**Chapter 3**

**Methodology of the Proposed System**

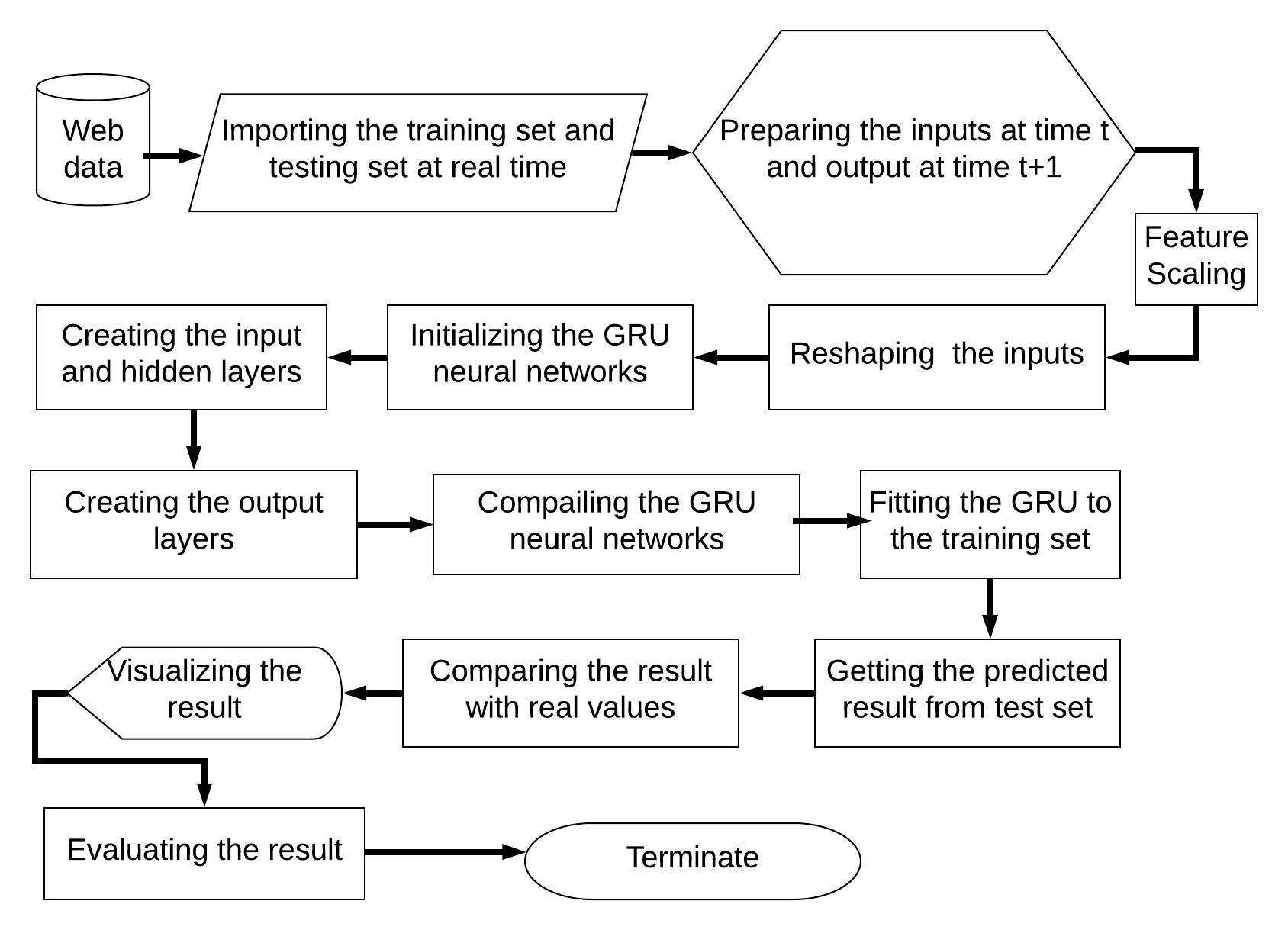
In this work, we focused on system analysis and designing the system which can predict the future prices of stock market based on an advanced deep neural networks system at real time. This chapter mainly focuses on the overall system architecture of the proposed system and procedure to achieve this in details.

Introduction:

Learning from past history and predict the future is a fundamentally tough challenge. A model may fit historical data well but not perform well when presented with new inputs. With gated recurrent neural networks (GRUs), we leverage the modeling abilities of artificial neural networks for time series forecasting. With this networks we also overcome the vanishing gradient problems of recurrent neural networks. When training, we also overcome the local minimum problem of gradient descent by using Stochastic gradient descent.

