Assignment 2 (ANN & Deep learning)

Student: Afraz Salim *r-number: r0439731*

In this section, we will discuss the hopfield network using hebbian learning and issues related to hebbian learning. We build a hopfield network to store the following attractors: $T = \begin{bmatrix} 1 & -1 & 1 \\ 1 & -1 & -1 \end{bmatrix}$. After training the network, will first control whether the patterns we wanted to store, are stable or not. This can be easily checked by giving the same input patterns that we used to train the model. If we use the trained pattern as input, we get the same pattern as result. Hence the target pattern is stored. Next, we will try to find out if there are any spurious attractors. We will start from many random locations and if there is any spurious pattern then one of these starting point will converge to that spurious pattern.
The following figure is obtained by starting from 20 different points and 30 iterations. We have chosen so many iteration to let the model converge to atleast one attractor. We can see in figure 1 that one

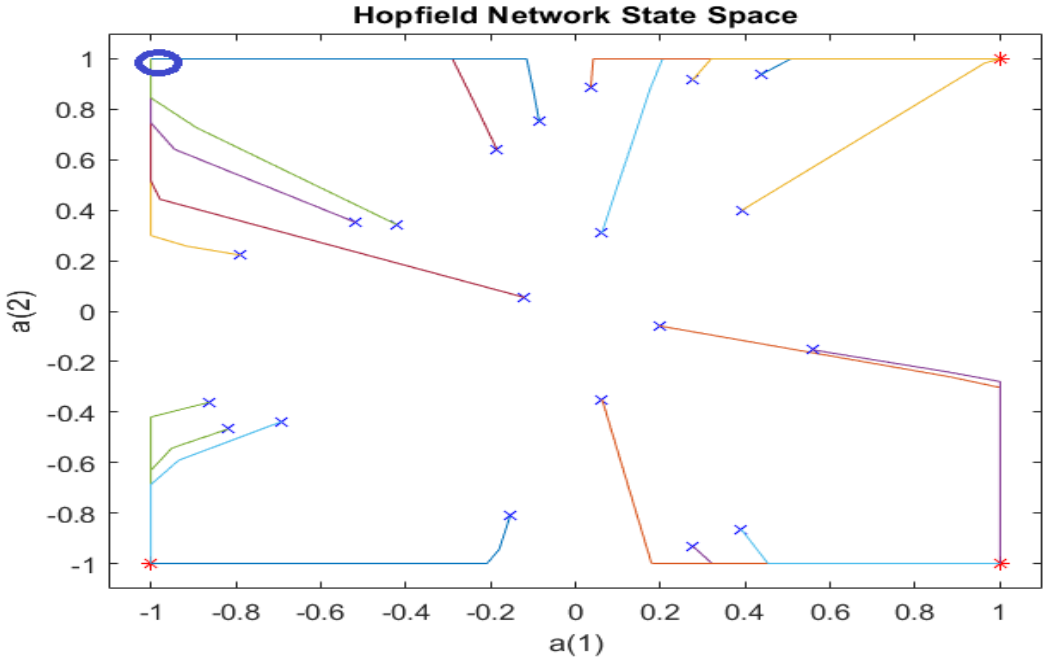


Figura 1: Hopfield network with one spurious attractor

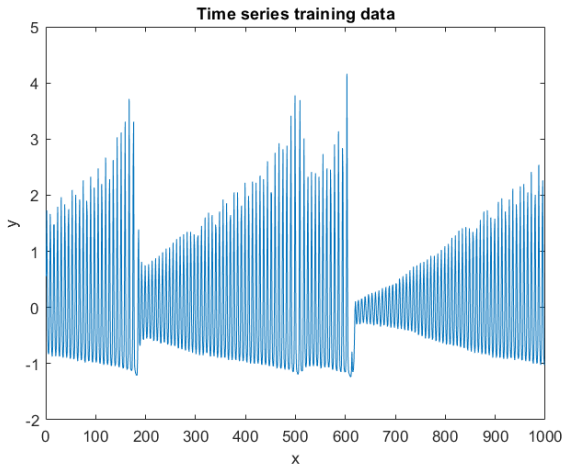
false pattern is stored which does not appear in our training patterns. There are many reasons for such parasitic patterns. One way to explain this behaviour is by using bits/karnaugh map. We are trying to store 3 patterns and we need atleast 2 bits to represent this number. We can build 4×4 cells karnaugh map. Now if we represent each cell by two bits where each bit can take -1 or 1 instead of 0 and 1. Now if we flip a bit out of two bits then such flipping has no impact on the energy function. In our model, we have parasitic pattern at $(-1, 1)$. If we generalize the above explanation and take the sum of three training input patterns then we get exactly the pattern that we have as parasitic. $Parasitic\_pattern \rightarrow (1, 1) + (-1, -1) + (1, -1) = (-, +)$. Here we used components wise addition and left the number out as we need only signs. Sum of odd number of stored patterns is also stable for hebbian learning.
Number of iterations depend on where our start vector lies. The distance is greatest between the starting vector and point to which it coverges when starting point and convergence point are orthognal.

