Temporal Recurrent Networks in Review

Sheida Etemadidavan
Department of Computer Science
Old Dominion University
5115 Hampton Blvd, Norfolk, VA 23529
setem001@odu.edu

Joseph S. Cannella
Department of Computer Science
Old Dominion University
5115 Hampton Blvd, Norfolk, VA 23529
jcann009@odu.edu

Abstract

Recent developments in computer vision and video processing techniques have allowed machines to now detect complex and abstract information i.e., actions form raw video frames. There are several methods capable of labeling and times-stamping actions on fully recorded videos, a process known as Action Detection. The accuracy of existing techniques is diminished when attempting to perform this detection in real-time, referred to as Online Action Detection. This paper details the primary principles of the Temporal Recurrent Network proposed by the authors [1] and to highlight our own attempts at implementing our own TRN and reproducing the authors results. In this paper we explore and asses the Temporal Recurrent Network when applied to stock market data.

1 Overview of problem

There are numerous problems that require rapid response to rapidly changing systems. Any highly time sensitive process must be able to quickly and efficiently react to its inputs before it is too late, or the information becomes stale. The ability to quickly detect an action in real time is referred to as Online Action Detection. In the context of computer vision and online action detection requires the ability to immediately process each image frame as soon as it is received.

In the literature most of action detection is performed offline. Offline means that start and end times of actions are determined after the entire action (video) are fully observed. But due to nowadays needs, require us to react in real-time. This means to detect actions online. Online action detection refers to recognize actions after observing a fraction of request. To address this problem the authors of the paper introduce a new model to estimate and use future information to improve performance of current online action. The proposed model called Temporal Recurrent Network (TRN) which will be discussed briefly in the following section. In the context of live video, the original context of the paper, for instance for one or more videos, the goal is to estimate a probability distribution over k possible actions for each frame of an image sequence.

Mathematically, this mean approximating the probability P_t of a currently occurring action's membership to one of K possible action classes by considering the current image frame. It as well as information derived from all previous frames $I_{<t}$. Note that "no action" is considered one of the valid action classes where in which no action is deemed to be taking place at that time.

2 Proposed method: Temporal Recurrent Network

In a TRN, future information is predicted as an anticipation task and used together with historical evidence to recognize action in the current frame. In everyday life we are constantly assessing our environment and events occurring in it in real-time. Our assessments of events unfolding around us is based on a continuous flow stimulus received from our surroundings. Our inferences are based

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

not only on instantaneous stimuli, but also from recent preceding stimuli. This notion of utilizing past information to draw conclusion about the present makes a lot of sense when temporal locality is considered, and this forms the foundation and theory behind Recurrent Neural Networks.

In Recurrent Neural Networks information from each preceding input is fed into the next analysis to augment and improve the estimation regarding the present input. What one may find surprising is that there is a great deal of evidence that biological organisms not only consider the past and present when drawing inferences about the present, but that they also consider predictions of the future. It is this anticipatory behavior that the authors of the authors seek to exploit via an algorithmic structure they refer to as a Temporal Recurrent Network (TRN).

Fundamentally, a Temporal Recurrent Network operates very similarly to a standard Recurrent Neural Network and is even comprised of many of the same components. In fact, without examining a the TRN cell further, the flow diagram appears identical to that of a RNN where the hidden layer of the previous cell is fed into the next cell along with the current input data and each cell outputs a prediction. TRN anticipates the future via a temporal decoder. TRN has a core called TRN cell. TRN cell controls flow of internal information by

The TRN cell is the core of the Temporal Recurrent Network and itself is a recurrent unit. The TRN cell receives both a feature vector x_t corresponding to an observation a time t as well as the hidden state h_{t-1} from the previous TRN cell. However, a TRN cell diverges from functionality of a standard RNN cell because in addition to taking into account the prior temporal dependencies, a TRN cell also seeks to utilize a relations due temporal locality of the present and near future states.

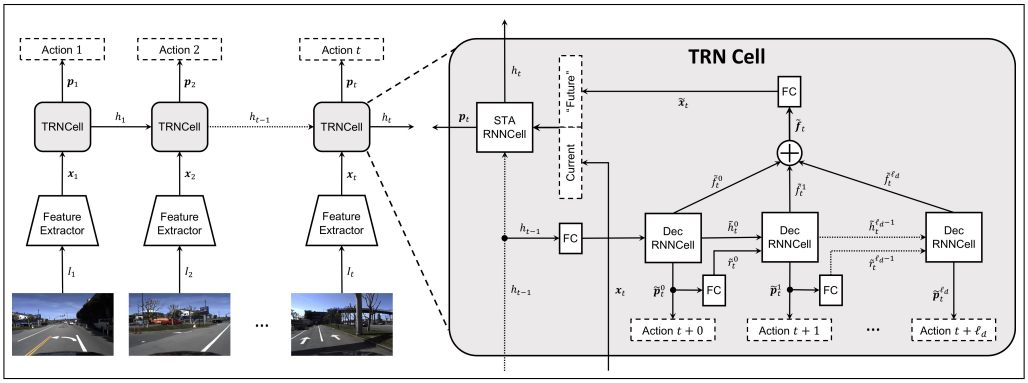

Figure 1: TRN Structure [1]

Note that because this problem requires real-time action prediction, even the proximal future states are not known to the network at any instant and thus must be approximated. This future prediction is employed by extending the current estimate forward by chaining a few RNN cells in series, feeding the previous hidden state $ht - 1$ as the input and conglomerating each of the RNN's outputs into a predicted future state.

The predicted future state is then concatenated with the current input feature vector and conglomerate becomes the input to the a primary RNN cell along with the previous hidden state. It is this primary RNN that actually classifies the current action.

3 Implementation

3.1 Frameworks and libraries

The algorithm for the Financial TRN was written entirely in Python utilizing numerous machine learning and data science libraries. These include SciKitLearn Numpy, Pandas, and Pytorch. SciKitLearn is a very popular machine learning library that provides several high-level features useful in the realm of data science, for our purposes we primarily used it for preprocessing the data. Pandas is another library very common in the data science field and provides highly efficient in memory

operation much like to Numpy, but with added functionality and column labels via it's DataFrame object. Again this was primarily used to select and prepare data for out primary machine learning algorithm.

