BIO301 End Semester Project Report

Abhishek Tilva (20141131)
Mentor: Pranay Goel

1 Introduction

Clinical data is one resource which has the potential of providing with very valuable information regarding diagnosis and prevention of a condition which is about to occur or is prevalent in patients. One such case is that of diabetic patients where in the historic blood glucose levels can be used to predict the same in short future. Here we use data of blood glucose concentration of a diabetic patient which is recorded every fifteen minutes and use a recurrent neural network model for prediction in future. Particularly we use Long Short Term Memory (LSTM) units for recurrent network architecture as LSTMs have the ability to learn long-term dependencies in sequential data and have been successfully applied for a lot of sequential learning tasks as in [5].

1.1 Original data and task

The original data which is to be modeled is temporal in nature where in the concentration of blood sugar of a patient is recorded every 15 minutes. Therefore, the given time series data is univariate in nature. The corresponding task is of predictive modelling of this temporal data so as the model should be able to forecast/predict the blood sugar levels at least a few time steps ahead (eg. an hour later).

1.2 Approach

The approach taken is that of modelling using recurrent neural networks. In general using neural networks for modelling time series provide certain advantages over classical models as neural networks in general do not have to assume stationarity and theoretically do not require any preprocessing of the time series. Also the non linear nature of the model allows for modelling of complex characteristics which a linear model might fail to. Moreover, the recurrent neural networks class of artificial neural networks provide rich dynamic behaviour due to their structure which allows for retaining information through time.

2 Recurrent Neural Networks (RNN)

Recurrent neural networks are one class of artificial neural networks. A simple RNN can be constructed by having time delays on connections between units or by just having self recurrent loops with a unit time delay (i.e. the output of a unit will depend on the incoming input as well its own output at the previous time step) as in Figure 1. This structure would allow the RNN to be able to retain information through time when training for temporal data. RNNs are usually trained using BPTT (Backpropagation Through Time) and gradient descent based optimization methods.

2.1 Backpropagation Through Time (BPTT) and vanishing/exploding gradients

Backpropagation Through Time is extension of standard backpropagation algorithm which is used to calculate error gradients with respect to weights in normal feed-forward networks. BPTT calculates error gradients w.r.t weights by unfolding the given RNN in time and then backpropagating errors using chain rule of differentiation like standard backpropagation. The effect of vanishing gradients in the earlier layers of the network which is observed in the deep feed-forward networks with normal sigmoid activations is much more pronounced in RNNs as even a shallow RNN trained on a long enough sequence is very deep in time and so suffers vanishing gradients. Due to this vanilla RNNs trained with BPTT quickly lose information from back in time and are not able to model temporal data efficiently.

$$\boldsymbol{o_{t+1} = \phi(V s_{t+1} + b_o)}$$
$$\boldsymbol{s_{t+1} = \phi(W s_t + U x_{t+1} + b_s)}$$

Here ϕ is a sigmoid or tanh or any other activation function. Typically here o_t, s_t, x_t are all implemented as vectors and U, V, W are the weight matrices between the corresponding layers and b_o and b_s are the biases.

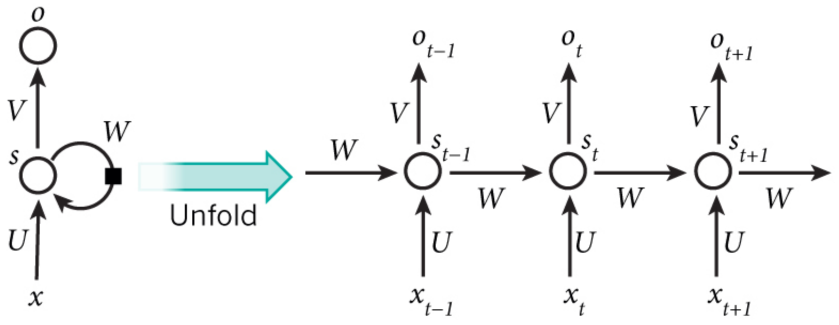

Figure 1: A simple RNN with self recurrent loop on left hand side and unrolled version of it on right hand side. Here x_t, s_t and o_t are the input, output of the hidden layer and the final output at time t respectively and U, V, W are the weights between the corresponding layers. Image from http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

2.2 Long Short Term Memory (LSTM)

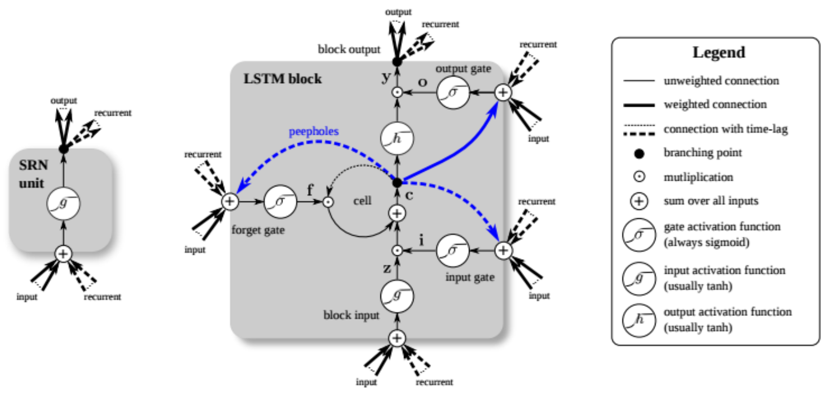

Figure 1. Detailed schematic of the Simple Recurrent Network (SRN) unit (left) and a Long Short-Term Memory block (right) as used in the hidden layers of a recurrent neural network.

Figure 2: Image from http://deeplearning4j.org/img/greff_lstm_diagram.png

LSTMs are a type of recurrent units which avoid vanishing gradients due to their specific structure. Central to an LSTM block is a cell state variable c which runs through time apart from the self recurrent loop which is similar to simple RNNs. The cell provides gates to append/remove information to/from the cell state using information from the input as well the previous output. Finally the cell feeds the updated cell state to itself identically (i.e. without applying any function) at the next time step and outputs value which depends on the input, previous output and the cell state.

