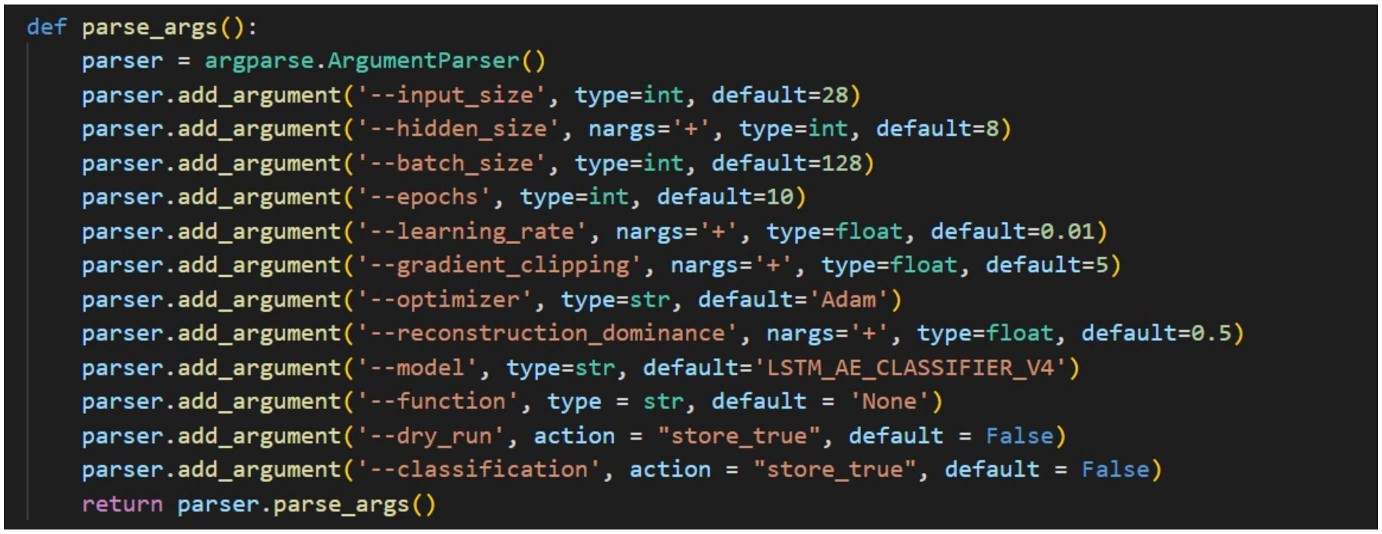
מגיש: אבינועם דוד
ת.ז: 206816910

<u>Submission Structure:</u> in addition to the scripts noted in the assignment, the code involves a few more files. The following is a quick guide-map for them:

1. scripts - organised in a way that each question has it's own function and can be called with any parameter you like:



```python
def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('--input_size', type=int, default=28)
    parser.add_argument('--hidden_size', nargs='+', type=int, default=8)
    parser.add_argument('--batch_size', type=int, default=128)
    parser.add_argument('--epochs', type=int, default=10)
    parser.add_argument('--learning_rate', nargs='+', type=float, default=0.01)
    parser.add_argument('--gradient_clipping', nargs='+', type=float, default=5)
    parser.add_argument('--optimizer', type=str, default='Adam')
    parser.add_argument('--reconstruction_dominance', nargs='+', type=float, default=0.5)
    parser.add_argument('--model', type=str, default='LSTM_AE_CLASSIFIER_V4')
    parser.add_argument('--function', type = str, default = 'None')
    parser.add_argument('--dry_run', action = "store_true", default = False)
    parser.add_argument('--classification', action = "store_true", default = False)
    return parser.parse_args()
```

   to run a function, use the - - function flag followed by the exact name of the function created for the question in the script, and fill all parameters you would like to run with. Note that for hyperparameter tuning, you can supply a list of each hyper-param you wish to tune. For example you can run with - -hidden_size 8 16 32.
   the 'dry_run' argument is meant for enabling you to preload a certain model from a previous run of a function to avoid having to train again to test something. However, since the assignment required no files other than ".py" and ".pdf", my pre-trained models have been deleted and you will need to do a full run (without the dry_run feature) for each function you run for the first time.

1. "logic" folder files - a self explanatory name for the content of it, and so is the name of each file inside it. The file "Function_Dump.py" consists of many previous attempts to implement each phase of the assignment. For example you'll find naive implementations of cross validation, basic scaling and normaising data methods, etc. These funcyions are not part of the code and should not be executed, but are rather there to show my learning process.
2. There are also empty folders such as "data" and "outputs". These are the target directories for scripts to save data for inter scripts communication and also to allow the user (you the reader) to not have to download the same data or train the same model twice in order to run a graph or a test (i.e the dry_run flag).