Pytorch is a machine learning framework that is rapidly gaining traction in research fields. Unlike SciKitLearn, Pytorch is not a library of predefined machine learning algorithms, but instead allows researchers to construct their own machine learning algorithms by linking together numerous low level ML components. These components include things like Fully Connected Layers, Convolutional Layers, RNN cells, and CrossEntropyLoss.

The documentation for the Pytorch framework can be found at

`https://pytorch.org/`

3.2 Network structure

Like the authors' work our Temporal Recurrent Network consisted primarily of Long Short Tem Memory (LSTM) cells. These LSTM cells act as both the SpatioTemporal Accumulator (STA) as well as the individual Decoder cells. Like a typical LSTM cell each instance STA receives the hidden state output by the previous STA. However, instead of simply receiving data at the current timestep as an input, the LSTM receives the concatenation of both the present data vector with the predicted future data vector as it's input. This "future" state was itself derived from the pooling average of each decoder cells output, where decoders are linked together in series.

Unlike the authors implementation, our data TRN does not rely on a external feature extractor prior to entering the TRN, and instead simply the min-max scaled data is fed directly into the algorithm. Additionally, while the previous work operates on 2D images, we adapted it to operate on a single scalar input at each time step, the stock price. We chose not to apply dropout between fully connected layers in order ensure the different models we tested only differed the parameters we were tracking. While the authors chose to create a complex command line interface for using their TRN, we instead thinned down the supporting code and extract just the fundamental operational aspects. Outside of the TRN class we created a few additional modules to assist in data parsing, model training, and evaluation.

3.3 Support structures

The Pipeline class was created to provide a simple means for selecting an input file, trimming data, selecting the column of interest, and specifying the number of decoders that the TRN will utilize. Note that a single input file is passed to the pipeline. No additional labeled target data is required, because our task at each time-step is to predict the following time-steps value, which simply means the target "truth" data can be derived by simply shifting the input data to the left.

The number of decoders are specified here because the TRN not only updates the STA based on its output error, but also each of the Decoder cell's individual errors. Like the target data for the STA, the decoder data is also derived from the input data but each additional decoder step would correspond to an additional shift of the data. For example, the expanded form of an time series [1,2,3,4,5,6,7] for three decoders would be [1,2,3,2,3,4,3,4,5,4,5,6,5,6,7]. Which turns out to be of length $L = (N - (D - 1)) * D$ where N is the number of original data points and D is the number of decoder steps.

The TrainModel method in RunAll.py allows a model to be run on a given data set with a variable number of epochs. This method utilizes the ADAM optimizer for back propagation and uses Mean Squared Error to compute loss. We chose to use MSE instead of the Cross Entropy Loss utilized in the paper because our target in not a class but a value so Cross Entropy would be inappropriate for this use.

Finally, the VizHelp class was created to assist with the visualization of the results and provides two generic methods allowing the Training Error vs Epoch to be easily plotted as well as the predicted values.

3.4 Hardware and training details

All training and testing instances were run with 50 Epochs and experimentally the number of decoders were varied to assess the impact. The models and date were ported over to a NVIDIA GeForce GTX 1660 GPU at clock speed of 2.67 GHz.

4 Dataset

The data set utilized in this paper is the "Huge Stock Market Dataset" which is publicly available. The full data set consists of 1344 files containing stock market data from a wide range of companies. Each file consists of Date, Open, Low, Close, Volume, and OpenInt columns for each trading day, but for our research we only extracted the Close price. The stocks we chose to target are Google, Amazon, Apple, and IBM.

Any missing values were populated with the temporally nearest value. Furthermore, we chose to focus on only four of the available stocks. The reasoning for not including the entire data set in training was because we want to predict the future price of a specific stock, we decided to cater our model training and testing instances to specific stocks. This was done under the hypothesis that one company may have different price characteristics than another. Additionally, we trimmed the data to a more recent interval by setting the date range between January 2nd 2010 and October 11th 2017. The input data is for all four stock over this interval is shown below.

Figure 2: Raw Data: Google

Figure 3: Raw Data: Amazon

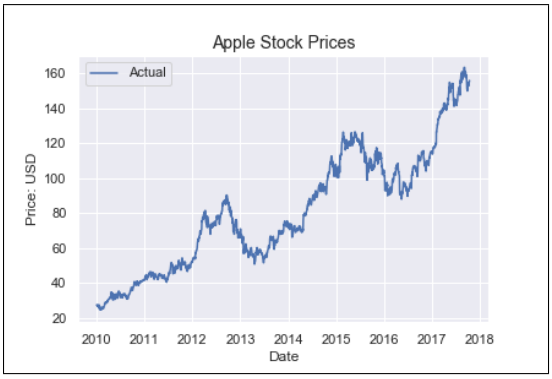

Figure 4: Raw Data: Apple

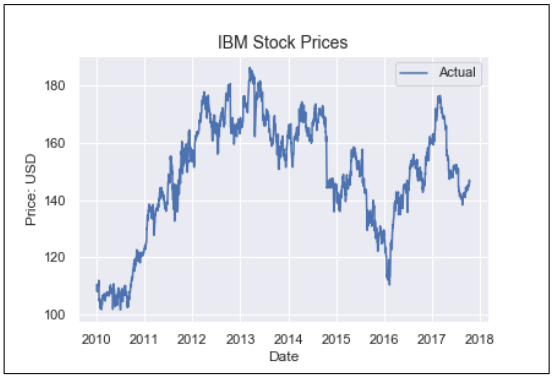

Figure 5: Raw Data: IBM

The Huge Stock Market Dataset may be found at

https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs

5 Results

The results of or trial runs were mixed. While the training errors for each model did decrease dramatically by the final epoch, the testing results varied greatly between each data set as well as with the number of decoders used. It's interesting to note that in many cases the variation in test error for a given data set seemed to vary non-linearly with the number of decoders. Additionally, it appears as though the TRN performed much worse on data sets that were generally stably increasing.

Note that the IBM stock seems to follow a more periodic trend and achieves the best results. While Amazon, which appears to be generally monotonically increasing performed much worse. These results may be due to an over fit of the data to the training set. Upon closer inspection of the IBM data the training set appears to nearly mirror the test set over this interval, while the other stocks transition from a flatter region into a steeper slope just around the training cutoff.

