Deep Learning – Assignment 2

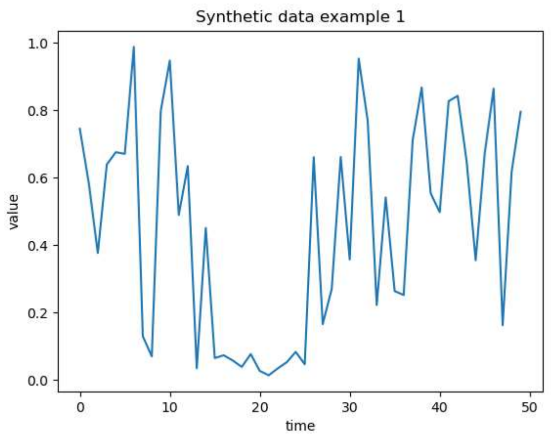
Synthetic Data

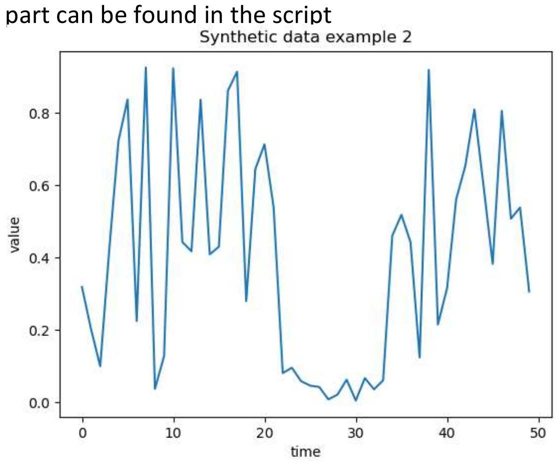
1) First, we generated synthetic data according to the question requirements:

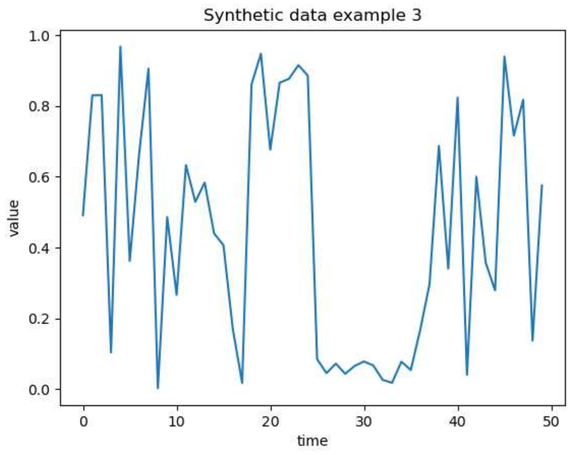
10,000 input sequences (signals), each contains uniformly distributed random 50 points, with values ranging from 0 to 1. Also, for each input sequence $\{x_t\}$ we randomly selected an index i_t between 20 to 30, and multiplied all the points in indices $i_t - 5$ to $i_t + 5$ by 0.1. That is:

$$\forall\{x_t\}, i_t \ random \ index\colon \ x_j = 0.1 \cdot x_j \ \ , \ \ j \in [i_t - 5, i_t + 5]$$

We then split the generated data (10,000 sequences) to train, validation and test sets according to the most common division, which is 60% of the data is the train set (6,000 sequences), 20% of the data is the validation set (2,000 sequences) and another 20% of the data is the test set (2,000 sequences). To demonstrate the generated data signals, we show plots of 3 generated signals (code for this part can be found in the script







2) **LSTM AE**

We've implemented an LSTM AE (code can be found in the script `encoder_decoder.py`) according to the required architecture. To construct the model EncoderDecoder, the following parameters should be given in the constructor:

input_size: the dimension of each data point in the series $\{x_t\}$

hidden_size: the desired hidden state size for the encoding and decoding process

output_size: the dimension of each data point in the reconstructed series $\{\tilde{x}_t\}$

The AE is composed of:

Encoder – an LSTM network which gets the input (of size input_size) and the hidden layer size, and encodes the series $\{x_t\}$ into a context vector z which is practically the last hidden state output of the LSTM network.

Decoder – an LSTM network which gets the series $\{z_t\}$ which is the output vector of the encoder z repeated T times (where T is the size of the original sequence), and outputs a sequence $\{\tilde{h}_t\}$ of the same size as the input sequence and is composed of the hidden states calculated during the forward pass of the decoder.