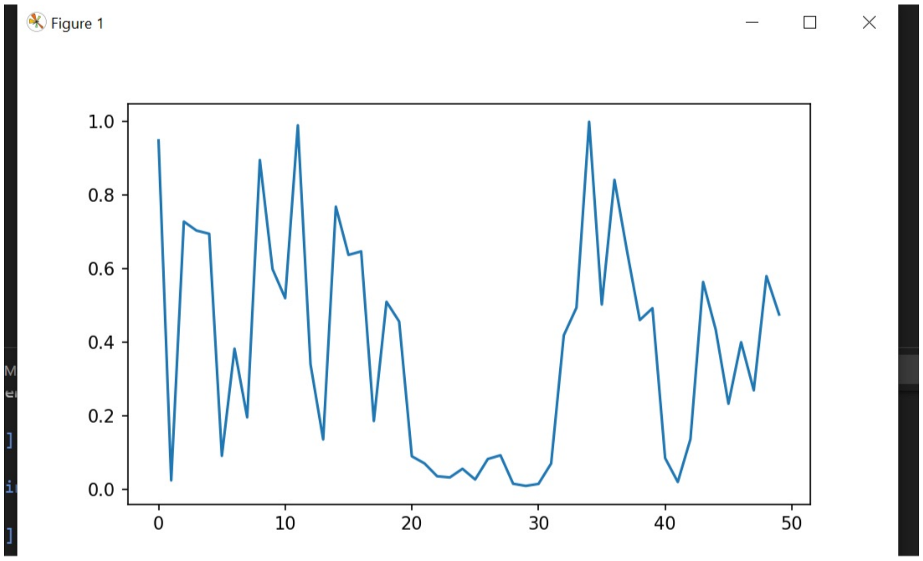
****Note that some files import others within the code so please retain the relative paths between them and do not delete or re-organise folders inside the code.**

*****Also, make sure you run all scripts from the same path they are located in inside your computer.**

Part 1 - Toy Model

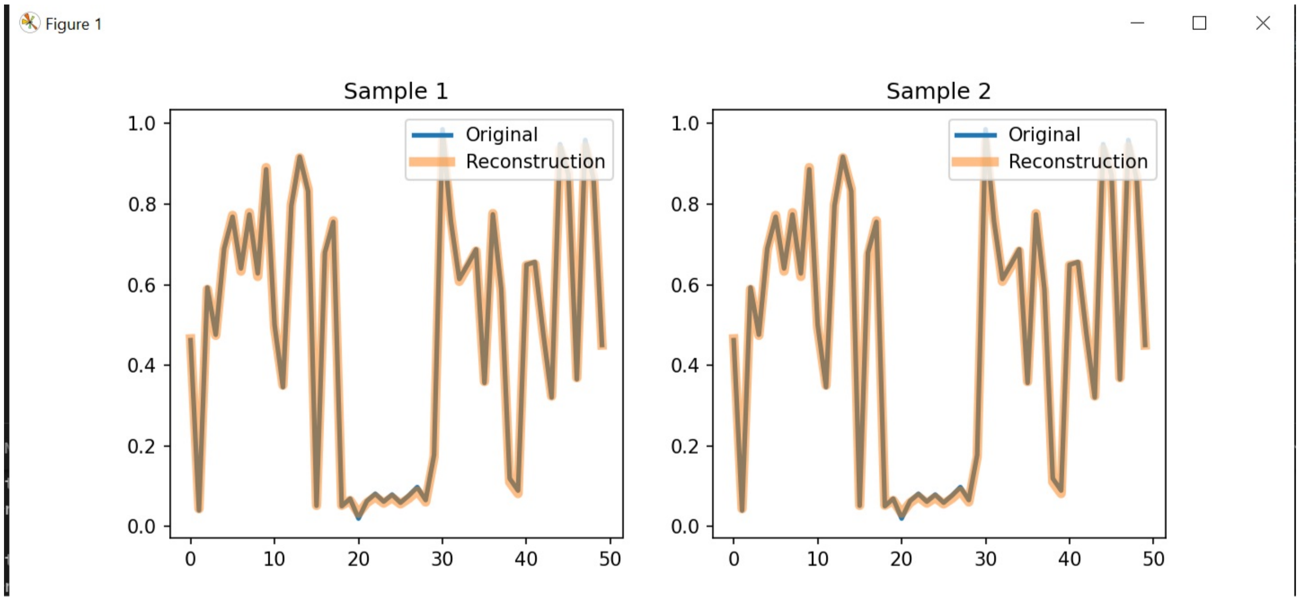
Q1:

The limitations of randomisation in the following documented run was shown in [20,30]

