Information Retrieval and DataMining: Memory Models for Time Series Analysis

Mark Neumann, Alexander Chapovskiy and Zheng Tian
UCL
mark.neumann.15@ucl.ac.uk
alexander.chapovskiy.15@ucl.ac.uk
zheng.tian.11@ucl.ac.uk@ucl.ac.uk

Abstract

This is a report for group project in the Information Retrieval and Data Mining course. In this report we investigate different deep neural network architectures on the time series. In particular, we apply neural networks to energy load and household energy consumption datasets.

1 Introduction

1.1 Memory Models

Long Short-Term memory[6] has been utilised widely across many disciplines in Machine Learning, with particularly notable performance in domains such as Machine Translation[3], Constituency Parsing[5], Speech Recognition and even generative models, such as human level handwriting generation.[4]

Particularly key in the success of LSTM and other variants, such as the GRU, has been their perceived ability to selectively encode long-term dependencies and learn precise timings. In this work, we investigate the effect of precise timings in accurately modelling time-series data, an area which should particularly benefit from the accuracy features provided by Long Short-Term memory.

1.2 Other Model Architectures

One perceived failure of neural networks until recently was their inability to deal with arbitrary sequence lengths. The input dimensions to a standard LSTM classifier/regressor are fixed, meaning that data which intrinsically contains irreducible sequences of different lengths was difficult or impossible to process. This clearly is a problem with time series data, as the noisy nature of real world data means that we are rarely going to achieve perfectly timed and sanitised data.

In [3], Sutskever introduces sequence to sequence neural networks, which utilise two separate networks, one to encode a sentence into a vector representation and the other to decode a vector representation into the model ouptut. These are trained jointly, enabling arbitrary length sequences and particularly enabling non-monotonic transformations. We note that although we do not explore sequence to sequence models in this paper, the underlying cell structure we explore can be used in this architecture and that this is a very interesting area of development for time series models given their nature.

1.3 Report organisation and code

First, we will describe in some detail the differences between the three memory cell configurations we are testing. We then go on to describe the datasets we have produced results on and then we present the results on each of these datasets in order. Note that in order to investigate a multitude of ideas, we have not completed the same analysis on both data sets. On the UCI dataset we explore memory models in some depth, focusing on the internal cell mechanisms. On the Kaggle energy dataset, we explore some more complex architectures, to see if combining sources of information would improve performance.

All code used to generate the results in this paper can be found in the following Github repository: `https://github.com/ucabmrn/ir_group_project`.

2 Model definitions and diagrams

Below we describe the models applied to the dataset, for which performance can be found in the next section. In addition to vanilla Long Short-Term Memory, we explore several recent innovations over the last few years, including Gated Recurrent Units and peephole connections.

2.1 LSTM

Given an input vector $x_t \in \mathbb{R}^N$ at time t and the previous output of the LSTM $h_{t-1} \in \mathbb{R}^K$, the forget gate(f_t), the input gate(i_t), the output gate(o_t) and the cell input(z_t) are computed as follows:

$$\begin{aligned} H &= \begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} \in \mathbb{R}^{N+K} & o_t &= \sigma(W_o H + b_o) \\ & & f_t &= \sigma(W_i H + b_i) \\ z_t &= Tanh(W_z H + b_z) & c_t &= f_t \odot c_{t-1} + i_t \odot z_t \\ i_t &= \sigma(W_i H + b_i) & h_t &= o_t \odot Tanh(c_t) \end{aligned} \tag{1}$$

Where $W_z, W_i, W_o, W_f \in \mathbb{R}^{K\times(N+K)}$ and $b_z, b_i, b_o, b_f \in \mathbb{R}^K$ are trained weight matrices and bias vectors for N, the dimension of the input vector and K the hidden size of the LSTM. σ and $Tanh$ are element-wise activation functions representing the sigmoid and hyperbolic tangent functions, with $\odot$ representing element-wise multiplication.

Below we present a pictorial demonstration of the connections in the LSTM cell.