Additionally, it was observed that there was a general increase in the run-time when a greater number of decoder steps were used in the model.

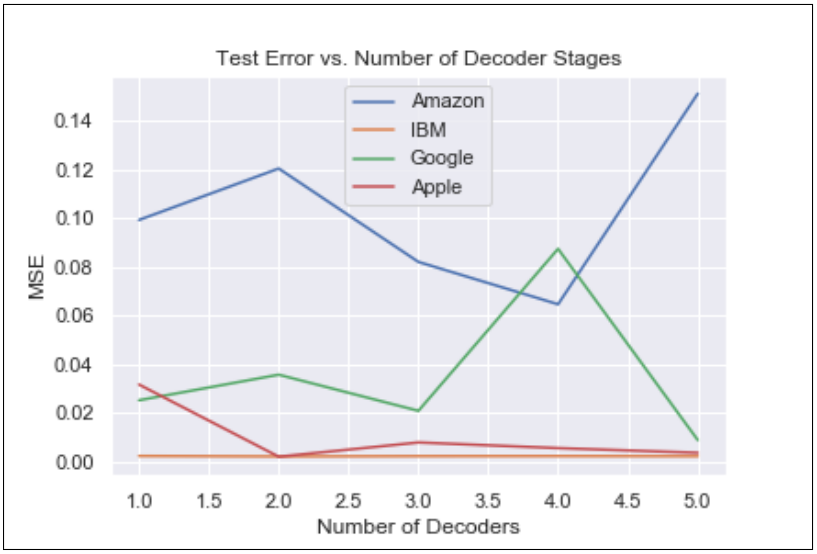

Figure 6: Test Errors vs Decoder Steps

6 Conclusions

While RNNs (in particular LSTMs) are well suited for time series data this does not mean they will successfully predict stock data when trained to target a specific stock. The TRN may yield some benefits in prediction, but we were unable to determine a direct relationship between the number of decoder steps and test error. However, it can generally be observed in figure 6 that three decoder steps appear to perform the best when all four test runs are taken into account.

It should also be noted that any performance improvements detected may simply be due to an added hyper parameter (number of decoder steps) which provides an additional dimension for tuning the models. Additionally, there may have been other hyper-parameters in our model that if tuned properly might significantly improve or degrade the results. For instance the batch size or "window" the STA receives. In these tests the windows was set to a single scalar, and this may have led to a degradation of the model.

Furthermore, stock prediction is a notoriously difficult task and there is some debate as of whether the actual time variant trends exist when observing the price alone. If this is not the case then

external variables must be taken into account to detect anomalies such as market jumps and crashes. From what we have observed, the hypothesis that individual stocks may exhibit very characteristics and thus may produce very different models in training appears to be maintained. However, to improve the overall accuracy of the model may require a broader frame of data during training to expose it to previously unseen scenarios and provide a more general interpretation of the data.

Table 1: Test results after 50 epochs: Amazon

Decoders	MSE	Runtime (s)
1	0.099	145.092
2	0.120	203.041
3	0.082	309.170
4	0.065	408.817
5	0.151	480.722