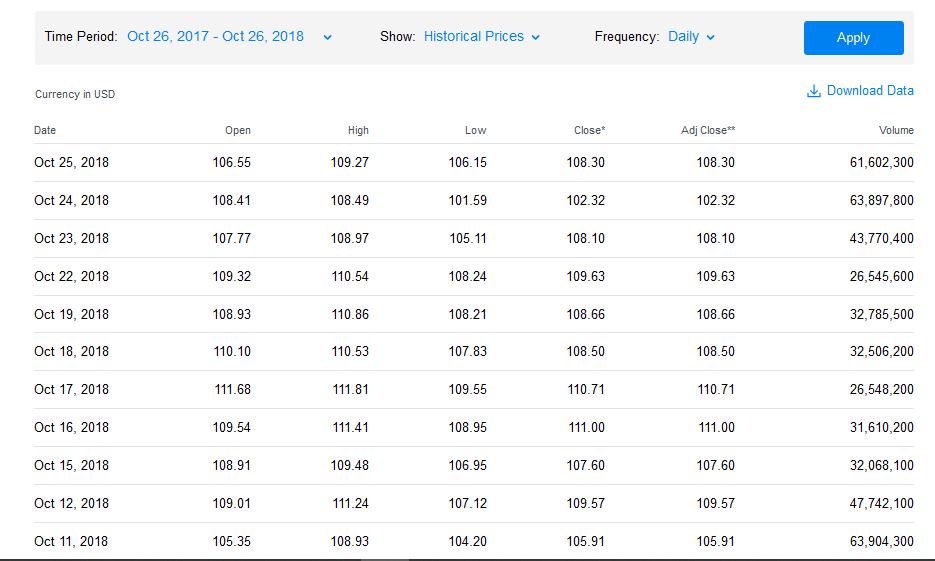
Overview:

Our system consists of fifteen steps. First we have to collect data from web at real time. Second we have to select training set and testing set. Third we have to prepare inputs at time t and outputs at time t+1. Forth we have to apply feature scaling on our data so that our data will stay between a range and also between the threshold. After that we reshape our data so that it has to be into proper form and can be used as the inputs of the networks. The sixth steps is about initializing the networks and at next step we creating the input and hidden layers and at next steps we creating the output layers. At ninth steps we compile our neural networks. After that we fit our training data into the networks to extract the hidden features. At eleventh step we get the predicted results from test set and at next steps we compare the values with real values. At thirteen step we visualize our data and after that we evaluating the results. Finally we terminate from our program.



Web Data Source:

The data are collected from Yahoo! Finance at real time. Yahoo! Finance is a media property that is the part of Yahoo! Network [yahoo finance].



Here every company has a ticker name like ‘INTL’ for the company name Intel. For every date that website has opening, highest, lowest, closing, adjusted closing and volume values.

Real Time Data Collection:

Python’s rich online connectivity capabilities offer a proper way to collect data from the Internet, and its ability to write CSV files provide a medium for sharing collected data. We use pandas\_datareader library to collect data at real time.

Training Set and Testing Set:

In deep learning, the study and construction of algorithms that can learn from and make predictions on data is a familiar task. Such algorithms work by making data-driven predictions or decisions through building mathematical model by using input data. We use Pandas library to import training set and testing set.

Training Set:

A training data set is a data set of examples used for leaning which means to fit the parameters or weights.

Ten years data have been used to train our model.

Testing Set:

A test data set is a set of examples used only to evaluate the performance that is independent of training dataset but that follows the same probability distribution as the training dataset. If a model fits to the training data set also fits the test data set.

Preparing the inputs and outputs:

Our first task is to prepare the inputs and outputs. Our inputs are opening price, closing price, highest value and lowest value at time t and output is opening/closing values at time t+1. To do that we shifted our output one timestamp and remove the row containing NAN value.

If we have two thousand one rows in our data set that means we have two thousand input vectors each containing the opening price, closing price, highest value and lowest value at time t. We have output vectors at next time step that means at time t+1 which is basically opening/closing price of that time.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Inputs | | | | Outputs |
| Open | Close | High | Low | Open/Close |
| timet | timet | timet | timet | timet+1 |
| timet+1 | timet+1 | timet+1 | timet+1 | timet+2 |
| ……………………………… | | | | ………… |
| timen | timen | timen | timen | timen+1 |

Feature Scaling:

Feature scaling is generally performed during the data preprocessing step which is a method used to standardize the range of independent variables or features of data which is also known as data normalization.