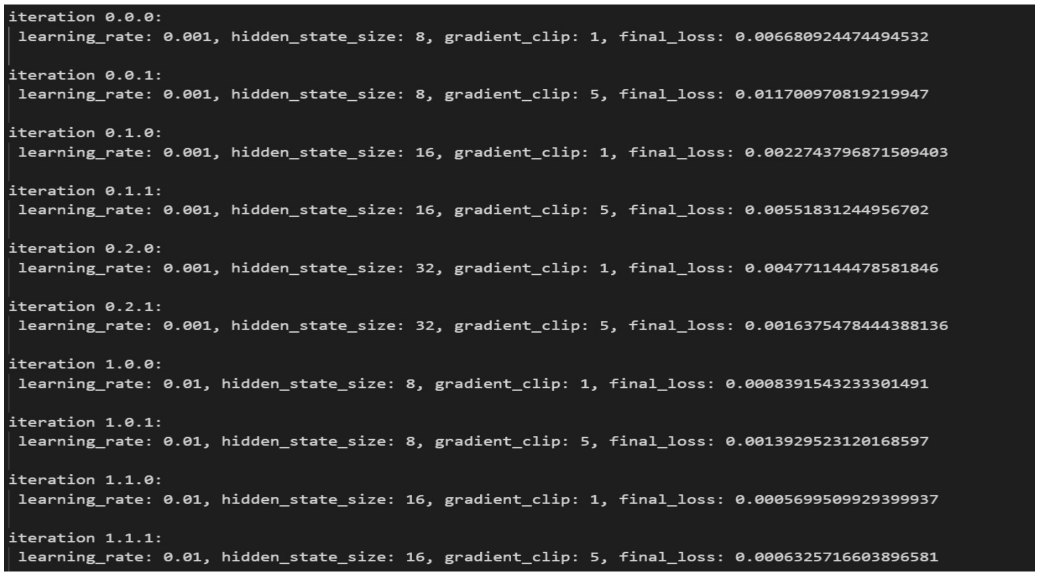


Q2:

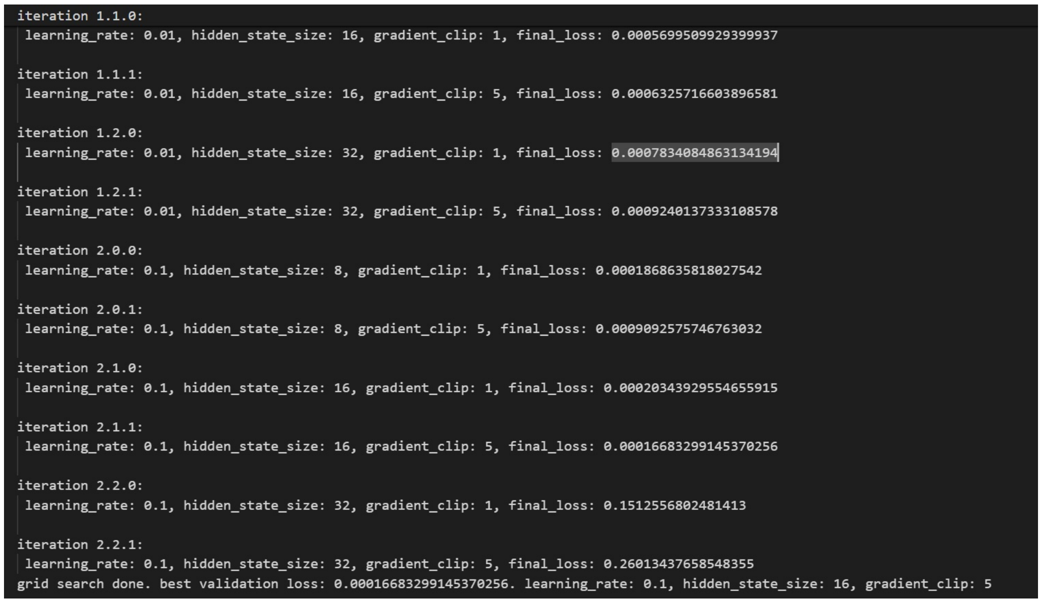
The first image is a run of the best performing model over 2 samples:



Learning rates in [0.1, 0.01, 0.001], hidden state sizes in [8,16,32], gardient clipping in [1,5].



```
iteration 0.0.0:
 learning_rate: 0.001, hidden_state_size: 8, gradient_clip: 1, final_loss: 0.006680924474494532

iteration 0.0.1:
 learning_rate: 0.001, hidden_state_size: 8, gradient_clip: 5, final_loss: 0.011700970819219947

iteration 0.1.0:
 learning_rate: 0.001, hidden_state_size: 16, gradient_clip: 1, final_loss: 0.0022743796871509403

iteration 0.1.1:
 learning_rate: 0.001, hidden_state_size: 16, gradient_clip: 5, final_loss: 0.00551831244956702

iteration 0.2.0:
 learning_rate: 0.001, hidden_state_size: 32, gradient_clip: 1, final_loss: 0.004771144478581846

iteration 0.2.1:
 learning_rate: 0.001, hidden_state_size: 32, gradient_clip: 5, final_loss: 0.0016375478444388136

iteration 1.0.0:
 learning_rate: 0.01, hidden_state_size: 8, gradient_clip: 1, final_loss: 0.0008391543233301491

iteration 1.0.1:
 learning_rate: 0.01, hidden_state_size: 8, gradient_clip: 5, final_loss: 0.0013929523120168597

iteration 1.1.0:
 learning_rate: 0.01, hidden_state_size: 16, gradient_clip: 1, final_loss: 0.0005699509929399937

iteration 1.1.1:
 learning_rate: 0.01, hidden_state_size: 16, gradient_clip: 5, final_loss: 0.0006325716603896581
```