We carried out similar experiments in 3 dimensional space and noted that if our training patterns contain any two patterns, A & B, such that flipping exactly one bit of A, results in B, then there will be a spurious attractor.
We know that in hebbian learning, if y is a stable state then $-y$ is also a stable state.
If we ignore the bias term then we can write the enerygy function as : $E = -\sum_{i,j<i} w_{i,j} y_i y_j$. We can re-write this term as: $E = -\frac{1}{2} Y^T.W.Y$ We used $\frac{1}{2}$ because we have a symmetric matrix. Next if we replace the Y vector with $-Y$ then the energy does not change.
$E \rightarrow -\frac{1}{2} Y^T.W.Y = -\frac{1}{2}(-Y)^T.W.(-Y)$.
However, hebbian learning was not investigated in depth because better results can be obtained with simple gradient descent. Hebbian learning has a drawback that it only tries to minimize the energy function where an attractor lies while this problem can be solved with SGD(simlpe gradient descent). We can try to maximize the energy of the patterns which are not target patterns.
$E = -\frac{1}{2} * Y^T.W.Y$ and weight matrix W can be obtained by using:
$W = W + \eta(\sum_{y \in Y_p} Y.Y^T - \sum_{y \notin Y_p} Y.Y^T)$.

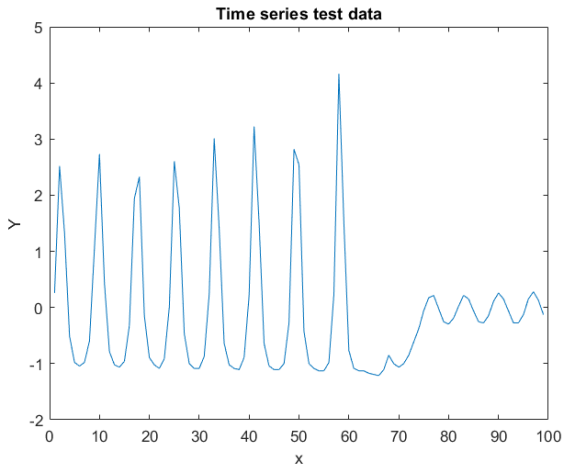
We experimented with hopdigit function which reconstructs a digit from it's noisy version. However,after tuning noise and numiter parameter, we concluded that the digits are not always reconstuctable if the noise crosses a certain threshold. However, noise plays an important role and reconstruction depends on which pixels are mostly effected. Noise can affect a digit in such a way that the noisy version lies at a place from where it will always converge to some other attractor. In short, noise places the noisy version of the digits in the neighbourhood of some attractor and independent of number of iteration, it will always(for most of the cases) converge to that attractor.

Time series prediction with LSTM

In this section, we will predict timeseries data with LSTM. We will also make a comparison between MLP and LSTM at the end of this section. We will discuss the LSTM's model and investigate the impact of lagged observations. We first plot the data that we will be using to train and test the model. The following two figures show the train and the test data. We create and train an LSTM model. We first



