In the name of God

Final Project Report

Course: Machine Learning

Instructor: Dr. Sajedi

Semester: Fall 1400

Presenter: Alireza Kazemipour (610300171)

- ## Dataset

  Among the 3 available dataset options, the [Top 50 Cryptocurrencies Historical Prices] was chosen. For the sake of better representation, we chose 11 cryptocurrencies from 50 ones to show our results.

  The objective we set to solve by **Deep Learning** approaches, was predicting future prices based on historical data of the past.

In order to obtain the dataset a module *downloader.py* was written to download and unzip the data from Kaggle (Kaggle's username and token key should be provided):

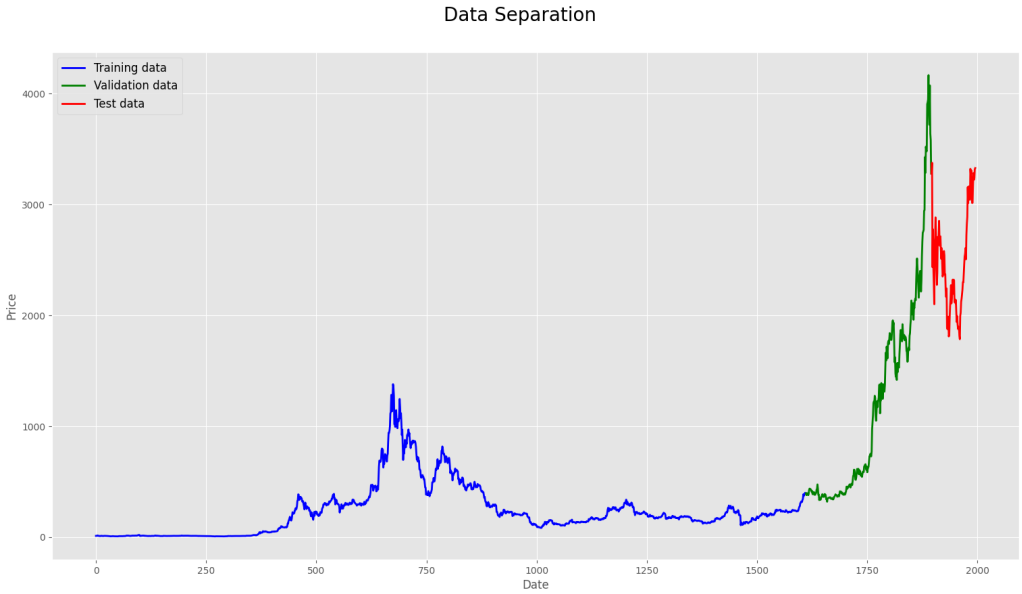
```
def download_dataset(args):
    print("***If you live in Iran, turn on your VPN!***")
    os.environ["KAGGLE_CONFIG_DIR"] = os.path.dirname(os.path.realpath(__file__))
    if not os.path.exists("kaggle.json"):
        os.environ["KAGGLE_USERNAME"] = args.kaggle_username
        os.environ["KAGGLE_KEY"] = args.kaggle_key
    os.system("kaggle datasets download -d " + args.dataset_name)


def prepare_dataset(dataset_name):
    if os.getcwd().split(os.sep)[-1] == "data":
        path = "../datasets/" + dataset_name
    else:
        path = os.path.join(os.getcwd(), "datasets", dataset_name)
    os.makedirs(path, exist_ok=True)
```

```
with zipfile.ZipFile(dataset_name + ".zip", mode='r') as f:
    f.extractall(path)
os.remove(dataset_name + ".zip")

for subpath in os.listdir(path):
    folder = os.path.join(path, subpath)
    if os.path.isdir(folder):
        for dir in os.listdir(folder):
            file = os.path.join(path, subpath, dir)
            shutil.move(file, path)
        os.removedirs(folder)
```

- Methodology

We splitted each dataset such that the last 5% of the data that is the most up-to-dated one is used for testing phase, the 15% prior to the testing portion was dedicated for validation phase and all the remaining for training.