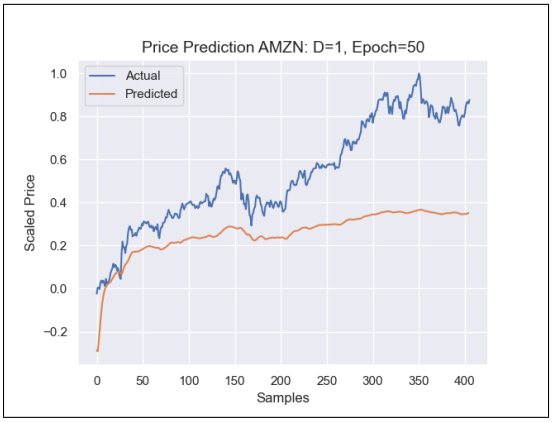

Figure 7: Amazon Training Error

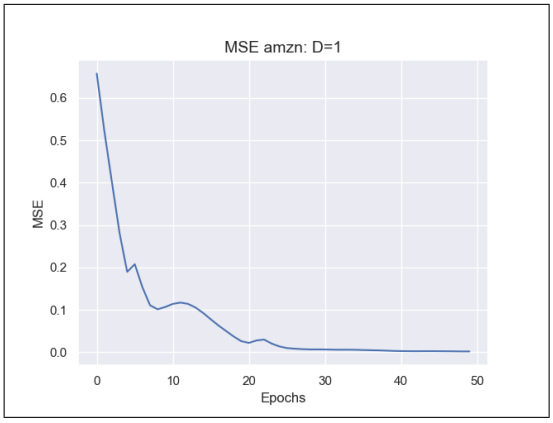

Figure 8: Amazon Test Prediction

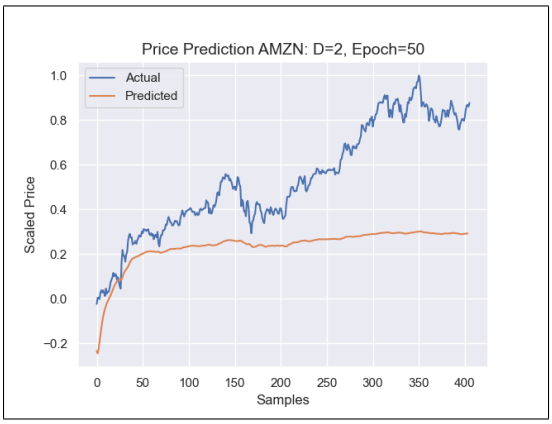

Figure 9: Amazon Training Error

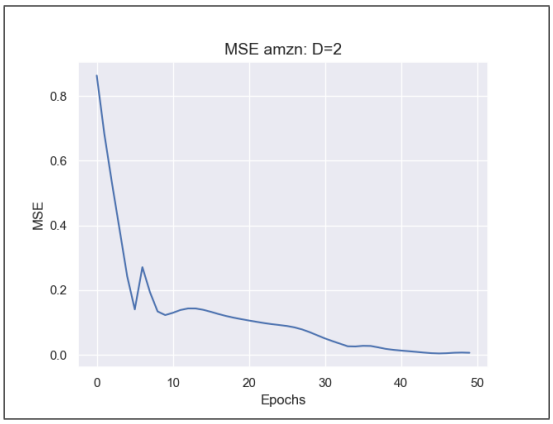

Figure 10: Amazon Test Prediction

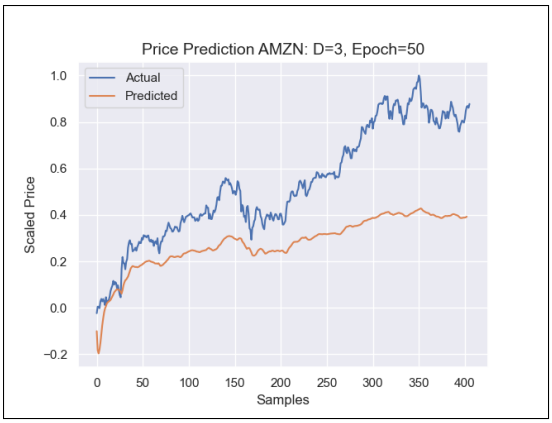

Figure 11: Amazon Training Error

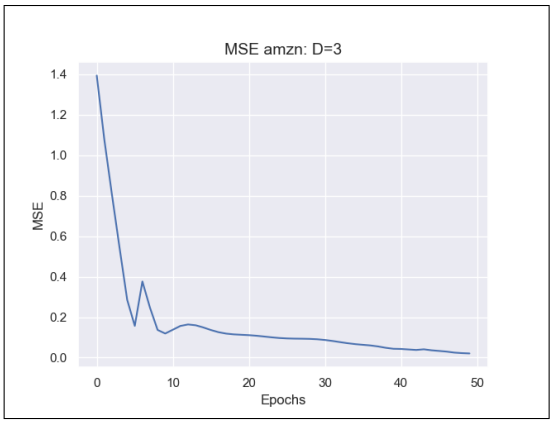

Figure 12: Amazon Test Prediction