LSTMs are able to avoid vanishing gradient as the cell state can pass through time easily for it undergoes changes only by linear interaction of gates and it is fed back identically at next time step (Constant Error Carousel).

We use vanilla LSTMs for our network architecture and thus will not have any peephole connections.

2.2.1 Inside LSTM block

Forget Gate Action: As seen in the figure, the lstm block feeds the cell state c at previous time step and updates it by *multiplication* with the output of the forget gate *(f)*. The output from the forget gate is calculated by passing the output of the block at previous time step (denoted by *"recurrent"*) and the input at current time step (denoted by *"input"*) through a normal layer of sigmoid units. Here the sigmoid squash value between 0 to 1 acts a factor of the amount of information which should be retained from the previous cell state to the updated one.

$$\boldsymbol{f_t = \sigma(W_f[y_{t-1}, x_t] + b_f)}$$

Here x_t is the input at time t and y_{t-1} is the output of the unit at time $t-1$ and W_f is the corresponding weight matrix and b_f are biases. $[y_{t-1}, x_t]$ denotes concatenation of y_{t-1} and x_t

Input Gate Action: As seen in the figure, the input gate provides with new information from the current input (*"input"*) and the previous output(*"recurrent"*) by passing them through sigmoid layer. And the *tanh* layer (*"block input"*) acts as a generator of new values for the cell state c. Finally the cell state modified by the forget layer is *added* with *multiplication* of values from the *sigmoid (i)* and the *tanh (z)* layers.

$$\boldsymbol{i_t = \sigma(W_i[y_{t-1}, x_t] + b_i)}$$
$$\boldsymbol{z_t = g(W_z[y_{t-1}, x_t] + b_z)}$$
$$\boldsymbol{C_t = f_t \star C_{t-1} + i_t \star z_t}$$

Here g is the *tanh* activation function. $\star$ denotes dot product.

Output Gate Action: Finally to output a value from the block at current time step, the output gate takes input from the current time step (*"input"*) and the output from the previous time step (*"recurrent"*) and passes them through a sigmoid layer and then append (by *multiplication*) this output o to the value from the current cell state which is first passed through a *tanh* function. Finally the block outputs this value y which is also fed back to itself at the next time step. The cell state c is fed back identically to be appended with the output of the forget layer at the next time step and so on.

$$\boldsymbol{o_t = \sigma(W_o[y_{t-1}, x_t] + b_o)}$$
$$\boldsymbol{y_t = o_t \star g(C_t)}$$

3 Tensorflow and TFLearn

We use TFLearn python library [4] for defining and training our RNNs. TFLearn provides a Higher level API to Tensorflow [3] which is a deep learning framework developed and open sourced by Google in November 2015. Tensorflow in itself provides a python API which is used to define computation graphs for building and training network architectures. These computation graphs are then executed by C/C++ backend to the python API which tends to be much faster when compared to standard python code.

4 Methods

For carrying out training we use time windows as input to the network. The network consists of an input layer, a hidden layer with LSTM units and an output layer with sigmoid/tanh/linear units. The weights and biases of the network are initialized using truncated normal. *Target replication* strategy [5] is used where in outputs are obtained at each time step and total error over all time steps is accumulated. The network is trained using stochastic gradient descent (SGD or momentum without using any mini batches. In all the experiments, the data was prepared by sliding windows where slide step of 1 is used for all experiments. ($slide\_time\_window = 1$)

For forecasting values, we train this network against the objective of minimizing total mean squared error over all samples.
Error in a unit of output layer:

$$L = (t - y)^2$$

For the classifier, if the glucose level is < 80 mg/dL, the condition represents hypoglycemia and the corresponding output vector is set as [1,0,0]. Similarly for glucose level between 80 and 140 mg/dL (normoglycemia), the output vector is [0,1,0] and for > 140 mg/dL (hyperglycemia) is [0,0,1]. The output layer is a softmax output layer which gives the probability distribution over all the classes and the model is trained against minimizing a cross entropy loss which is given by:

$$L = -(t \log_e y + (1 - t) \log_e (1 - y))$$

Where t is the target (0 or 1) and y is the output of a unit (value between 0 and 1).
Activation of a softmax unit:

$$y_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Where y_j is the output of the j^{th} unit in the output layer and z_j is the total weighted input (logit) in the j^{th} unit and the summation is over all the k units of the output layer.

For multi step prediction, mainly three different approaches are used:
(i) Train the network to predict the corresponding value directly (*model-1*) (referred to as *direct* strategy in [6])
(ii) Train the network to predict all values of the series till the required value and compare the last required value (*model-2*) (referred to as *MIMO* (multiple input multiple output) in [6])
(iii) Train the network to predict only the next value and then obtain the value at required number of steps ahead in time by sequentially using the predicted value as input and reaching to the required value (*model-3*) (referred to as *recursive* strategy in [6])

5 Results

5.1 Sine Curve

We initially test the model on a sine curve to see how the model performs for single and multi step ahead forecasting on it. For this sine data was generated between 0 to 20*pi in steps of 0.01 radians for testing and test data from 20*pi to 25*pi again in steps of 0.01 radians was used to evaluate the model. Mean Squared Error (MSE) is reported for each experiment along with all other details of the experiment run like number of units in hidden layer, optimizer used, learning rate, total number of training epochs and the prediction horizon in the plots itself. For the sine plots, the x-axis represents time step and on the y-axis is the value of sine. Here for the sine case, minibatches are used to speed up training but the batches are not shuffled.