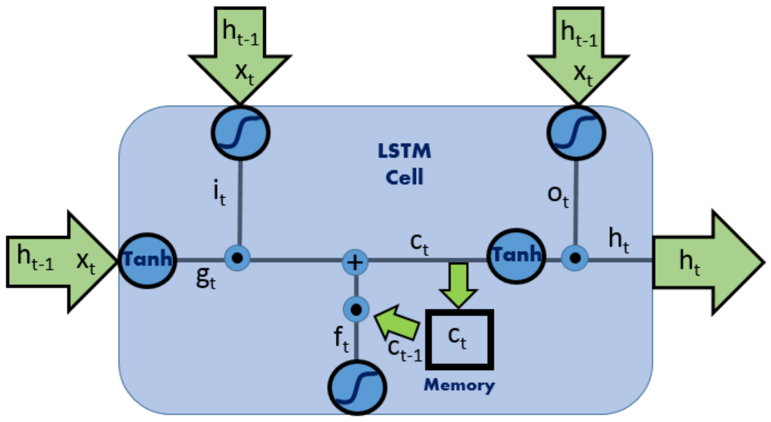

Figure 1: A demonstration of the interactions within an LSTM cell. Here we can see clearly how the input and output control the respective features of the cell: the closer to zero these vectors are, the smaller the amount of information which can travel through the dot product.

2.2 Gated Recurrent Unit

The Gated Recurrent Unit[1] can be seen as a simplified LSTM with a tied output and forget gate, merging two of the above equations.

Given an input vector $x_t \in \mathbb{R}^N$ at time t and the previous output of the GRU $h_{t-1} \in \mathbb{R}^K$, the forget gate(f_t), the input gate(i_t) and cell output(h_t) are computed as follows:

$$\begin{aligned} &Input : x_t && f_t = \sigma(W_f^x x_t + W_f^h h_{t-1} + b_f) \\ &PrevState : h_{t-1} && \tilde{h_t} = Tanh(W x_t + f_t \odot h_{t-1}) \\ &i_t = \sigma(W_i^x x_t + W_i^h h_{t-1} + b_i) && h_t = i_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h_t} \end{aligned} \tag{2}$$

Where $W_i^x, W_f^x, \in \mathbb{R}^{K \times (N)}, W_i^h, W_f^h \in \mathbb{R}^{K \times (K)}$ and $b_i \in \mathbb{R}^N, b_f \in \mathbb{R}^K$ are trained weight matrices and bias vectors for N, the dimension of the input vector and K the hidden size of the GRU. σ and $Tanh$ are again element-wise activation functions representing the sigmoid and hyperbolic tangent functions, with $\odot$ representing element-wise multiplication as above.

It should be noted that subsequent papers by Cho expand on the GRU, particularly to incorporate "recurrent feedback", intuitively described as a linear combination of memory storage across hidden layers. This would certainly be an interesting extension to the work in this paper.

2.3 Peephole Connections

Peephole connections are a popular addition to the Long Short-Term memory model[2]. Originating in a paper by Schmidhuber and Gers whilst working at the IDSIA lab in Switzerland, they quickly became popular in an array of literature. The basic premise is that the gates of the LSTM mechanism become a function not only of the current input x_t and previous LSTM state h_{h-1}, but also a function of the current cell memory state which persists from the last timestep, c_{t-1}. The input, output and forget gates are therefore defined as follows:

$$H = \begin{bmatrix} x_t \\ h_{t-1} \\ c_{t-1} \end{bmatrix} \in \mathbb{R}^{N+2K} \qquad \begin{aligned} i_t &= \sigma(W_i H + b_i) \\ f_t &= \sigma(W_f H + b_f) \\ o_t &= \sigma(W_o^x x_t + W_o^h h_{t-1} + W_o^c c_t + +b_o) \end{aligned} \tag{3}$$

where the memory is simply appended onto the vector used in the LSTM mechanism for the calculation of the input and forget gates, with the output gate using a different format only because it uses the *current* state's memory, rather than the previous one as in the input and forget gates. The other functionality remains unchanged. Additionally, some practitioners choose to only apply peephole connections to gates selectively - often, they are only applied to the input and output gates and not the forget gate. Note the effect of peephole connections is difficult to measure accurately, as in a vanilla LSTM, all the gates are implicitly a function of the memory vector, as they are a function of the previous state which is informed by the memory at that point. Below we present a pictorial representation of how the addition of peephole connections changes the internal cell architecture.