*Why Feature Scaling:*

The main advantage of scaling is to avoid attributes in greater numeric ranges dominating those in smaller numeric ranges. To apply the gradient descent algorithms properly there are some techniques that can be applied on both training set and testing set. If the features applied on input vectors, are out of scale then loss space will be somehow stretched and this will make the gradient descent convergence harder or at least slower.

*Methods Feature Scaling:*

The main adhere are four common methods to perform Feature Scaling.

Min-max normalization:

To scale the values between [0, 1], we use min-max normalization.

Mean normalization:

To scale the values between [-1, 1], we use mean normalization.

Standardization:

After applying them data will have zero mean and unit variance.

Standardization:

Scaling is done considering the whole feature vector to be of unit length.

Implementation at Our Data Set:

Min-max normalization is used on our data to scale the values between [0, 1]. The formula is:

As our neural networks need sigmoid activation functions and it scale the values between [0,1] that is the main reason to scale our data with min-max normalization.

Reshaping the Inputs:

Reshaping is about changing the format of inputs. Before reshaping our input was a two dimensional array containing number of observations and features and we have four features. After reshaping our input data is a three dimensional array as time step is also included. We implemented this by using NumpPy library.

Gated Recurrent Unit (GRU) Neural Networks:

Gated Recurrent Unit networks usually just called “GRUs” – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Cho, et al. (2014) [gru] .They work tremendously well on a large variety of problems and they are one of the most modern, powerful and effective neural networks. GRUs are explicitly designed to avoid the long-term dependency problem. GRU has fewer parameter than LSTM and thus may train a bit faster or need less iterations to generalize. Writers of the paper ‘An Empirical Exploration of Recurrent Network Architectures’ showed that the GRU outperformed the LSTM on most tasks with the exception of language modeling [gru better].

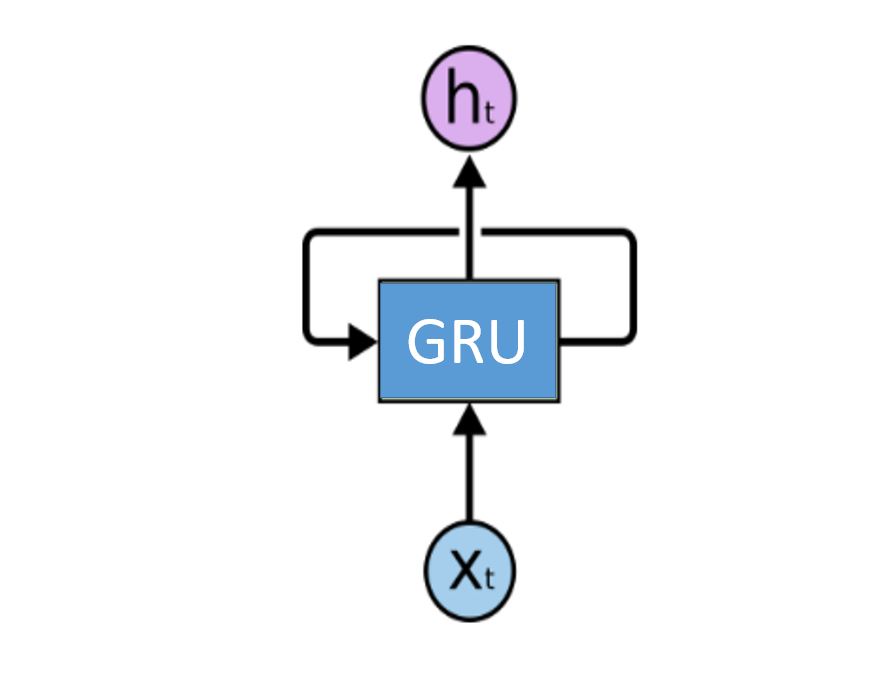


Fig: Rolled Gated Recurrent Unit (GRU) Neural Networks

GRUs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Another way to think about GRUs is that they have a “memory” which captures information about what has been calculated so far.

In the above figure, a neural network, ‘GRU’ looks at some input xtand outputs a value ht which is also the input of next step and thus a loop allows data to be passed from one step of the network to the next.