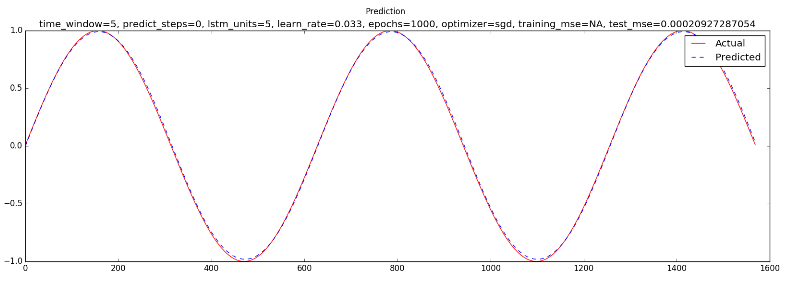

Figure 3: Model-1, test data, predict-steps=0

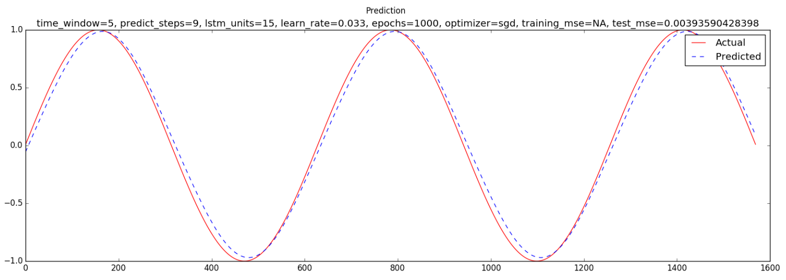

Figure 4: Model-1, test data, predict-steps=9

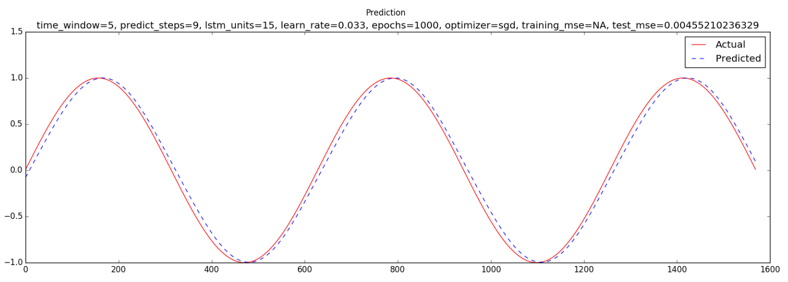

Figure 5: Model-2, test data, predict-steps=9

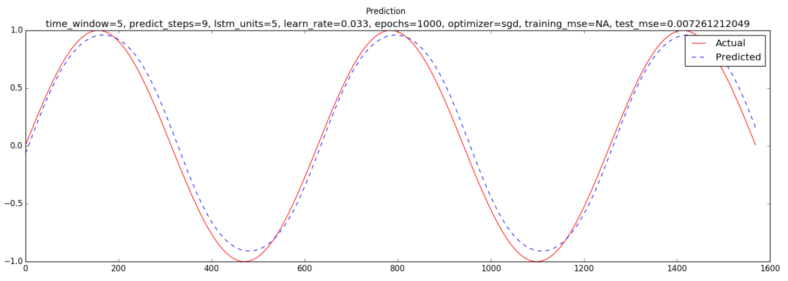

Figure 6: Model-3, test data, predict-steps=9

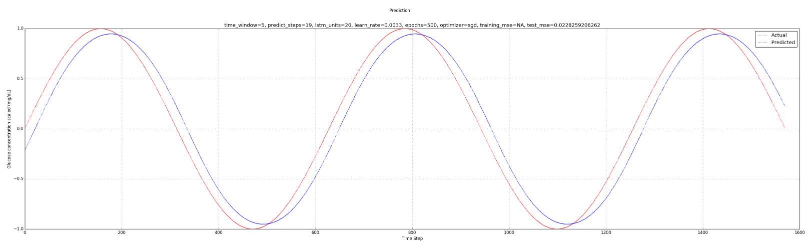

Figure 7: Model-1, test data, predict-steps=19

As observed from the plots, a systematic phase shift was observed between predicted and the actual waveform when prediction horizon is increased and this shift increases as the prediction horizon increases. This turns out only to be an optimization hurdle for the model rather than a problem with the model as when using the adaptive moments estimation (ADAM) [7] optimizer, the model was able to achieve the optimization objective for sine and able to predict for multiple number of steps ahead in time as is evident from the further plots. ADAM calculates "individual adaptive learning rates for different parameters from estimates of first and second moments of the gradient" and so provides an advantage over vanilla SGD which has a single learning rate for all the parameters of the model which also might be the reason for SGD optimization to get stuck in local error minima rather than reaching global minima.

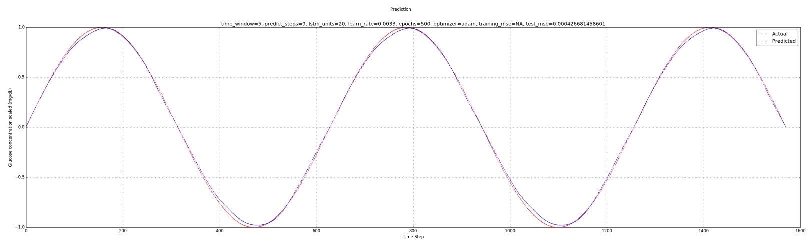

Figure 8: ADAM, Model-1, test data, predict-steps=9

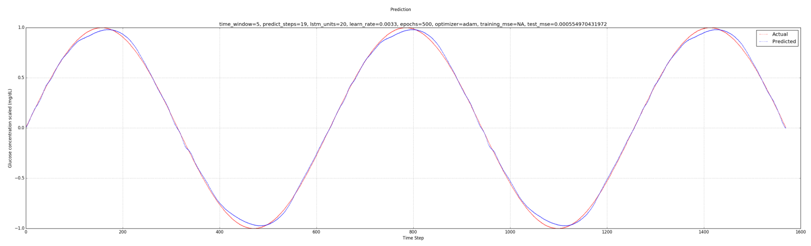

Figure 9: ADAM, Model-1, test data, predict-steps=19

5.2 Glucose Data

While working with actual data, we first scale the data between 0 to 1 so as to be able to use a simple network architecture as described earlier of only one hidden layer of lstm units. The scaling is done as following:

$$y_i = \frac{x_i - min\{x_k\}}{max\{x_k\} - min\{x_k\}}$$

Moreover the experiments for multi step prediction are mostly carried out using model-2 (the most) and model-1 rather than model-3. This is because, recursive strategy for multi step prediction is known to suffer from accumulation of errors at each step and generally compares poorly with MIMO or other strategy [6]. Also no minibatches are used here and so the weight updates are carried out at the end of each epoch only.

For single step prediction, the model performs well as evident from the plot.

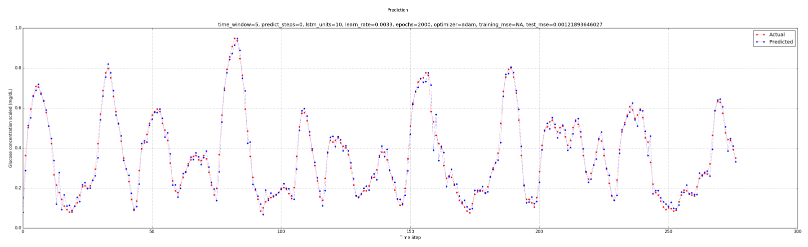

Figure 10: Model-1, test data, predict-steps=0

However, the model fares poorly for predicting 4 steps in future (i.e.) an hour later.
A variety of other experiments were run changing the size of the network, size of the input time window, training on unscaled data,etc. but none of them showed any improvement in the performance over 4-step ahead prediction on the glucose data. One of the reasons for this is that the model overfits the training data and so a large net using dropout [8] as regularisation in recurrent layers was also tried without success. Moreover a bidirectional [18] recurrent layer of lstm cells as hidden layer was also tried without success.

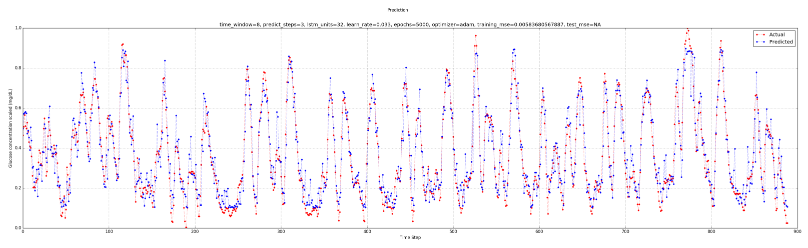

Figure 11: Dropout, Model-2, training data, predict-steps=3

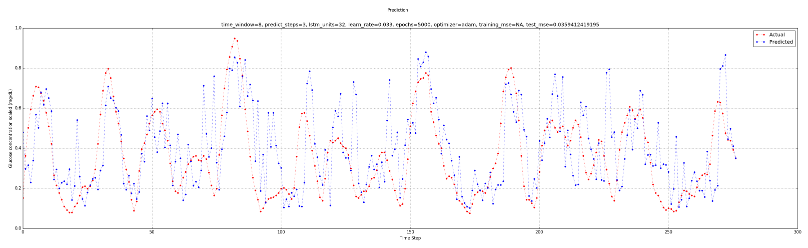

Figure 12: Dropout, Model-2, test data, predict-steps=3

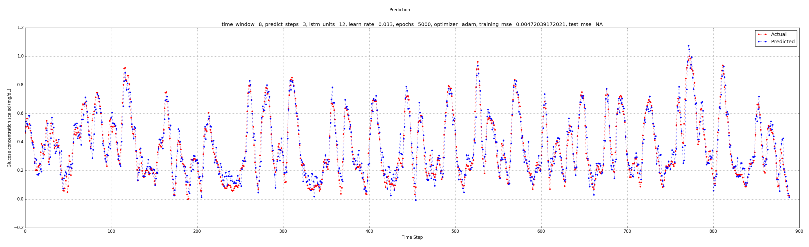

Figure 13: Bidirectional RNN, Model-2, training data, predict-steps=3

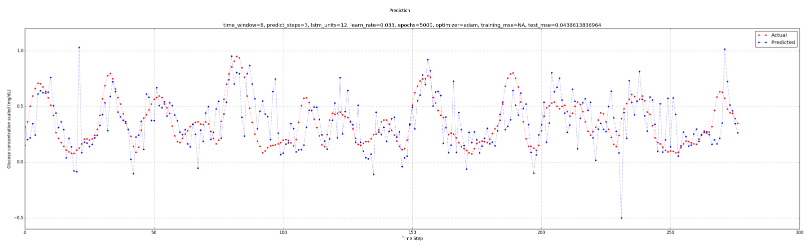

Figure 14: Bidirectional RNN, Model-2, test data, predict-steps=3

The glucose data waveform is also known to have a periodicity of 24 hours and so one more approach was tried wherein the input from a day ahead was also given to network along with the current input. This however didn't show any improvement in the performance of the model.