Re-construction layer – finally, in order to reconstruct the original sequence $\{x_t\}$, we used a reconstruction layer which takes each sequence point $\tilde{h}_t$ and applies a linear and a non-linear (we chose tanh) function on it to reconstruct the specific sequence point $\tilde{x}_t$. That is:

$$\tilde{x}_t = \tanh\left(U\tilde{h}_t\right)$$

We calculated the AE's loss using an MSE loss on the original sequence vs. the reconstructed sequence.

Then, we performed grid search for the following hyperparameters and values:

$$Batch\ size \in \{32, 64\}, Learning\ rate \in \{0.001, 0.01\},$$

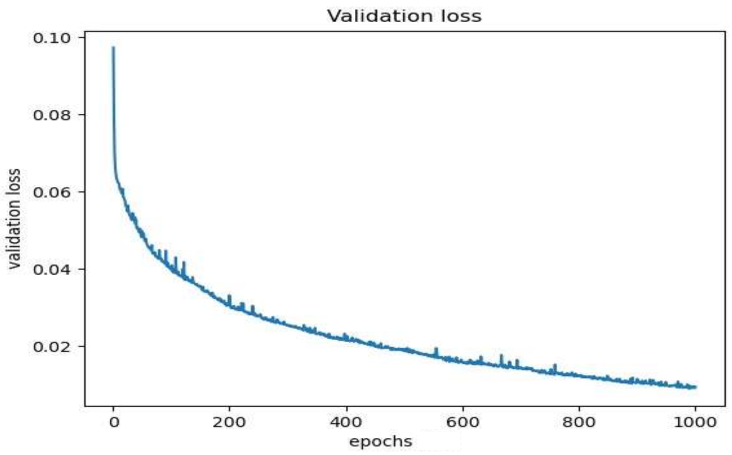
$$Gradient\ clip \in \{0, 1\}, Hidden\ state\ size \in \{10, 25, 40\}$$

This led to training 24 different models, and we chose the best hyperparameters according to the model which reached the best loss on the validation set. Note that we chose to run the grid search on hidden sizes smaller than the original sequence length (50) on purpose, as we didn't want our model to learn the identity function. Also, we used ADAM optimizer for the training, and trained for 1000 epochs.

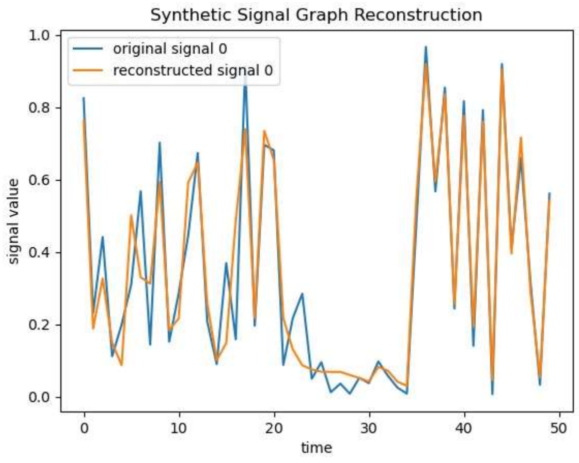
The model which performed best (reached validation loss of about 0.01) was the model with the following hyperparameters:

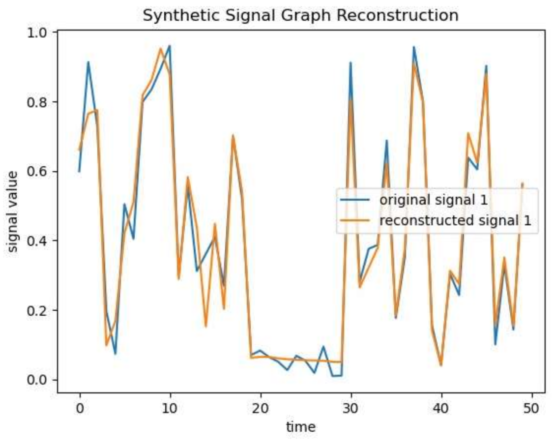
$$Batch\ size = 32, Learning\ rate = 0.001, Gradient\ clip = 1, Hidden\ state\ size = 40$$

Plot of the validation MSE loss graph per epoch during the training process:



The following plots demonstrate how our trained model reconstructed signals from the test data set:





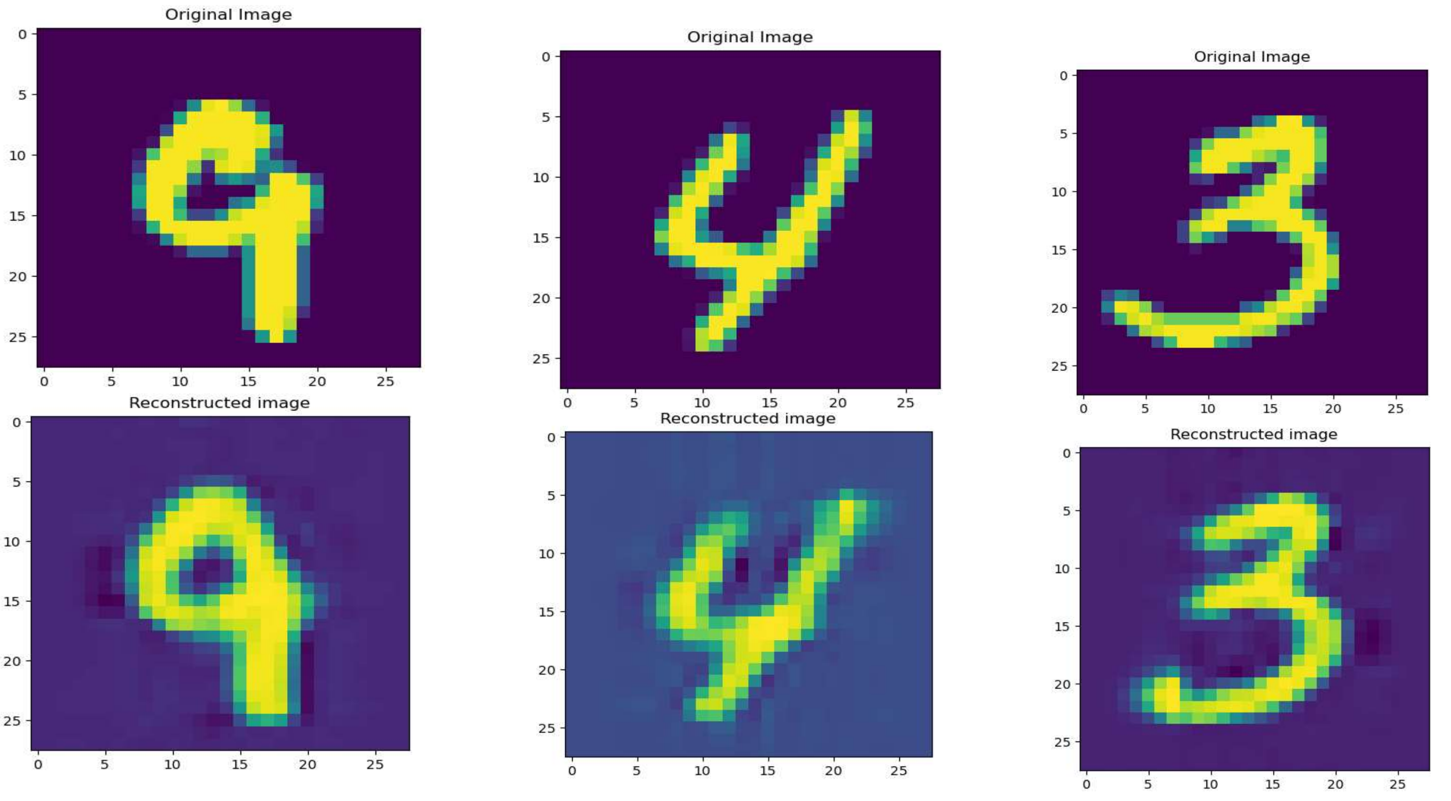
MNIST

1) We've trained the LSTM AE presented in the last section to now be able to reconstruct MNIST data. For each MNIST image I, we defined the sequence $\{I_t\}$ where $t \in [1, 28]$ and $I_t \in \mathbb{R}^{28}$. In simple words – each image is a sequence with 28 entries, where each entry as a pixel row of the image (28 rows in total for MNIST image), and each entry's (image row) dimension is 28 (as it holds values for 28 pixels).