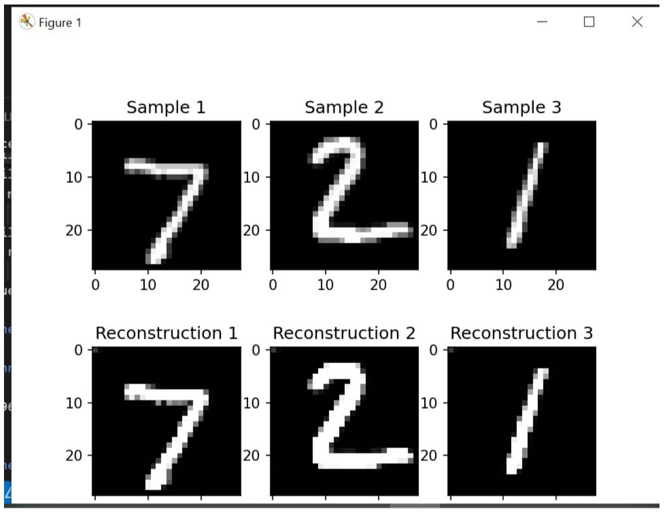


```
iteration 1.1.0:
 learning_rate: 0.01, hidden_state_size: 16, gradient_clip: 1, final_loss: 0.0005699509929399937

iteration 1.1.1:
 learning_rate: 0.01, hidden_state_size: 16, gradient_clip: 5, final_loss: 0.0006325716603896581

iteration 1.2.0:
 learning_rate: 0.01, hidden_state_size: 32, gradient_clip: 1, final_loss: 0.0007834084863134194

iteration 1.2.1:
 learning_rate: 0.01, hidden_state_size: 32, gradient_clip: 5, final_loss: 0.0009240137333108578

iteration 2.0.0:
 learning_rate: 0.1, hidden_state_size: 8, gradient_clip: 1, final_loss: 0.0001868635818027542

iteration 2.0.1:
 learning_rate: 0.1, hidden_state_size: 8, gradient_clip: 5, final_loss: 0.0009092575746763032

iteration 2.1.0:
 learning_rate: 0.1, hidden_state_size: 16, gradient_clip: 1, final_loss: 0.00020343929554655915

iteration 2.1.1:
 learning_rate: 0.1, hidden_state_size: 16, gradient_clip: 5, final_loss: 0.00016683299145370256

iteration 2.2.0:
 learning_rate: 0.1, hidden_state_size: 32, gradient_clip: 1, final_loss: 0.151255680248141З

iteration 2.2.1:
 learning_rate: 0.1, hidden_state_size: 32, gradient_clip: 5, final_loss: 0.26013437658548355
grid search done. best validation loss: 0.00016683299145370256. learning_rate: 0.1, hidden_state_size: 16, gradient_clip: 5
```

It seems learning rate = 0.1 was the best in our case and a strong tendency of gradient_clip = 1 to be better.
The bigger the hidden state size, the better the performance. That makes sense because it allows the model more flexibility.
For that reason, and for run-time, I was impressed by the smaller hidden layer size models who had very close performance.

Part 2 - MNIST (Classification)

Q1:



Q2:

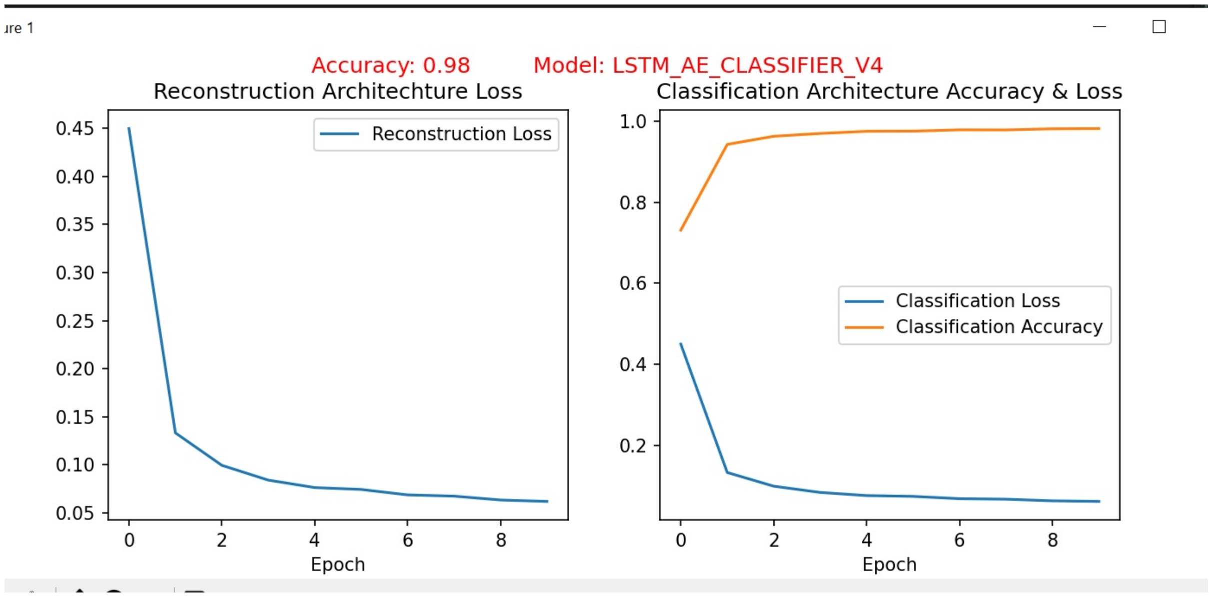
The process of surpassing 98% accuracy included many experiments. Below are some of them and their findings.