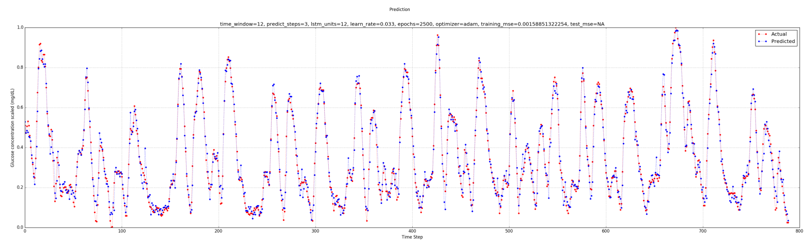

Figure 15: Previous Day input, Model-2, training data, predict-steps=3

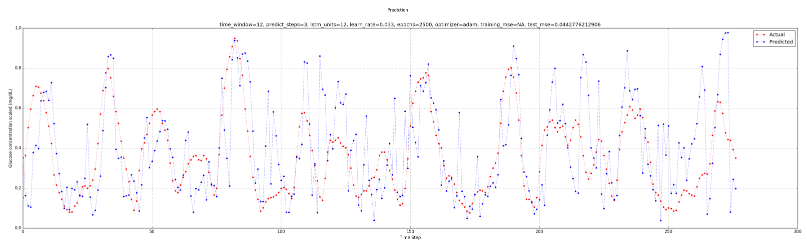

Figure 16: Previous Day input, Model-2, test data, predict-steps=3

Finally, we also tried a classifier based on the data where in the the objective was to predict the condition an hour later (hypoglycemia, normoglycemia or hyperglycemia) rather than predicting actual values. This was trained against the corresponding classification data which was prepared from the original glucose time series and trained on the network both using dropout as regularisation or not. For a classification accuracy of >90% on the training data, the accuracy obtained on test data was 60-70% (with or without dropout) which again shows overfitting and failing to learn a good enough representation of the entire process.

I also tried implementing architecture in which the units in the output layer are connected (for model-2). This would essentially allow the network internally access information of the prediction at the previous time step and so accordingly large/small dependecy on recent predictions can be adjusted within the network itself. However this was implemented too without any significant improvement.

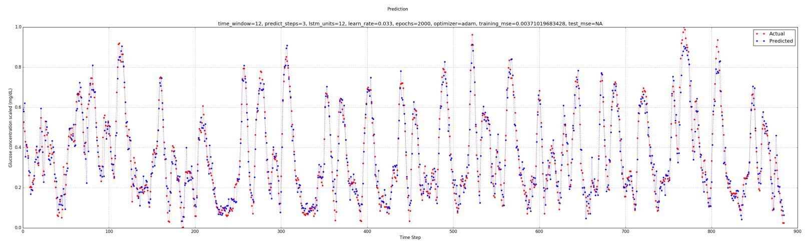

Figure 17: Output Units Connected, Model-2, training data, predict-steps=3

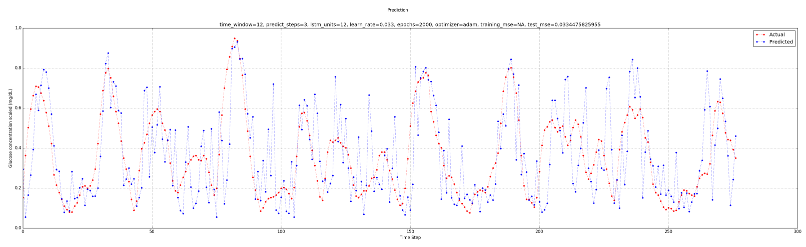

Figure 18: Output Units Connected, Model-2, test data, predict-steps=3

6 Conclusion and Further Possibilities

One of the main reasons which we suspect for the failure of the model for multi step prediction is the small amount of data. Total number of training samples are 900 and corresponding test samples are 277. Neural network models have known to require a lot of data for training efficiently and learn a good representation of the overall process [1]. This may be the primary reason that our model does not perform well for multi step ahead forecasting. However time series forecasting in general and particularly in it the multi step ahead forecasting are very difficult problems for which no general solutions exist [6] due to increase in uncertainity as the horizon of the prediction increases.

However, here our approach had been of modelling the time series without any preprocessing like removal of trends and deseasonalization which have potential for improvement in multi step ahead prediction [6].

References

[1] Dorffner, Georg. "Neural networks for time series processing." Neural Network World. 1996.

[2] Hochreiter, Sepp, and Jrgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.

[3] Abadi, Martn, et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." arXiv preprint arXiv:1603.04467 (2016).

[4] TFLearn: Deep learning library featuring a higher-level API for TensorFlow. *http://tflearn.org*

[5] Lipton, Zachary C., et al. "Learning to Diagnose with LSTM Recurrent Neural Networks." arXiv preprint arXiv:1511.03677 (2015).

[6] Taieb, Souhaib Ben, et al. "A review and comparison of strategies for multi-step ahead time series forecasting based on the NN5 forecasting competition." Expert systems with applications 39.8 (2012): 7067-7083.

[7] Kingma, Diederik, and Jimmy Ba. ”Adam: A method for stochastic optimization.” arXiv preprint arXiv:1412.6980 (2014).

[8] Srivastava, Nitish, et al. ”Dropout: a simple way to prevent neural networks from overfitting.” Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[9] Lu, Yinghui, et al. ”Predicting human subcutaneous glucose concentration in real time: a universal data-driven approach.” 2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE, 2011.

[10] Prasad, Sharat C., and Piyush Prasad. ”Deep Recurrent Neural Networks for Time Series Prediction.” arXiv preprint arXiv:1407.5949 (2014).

[11] Bon, Romuald, and Hubert Cardot. ”Advanced Methods for Time Series Prediction Using Recurrent Neural Networks.” Recurrent Neural Networks for Temporal Data Processing (2011): 15-36.

[12] Graves, Alex. ”Neural Networks.” Supervised Sequence Labelling with Recurrent Neural Networks. Springer Berlin Heidelberg, 2012. 15-35.

[13] Christopher Olah -Understanding LSTM Networks *http://colah.github.io/posts/2015-08-Understanding- LSTMs/*

[14] Guo, Jiang. ”Backpropagation through time.” Unpubl. ms., Harbin Institute of Technology (2013).

[15] Gers, Felix A., Douglas Eck, and Jrgen Schmidhuber. ”Applying LSTM to time series predictable through time-window approaches.” International Conference on Artificial Neural Networks. Springer Berlin Heidelberg, 2001.