Normalized training data



Normalized test data

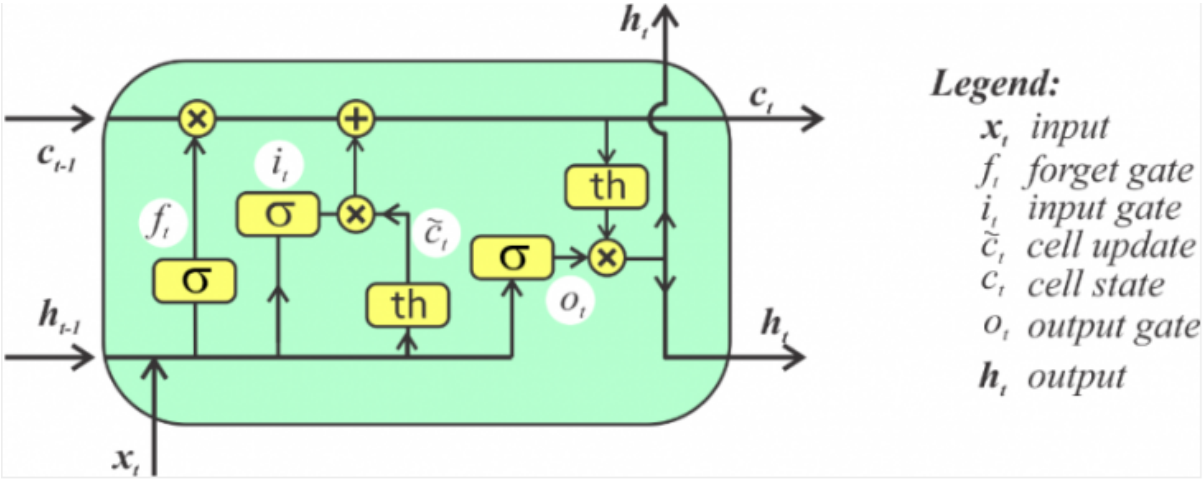
show the LSTM architecture that we use to predict the time series data. The following figure show the simple the LSTM architecture built in matlab.
We will later add multiple LSTM layers and analyze the results. We used the following code to create such architecture.

```
numFeatures = 1;
numResponses = 1;
numHiddenUnits = 500;
```


```
layers = [ ...
    sequenceInputLayer(numFeatures)
    lstmLayer(numHiddenUnits)
    fullyConnectedLayer(numResponses)
    regressionLayer];
```

```
options = trainingOptions('adam', ...
       'MaxEpochs',1, ...
       'GradientThreshold',1, ...
       'InitialLearnRate',0.005, ...
       'LearnRateSchedule','piecewise', ...
       'LearnRateDropFactor',0.2, ...
       'ExecutionEnvironment','gpu'
       'Verbose',0, ...
       'Plots','training-progress');
```

Except numHiddenUnits, the other two variables have clear meaning. In most of the online literature, LSTM layer is shown as a block. However, LSTM block is a layer in itself. numHiddenUnits represents the amount of information that we store between two time steps. To explain the numHiddenUnits, we add an image of LSTM layer/block below.



numHiddenUnits represents the dimension of the vector h_t and c_t. If we increase the dimension of h_t and c_t, then we can store more information and also more information can flow from previous states as C_t represents the cell state. We used the data with lagged value $= 1$. It means that, at timestep t we train the model to make a prediction for time step $t + 1$ by using only one previous value.
We choose numHiddenUnits as 400 because from the training plot, we can see that the pattern repeats itself after approximately 400 points. With the architecture shown above, we will plot different results that we obtain.
From the figure 3 ,we see that in the beginning we have good approximation but as the time passes the error goes up. We have chosen 400 hidden units that are capable of capturing the whole period of the function. Next,we will reduce the number of units but increase the depth of our model by adding one more LSTM layer.
Parameters in 'trainingOptions' are also tuned and can have dramatic impact on the performace.
Our new model has two LSTM layer instead of one and each layer has 150 hidden units as shown in 5.