- learning rate of 0.1, which was the best performer in the previous section, was horrible. Each training ranged in accuracy between just 9% (less than a coin flip) to 80%. When inspecting the losses of each iteration, it's clear that the lr=0.1 is too big and causes scattering.

  A grid search of hyperparameters like in the previous section helped conclude 0.01 gives the most steady decline, and also shows a hidden size of only 8 shows virtually no difference in accuracy or reconstruction quality - however FAR better on performance, since now its over a much longer sequence of data
- separating classification and reconstruction to 2 different LSTMs (as shows in the models of V3, V4 in the code) improves performance from 94% precision to around 97%.
- After passing through the LSTM AE, adding more layers of even regular FC layers improves performance by another 1% or more (like the model V4). On the classification side - the more complexity after reconstructing, the better. That being said, more LSTM layers don't do much.
- When distributing the loss between the classification and reconstruction unevenly - one part gets better results. However this difference is not much noticeable when done over an already well proven model. Its seems like a 50:50 split (like summing both losses) seems the best compromise.

  Interestingly enough, on a model that passes through the same LSTM for both the reconstruction and classification (like model V2), the best accuracy was achieved by giving a slight weight to the reconstructed penalty.

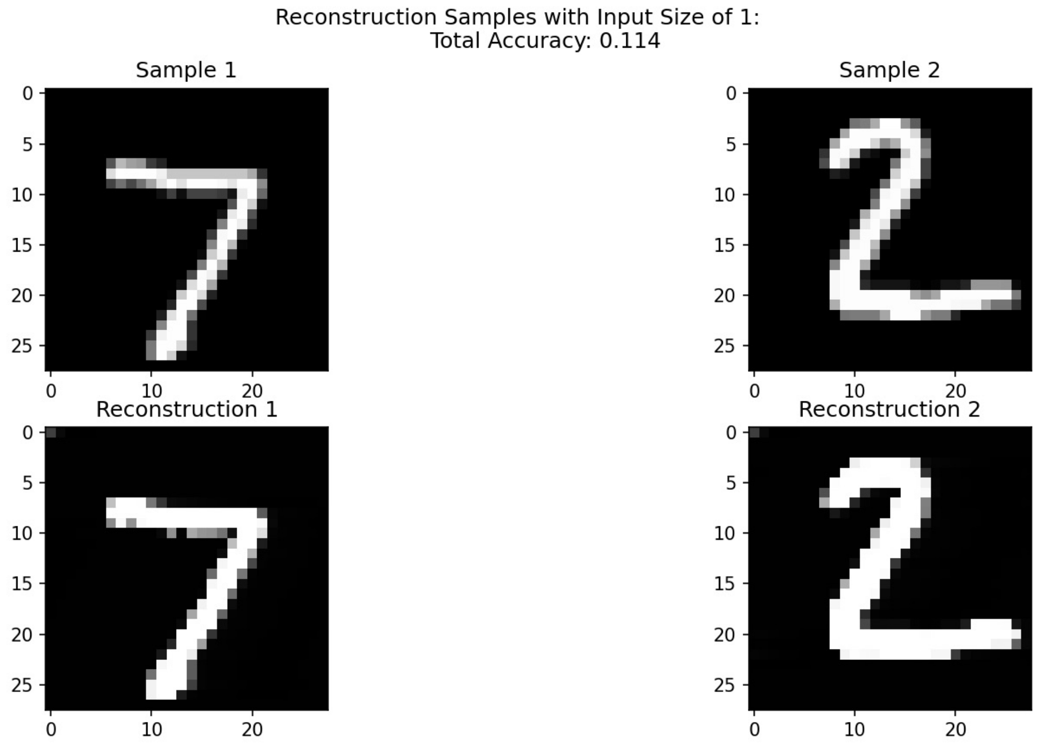
For the sake of keeping a concise report, all findings have dedicated functions for your free use in Function_Dump.py, and will not be attached to this report. Below is the plot of accuracy and loss vs Epochs of the best performing model:



<u>Q3:</u>

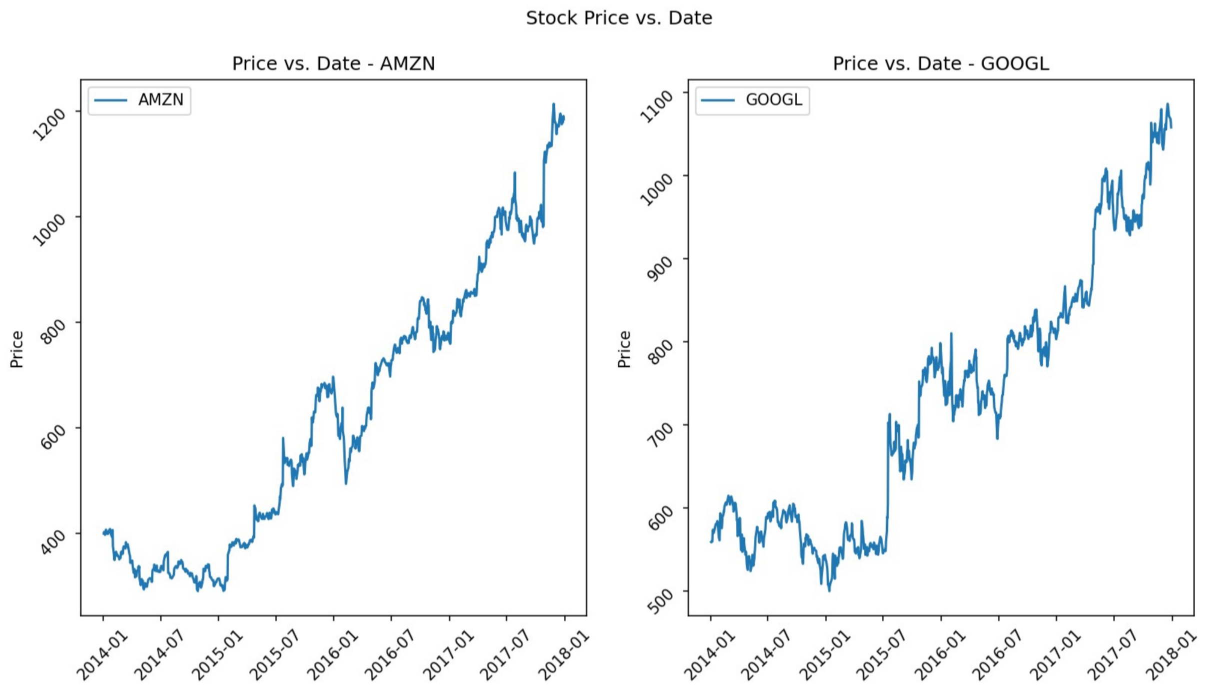
When downgrading to feeding 1 pixel at a time:

- Accuracy dropped drastically. Runs varied with 15% success on average, and one run (presented below) did as bad as ~10%. It might be because it causes a sequence of 784 single pixels to nerf into an output with hidden size of 16 just can't contain enough information about the picture to classify with. Or it could just be that it's too much "to remember" for an LSTM in terms of classification, or at least requires more hyper_parameters tuning.
- Nevertheless, reconstruction works seamlessly. I can assume it's either a simpler task or 1 that's more suitable for an LSTM
- In terms of run-time, results are also far worse, even though the data was the same and the network was of the same size. This comes to show the iterative nature of feeding RNNs and LSTMs specifically with data - as discussed in class.