[16] Hunter, John D. ”Matplotlib: A 2D graphics environment.” Computing in science and engineering 9.3 (2007): 90-95.

[17] Michael A. Nielson, ”Neural Network and Deep Learning”, Determination Press, 2015

[18] Schuster, Mike, and Kuldip K. Paliwal. ”Bidirectional recurrent neural networks.” IEEE Transactions on Signal Processing 45.11 (1997): 2673-2681.

Appendix

A Code

```
from __future__ import division, print_function, absolute_import

import tflearn
import numpy as np
import math
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import pandas
import tensorflow as tf

time_window = 12
slide_time_window = 1
predict_steps = 3 #Number of steps in future to predict. Value of zero will
#imply the next step. Value of 1 implies to predict the next to next step in
#future and so on. Use this for model 1 and 2
#learn_rate = 0.33
learn_rate = 0.033
epochs = 2000
future_steps = 3 #Similar to predict_steps. Use this for model3. When using
#model 3, keep slide_time_window = 1
optimizer = 'adam'
lstm_units = 12
minibatch = 885

def generate_sin_data(step_radians=0.01, n=20):
    """Generates sine data points between 0 to n*pi in steps of step_radians."""
    num = np.arange(0, n*math.pi, step_radians)
    x_train = np.sin(num)
    test_num = np.arange(num[len(num) - time_window - predict_steps],
    (n+5)*math.pi, step_radians)
    x_test = np.sin(test_num)
    return x_train, x_test

def load_glucose_data():
    """Loads glucose data and scales the data values between 0 and 1"""
    df=pandas.read_csv("glucose_original.csv")
    df=df["Historic_Glucose_(mg/dL)"]
    f=df.tolist()
    f=np.array(f)
    max_f = np.amax(f)
    min_f = np.amin(f)
    f = (f - min_f)/(max_f - min_f)
    f_train = f[0:900]
    f_test = f[len(f_train)-time_window-predict_steps:]
    return f_train, f_test


#Prepare Data For Training/Testing
def model1_prepare_data(x):
    seq = []
    next_val = []

    for i in range(0, len(x) - time_window - predict_steps, slide_time_window):
        seq.append(x[i: i + time_window])
        next_val.append(x[i + time_window + predict_steps])

    next_val = np.reshape(next_val, [-1, 1])
    seq = np.reshape(seq, [-1, time_window, 1])

    inputs = np.array(seq)
    targets = np.array(next_val)

    return inputs, targets

def model2_prepare_data(x):
    seq = []
    next_val = []

    for i in range(0, len(x) - time_window - predict_steps, slide_time_window):
        seq.append(x[i: i + time_window])
        next_val.append(x[(i + time_window): (i+time_window+predict_steps+1)])

    seq = np.reshape(seq, [-1, time_window, 1])
    #next_val = np.reshape(next_val, [-1, predict_steps+1, 1])
    inputs = np.array(seq)
    targets = np.array(next_val)

    return inputs, targets

def model2_last_values(actual, predict):
    last_actual = []
    last_predict = []

    for i in range(0, len(actual)):
        last_actual.append(actual[i][predict_steps])
        last_predict.append(predict[i][predict_steps])

    return last_actual, last_predict

def model3_prepare_train_data(x):
    seq = []
    next_val = []

    for i in range(0, len(x) - time_window, slide_time_window):
        seq.append(x[i: i + time_window])
        next_val.append(x[i + time_window])

    next_val = np.reshape(next_val, [-1, 1])
    seq = np.reshape(seq, [-1, time_window, 1])

    inputs = np.array(seq)
    targets = np.array(next_val)

    return inputs, targets

def model3_prepare_test_data(x):
    seq = []
    next_val = []

    for i in range(0, len(x) - time_window - future_steps, slide_time_window):
        seq.append(x[i: i + time_window])
        next_val.append(x[i + time_window + future_steps])

    next_val = np.reshape(next_val, [-1, 1])
    seq = np.reshape(seq, [-1, time_window, 1])

    inputs = np.array(seq)
    targets = np.array(next_val)

    return inputs, targets

def model3_predict(seq, model):
    predict_Y = []

    for j in range(0, len(seq)):
        testwindow = seq[j]
        rstestwindow = np.reshape(testwindow, [1,time_window,1])
        for i in range(0, future_steps+1):
            predict_tw = model.predict(rstestwindow)
            nextwindow = np.append(rstestwindow[0][1:], predict_tw)
            rstestwindow = np.reshape(nextwindow, [1,time_window,1])
        predict_Y.append(predict_tw)

    rspredict_Y = np.reshape(predict_Y, [-1,1])
    return rspredict_Y

def add_noise(x, std_dev=0.08, mean=0):
    """Adds gaussian noise to the input array"""
    x = np.array(x)
    noise = std_dev*np.random.randn(x.shape[0]) + mean
    x = x + noise
    return x

def glu_previous_window(train, test):

    def prepare(x, now):
        x = [list(i) for i in x]
        windows=[]
        for i in range(0, len(x) - time_window - predict_steps,
        slide_time_window):
            windows.append(x[i: i + time_window])
        inputs = np.array(windows)

        targets=[]
        for i in range(0, len(now) - time_window - predict_steps,
        slide_time_window):
            #targets.append(now[(i + time_window):
            #(i+time_window+predict_steps+1)])
            #targets.append(now[i+time_window+predict_steps])
            targets.append(now[(i+time_window):(i+time_window+predict_steps+1)])
        targets = np.array(targets)
        #targets = np.reshape(targets, [-1,1])
        return inputs, targets
    train_now = train[96:]
