted_values_TLNN
```

In [0]:

```
def get_accuracies_TLNN(train_data, test_data, parameters):
    combination_of_params = get_combinations(parameters)
    information_TLNN = []
    iterator = 0
    print('TLNN - Number of combinations: ' + str(len(combination_of_params)))

    for param in combination_of_params:
        if (iterator+1) != len(combination_of_params):
            print(iterator+1, end=' -> ')
        else:
            print(iterator+1)
        iterator = iterator+1

        time_lagged_points = param[0]
        hidden_nodes = param[1]
        output_nodes = param[2]
        epochs = param[3]
        batch_size = param[4]
        future_steps = param[5]

        model_TLNN, forecasted_values_TLNN = TLNN(train_data, time_lagged_points, hidden_nodes,
                                                  output_nodes, epochs, batch_size, future_steps)

        y_true = test_data[:future_steps]
        y_pred = forecasted_values_TLNN
        rmse = np.sqrt(mean_squared_error(y_true, y_pred))

        info = list(param) + [rmse] + forecasted_values_TLNN
        information_TLNN.append(info)

    information_TLNN_df = pd.DataFrame(information_TLNN)
    indexes = [str(i) for i in list(range(1, future_steps+1))]
    information_TLNN_df.columns = ['look_back_lags', 'hidden_nodes', 'output_nodes', 'epochs',
                                   'batch_size', 'future_steps','RMSE'] + indexes
    return information_TLNN_df
```

In [0]:

```
def analyze_results(data_frame, test_data, name, flag=False):
    optimized_params = data_frame.iloc[0]
    future_steps = optimized_params.future_steps
    forecast_values = optimized_params[-1*int(future_steps):]
    y_true = test_data[:int(future_steps)]

    model = create_NN(len(optimized_params.look_back_lags), optimized_params.hidden_nodes,
                      optimized_params.output_nodes)
    s = ''
    for i in optimized_params.look_back_lags:
        s = s+' '+str(i)
    print('Look back lags: ' + s)

    print('Number of epochs: ' + str(optimized_params.epochs))
    print('Batch size: ' + str(optimized_params.batch_size))
    print('Number of future steps forecasted: ' + str(optimized_params.future_steps))
    print('Root Mean Squared Error(RMSE): ' + str(optimized_params.RMSE))
    print('\n\n')

    plt.figure(figsize=(14,5))
    plt.plot(y_true, color='green', label='Actual values')
    plt.plot(forecast_values, color='red', label='Forecasted values')
    plt.legend()
    return optimized_params
```

In [0]:

```
train_data = original_data[:int(10/12*length_data)]
train_data = train_data.reshape((len(train_data), 1))
test_data = original_data[int(10/12*length_data):]
test_data = test_data.reshape((len(test_data), 1))
```

In [0]:

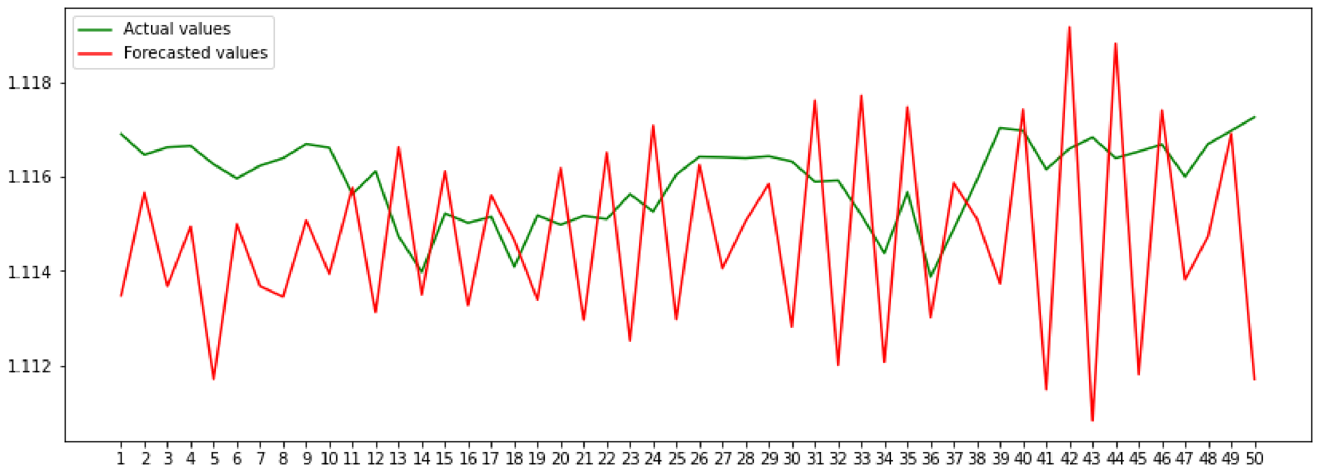
```
output_nodes = 1
epochs = 5
batch_size = 20
future_steps = 50
tlp = [1,2,3,4,5,6,10,11,12]