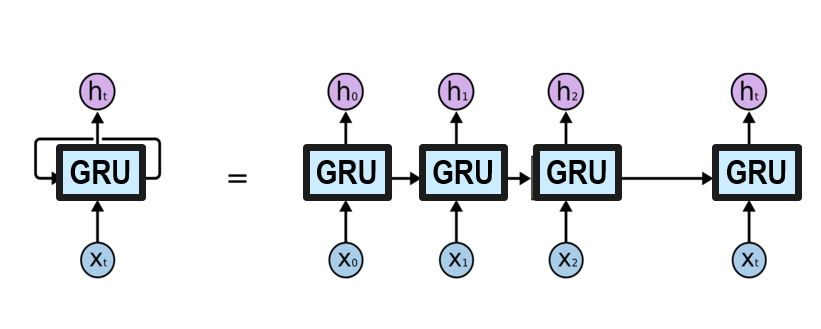
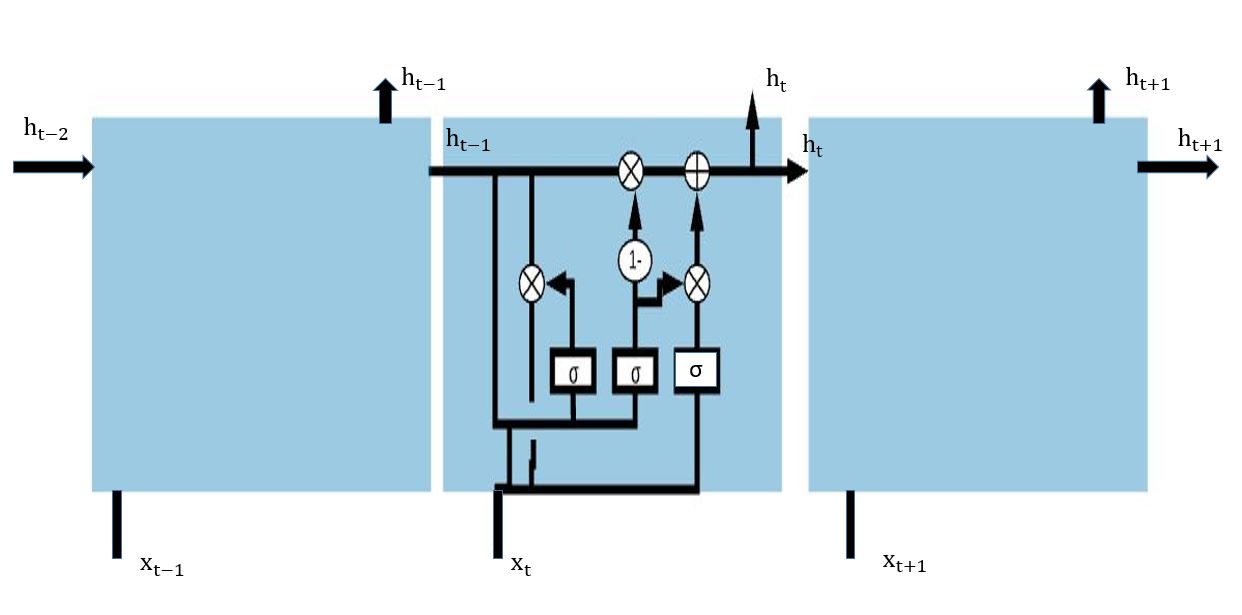


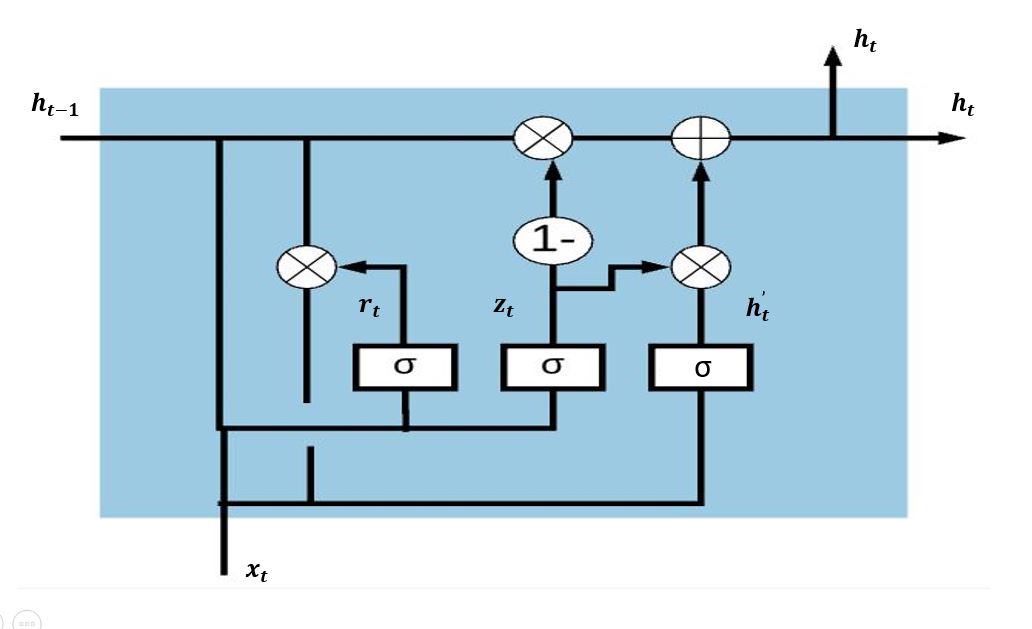
Fig: unrolling the rolled GRU neural network

This neural network is called recurrent because of multiple copies of the same network, each passing a message to next network and that chain-like nature is the reason that recurrent neural networks are basically related to sequences and lists like stock market data. So recurrent neural networks have the natural architecture of neural network to use for extracting the hidden pattern of stock market.