Figura 2: Forecast vs test data: 1 LSTM layer with 400 hidden units & RMSE = 77.9845

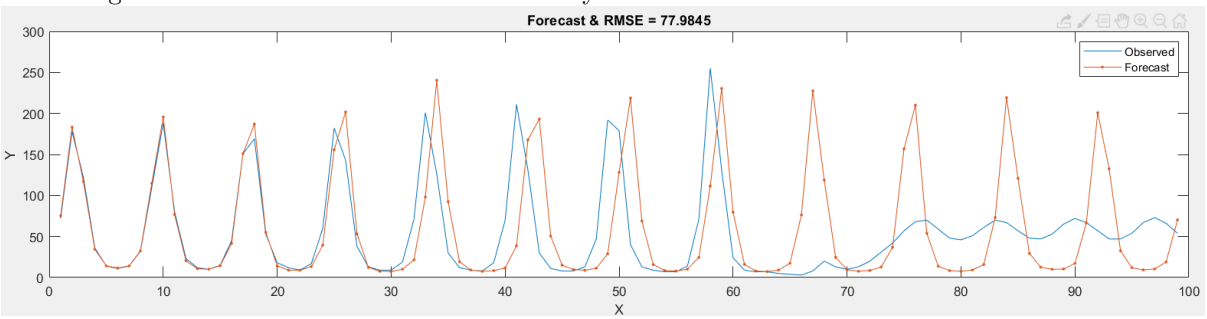


Figura 3: LSTM Model

Figura 4: Forecast vs test data: 2 LSTM layers with 150 hidden units & RMSE = 43,2987

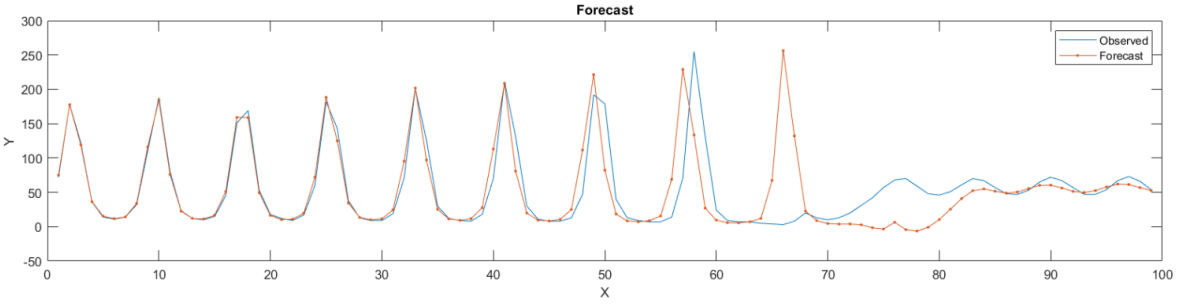


Figura 5: LSTM Model

Figura 6: Forecast vs test data: Lag = 1, 5 LSTM layers with 300 hidden units & RMSE = 22,455

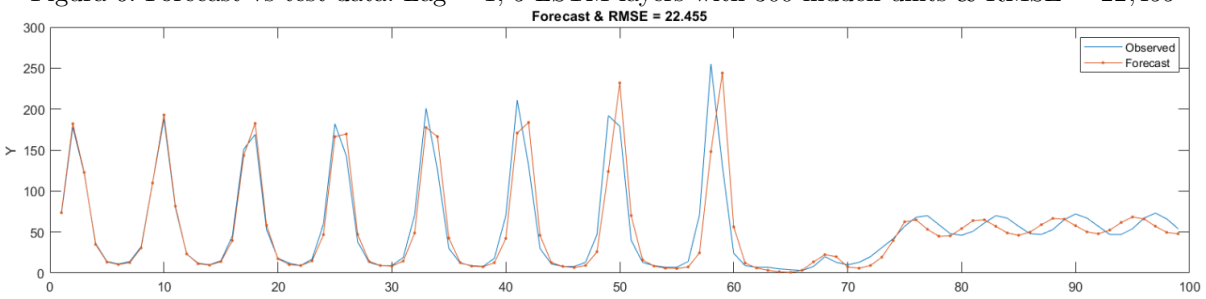


Figura 7: LSTM Model

Impact of lag value

We performed different experiments with different architectures to see the impact of lag values on the final result. However,the results shown in figure 7 are obtained from the model with 5 LSTM layers. When we include many lagged features then the forecasted curve shows an averaging behaviour. The error gets minimized but the peaks have same height everywhere. One example of such averaging behaviour is shown in 9.

In our case, a model with one lagged value performed better than all other combinations of parameters.