# time_lagged_points, hidden_nodes, output_nodes, epochs, batch_size, future_steps
parameters_TLNN = [[tlp], [5], [output_nodes], [epochs], [batch_size], [future_steps]]
```

In [19]:

```
information_TLNN_df = get_accuracies_TLNN(train_data, test_data, parameters_TLNN)
optimized_params_TLNN = analyze_results(information_TLNN_df, test_data, 'TLNN')
```

```
TLNN - Number of combinations: 1
1
Look back lags:  1 2 3 4 5 6 10 11 12
Number of epochs: 5
Batch size: 20
Number of future steps forecasted: 50
Root Mean Squared Error(RMSE): 0.002529694003500197
```



Long Short Term Memory (LSTM) - Implementation in python

In [0]:

```
import numpy as np
import matplotlib.pyplot as plt

from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error

import itertools
import warnings
warnings.filterwarnings('ignore')
```

```
Using TensorFlow backend.
```

In [0]:

```
def get_combinations(parameters):
    return list(itertools.product(*parameters))
```

In [0]:

```
def create_dataset(dataset, look_back=1):
    dataX, dataY = [], []

    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        dataX.append(a)
        dataY.append(dataset[i + look_back, 0])

    return np.array(dataX), np.array(dataY)
```

In [0]:

```
def LSTM_model(train_data, test_data, look_back=1, units=4):
  trainX, trainY = create_dataset(train_data, look_back)
  testX, testY = create_dataset(test_data, look_back)
  trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
  testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

  model = Sequential()
  model.add(LSTM(units=units, input_shape=(1, look_back)))
  model.add(Dense(1))
  model.compile(loss='mean_squared_error', optimizer='adam')
  print('-----> look back = {l} , units = {u}'.format(l=look_back,u=units))
  model.fit(trainX, trainY, epochs=3, batch_size=1, verbose=2)
  trainPredict = model.predict(trainX)
  testPredict = model.predict(testX)
  trainPredict = scaler.inverse_transform(trainPredict)
  trainY = scaler.inverse_transform([trainY])
  testPredict = scaler.inverse_transform(testPredict)
  testY = scaler.inverse_transform([testY])
  train_RMSE = np.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
  test_RMSE = np.sqrt(mean_squared_error(testY[0], testPredict[:,0]))

  return [trainY[0], trainPredict[:,0], train_RMSE, testY[0], testPredict[:,0], test_RMSE]
```

In [0]:

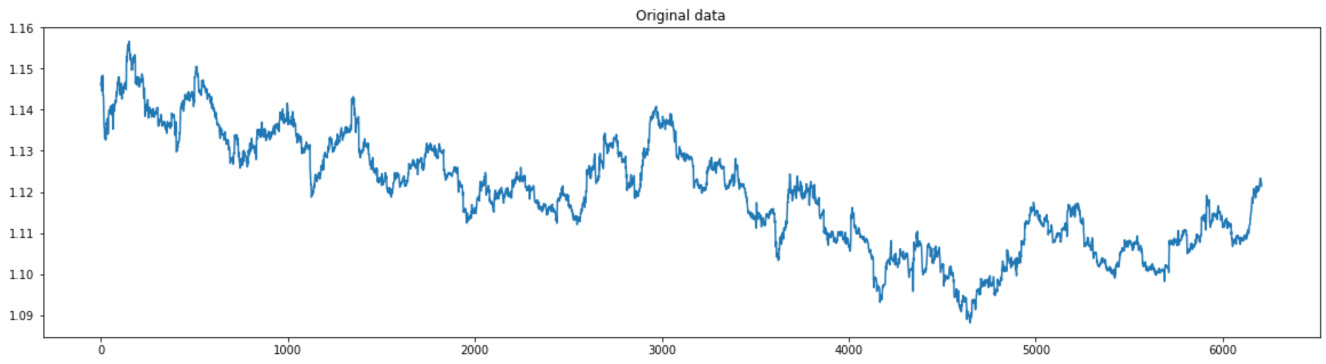
```
# We take samples every 60 minutes from the original data
original_data = np.genfromtxt('DAT_MT_EURUSD_M1_2019.csv', delimiter=',')
original_data = original_data[1::60, 2]
length_data = len(original_data)
```

In [0]:

```
plt.figure(figsize=(20,5))
plt.title('Original data')
plt.plot(original_data)
```

Out[0]:

```
[<matplotlib.lines.Line2D at 0x7f90eee4c1d0>]
```



In [0]:

```
train_data = original_data[:int(10/12*length_data)]
train_data = train_data.reshape((len(train_data), 1))
test_data = original_data[int(10/12*length_data):]
test_data = test_data.reshape((len(test_data), 1))
```

In [0]:

```
# Moving the data values to the range (0,1)
scaler = MinMaxScaler(feature_range=(0, 1))
train_data = scaler.fit_transform(train_data)
test_data = scaler.fit_transform(test_data)
```

In [0]:

```
print("Shape of train data: " + str(train_data.shape))
print("Shape of test data: " + str(test_data.shape))
```

```
Shape of train data: (5174, 1)
Shape of test data: (1035, 1)
```

In [0]:

```
# information_LSTM:  for each key (look back, units) we have the next list:
# [trainY, trainPredict, train_RMSE, testY, testPredict, test_RMSE]

look_back, units = [i for i in range(5,16,5)], [5,10,15]
parameters = get_combinations([look_back, units])
information_LSTM = {}

for lb, u in parameters:
  info = LSTM_model(train_data.copy(), test_data.copy() ,lb, u)
  information_LSTM[(lb,u)] = info
```

```
-----> look back = 5 , units = 5
Epoch 1/3
 - 33s - loss: 0.0025
Epoch 2/3
 - 31s - loss: 3.6461e-04
Epoch 3/3
 - 32s - loss: 3.1685e-04
-----> look back = 5 , units = 10
Epoch 1/3
 - 32s - loss: 0.0033
```

```
Epoch 2/3
 - 31s - loss: 3.0675e-04
Epoch 3/3
 - 31s - loss: 2.5748e-04
-----> look back = 5 , units = 15
Epoch 1/3
 - 32s - loss: 0.0038
Epoch 2/3
 - 31s - loss: 2.5321e-04
Epoch 3/3
 - 31s - loss: 2.3215e-04
-----> look back = 10 , units = 5
Epoch 1/3
 - 32s - loss: 0.0039
Epoch 2/3
 - 32s - loss: 3.6420e-04
Epoch 3/3
 - 32s - loss: 2.9783e-04
-----> look back = 10 , units = 10
Epoch 1/3
 - 32s - loss: 0.0010
Epoch 2/3
 - 31s - loss: 3.3240e-04
Epoch 3/3
 - 31s - loss: 2.6266e-04
-----> look back = 10 , units = 15
Epoch 1/3
 - 32s - loss: 0.0019
Epoch 2/3
 - 32s - loss: 3.0682e-04
Epoch 3/3
 - 32s - loss: 2.6310e-04
-----> look back = 15 , units = 5
Epoch 1/3
 - 31s - loss: 0.0079
Epoch 2/3
 - 31s - loss: 3.9044e-04
Epoch 3/3
 - 31s - loss: 3.2126e-04
-----> look back = 15 , units = 10
Epoch 1/3
 - 33s - loss: 0.0025
Epoch 2/3
 - 32s - loss: 3.2803e-04
Epoch 3/3
 - 32s - loss: 2.8090e-04
-----> look back = 15 , units = 15
Epoch 1/3
 - 32s - loss: 0.0013
Epoch 2/3
 - 31s - loss: 3.4375e-04
Epoch 3/3
 - 31s - loss: 2.8727e-04
```

In [0]:

```
k, min_rmse = (look_back[0],units[0]), information_LSTM[k][5]

for key in information_LSTM:
  if information_LSTM[key][5] < min_rmse:
    min_rmse=information_LSTM[key][5]
    k=key

trainY,trainPredict = information_LSTM[k][0],information_LSTM[k][1]
testY, testPredict = information_LSTM[k][3],information_LSTM[k][4]
train_RMSE, test_RMSE = information_LSTM[k][2], information_LSTM[k][5]
```

In [0]:

```
print('The best parameters: look back = {l}, units={u}'.format(l=k[0], u=k[1]))
print('The best RMSE = {err}'.format(err=min_rmse))
```

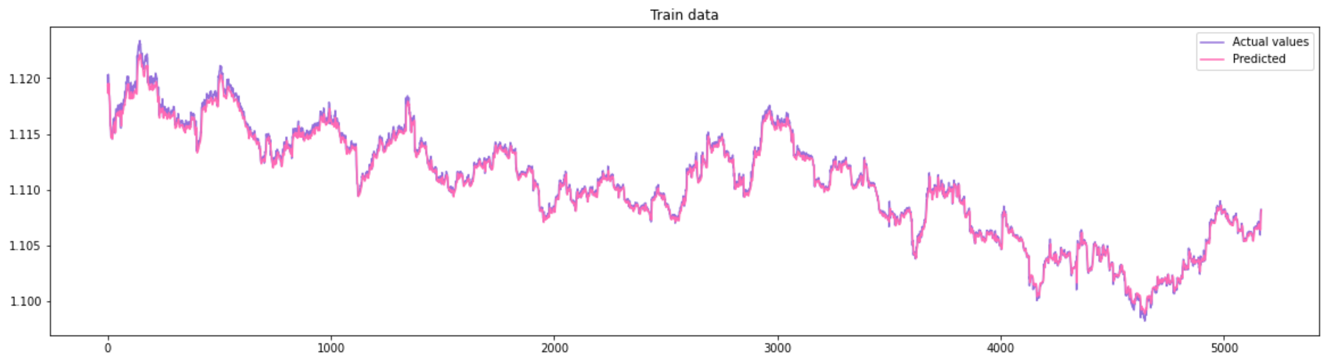
```
The best parameters: look back = 5, units=15
The best RMSE = 0.0007136265977746055
```

In [0]:

```
plt.figure(figsize=(20,5))
plt.title('Train data')
plt.plot(trainY, color='mediumpurple', label='Actual values')
plt.plot(trainPredict, color='hotpink', label='Predicted')
plt.legend(loc='best')
```

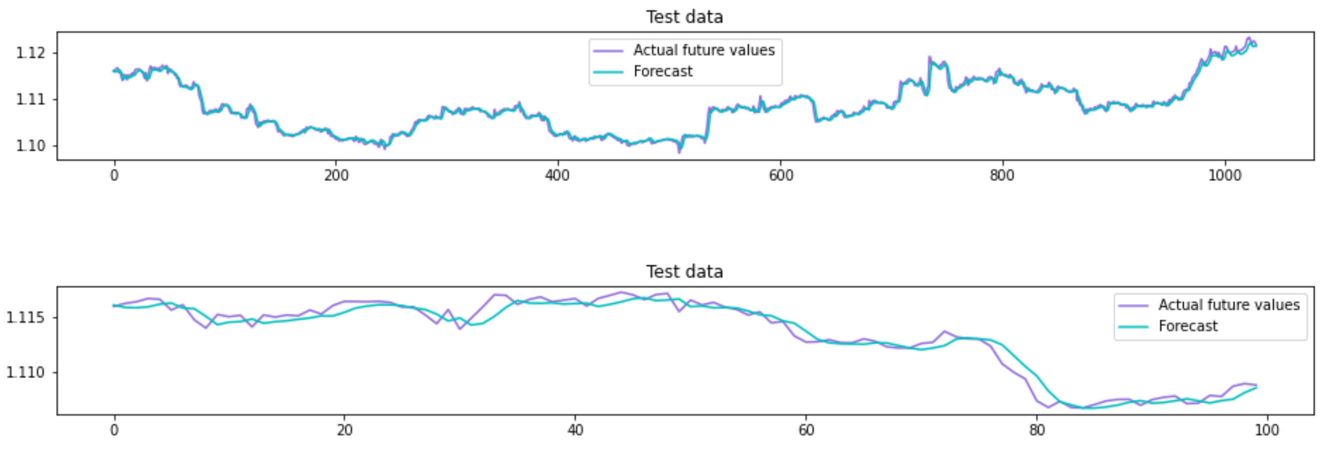
Out[0]:

```
<matplotlib.legend.Legend at 0x7f90e4db3048>
```



In [0]:

```
plt.figure(figsize=(16,5))
plt.subplot(211)
plt.title('Test data')
plt.plot(testY, color='mediumpurple', label='Actual future values')
plt.plot(testPredict, color='c', label='Forecast')
plt.legend(loc='best')
plt.subplot(212)
plt.title('Test data')
plt.plot(testY[:100], color='mediumpurple', label='Actual future values')
plt.plot(testPredict[:100], color='c', label='Forecast')
plt.legend(loc='best')
plt.subplots_adjust(hspace=1)
```



Conclusion:

After trying many different configurations, we've come to the conclusion that the TDNN model doesn't provide a good enough estimation for this particular time series. This does not imply that TDNNs are useless, for they have proven to be very effective in some cases.
In our case, we've found that an LSTM model works much better, providing a very accurate estimation with minimal error. After testing many different configurations, we've found that the best estimation is achieved by using 15 LSTM units with a 5 time step look-back.
We were able to achieve a future prediction of a 1000 time steps with a Root Mean Square Error (with respect to the actual test data) of approximately 7×10^{-4}.