We trained the model with the following parameters (most of them based on the last section, and we kept the hidden state size lower than 28 so the AE won't learn the identity function):

$$Batch\ size = 32, Learning\ rate = 0.001, Gradient\ clip = 1, Hidden\ state\ size = 20$$

The following plots demonstrate the reconstruction capabilities of our trained LSTM AE for MNIST images (the original images are drawn randomly from the test data set):

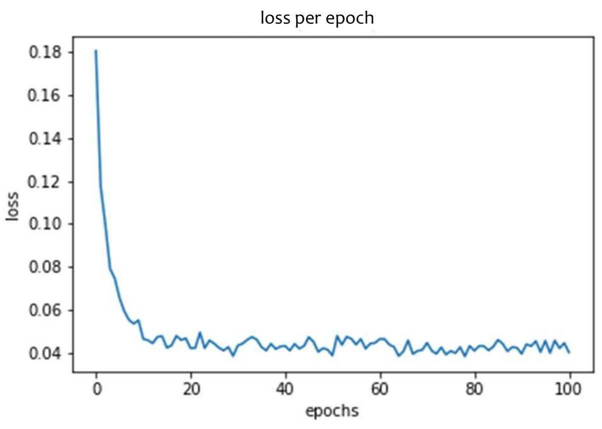


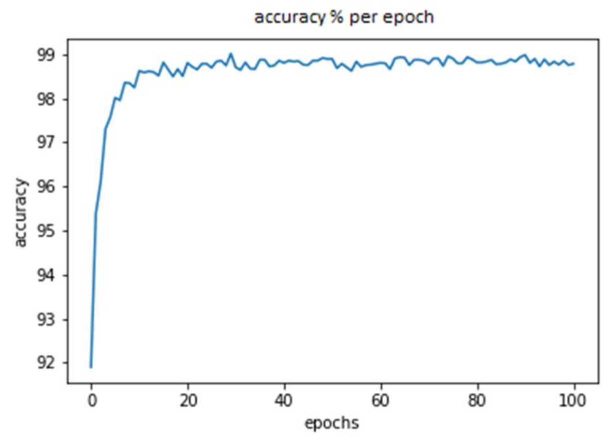
2) In order to classify MNIST images, we extended the LSTM AE presented in the previous section by adding a flag is_classification to the model's constructor. When this flag is set to True, two things change in the way our model works:

- we add to our model a linear classification layer which produces the probabilities vector for each of the 10 possible labels by activating the linear classification layer on the last hidden state received from the encoder
- We use the cross entropy loss function combined with the already existing MSE loss function (for the reconstruction loss) to calculate the total loss by averaging both losses:

$$Loss_{total} = \frac{MSE(original\ image, reconstructed\ image) + Cross\ Entropy(predicted\ labels, actual\ labels)}{2}$$

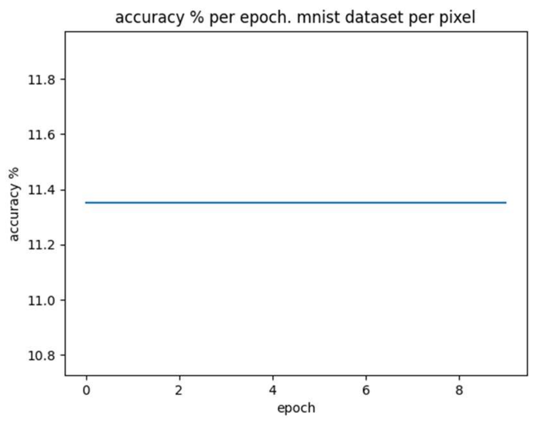
We've trained the new LSTM AE model for 100 epochs. The model reached total loss of approximately 0.04, while producing almost 99% accuracy in prediction. The following plots show the loss per epoch and accuracy percentage per epoch our AE produced for the validation data set:

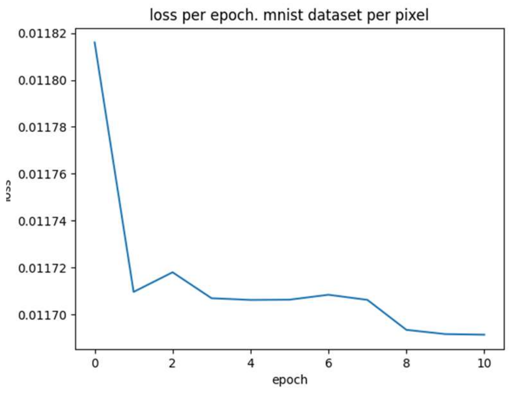




3) At this section we again tried to train a model the will both reconstruct and classify images as before. This time, we used a pixel-by-pixel strategy, meaning we treated the image as a time series of length 784 where x_t is a single pixel. Since the sequence was much longer, we run trained the model with larger hidden state sizes: 100, 150, 300, also trying different learning rates: 0.01, 0.001. The results for all the experiments were similar.