An example of the split:



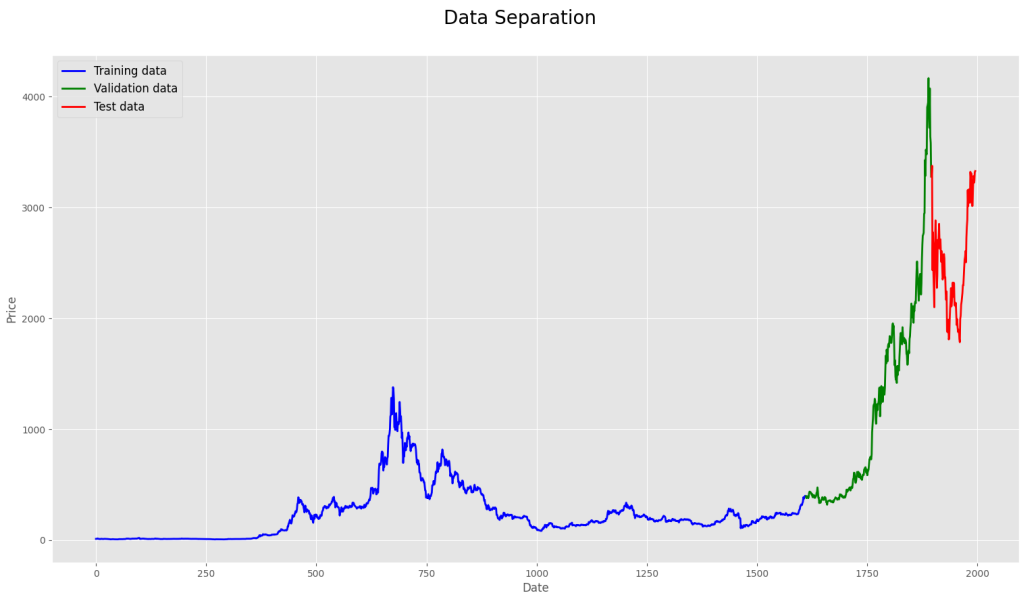
Also, we used 5 features: *Price, Low, High, Volume, Open*. We fed the features of the last 4 days and the model should have predicted the ***Price* of the 5th day.**

The split was done in *data/data_loader.py*:

```
def get_sets(dataset_path,
             valid_portion=0.15,
             test_portion=0.05
             ):
    df = pd.read_csv(dataset_path)
    data = df[["Price", "Open", "High", "Low", "Vol."]]

    train_portion = 1 - test_portion
    train_size = int(train_portion * len(data))
    train_data = data.loc[:train_size]
    test_df = data.loc[train_size:]

    train_portion = 1 - valid_portion
    train_size = int(train_portion * len(train_data))
    train_df = train_data.loc[:train_size]
    valid_df = train_data.loc[train_size:]

    return df, train_df, valid_df, test_df
```

- Input/Output Normalization

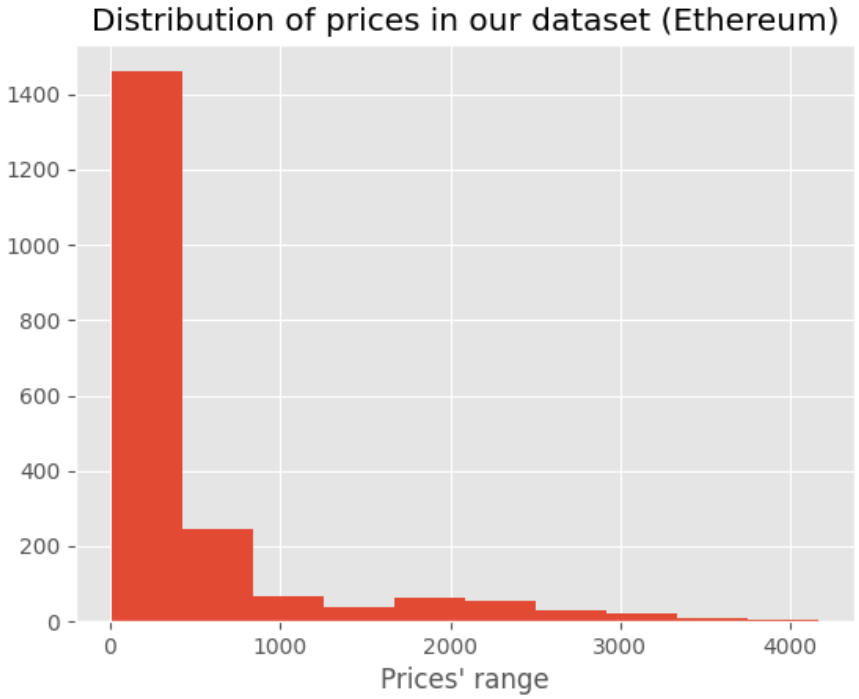
  Common strategies of normalization/standardization of data is not applicable. Because we look at the prices of the Ethereum coin over time:



We see that all of the data in our training set is less than the validation and test sets thus, the statistics computed during training can not be extended to normalize validation

and test sets. In other words, the price of Ethereum was low in the past and that does not correspond to its recent upward trend.

This problem can be viewed by evaluating the distribution of the data and we we can see that it is not Gaussian either:



Therefor to address this problem, we have used another trick to normalize the data; all the data in a same window (4 days in a row) are normalized with respect the the first day in that window and the target day's (5th's) price is also normalized with respect to the price of the first day of the window.

Why is this kind of normalization valid and when does it fail?

It's valid because if we have assumed that we can estimate the value of the 5th day by using 4 days prior to that then, they must have come from the same distribution and that distribution.

It fails when one feature of the first day is exactly zero! (as a result of division by zero)

This procedure of windowing the dataset (grouping 4 days in a row and selecting the 5th day as the target) and normalization was done in *utils/common.py* (we used eps=1e-6 to avoid division by zero problem but it caused another problem which is making values very large):

```
def make_sequence(data, n, offset, eps=1e-6):
    #
https://dashee87.github.io/deep%20learning/python/predicting-cryptocurrency-prices-
with-deep-learning/
    data = data.values
    x, y = [], []
    for i in range(offset, len(data)):
        x.append(data[i - n:i] / (data[i - n] + eps) - 1)
        y.append(data[i][0] / (data[i - n][0] + eps) - 1)

    return np.stack(x), np.hstack(y)
```

- ## Model architectures

  In order to estimate the 5th day's price we deployed 5 strategies that 2 of them were statistical and 3 of them were based on Deep Learning.

1. ### Simple Moving Average:

   The target price is roughly the mean of the prices of the window.