*Internal Architecture of* *Gated Recurrent Unit (GRU) Neural Networks:*

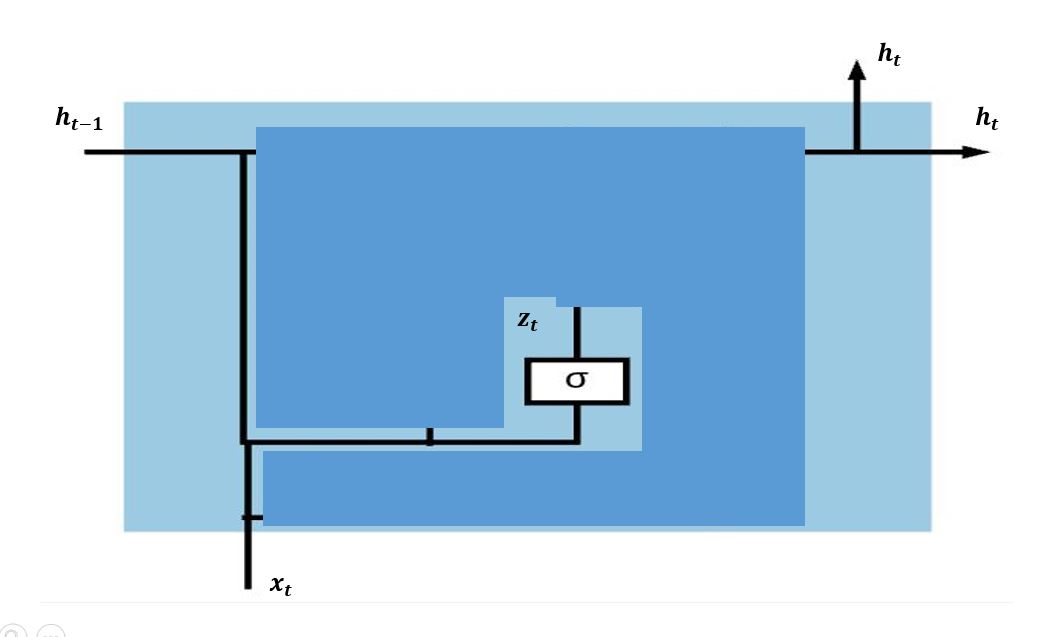
GRU uses update gate and reset gate to solve the vanishing gradient problem of a standard RNN. Actually update gate and reset gate are two vectors which decide what information should be passed and what information should not be passed to the output. The best thing about them is that they can be trained to keep information from long ago, without washing it through time or remove information which is irrelevant to the prediction. With the help of [gru blog] we explain the mathematics behind a single cell of this networks.

**



Update gate:

The update gate helps the network to control how much of the past information (from previous time steps) needs to be passed along to the future. That is really effective because the network can decide to copy all the information from the past and eliminate the risk of vanishing gradient problem.