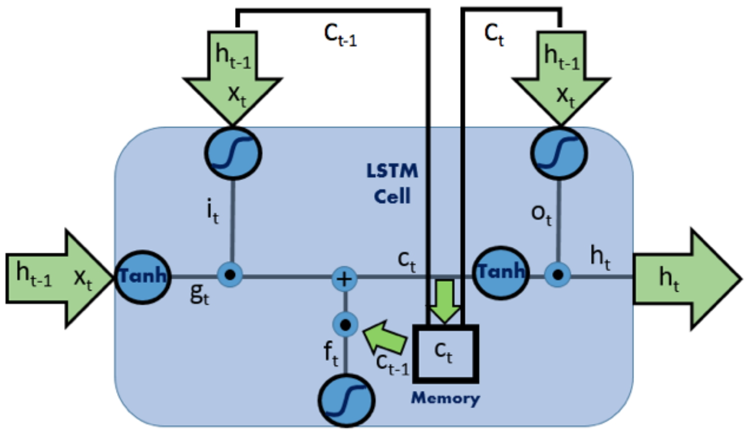

Figure 2: Here we can clearly see how all gates now have direct interaction with the memory state.

3 Datasets

3.1 Kaggle Energy Load Data

In this section we describe a dataset of Energy Load. The dataset, originally used in a Kaggle competition, contains energy load for 20 power stations over approximately 4 years every hour with some weeks missing. The dataset also provides temperatures at 11 weather stations.

Here our goal is to investigate application of neural networks to time series data. Therefore we focus on one week ahead prediction of the energy load at each of 20 power stations using temperatures during that period and temperatures and loads prior to that week.

After the competition the winners of the competition published their approaches, [7]. Mostly the approaches were based on linear regression with hand crafted non-linear features using various improvements such as gradient boosting.

In Section 4 we apply a regression model with handcrafted non-linear features to serve as a benchmark. We then discuss deep feed-forward Neural Network with simple variables, such as temperature, day of the week, hour, *etc*, for week ahead prediction to see if the network is able to discover relevant features. We then discuss a Recurrent Neural Network, which additionally takes prior week loads and temperatures, in order to see if RNN is able to discover additional regularities, not captured by the variables above.

3.2 UCI Household Energy Consumption Data

This dataset comes from UCI-Machine Learning Repository. It records an individual household electric power consumption with a one-minute sampling rate over a period of almost 4 years. Different electrical quantities and some sub-metering values are included. Additionally, this dataset contains some missing values in the measurements (nearly 1.25 percent of the rows), which will provide an opportunity to measure model robustness in addition to performance.

Our task is to predict household global minute-averaged active power (third attribute in raw data) from history. We build a 50 step-length window where we predict the last step's global active power based on the counterpart from previous 49 steps. We slide the window from the beginning to the end of the data set so that we could predict minute by minute power output over the whole window.

For this dataset, we examine the performance of recurrent neural networks making use of LSTM, GRU and LSTM + Peephole Connection cells.

4 Results on the Kaggle Energy Load Dataset

In this section we discuss different models applied to the Energy Load dataset described in Section 3.1. We first discuss a simple regression model with non-linear features in Section 4.1, which serves as a benchmark for more complicated models. In Section 4.2 we apply deep neural network with simple inputs to the dataset in order to see if the network is able to find relevant non-linear features. In Section 4.3 we apply Recurrent Neural Network, with prior week's temperatures and load as inputs, to the dataset in order to see if the network is able to find some additional regularities in the historical information.