    train_previous = train[:-96]
    tr = zip(train_now, train_previous)
    train_inputs, train_targets = prepare(tr, train_now)

    leftover = train[-96:]
    total_test = np.concatenate((leftover, test))
    test_now=test
    test_previous=total_test[:-96]
    ts = zip(test_now, test_previous)
    test_inputs, test_targets = prepare(ts, test_now)

    return train_inputs, train_targets, test_inputs, test_targets

def model4_classifier(train, test):

    def prepare(x):
        seq = []
        next_val = []

        for i in range(0, len(x) - time_window - predict_steps, slide_time_window):
            seq.append(x[i: i + time_window])
            if x[i + time_window + predict_steps]<80:
                next_val.append([1,0,0])
            elif x[i + time_window + predict_steps]<140:
                next_val.append([0,1,0])
            else:
                next_val.append([0,0,1])

        seq = np.reshape(seq, [-1, time_window, 1])

        inputs = np.array(seq)
        targets = np.array(next_val)

        return inputs, targets

    train_inputs, train_targets = prepare(train)
    test_inputs, test_targets = prepare(test)

    return train_inputs, train_targets, test_inputs, test_targets

#16/237, 76/327

# Build Network
def build_net():
    net = tflearn.input_data(shape=[None, time_window, 1])
    #net = tflearn.fully_connected(net, 10, activation='sigmoid')
    #net = tflearn.reshape(net, [-1, 10, 1])
    #net = tflearn.lstm(net, dropout=(1.0, 0.5), n_units=lstm_units, return_seq=True)
    #net = tflearn.lstm(net, n_units=lstm_units, return_seq=True)
    #net = tflearn.lstm(net, n_units=lstm_units, return_seq=True)
    #net = tflearn.lstm(net, n_units=lstm_units, return_seq=True)
    #net = tflearn.lstm(net, n_units=lstm_units, return_seq=True)
    #net = tflearn.lstm(net, n_units=lstm_units, return_seq=True)
    #net = tflearn.lstm(net, n_units=lstm_units, return_seq=True)
    #net = tflearn.lstm(net, n_units=lstm_units, return_seq=True)
    #net = tflearn.lstm(net, n_units=lstm_units, return_seq=True)
    #net = tflearn.lstm(net, dropout=(1.0, 0.5), n_units=lstm_units, return_seq=True)
    net = tflearn.lstm(net, n_units=lstm_units, return_seq=False)



    #net = tflearn.fully_connected(net, 8, activation='sigmoid')
    #net = tflearn.dropout(net, 0.5)
    #net = tflearn.fully_connected(net, lstm_units, activation='relu')
    #net = tflearn.dropout(net, 0.5)

    #For model 2
    net = tflearn.fully_connected(net, predict_steps+1, activation='sigmoid')

    #For model 1 and 3
    #net = tflearn.fully_connected(net, 1, activation='linear')

    #For model 4
    #net = tflearn.fully_connected(net, 3, activation='softmax')
    #net = tflearn.regression(net, metric='accuracy', optimizer=optimizer,
    #loss='categorical_crossentropy',
    #learning_rate=learn_rate, shuffle_batches=False)

    net = tflearn.regression(net, metric=None, optimizer=optimizer, loss='mean_square',
    learning_rate=learn_rate, shuffle_batches=False)

    model = tflearn.DNN(net, clip_gradients=0.0, tensorboard_verbose=0)
    return model

def build_net2():
    """Output units connected"""
    net = tflearn.input_data(shape=[None, time_window, 1])
    #net = tflearn.fully_connected(net, 10, activation='sigmoid')
    #net = tflearn.reshape(net, [-1, 10, 1])
    #net = tflearn.lstm(net, dropout=(1.0, 0.5), n_units=lstm_units, return_seq=True)
    net = tflearn.lstm(net, n_units=lstm_units, return_seq=False)
    #net = tflearn.fully_connected(net, 32, activation='relu')
    unit1 = tflearn.fully_connected(net, 1, activation='relu')
    merge1 = tflearn.merge([net, unit1], 'concat', axis=1)
    unit2 = tflearn.fully_connected(merge1, 1, activation='relu')
    merge2 = tflearn.merge([net, unit2], 'concat', axis=1)
    unit3 = tflearn.fully_connected(merge2, 1, activation='relu')
    merge3 = tflearn.merge([net, unit3], 'concat', axis=1)
    unit4 = tflearn.fully_connected(merge3, 1, activation='relu')
    outputs = tflearn.merge([unit1, unit2, unit3, unit4], 'concat', axis=1)

    net = tflearn.regression(outputs, metric=None, optimizer=optimizer, loss='mean_square',
    learning_rate=learn_rate, shuffle_batches=False)

    model = tflearn.DNN(net, clip_gradients=0.0, tensorboard_verbose=0)
    return model

def build_net3():
    """Bidirectional RNN"""
    net = tflearn.input_data(shape=[None, time_window, 1])
    lstm1 = tflearn.BasicLSTMCell(num_units=lstm_units)
    lstm2 = tflearn.BasicLSTMCell(num_units=lstm_units)
    #net = tflearn.fully_connected(net, 10, activation='sigmoid')
    #net = tflearn.reshape(net, [-1, 10, 1])
    #net = tflearn.lstm(net, dropout=(1.0, 0.5), n_units=lstm_units, return_seq=True)
    net = tflearn.bidirectional_rnn(net, lstm1, lstm2, return_seq=False)
    #net = tflearn.fully_connected(net, 5, activation='linear')
    #net = tflearn.dropout(net, 0.5)
    #net = tflearn.fully_connected(net, lstm_units, activation='relu')
    #net = tflearn.dropout(net, 0.5)

    #For model 2
    net = tflearn.fully_connected(net, predict_steps+1, activation='linear')

    #For model 1 and 3
    #net = tflearn.fully_connected(net, 1, activation='linear')

    #For model 4
    #net = tflearn.fully_connected(net, 3, activation='softmax')
    #net = tflearn.regression(net, metric='accuracy', optimizer=optimizer,
    #loss='categorical_crossentropy',