2. ### Exponentially Moving Average

$$EMA = P_t * k + EMA_{t-1} * (1 - k)$$

P_t: Price at the timestep t

N: Number of timesteps in the window

K: Weight factor $= \frac{2}{N+1}$

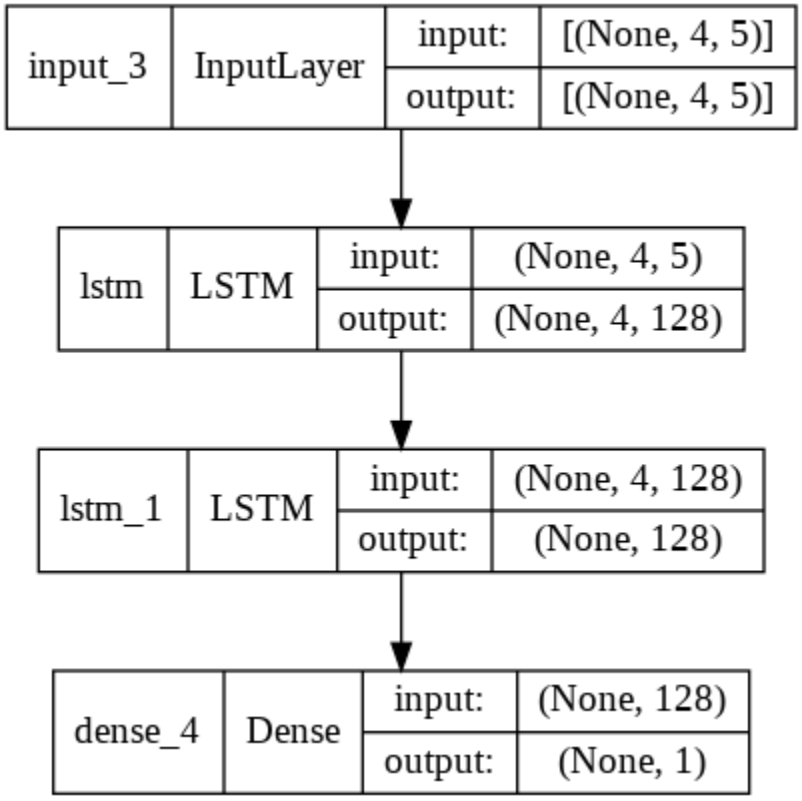
3. ### LSTM

   Long-Short Term Memory cells are very typical when we are processing sequential data that is our case too. LSTMs are able to process each timestep in the window and flow the data between each step via their hidden/cell states connections.

   We used this architecture:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 4, 128)            68608
_________________________________________________________________
```