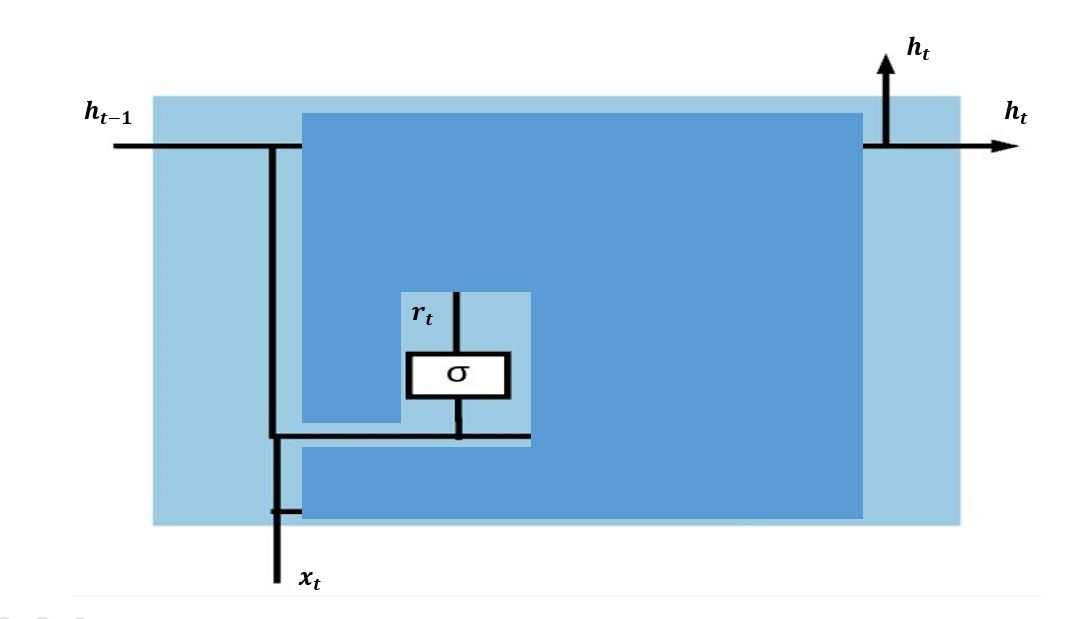


The formula for update gate is:

Here is the into the network unit, it is multiplied by its own weight. The same goes for which holds the information for the previous t-1 units and is multiplied by its own weight . Both results are added together and a sigmoid activation function is applied to squash the result between 0 and 1.

Reset gate:

The main purpose of reset gate in the network is basically to decide how much of the past information to forget.

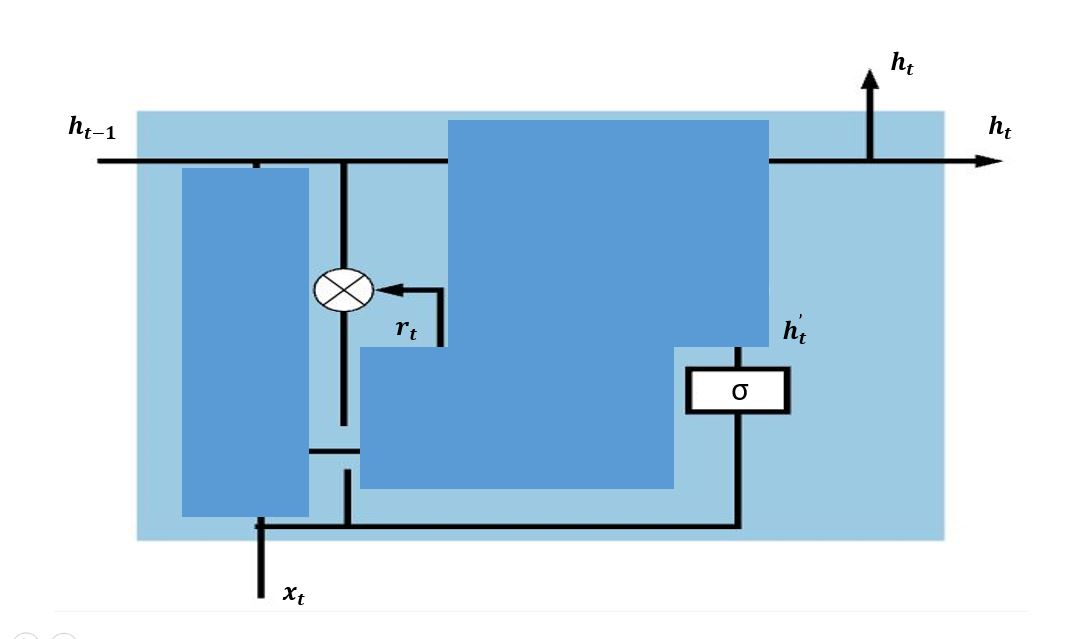


The formula for reset gate is:

We plug in previous output as input and , multiply them with their corresponding weights, sum the results and apply the sigmoid function.

Current memory content:

This memory content will use the reset gate to store the relevant information from the past. Here we changed our activation function from hyperbolic tangent to sigmoid