Conclusion on LSTM

We noticed that creating one LSTM layer with many units can cause overfitting. Instead of using one layer with too many hidden units, a model with more than one LSTM layer and each layer having few

Figura 8: Forecast vs test data: Lag = 1, 5 LSTM layers with 300 hidden units & RMSE = 22,455

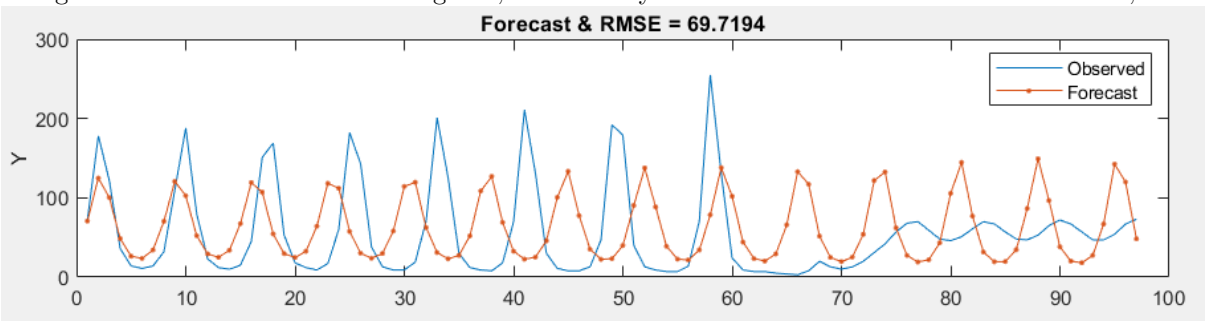


Figura 9: Forecast vs test data: Lag = 3, 5 LSTM layers with 300 hidden units & $RMSE = 69{,}7194$

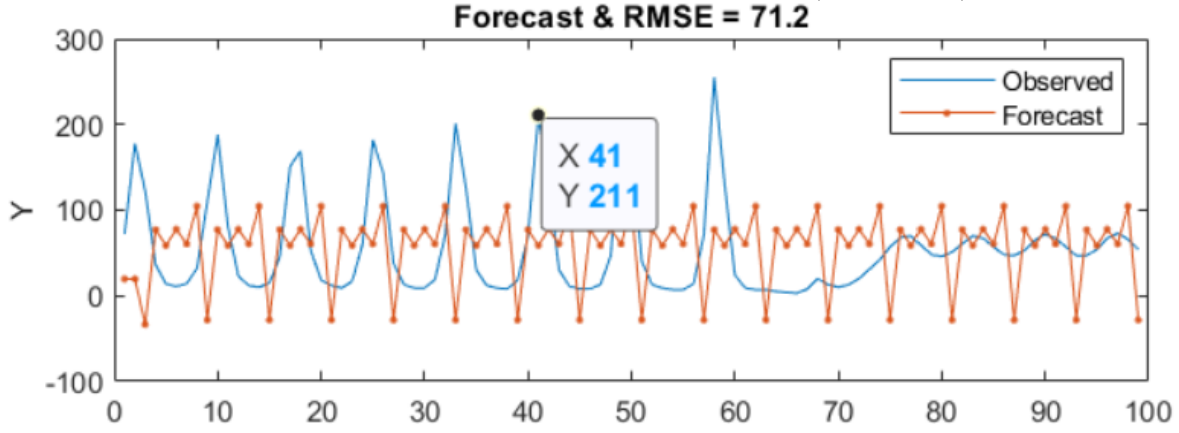
units performs better. The idea is to let next layer learn those things which first layer has not learned instead of learning all the weight in one layer.

MLP for time series data

We tried different architectures of MLP to predict same time series data that we used in LSTM. From the universal approximation theorem, we know that any function can be approximated by a MLP with one hidden layer. However, Feedforward neural networks are not capable of learning long-term dependencies. Below we show different results that we obtained with MLP.

Epoch	Neurons	max_fail	RMSE
1000	400	10	71.2
1000	600	10	119.2393

Figura 10: Forecast vs test data: Lag = 1, MLP with 1 hidden layer(400 neurons) & RMSE = 71.2



In MLP, as we start increasing the hidden units, the model starts getting overfitting. Hence, simple MLP is not a good choice for sequential data.

Figura 11: Forecast vs test data: Lag = 1, MLP with 1 hidden layer(600 neurons)& RMSE = 119.2393