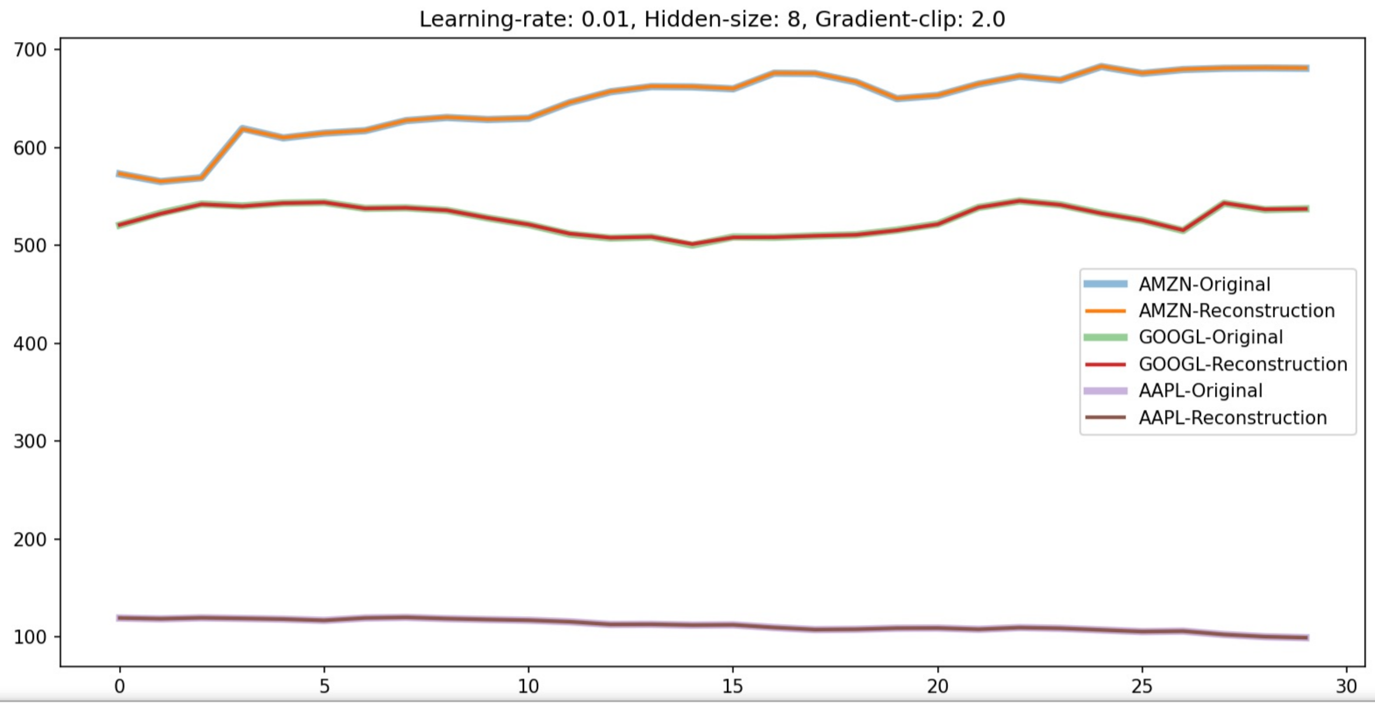
Part 3 - S&P500 (Prediction)

Q1:



Q2:

- At first, reconstruction took about 10,000 epochs to fit the data. This popped the idea of a meed of scaling the data. By that time only 2-3 epochs were actually enough to perfectly fit.