Due to limited computation resources, we run the training for only 10 epochs for each hidden layer size and learning rate (a single epoch in this setting took almost 20 minutes!).

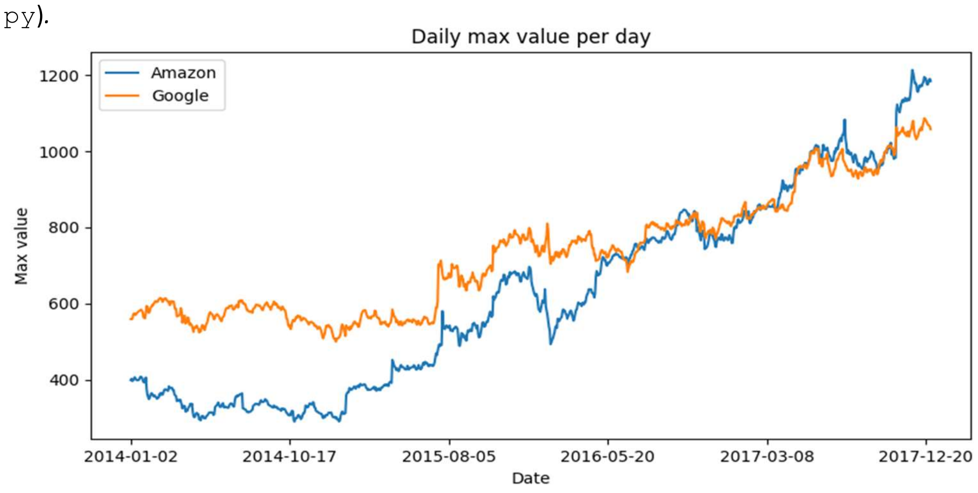




We can see that our model was unable to achieve better accuracy then 11.4% in this setting, also the loss decreased in less then 1% in 10 epochs.