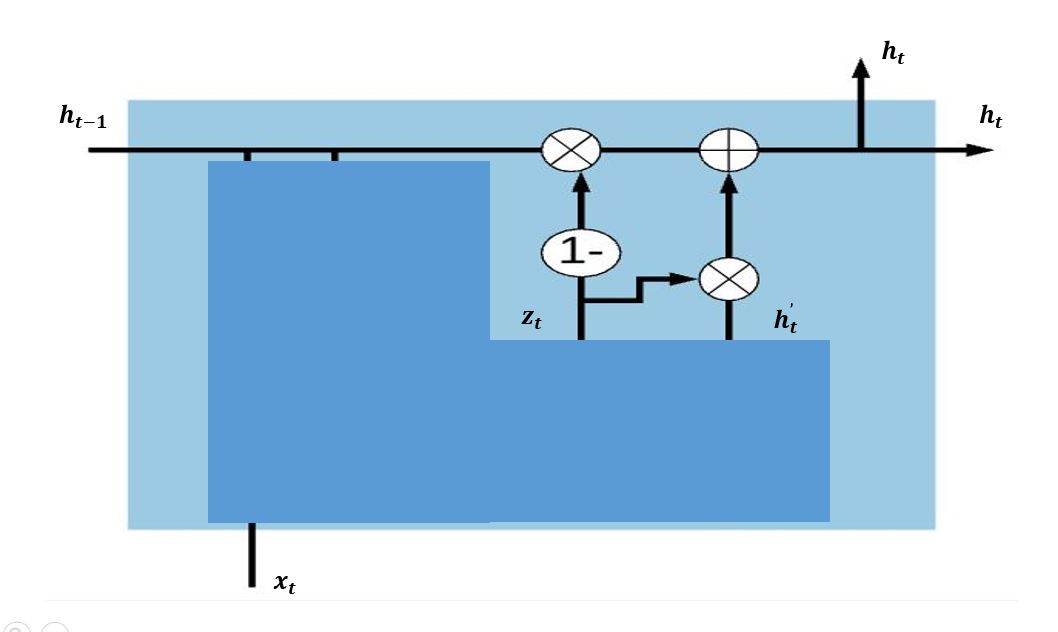


The formula for current memory content is:

Here an element-wise multiplication happened between and  *line* and then sum the result with the inputfinally, σ is used to produce

Final memory at current time step:

Finally the network needs to determine which is the output of current unit and passes it down for next unit. In order to do that the update gate is needed which control what to collect from the current memory content  and what from the previous steps    .



The formula of this content is:

Initializing the GRU Neural Networks:

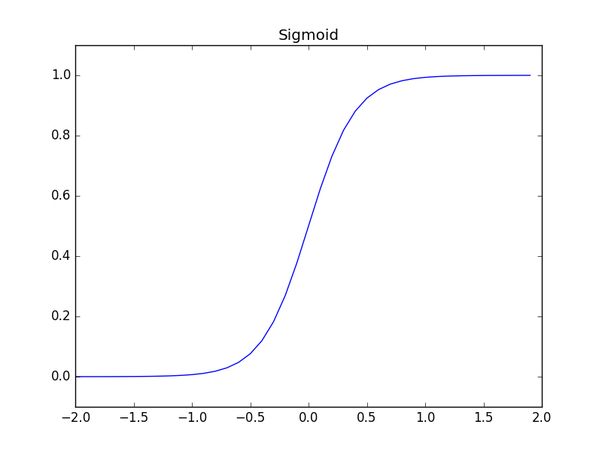
Our data has a continuous sequence. As we are predicting continuous outcome so we make regression model. Regression is used when we have continuous outcome and classification is used when we have categorical outcome. So we called our object regressor and use sequential class as our data has a continuous sequence.

Creating the Inputs and Hidden Layers:

We changed our hidden layers from traditional GRUs by using sigmoid activation function instead of hyperbolic tangent.

Sigmoid activation function:

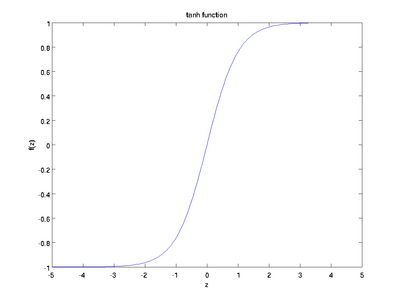
The sigmoid function curve looks like a S-shape and the main reason why we use sigmoid function is because it exists between the range zero to one.



The equation for sigmoid is:

Hyperbolic tangent activation function:

The hyperbolic tangent activation function curve looks like a S-shape and it exists between the range minus one to one.



The equation for hyperbolic tangent activation function is:

As our stock price values never be less than zero and that is the main reason for using sigmoid instead of hyperbolic tangent. That is why we changed the activation functions.

We use Keras to create this model with four input features and one time step using sigmoid activation function.

Creating the outputs Layers:

We used Keras, Densed class to create the output layers which is opening/closing prices at next time step.

Compiling GRU Neural Networks:

To compile the GRU neural networks, we first need to know how we train our neural networks. We use back Backpropagation through time (BPTT) to train our neural networks.