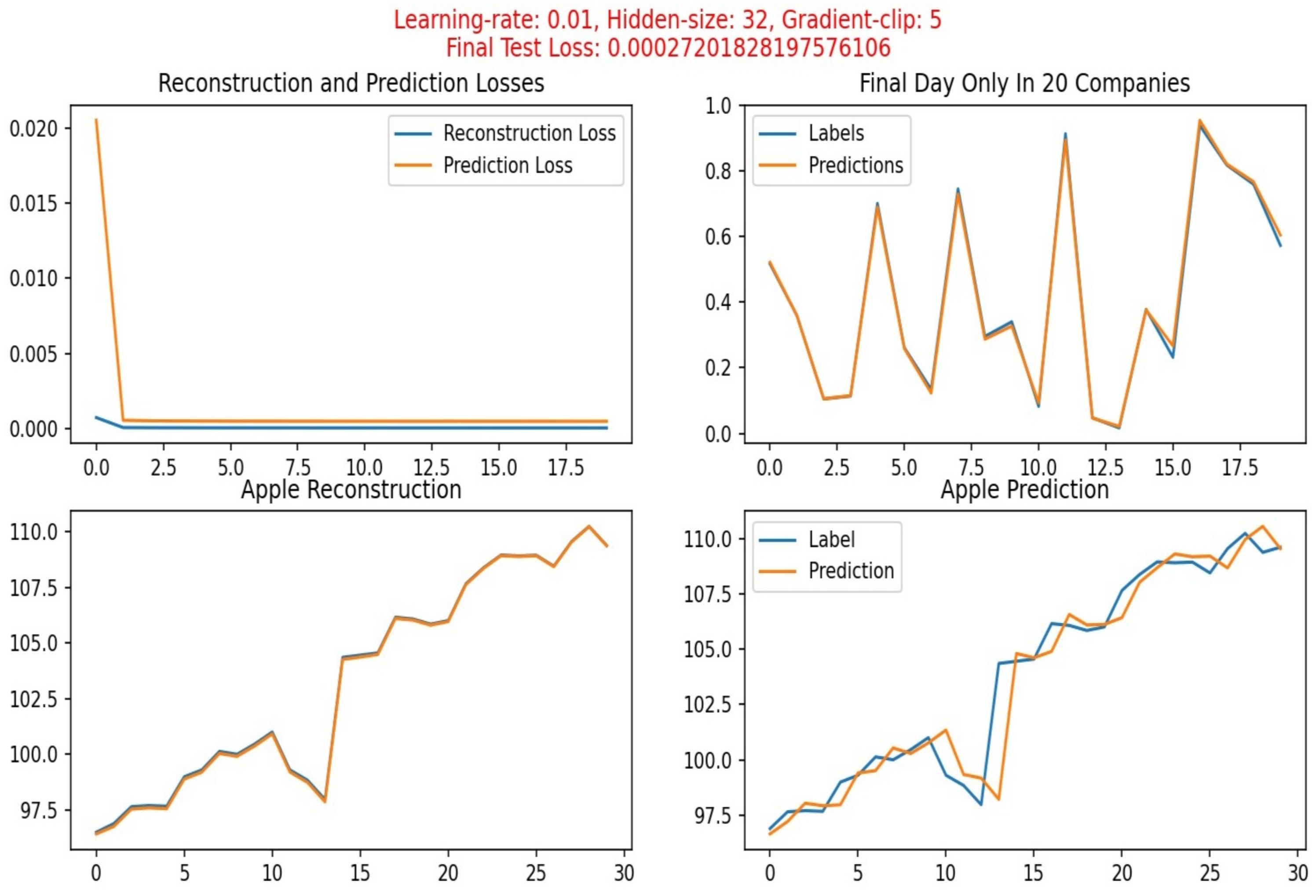


Q3:

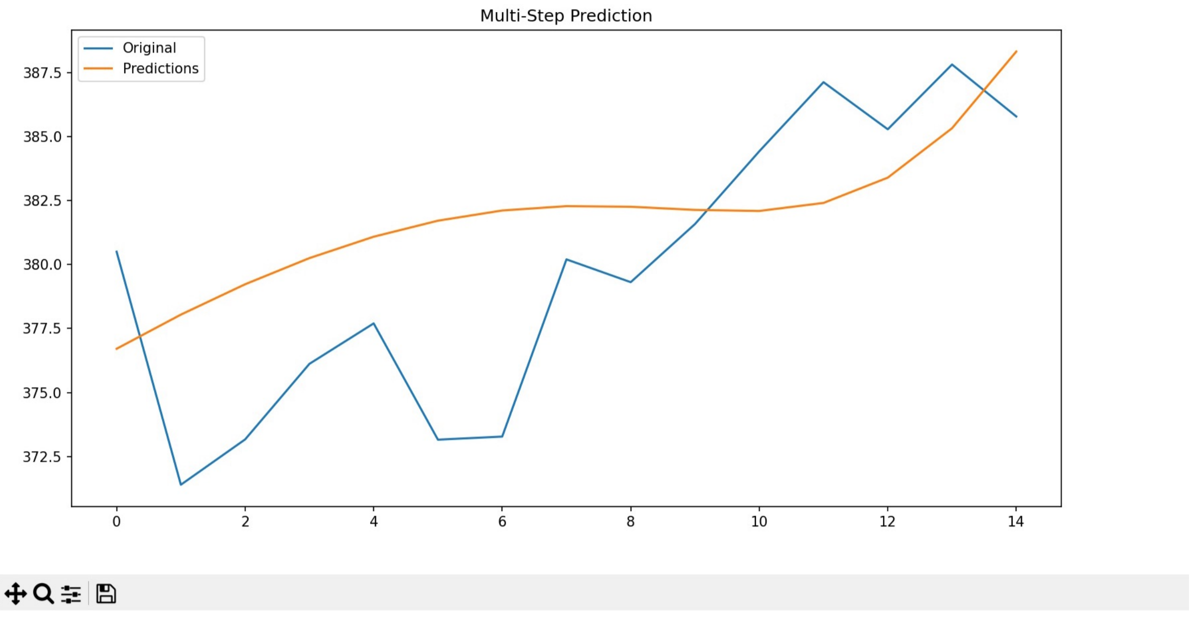
- The task of prediction seems much harder for the model to learn than the reconstruction.
- Attempts to succeed relatively well included - different normalisation methods and, hyper-parameters, cleaning and organising the data for feeding the network with, and more.
- They key objectives that did the most significant effect:

  - Using a MinMaxScaler() model and fitting it separately for each company.
  - Composing each sequence as sequential days of the same company (no two companies in the same sequence).
  - shuffling the sequences before training.
  - using KFolds to filter out bad hyper-params combinations and then running KFolds AGAIN for each selected hyper-param set and choosing the best one.
  - Tuning down sequence lengths to about 30.
- Although not asked for, a version of the assignment where the model is required to predict only the last (and the only unseen) day of the sequence was implemented and results were far better. Further research showed that even the model we WERE required to implement (one that predicts all days of the sequence with a right shift of one day) did stunningly better on the last and unseen day than on the rest of the days, which it already seen. This insight is viewed in the top right figure in the image below - which consists of only the last day of a random sequence of 20 different companies.

You can see the difference between the last day matches and the whole sequence's prediction in the right side figures, with the bottom figure showing the prediction of a single company (Apple) along all 30 days in it's sequence.



Q4:



- The result graph is a more "smoothed" and less accurate version of the original graph. Though It does describe pretty well the general trend of the sequence.
- This can probably explain that the model simply follows the momentum he has seen in previous days. It shows that when given his own previous output, it doesn't hurry to throw himself off track so aggressively because the difference between his input and his t-1 first predictions is little.

Some key take-aways:

- reconstruction is a far easier task for an LSTM that classification/prediction.
- Hyper-params tuning is key.
- The amount of hidden-layers should be large enough to contain enough info
- data cleaning and engineering is key.