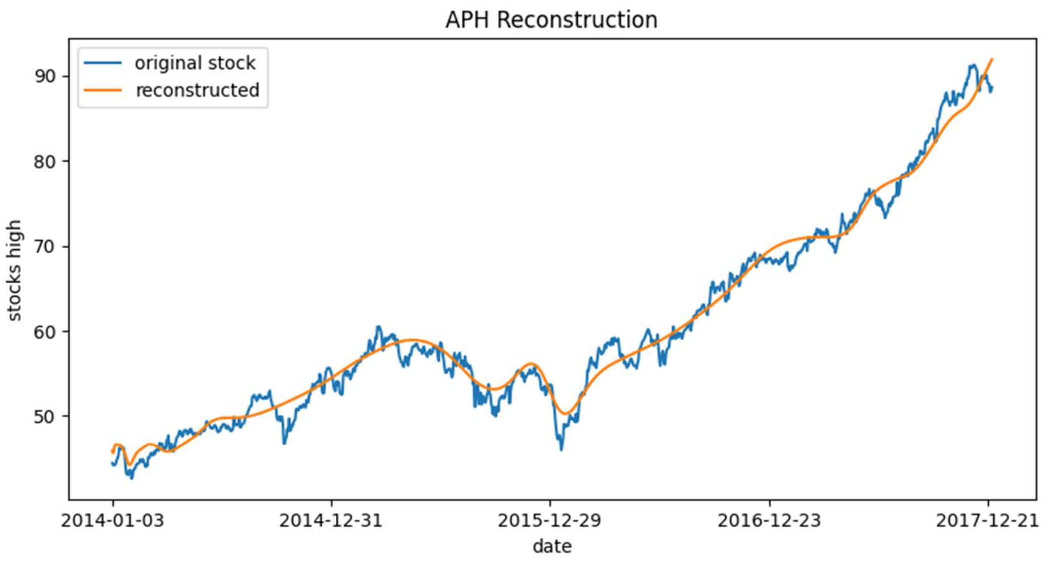
Snp500

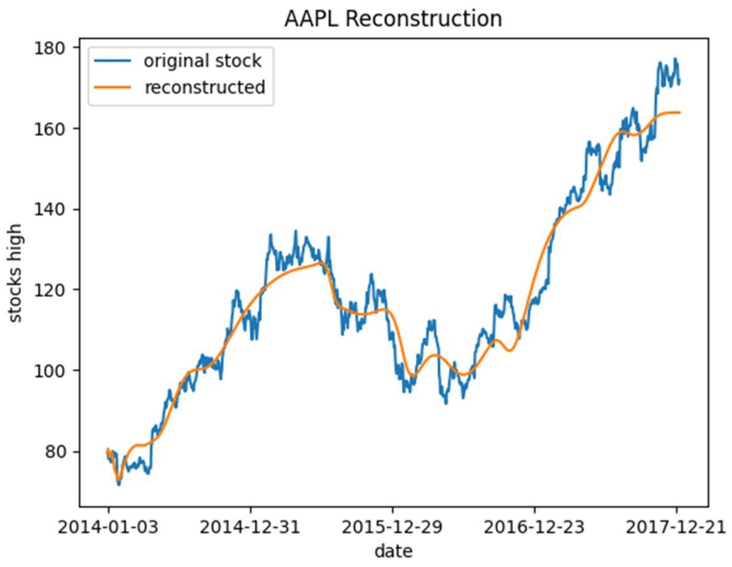
1) Below is the graph of daily max value for stocks *AMZN* and *GOOGL* (code can be found at the script `snp500_dataset.py`).

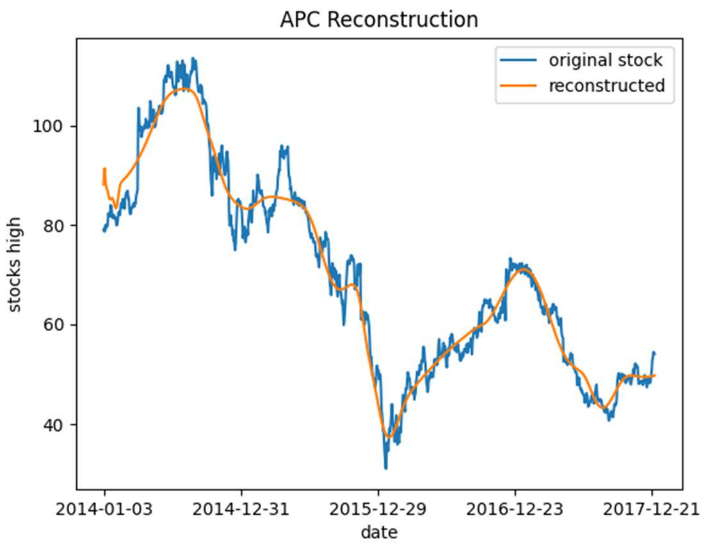


2) In order to train our model to reconstruct stocks signal, we first normalized the data. We defined a 'Normalizer' class, acts as follows: $stock\ sequence\ rescaled\ = \frac{stock\ sequence - min(stock\ sequence)}{max(stock\ sequence)}$, making the sequence values range in $[0,1]$. Then $normalized\ stock\ sequence\ = \frac{stock\ sequence\ rescaled}{2*mean(stock\ sequence\ rescaled)}$ setting the mean as $\frac{1}{2}$. Using

grid search again, we find optimal parameters: $hidden\ size = 120, learnig\ rate = 0.001, gradient\ clip = 1, batch\ size = 32$ for 1000 epochs. Below the plots of three random signals reconstructed:





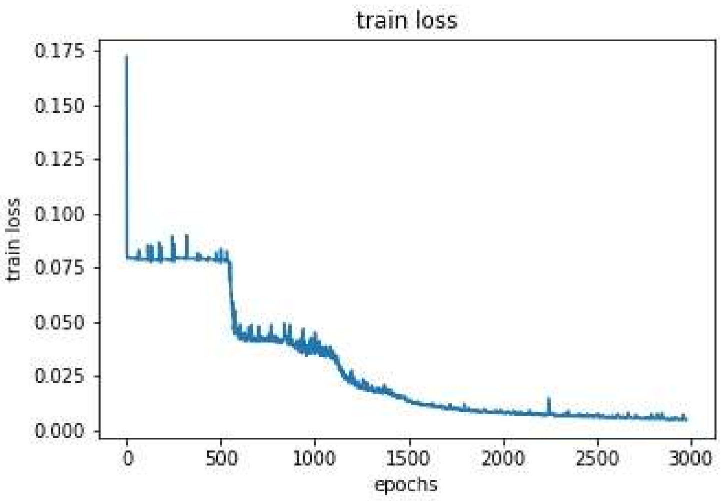


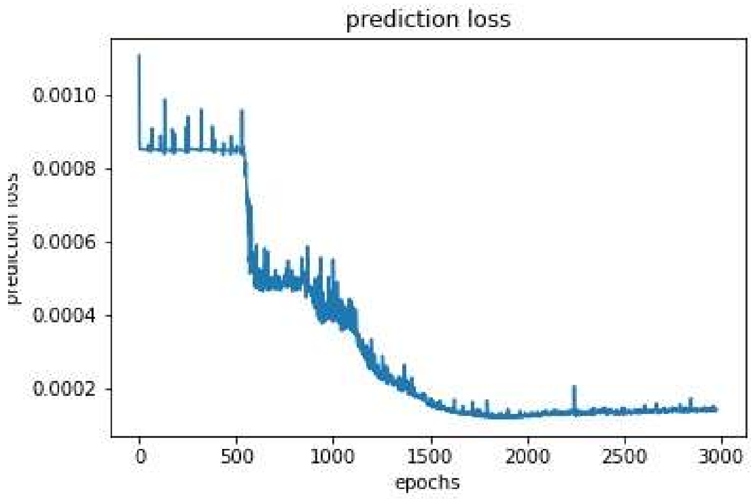
3) In order to predict value of stock in time x_{t+1}, we extended the LSTM AE presented before by adding a flag is_prediction to the model's constructor. When this flag is set to True, two things change in the way our model works:

- we add to our model a linear layer which predicts the value of the signal in the next day based on the values of previous days.
- We calculated to cost of the prediction with new MSE loss function combined with the already existing MSE loss function (for the reconstruction loss) to calculate the total loss by averaging both losses:

$$Loss_{total} = \frac{MSE(original\ image, reconstructed\ image) + MSE(predicted\ value, actual\ value)}{2}$$

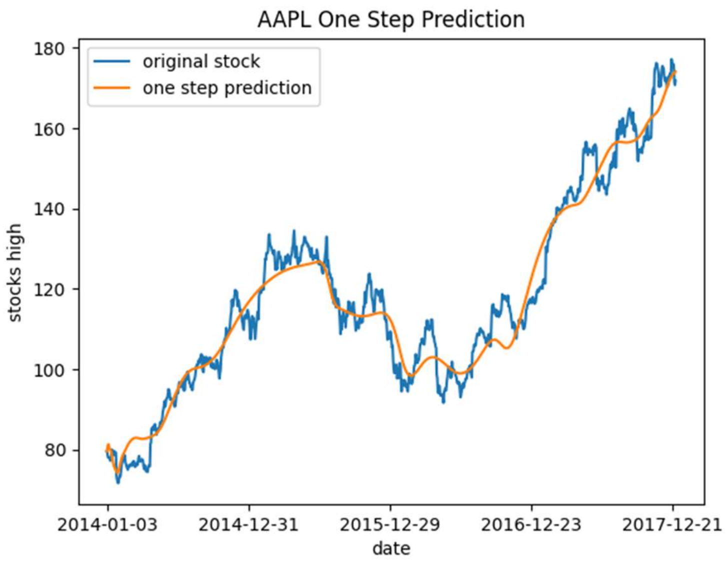
We've trained the new LSTM AE model for 3000 epochs. The following plots show the loss per epoch both for on training set and validation (prediction) set:

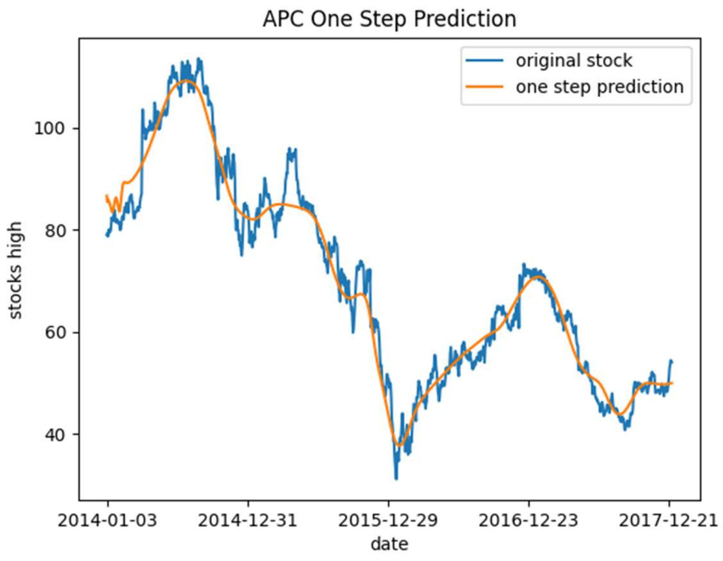




4) Using the trained model from the previous section we tested its prediction abilities.