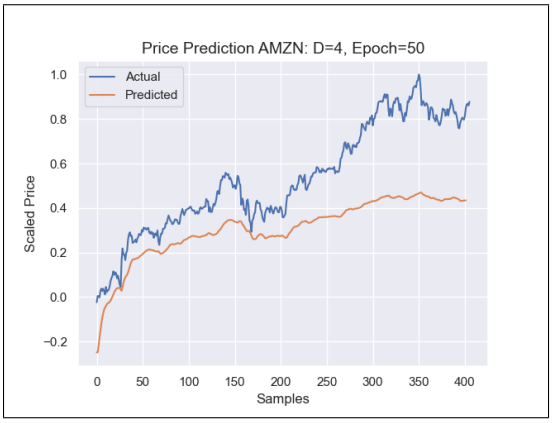

Figure 13: Amazon Training Error

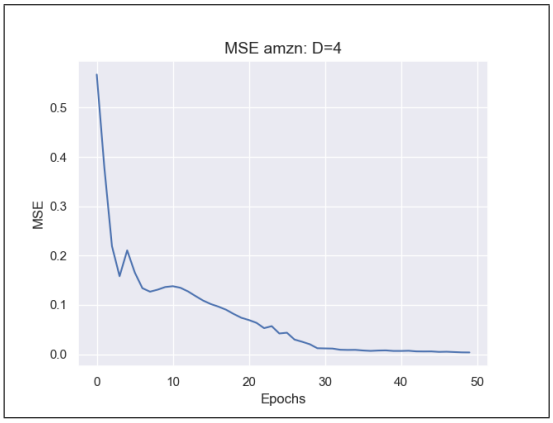

Figure 14: Amazon Test Prediction

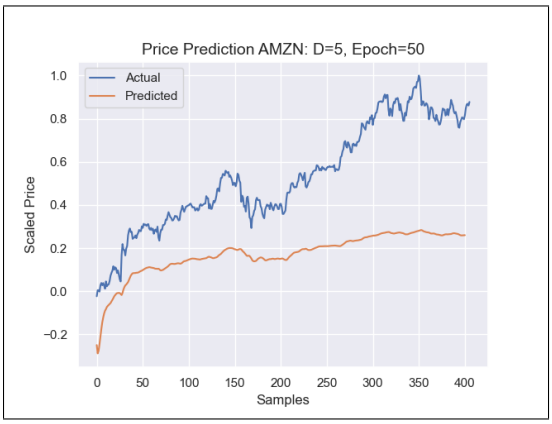

Figure 15: Amazon Training Error

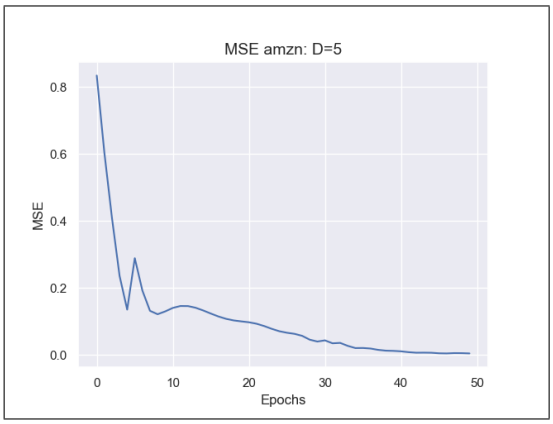

Figure 16: Amazon Test Prediction

Table 2: Test results after 50 epochs: IBM

Decoders	MSE	Runtime (s)
1	0.002	272.544
2	0.002	225.033
3	0.002	314.949
4	0.002	414.227
5	0.002	484.644