In order to compare the performance of different models we formulate the problem in the following way. We split all the data into overlapping 2 week periods, each week from Monday to Sunday. In each pair of weeks the goal is to predict energy loads for 20 stations in the second week using temperatures from 11 weather stations in the second week and energy loads and temperatures in the first week. Thus, for every pair of weeks we have to make $20 \times 7 \times 24$ load predictions. This way we have about 200 week-pairs in total. We allocate 20% of the data for testing, with remaining 80% of the data used for training and validation. We optimise all models using mean square error objective function

$$E_{\text{mse}} = \frac{1}{N} \sum_i \left(L^{(i)}_{\text{predicted}} - L^{(i)}_{\text{actual}} \right)^2 , \tag{4}$$

where $L^{(i)}_{\text{predicted}}$ and $L^{(i)}_{\text{actual}}$ are predicted and actual energy load respectively and the sum is over all week, week day, hour, zone combinations. For convenience we compare performance of different models using dimensionless measure (relative square root of mean square error)

$$E_{\text{rmse}} = \frac{\sqrt{E_{\text{mse}}}}{\overline{L}_{\text{actual}}}, \tag{5}$$

where $\overline{L}_{\text{actual}}$ is the average load over the entire dataset.

4.1 Features and Regression Model

In order to construct a baseline to examine our results, we perform linear regression with non-linear features on batches of one week's worth of data. It is natural to expect that energy load in zone, $L_z(t)$, is a function of hour, h, week day, w, and temperature at all weather stations, T_i, as it is not know which weather station is relevant for zone z. It is also natural to expect that the load is correlated with the load at the same zone during the prior week. Therefore we use the following features

- **Bias**.
- **Temperature**. We use linear and quadratic features, T_i and T_i^2, for all weather stations, i.
- **Hour**. We use features up to the third power, h, h^2 and h^3.
- **Day of the week**. We use linear and quadratic features, w and w^2.
- **Previous Load**. 20 zone loads averaged over the previous week.

Thus we have 48 features in total.

We calculate feature vector for every every week/week-day/hour combination, normalise the feature vector over the data and run regularised regression on normalised feature vector. The output of the regression is 20-dimensional vector of energy load predictions for every zone and every week/week-day/hour. We obtain relative error of $E_{\text{rmse}} = (15\%, 16\%)$ for training and test sets respectively[1]. The errors for training and test sets are the same indicating lack of over-fitting. Fig. 3 shows predicted and observed energy loads for two different zones and weeks. One can see that our simple regression model, while not perfect, captures roughly the behaviour of the energy loads.

[1] The model is implemented in `model_reg.py`.

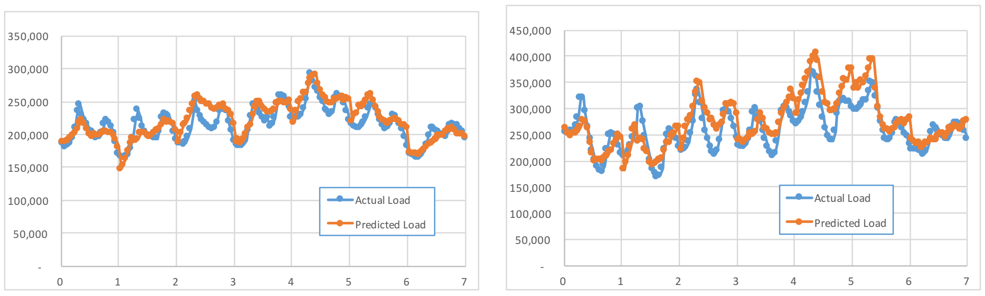

Figure 3: Predicted and actual energy loads for two different weeks and zones. Prediction is done by regression model.

4.2 Feed-Forward Neural Network

In Section 4.1 we implemented a simple regression model with non-linear features. There we chose non-linear features by hand using domain knowledge. In this section we implement a deep feed-forward neural network in order to see if the networks will be able to find relevant features automatically.

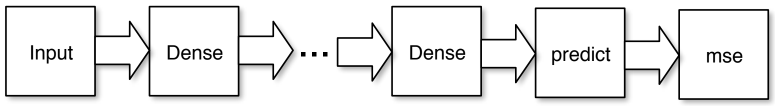

Figure 4: Architecture of feed-forward neural network.

Fig 4 shows schematically the architecture of the model. Different elements are the following

- **Input**. The input contains the following information: week-day, hour, 11 temperatures at the same hour, energy load averaged over the previous week for 20 zones. There are 33 inputs in total.
- **Dense**. The input is passed to a series of standard fully connected layers with `ReLu` non-linearity. By looking at the validation set error we found that 8 dense fully connected layers each with dimension 33, the same as the input dimension,works the best.
- **Predict**. The output of the dense network is passed to 20-dimensional prediction layer.
- **MSE**. Mean square loss objective, which is minimised.