```
lstm_1 (LSTM)                (None, 128)               131584
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 129
=================================================================
Total params: 200,321
Trainable params: 200,321
Non-trainable params: 0
_________________________________________________________________
```



4. CNN

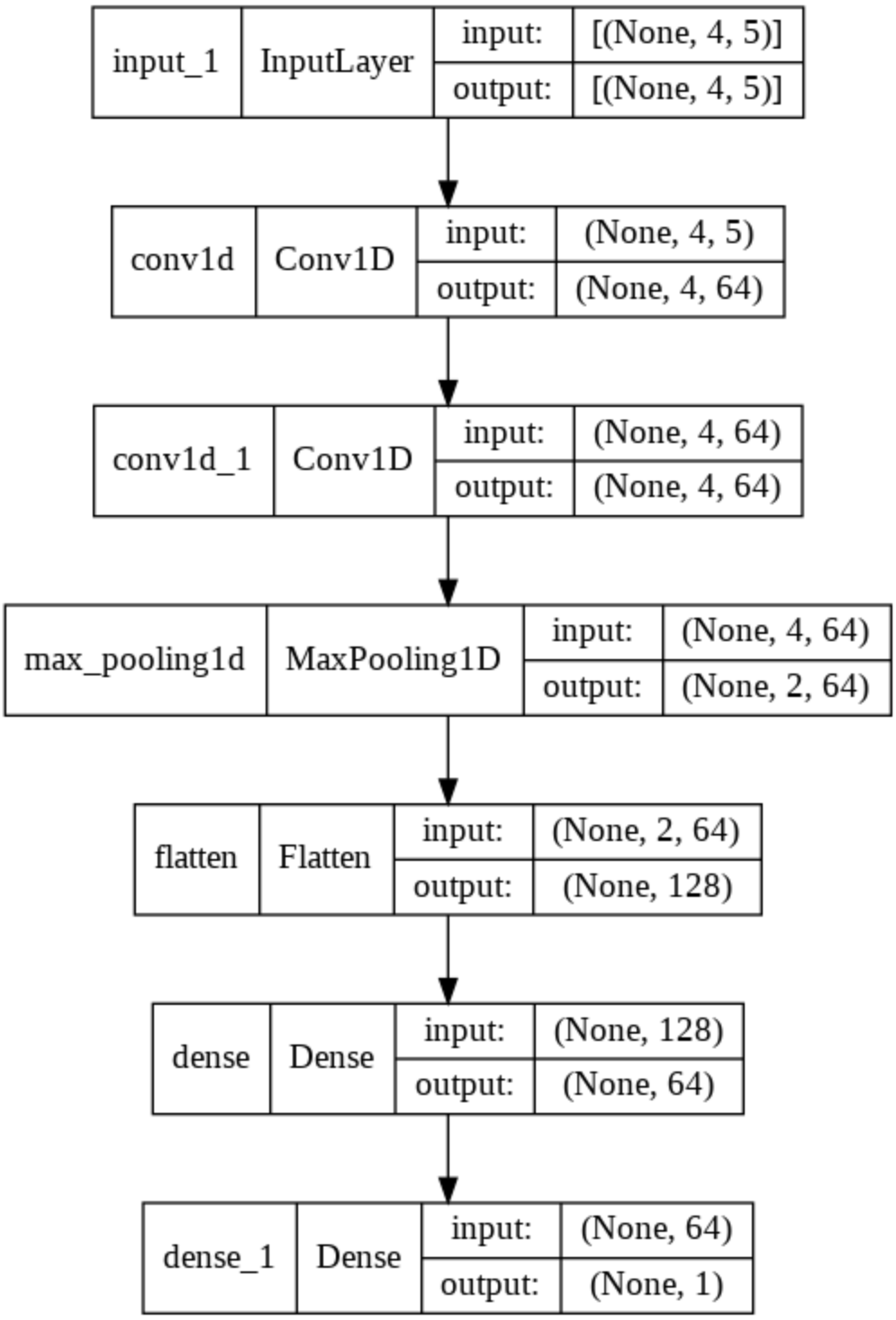
Convolutional Neural Networks are especially successful architectures to handle 2D data like images. In our case our data is a special case of 2D instances where the number of rows are equal to 1, number of columns are equal to 4 (sequence length) and the number of channels (5) are equal to the number of input features.

This special case of 2D CNNs are known as Conv1D neurons. They will sweep our window with unique weights and produce outputs based on all timesteps but in a sparse way.

We used this architecture (we used MaxPooling for reducing number of features and extracting important ones and it was optional):

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 4, 64)	1024
conv1d_1 (Conv1D)	(None, 4, 64)	12352
max_pooling1d (MaxPooling1D)	(None, 2, 64)	0
flatten (Flatten)	(None, 128)	0
dense (Dense)	(None, 64)	8256
dense_1 (Dense)	(None, 1)	65

Total params: 21,697
Trainable params: 21,697
Non-trainable params: 0



5. Transformer

   One of the downsides of using LSTMs is that they process sequences serially and they can not be used in parallel ways. Further, they suffer from exploding/vanishing gradients as the sequential computations alter the gradient

flow to the old timesteps. As a result, in NLP field another idea came up that could be implemented in parallel and also at a sme time it pays attention to all timesteps so, there is no need to process the sequence serially. This idea is called Self-Attention and it is based on the 3 quantities known as *Key, Value* and *Query.*

For each timestep, Key, Value and Query are computed and based on the following formula the quantity attention is obtained. It simply says that to make the prediction, the network should how much pays attention to each timestep based on this quantity.

$$attention = softmax(\frac{q \times k}{dimension}) \times v$$

It is also possible to have multiple attention heads to build a multi-head attention mechanism and improve accuracy (as we have done). A non linear combination of multi-head attentions can estimate the target.

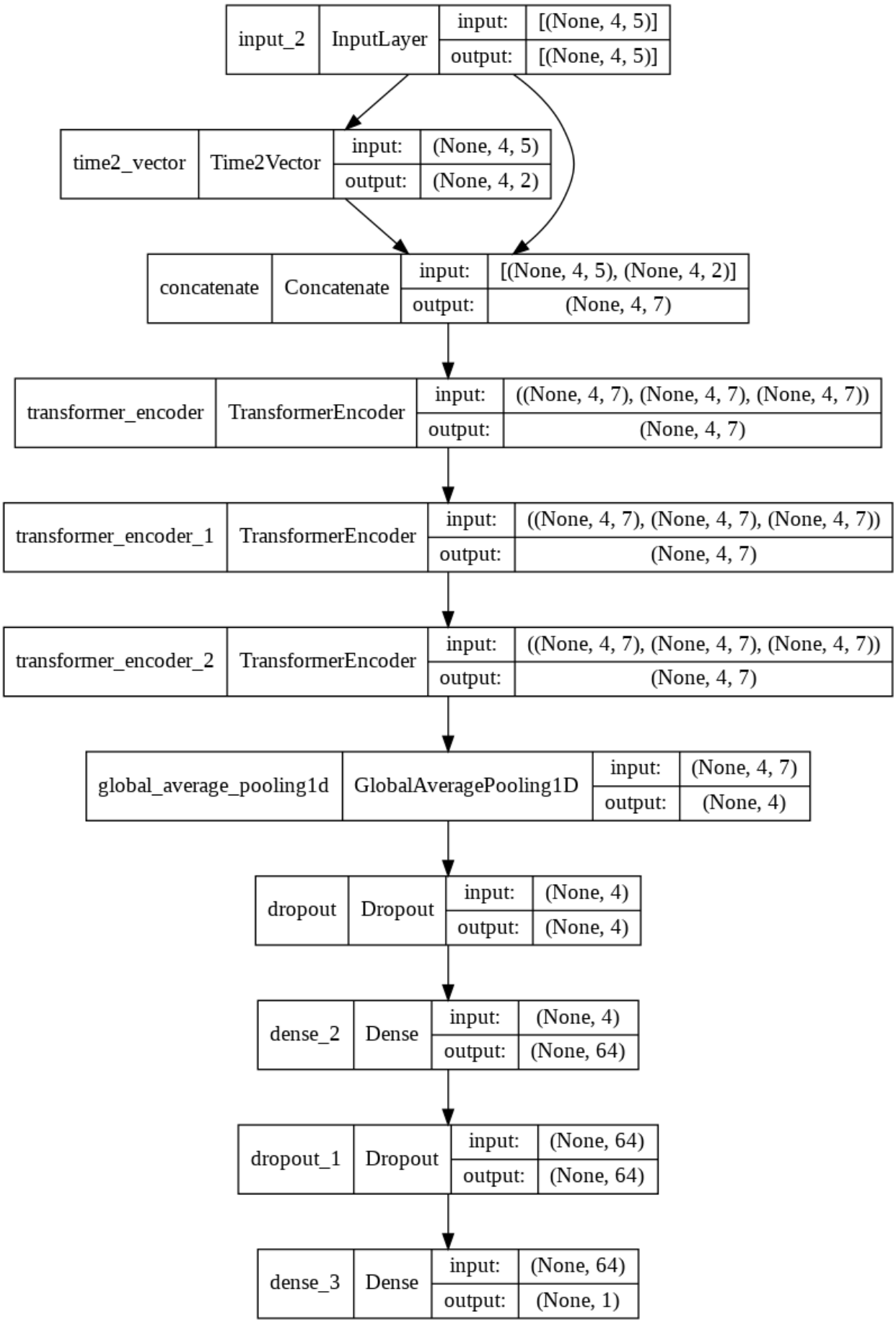