Backpropagation:

Backpropagation is shorthand for “the backward propagation of errors” which is a method used in deep learning to calculate a gradient that is needed in the calculation of the weights to be used in the neural networks. Backpropagation is commonly used by the gradient descent optimization algorithm to adjust the weight of neurons by calculating the gradient of the loss function.

Loss Function:

Here our loss function is mean squared error (MSE) which is an estimator measures the average of the squares of the errors.

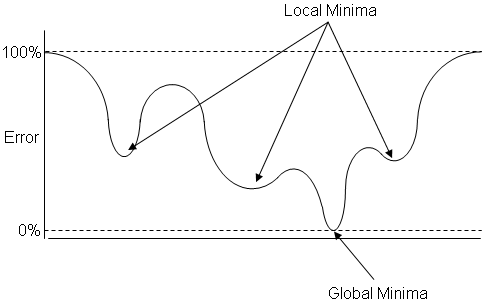
If is a vector of predictions generated from a sample of *n* data points on all variables, and is a vector of observed values of the variable being predicted, then the with in sample mean squared error (MSE) of the predictor is computed as

Local Minima Problem with Gradient Descent:

The gradient descent optimization algorithm targets to minimize some cost/loss function based on that functions gradient. When training weights in a deep neural network, normal batch gradient descent usually takes the mean squared error of all the training samples when it is updating the weights of the network:

Where are the weights, *α* is the learning rate and ∇ is the gradient of the cost function with respect to changes in the weights.

At complex structure, gradient descent can detect a gradient which might be the gradient of a smaller sub part but that is not the optimal gradient. That is known as local minima problem.



The problem with gradient descent is that it can detect a minimum point but that point is not the optimal minimum or global minima, that point will be a local minima point.

Problem with Stochastic Gradient Descent:

Stochastic gradient descent updates the weight matrix after evaluation the cost/loss function after each sample.  That is, rather than summing up the cost/loss function values for all the sample then taking the mean, stochastic gradient descent (SGD) updates the weights after every training sample is analyzed. The formula is:

Where are the weights, *α* is the learning rate and ∇ is the gradient of the cost function with respect to changes in the weights, here an update to the weights and also to the bias is performed after every sample *z* in *m*.

It can solve local minimum problem but it takes much computation time. It responds to the effects of each and every sample, and the samples themselves will contain an element of noisiness that will make the result noisy moreover it takes much computation time.

Mini-batch Gradient Descent:

Mini-batch gradient descent is a good trade-off between stochastic gradient descent and batch gradient descent. In this techniques, the cost function (and therefore gradient) is averaged over a small number of samples, from around 10-500.  This is opposed to the stochastic gradient descent batch size of 1 sample, and the batch gradient descent size of all the training samples.  It looks like this:

Where are the weights, *α* is the learning rate and ∇ is the gradient of the cost function with respect to changes in the weights and is the mini-batch size.

We use ADAM optimization algorithm that can used instead of the classical stochastic gradient descent algorithm. ADAM stands for adaptive moment estimation was presented by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in their 2015 ICLR ICLR paper (poster) titled “Adam: A Method for Stochastic Optimization” [Adam]. Our loss function is mean squared error (MSE).

Fitting the GRU to the training set:

Here we import our inputs and outputs at the neural networks. As we use mini-batch gradient descent, we tested our neural networks on several batch size and find the best result at batch size 32. For good result and good fitting we iterated our networks 200 times and find very low lost function which is better for testing.

Getting the predicted result from test set:

After completing the training, our neural networks learn the hidden patterns of our data, So we test our model at test set. A test data set is a set of examples used only to evaluate the performance that is independent of training dataset but that follows the same probability distribution as the training dataset. If a model fits to the training data set also fits the test data set.

We also applied feature scaling at our inputs. Our predicted results is the opening/closing prices of the next day.

Comparing the results with real values:

First we applied inverse feature scaling on our predicted results. After that we compare our results with real values.

Visualizing the results:

We use pyplot module from matplotlib to plot the real socks price and predicted stocks and visualize the results.

Evaluating the results:

Our model is a regression model and regression model is generally evaluated by calculating root mean square error (RMSE). The root-mean-square error (RMSE) (or sometimes root-mean-squared error) is a frequently used measure of the differences between values predicted by a neural networks and the values observed.

If is a vector of predictions generated from a sample of *n* data points on all variables, and is a vector of real values of the variable being predicted, then the with in sample root mean squared error (RMSE) of the predictor is computed as

We used Scikit-learn and math library to calculate the root mean square error (RMSE) between the real stock price and the predicted stock price.

Conclusion:

In this chapter we will discuss about methodology related to this project. In the next chapter we discuss about the implementation of this project.