Minimization of all the weights is performed by stochastic gradient decent. Implementation[2] is using Keras/Theano[8] for NN code, which relies on Cuda to run on GPU. As before we use 20% of the whole data for final testing (7,056 observations). The remainder we split into 20% for validation set (5,544 observations), and the rest is used as a training set (2,2848 observations).

Fig. 5 shows relative error as function of learning epoch for training, validation and test data. One can see that in the very end training and validation errors start to diverge. We adopt early stopping by monitoring validation set error. At the minimum of validation error we obtain errors $(12.5, 12.5, 13.5)\%$ for training, validation and test sets respectively. Thus should be compared with $(15, 16)\%$ errors for training and test sets we obtained with regression. NN performance is slightly better. Thus we can conclude that NN did not only find the hand-made features we used in regression, but also some additional features, which we missed.

[2] Implementation of the model can be found in directory `KerasRNN` in `model1_2.py`, which is used by `train1.py`.

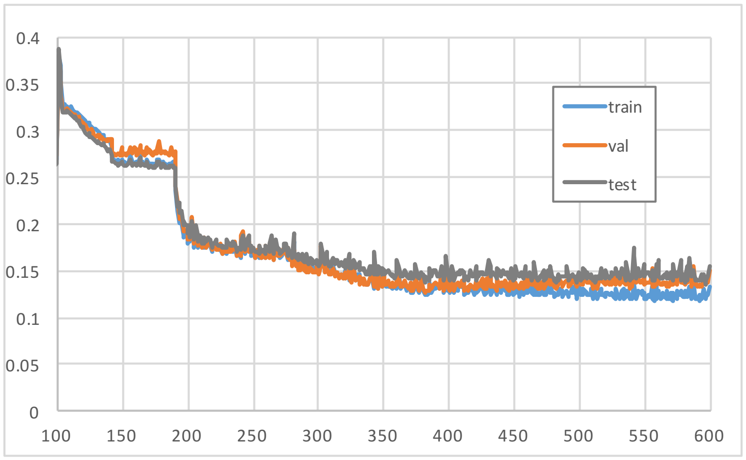

Figure 5: Relative error as function of learning epoch for training, validation and test data.

4.3 Recurrent Neural Network

In Section 4.2 we discussed a fully connected feed-forward neural network, which managed to discover non-linear dependence of load as a function of hour, week day, temperature, and average load in the previous week. Average load in the previous week was one of the inputs, which we had to provide explicitly. In this section we discuss Recurrent Neural Network to which we pass the previous week's history as an input. We would like to see if the network is able to automatically find relevant features of the history useful for predicting the load.

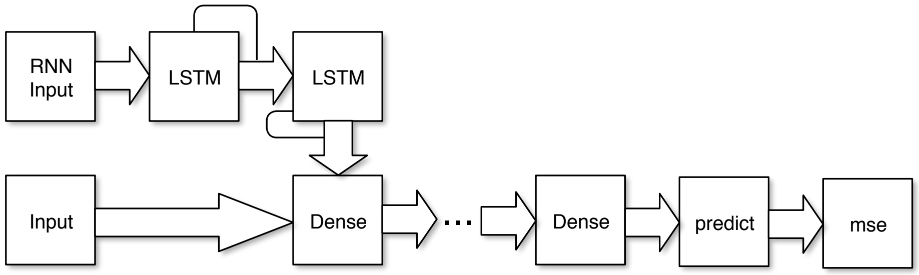

Figure 6: Architecture of RNN with feed-forward neural network.

Fig. 6 shows the architecture of the network.