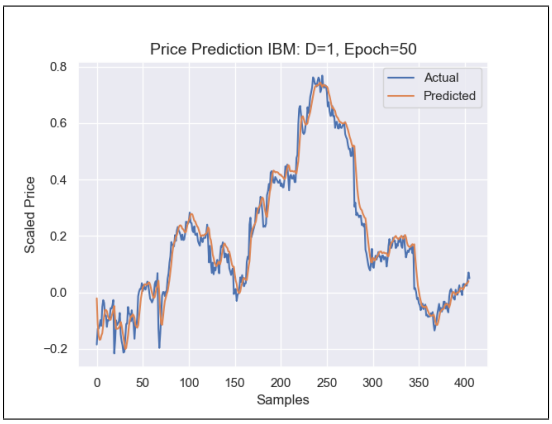

Figure 17: IBM Training Error

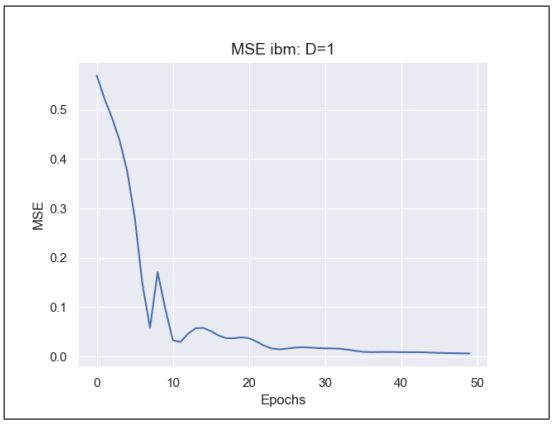

Figure 18: IBM Test Prediction

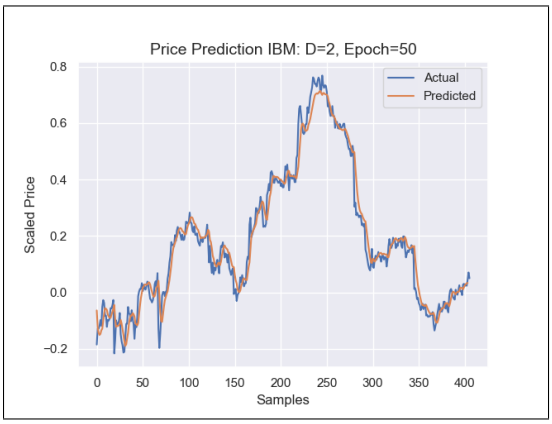

Figure 19: IBM Training Error

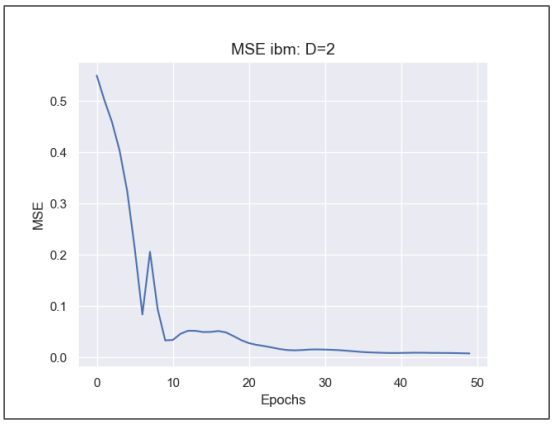

Figure 20: IBM Test Prediction

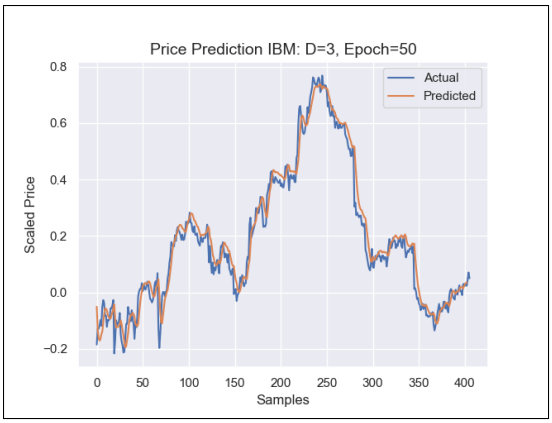

Figure 21: IBM Training Error

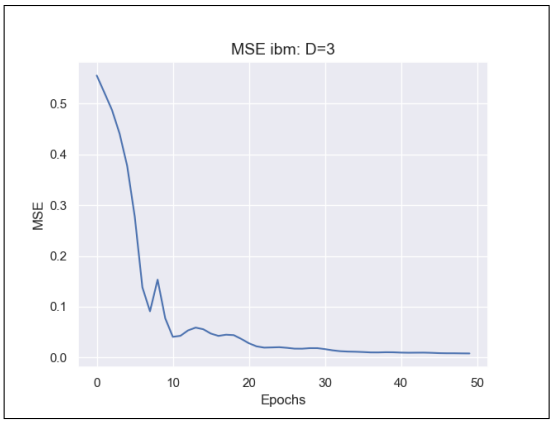

Figure 22: IBM Test Prediction

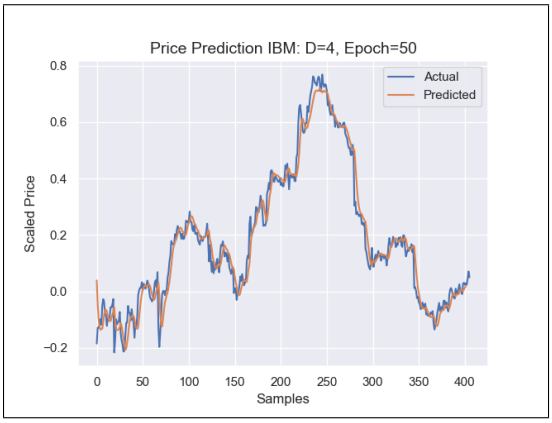

Figure 23: IBM Training Error

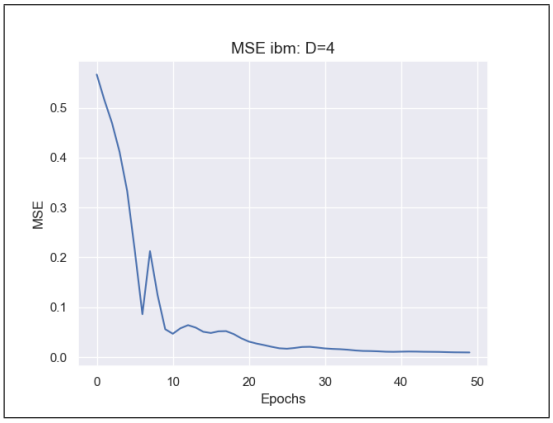

Figure 24: IBM Test Prediction

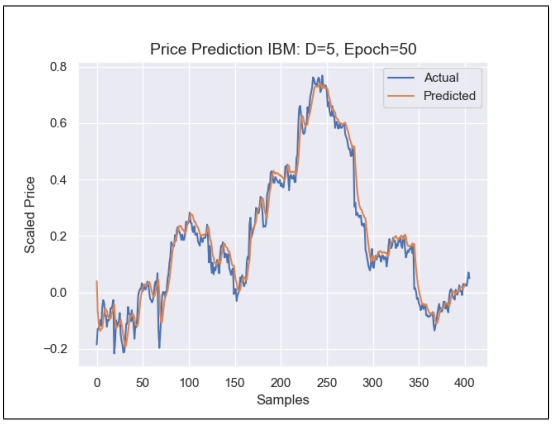

Figure 25: IBM Training Error

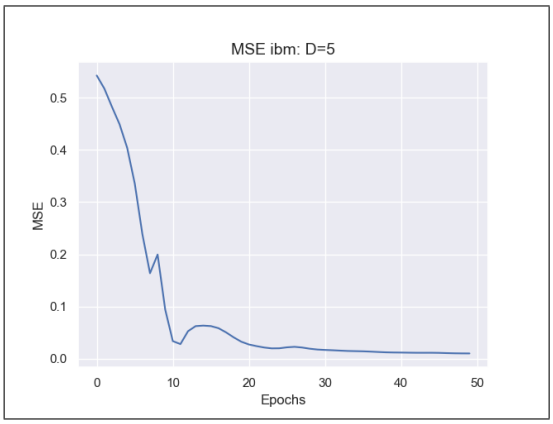

Figure 26: IBM Test Prediction

Table 3: Test results after 50 epochs: Apple

Decoders	MSE	Runtime (s)
1	0.032	374.953
2	0.002	663.007
3	0.008	325.118
4	0.005	400.273
5	0.004	479.471