    #learning_rate=learn_rate, shuffle_batches=False)

    net = tflearn.regression(net, metric=None, optimizer=optimizer, loss='mean_square',
    learning_rate=learn_rate, shuffle_batches=False)

    model = tflearn.DNN(net, clip_gradients=0.0, tensorboard_verbose=0)
    return model
#Train Network
def fit_model(model, trainX, trainY, num_epochs=epochs, validation=0.0,
    mini_batch=minibatch, save='No'):
    model.fit(trainX, trainY, n_epoch=num_epochs, validation_set=validation,
    batch_size=mini_batch, shuffle=False, show_metric=True)
    if save=='yes' or 'y' or 'Y' or 'Yes' or 'YES':
        model.save('model1')

#mini_batch = 887 for glucose
# for sine mini_batch = 6270

def MSE(actual, predict):
    actual = np.array(actual)
    predict = np.array(predict)
    squared_loss = np.square(predict - actual)
    total_squared_loss = (1/(actual.shape[0]))*np.sum(squared_loss)
    return total_squared_loss

def plot(actual, predict, train_loss='NA', test_loss='NA',
save='model1.png'):
    # Plot test results
    plt.figure(figsize=(40,10))
    plt.suptitle('Prediction')
    plt.title('time_window='+str(time_window)+
    ', predict_steps='+str(predict_steps)+
    ', lstm_units='+str(lstm_units)+
    ', learn_rate='+str(learn_rate)+
    ', epochs='+str(epochs)+
    ', optimizer='+str(optimizer)+
    ', training_mse='+str(train_loss)+
    ', test_mse='+str(test_loss))
    plt.xlabel('Time Step')
    plt.ylabel('Glucose concentration scaled (mg/dL)')
    #plt.plot(actual, 'r.', label='Actual')
    #plt.plot(predict, 'b.', label='Predicted')
    plt.plot(actual, color='red', linestyle='dotted', marker='.', markerfacecolor='red',
             markersize='8',            label='Actual')
    plt.plot(predict, color='blue', linestyle='dotted', marker='.', markerfacecolor='blue',
             markersize='8',    label='Predicted')
    plt.grid(b=True, which='major', linestyle='dotted')
    plt.grid(b=True, which='minor', linestyle='dotted')
    plt.legend()
    #plt.show()
    plt.savefig(save)

def model1_run():
    train, test = load_glucose_data()
    trainX, trainY = model1_prepare_data(train)
    testX, testY = model1_prepare_data(test)
    #trainX, trainY, testX, testY = glu_previous_window(train, test)
    model=build_net3()
    fit_model(model, trainX, trainY)

    train_predict = model.predict(trainX)
    test_predict = model.predict(testX)

    train_mse= MSE(trainY, train_predict)
    test_mse= MSE(testY, test_predict)

    print("Total training MSE is", train_mse)
    print("Total testing MSE is", test_mse)

    plot(trainY, train_predict,
    train_loss=train_mse, save='bi_glu_model1_train.png')
    plot(testY, test_predict, test_loss=test_mse,
    save='bi_glu_model1_test.png')


def model2_run():
    train, test = load_glucose_data()
    trainX, trainY = model2_prepare_data(train)
    testX, testY = model2_prepare_data(test)
    #print(np.shape(testX))
    #trainX, trainY, testX, testY = glu_previous_window(train, test)
    model = build_net()
    fit_model(model, trainX, trainY)


    test_predict = model.predict(testX)
    train_predict = model.predict(trainX)

    last_trainY, last_train_predict = model2_last_values(trainY, train_predict)
    last_testY, last_test_predict = model2_last_values(testY, test_predict)

    train_mse= MSE(last_trainY, last_train_predict)
    test_mse= MSE(last_testY, last_test_predict)

    def new1(allpredictions, lastvalues):
        """Ignore This"""
        lastvalues = np.array(lastvalues)
        lastvalues = np.reshape(lastvalues, [-1,1])
        inputs = allpredictions
        targets = lastvalues

        net = tflearn.input_data(shape=[None, 3])
        net = tflearn.fully_connected(net, 4, activation='sigmoid')
        net = tflearn.fully_connected(net, 1, activation='sigmoid')
        net = tflearn.regression(net, metric=None, optimizer='adam', loss='mean_square',
                                 learning_rate=0.01, shuffle_batches=False)
        model = tflearn.DNN(net, clip_gradients=0.0, tensorboard_verbose=0)
        model.fit(inputs, targets, n_epoch=20, validation_set=validation,
        batch_size=mini_batch, show_metric=True)


    print("Total training MSE is", train_mse)
    print("Total testing MSE is", test_mse)

    plot(last_trainY, last_train_predict,
    train_loss=train_mse,
    save='CONN_glu_model2_train.png')
    plot(last_testY, last_test_predict, test_loss=test_mse,
    save='CONN_glu_model2_test.png')

def model3_run():
    train, test = load_glucose_data()
    trainX, trainY = model3_prepare_train_data(train)
    testX, testY = model3_prepare_test_data(test)
    model = build_net()
    fit_model(model, trainX, trainY)
    test_predict = model3_predict(testX, model)

    test_mse= MSE(testY, test_predict)
    train_mse= 'NA'

    print("Total testing MSE is", test_mse)

    plot(testY, test_predict,test_loss=test_mse,
    save='glu_model3_test.png')

def model4_run():
    train, test = load_glucose_data()
    trainX, trainY, testX, testY = model4_classifier(train, test)
    #print(np.shape(trainX), np.shape(trainY))
    model=build_net()
    fit_model(model, trainX, trainY)

    print("Test accuracy is ", model.evaluate(testX, testY))


model2_run()
#train, test = load_glucose_data()
#plot(train, save='train_unscaled.png')
#plot(test, save='test_unscaled.png')
```