- **Input**. The input contains the following information: week-day, hour, 11 temperatures at the same hour. There are 13 inputs in total. Note, that we do not provide average of the history as before.
- **RNN Input**. Here we would like to provide the whole previous week history of loads and temperatures. However, there are $24 \times 7 = 168$ observations, which makes RNN network very deep and we were not able to successfully train it. In order to overcome this difficulty, we pass 7 loads and temperatures averaged over each day of the previous week. This way he hope the network will discover regularities over the previous week but not intra-day. Thus RNN input consists of 7 $(20 + 11)$-dimensional vectors.
- **LSTM**. RNN input is passed to two LSTM layers with $2 \times (20 + 11) = 62$ hidden states. Each LSTM uses $tanh$ non-linearity.
- **Dense**. Output of LSTM layer is combined with Input and is passed to a series of dense layers, similar to Section 4.2. By looking at validation error we found that 4 dense layers,

each layer having $2 \times (20+11)+13 = 44$ states, works pretty well. Each Dense layer has *ReLu* non-linearity.

- **Predict**. The output of the dense network is passed to 20-dimensional prediction layer.
- **MSE**. Mean square loss objective, which is minimised.

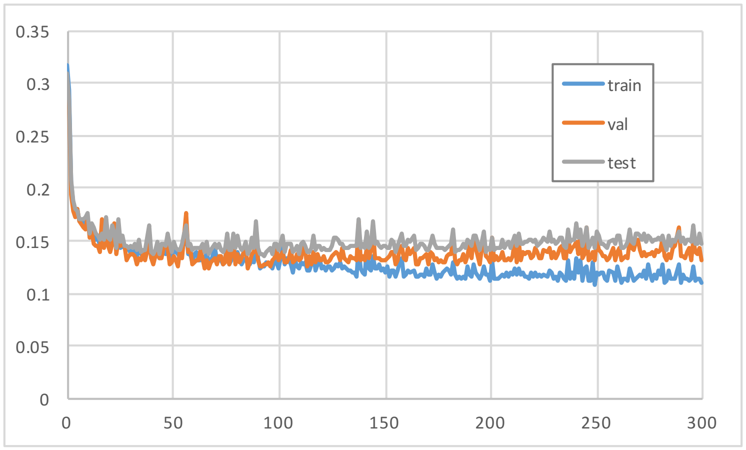

Figure 7: Error of the RNN-dense network as function of training epoch for training, validation and test sets.

We implemented[3] this model using Keras/Theano [8] for neural network implementation, which in turn uses Cuda for GPU calculations. The network is trained using stochastic gradient decent. Fig. 7 shows dependence of the relative mean square error as a function of training epoch for training, validation and test sets. One can see that three curves converge very quickly and are generally close to each other. In the later stages of the training they exhibit slight over-fitting, so we apply early stopping by monitoring validation error. **We obtain errors of (13,12.5,13.5)% for training, validation and test sets respectively. This should be compared with (12.5,12.5,13.5)% we obtained with NN in Section 4.2**. One can see that results are comparable. This means that our RNN managed to find the only relevant feature from the history, which is the average load over the previous week. This is quite disappointing as the model is quite a bit more complicated. However, it is quite possible that dependence on the history (additionally to average) is not large in our dataset. Also, it does find the average load, which was provided by hand previously, which is quite satisfying, as in principle this was not guaranteed.

It has to be noted that in the calculations above, instead of passing all 24×7 $(20+11)$-dimensional vectors which we could not do for computational reasons, we pass 7, each representing average over each day of the week. In this way we hoped that the network would find *"long range"* regularities on the scale of the week. It is possible that there are regularities, which are lost, when averaging intra-day. For example, it is possible that load over a day has some particular structure, which would help prediction.

In order to investigate this dependence we implemented[4] a model with architecture shown in Fig. 8. In that model we have three different inputs: Input and RNN1 Input are as before. RNN2 Input is new and consists of 24 $(20+11)$ dimensional vectors representing average load and temperature over the previous week for every hour. RNN2 input is passed to its own LSTM layer, which is then combined with the rest of the network and fed into dense layers, like before. We do not give further details of the calculations because, although this model produces results comparable to what we obtained before, unfortunately, it does not result in any significant improvements. Again, it is possible that there is no any further signal from history due to the nature of the data. On the other hand, this version of the model is quite a bit more complicated and is harder to train.