First, one step prediction. In the graph below we tried for $\{x_i\}_{i=1}^{t}$ to predict x_{t+1}.

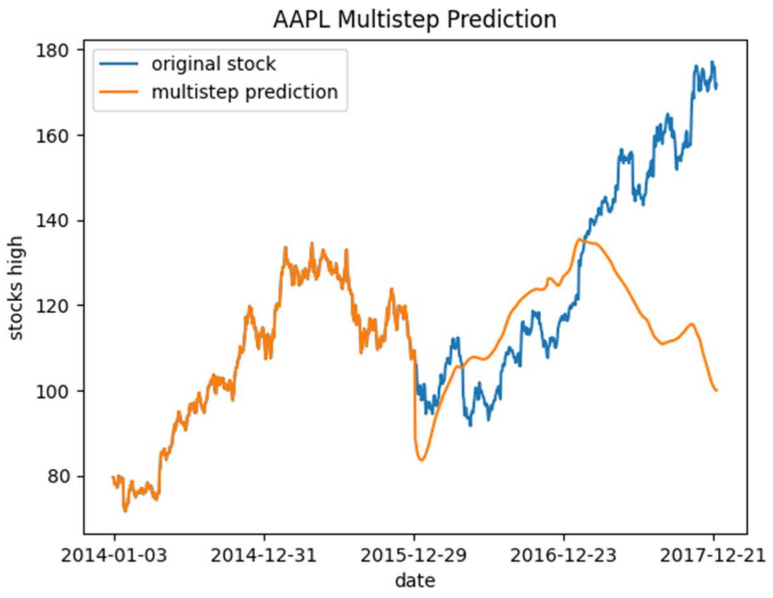


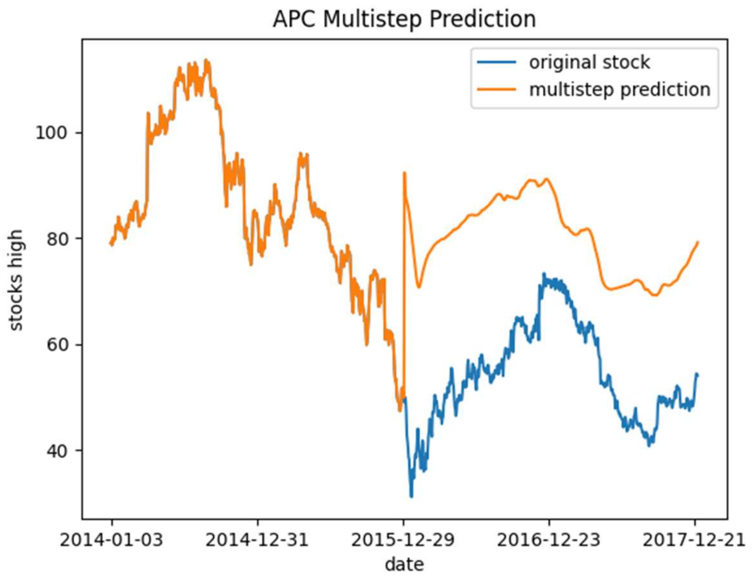


Then, we made another experiment for the same sequences, in which the model gets $\{x_t\}_{t=1}^{\frac{T}{2}}$ and tries to predict $x_{\frac{T}{2}+1}$. Using its prediction it tries to predict $x_{\frac{T}{2}+2}$ using $\{x_t\}_{t=2}^{\frac{T}{2}+1}$, and so on until it predicts all the missing value of the sequence.

The results for multistep prediction are at the plots below:

(notice that $\{x_t\}_{t=1}^{\frac{T}{2}}$ no prediction is made hence the graphs are the same)





Obviously, the accuracy of the multistep prediction is much worse than one step prediction. Although the error at one step prediction exists, it is independent from the errors it made before which in turn applies a small error overall. In contrast, at multistep prediction the error accumulates, meaning that error at time t depends on all errors at time $\leq t$. We can see that from nearly time $\frac{T}{2} + 1$ the prediction loses connection with the original signal, such an auto regression has bad accuracy results.

To conclude, we saw that our model was able to reconstruct the signal as well as predicting one step forward, but it failed to made a long-term multistep prediction due to the dependence of errors.