It is necessary to point out that in order to avoid eliminating the sequential dependency between timesteps we have to inject the concept of time to the network otherwise it does not inform that that the input constructs a sequence because it processes the data in parallel.

To capture time, we have used a trick named Time-to-Vec. It captures periodic and non-periodic relations within our dataset and is used as an extra feature during training. It produces periodic and non-periodic embedding and learns how during training how to produce these quantities. Thus the number input's features to the network are 7 in this setting.

We have used this architecture:

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	[(None, 4, 5)]	0	
time2_vector (Time2Vector)	(None, 4, 2)	16	input_2[0][0]
concatenate (Concatenate)	(None, 4, 7)	0	input_2[0][0] time2_vector[0][0]
transformer_encoder (Transforme	(None, 4, 7)	1770	concatenate[0][0] concatenate[0][0] concatenate[0][0]
transformer_encoder_1 (Transfor	(None, 4, 7)	1770	transformer_encoder[0][0] transformer_encoder[0][0] transformer_encoder[0][0]
transformer_encoder_2 (Transfor	(None, 4, 7)	1770	transformer_encoder_1[0][0] transformer_encoder_1[0][0] transformer_encoder_1[0][0]
global_average_pooling1d (Globa	(None, 4)	0	transformer_encoder_2[0][0]
dropout (Dropout)	(None, 4)	0	global_average_pooling1d[0][0]
dense_2 (Dense)	(None, 64)	320	dropout[0][0]
dropout_1 (Dropout)	(None, 64)	0	dense_2[0][0]
dense_3 (Dense)	(None, 1)	65	dropout_1[0][0]

Total params: 5,711
Trainable params: 5,711
Non-trainable params: 0


input_2 InputLayer input: [(None, 4, 5)] output: [(None, 4, 5)]
time2_vector Time2Vector input: (None, 4, 5) output: (None, 4, 2)
concatenate Concatenate input: [(None, 4, 5), (None, 4, 2)] output: (None, 4, 7)
transformer_encoder TransformerEncoder input: ((None, 4, 7), (None, 4, 7), (None, 4, 7)) output: (None, 4, 7)
transformer_encoder_1 TransformerEncoder input: ((None, 4, 7), (None, 4, 7), (None, 4, 7)) output: (None, 4, 7)
transformer_encoder_2 TransformerEncoder input: ((None, 4, 7), (None, 4, 7), (None, 4, 7)) output: (None, 4, 7)
global_average_pooling1d GlobalAveragePooling1D input: (None, 4, 7) output: (None, 4)
dropout Dropout input: (None, 4) output: (None, 4)
dense_2 Dense input: (None, 4) output: (None, 64)
dropout_1 Dropout input: (None, 64) output: (None, 64)
dense_3 Dense input: (None, 64) output: (None, 1)

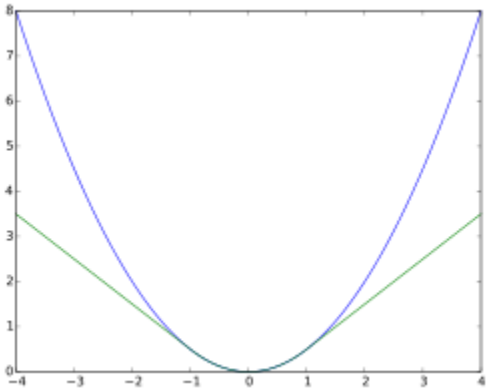
These models were defined in the models package.