[3] Implementation can be found in directory `KerasRNN` in file `model2_1.py`, which should be used together with `train2.py`

[4] Implementation can be found in directory `kerasRNN` in file `model.py` to be used with `train.py`.

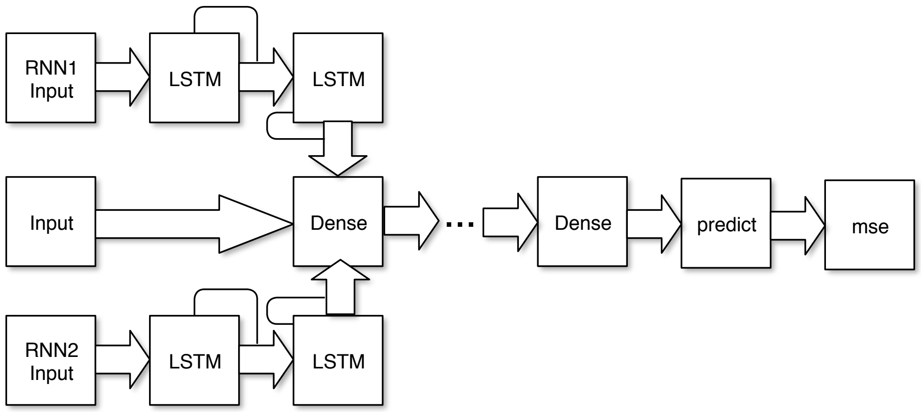

Figure 8: Architecture of double-RNN with feed-forward neural network.

5 Results on the UCI Household Energy Consumption Dataset

For the UCI dataset, we chose to implement the same neural network with 3 different layer structures, namely an LSTM cell, GRU cell and an LSTM with peephole connections, as described above.[5]

For the model architecture, we used a 3 layer Recurrent Neural Network, making use of the different memory cells described above. The hidden dimension of the RNN we used was 3,50,1 per layer respectively, where the last layer is a fully connected with linear activation functions. For the GRU and LSTM models, we made use of the Keras library with a Theano[8] backend. For the Peephole layer, we implemented a custom layer using Keras' RNN interface. [6]

We found that all three models performed extremely well on the prediction task, all achieving around 95% accuracy. However, we expected the Peephole connections to improve the deviation where timing is important, and as we expected, areas of volatility are more accurately matched by the LSTM with peephole connections than the other models. **We achieved a MSEs of 0.046, 0.048 and 0.042 for the LSTM, GRU and LSTM + Peephole models respectively**. Although this difference in performance seems minor, the peephole addition provides a **relative decrease in error of 9%**.

[5]all (separate) model scripts can be found in the code/LSTM_Household directory

[6]This custom layer implementation can be found in /code_keras/kerasRNN/PeepholeLayer.py