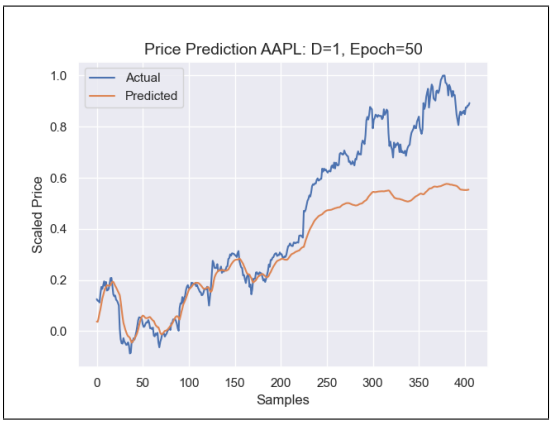

Figure 27: Apple Training Error

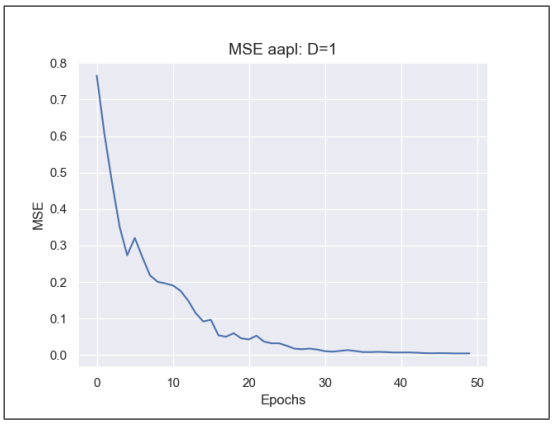

Figure 28: Apple Test Prediction

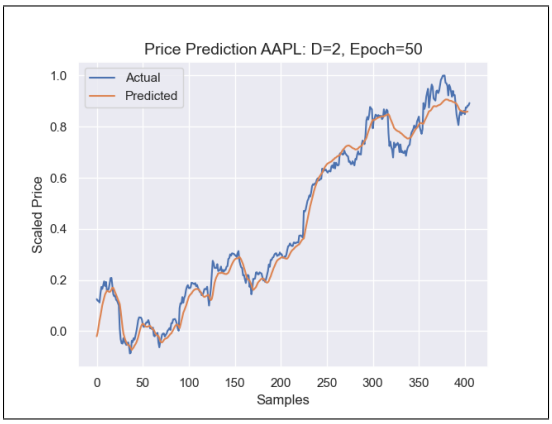

Figure 29: Apple Training Error

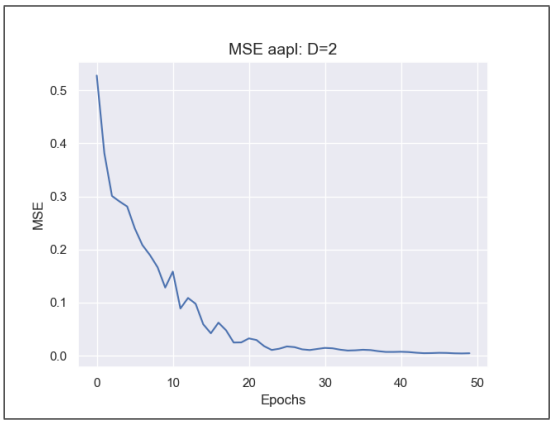

Figure 30: Apple Test Prediction

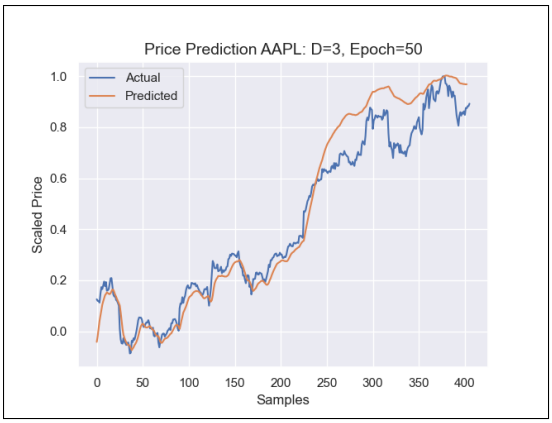

Figure 31: Apple Training Error

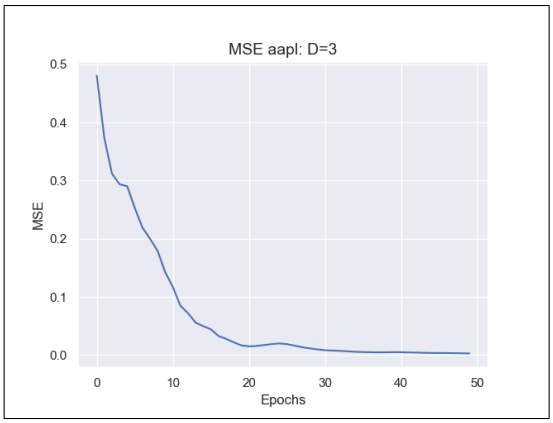

Figure 32: Apple Test Prediction

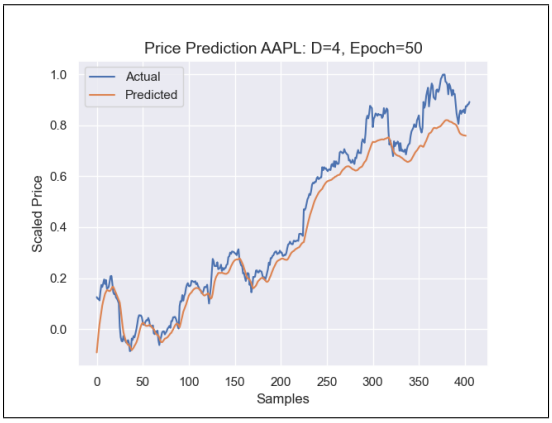

Figure 33: Apple Training Error

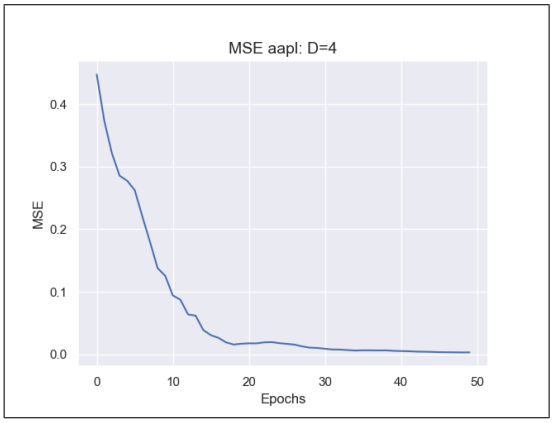

Figure 34: Apple Test Prediction

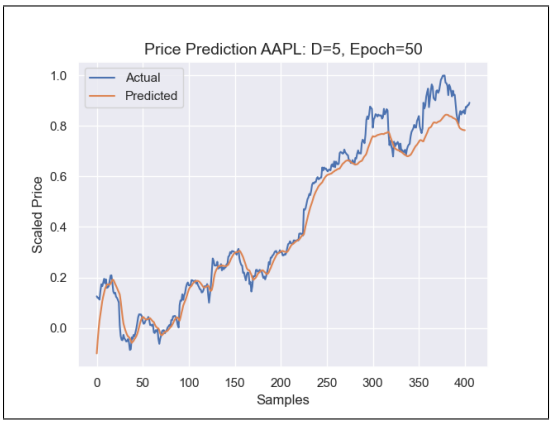

Figure 35: Apple Training Error

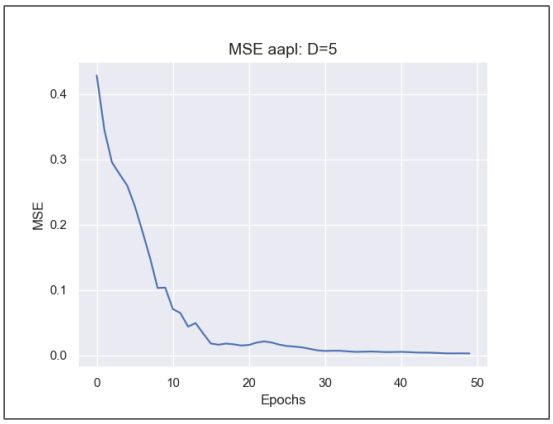

Figure 36: Apple Test Prediction

Table 4: Test results after 50 epochs: Google

Decoders	MSE	Runtime (s)
1	0.025	264.949
2	0.036	276.605
3	0.021	345.568
4	0.087	408.621
5	0.009	452.681