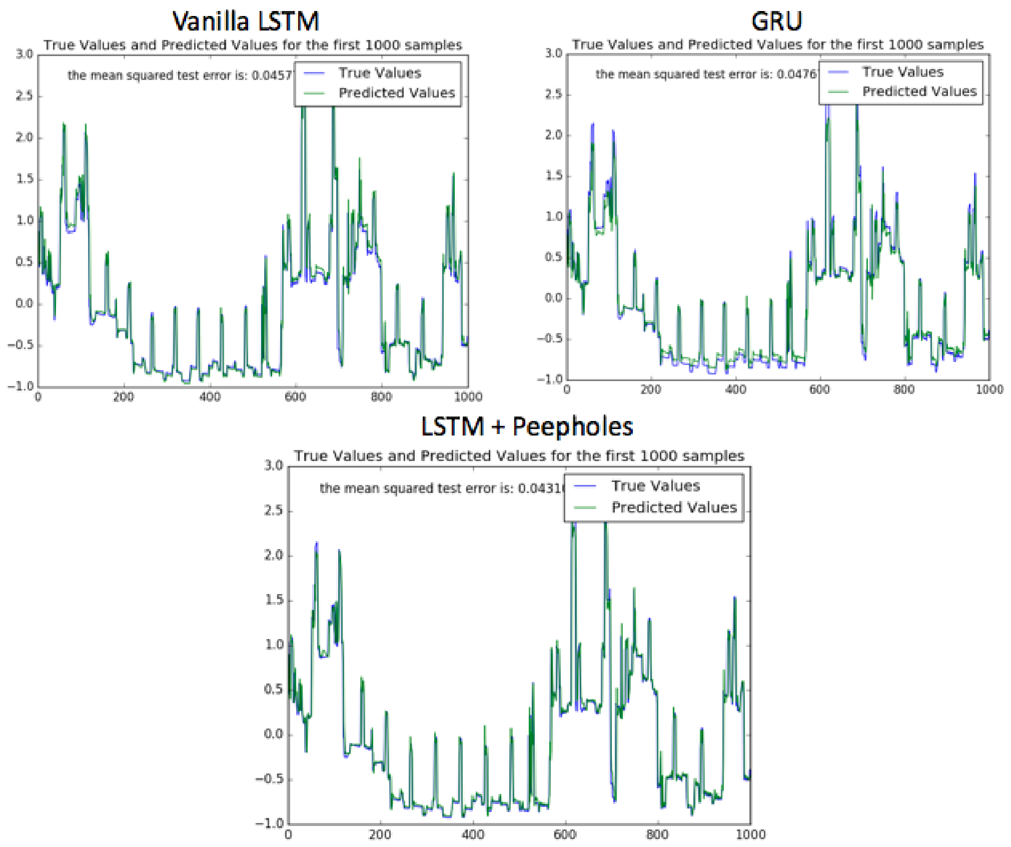

Figure 9: Example error graphs for all three models, trained on all available data with 5% held out for a test set. It is clear that around timestep 600, the peephole connections do provide some benefit over the other two models.

The above results demonstrate that the single step prediction task is extremely easy for neural networks. However, as we approach these levels of accuracy, we do begin to see distinctions between the LSTMs with and without peephole connections.

6 Conclusions

Overall, we have examined the performance of peephole connections and neural network architectures in general for several tasks, drawing mixed results. Although one of the prediction tasks proved to be easily predictable for the neural networks we tested, it is a reality that the entirety of the previous household energy information would be available for prediction, meaning that this does demonstrably map to a real world task. On balance, we suggest that more experimentation with peephole connections is needed, with more available time and computational resource, as the task that we tested them on was not at the limit of the neural network's capacity, meaning that distinguishing performance was difficult. We have demonstrated that there is the potential for the peephole connections to have a serious impact in the second experiment with the UCI dataset.

In the more difficult prediction task of energy load forecasting, we demonstrate how difficult complex neural networks can be to train - complex network architectures still provide serious problems to overcome in this area.

References

[1] Junyoung Chung and Çaglar Gülçehre and KyungHyun Cho and Yoshua Bengio (2014) *Gated Recurrent Neural Networks*

[2] Gers, F. and Schmidhuber, J. (2000) *Recurrent Nets that time and Count*

[3] Sutskever, I., Vinyals, O., Le, Q.V. (2015)
Sequence to Sequence Learning with Neural Networks

[4] Graves, A. (2013) *Generating Sequences With Recurrent Neural Networks*

[5] Oriol Vinyals and Lukasz Kaiser and Terry Koo and Slav Petrov and Ilya Sutskever and Geoffrey E. Hinton, (2015) *Grammar as a Foreign Language*

[6] Sepp Hochreiter and Jrgen Schmidhuber(1995) *Long Short-Term Memory*

[7] Tao Hong, Pierre Pinson and Shu Fan, (2014), *Global Energy Forecasting Competition 2012.* International Journal of Forecasting, 30(2), 357;
Charlton, N. and Singleton, C. (2014). *A refined parametric model for short term load forecasting.* International Journal of Forecasting, 30(2), 364-368.
Souhaib Ben Taieba and Rob J. Hyndmanb, (2014), *A gradient boosting approach to the Kaggle load forecasting competition.* International Journal of Forecasting, 30(2), 382;
Lloyd, J. R. (2014), *GEFCom2012 hierarchical load forecasting: Gradient boosting machines and Gaussian processes.* International Journal of Forecasting, 30(2), 369.

[8] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley and Y. Bengio. *Theano: new features and speed improvements*, NIPS 2012 deep learning workshop