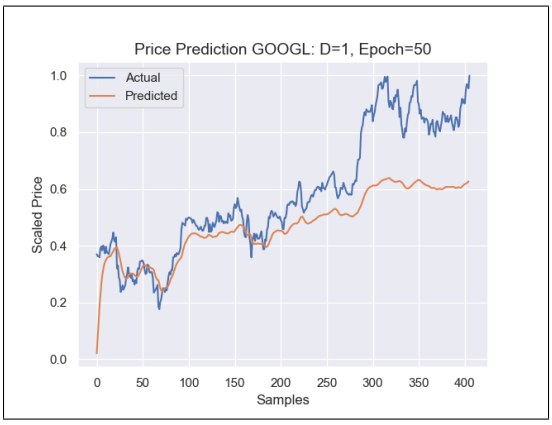

Figure 37: Google Training Error

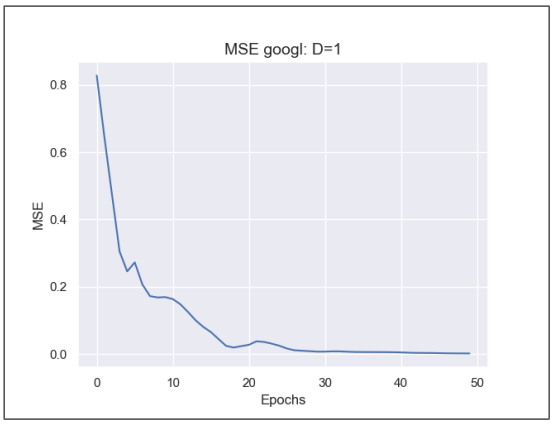

Figure 38: Google Test Prediction

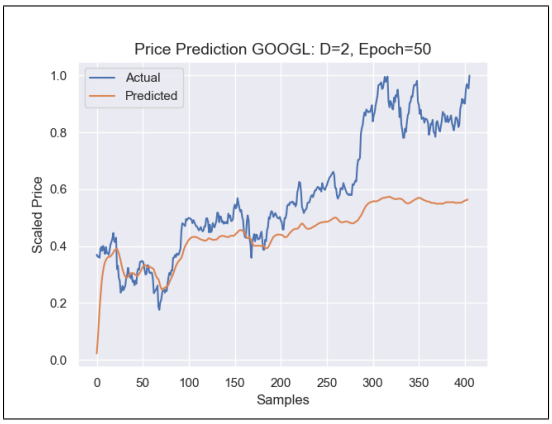

Figure 39: Google Training Error

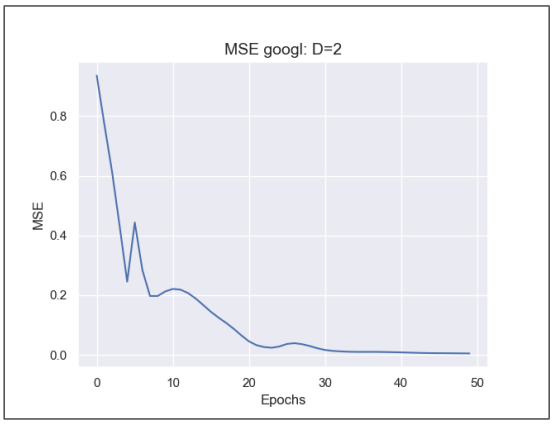

Figure 40: Google Test Prediction

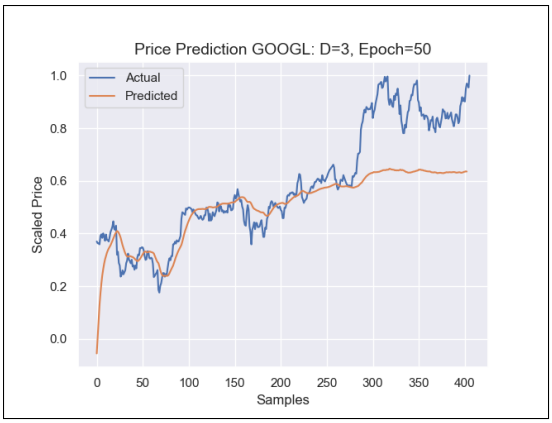

Figure 41: Google Training Error

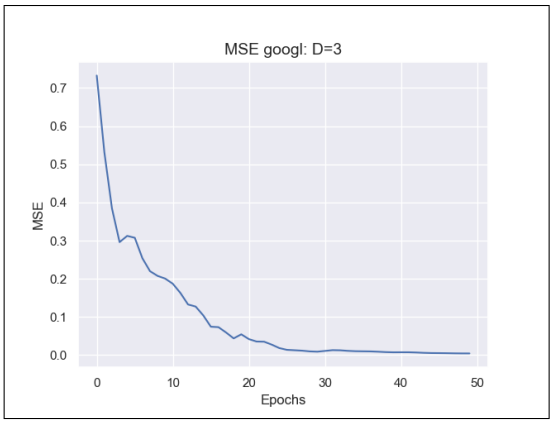

Figure 42: Google Test Prediction

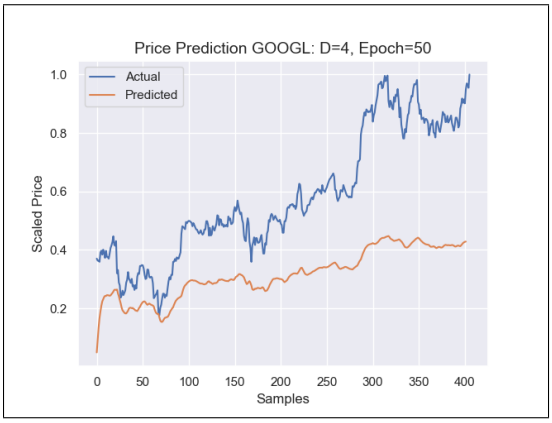

Figure 43: Google Training Error

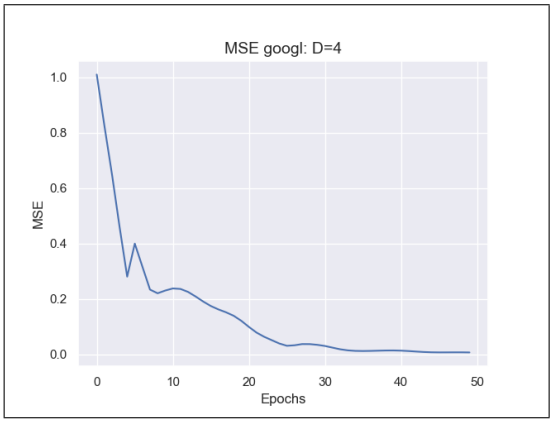

Figure 44: Google Test Prediction

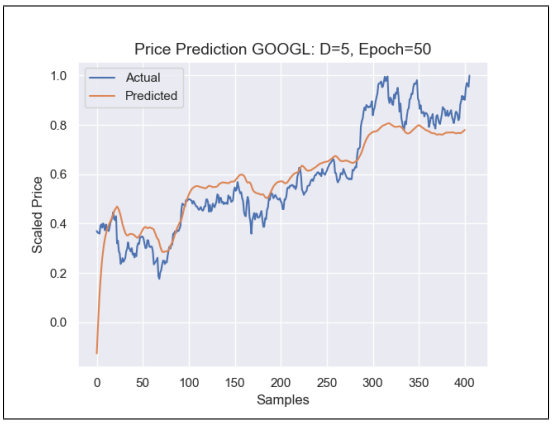

Figure 45: Google Training Error

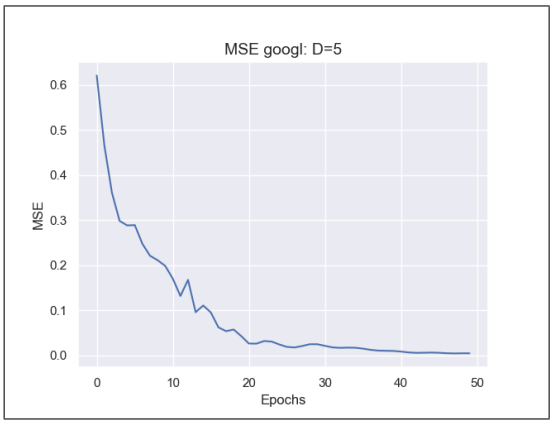

Figure 46: Google Test Prediction

References

[1] Mingze Xu, Mingfei Gao, Yi-Ting Chen, Larry S. Davis, David J. Crandall (2019) Trmporal Recurrent Networks for Online Detection, *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*,