```
def create_lstm(seq_len, in_dim):
    model = Sequential(
        [
            Input(shape=(seq_len, in_dim)),
            LSTM(128, return_sequences=True, ),
            LSTM(128),
            Dense(1)
        ]
    )
    return model


def create_cnn(seq_len, in_dim):
    model = Sequential(
        [
            Input(shape=(seq_len, in_dim)),
            Conv1D(64, kernel_size=3, activation="relu", padding="same"),
            Conv1D(64, kernel_size=3, activation="relu", padding="same"),
            MaxPooling1D(pool_size=2),
            Flatten(),
            Dense(64, activation="relu"),
            Dense(1)
        ]
    )
    return model

def create_transformer(seq_len, in_dim, d_k, d_v, n_heads, ff_dim):
    '''Initialize time and transformer layers'''
    time_embedding = Time2Vector(seq_len)
    attn_layer1 = TransformerEncoder(in_dim, d_k, d_v, n_heads, ff_dim)
    attn_layer2 = TransformerEncoder(in_dim, d_k, d_v, n_heads, ff_dim)
    attn_layer3 = TransformerEncoder(in_dim, d_k, d_v, n_heads, ff_dim)

    '''Construct model'''
    in_seq = Input(shape=(seq_len, in_dim))
    x = time_embedding(in_seq)
    x = Concatenate(axis=-1)([in_seq, x])
    x = attn_layer1((x, x, x))
    x = attn_layer2((x, x, x))
    x = attn_layer3((x, x, x))
    x = GlobalAveragePooling1D(data_format='channels_first')(x)
    x = Dropout(0.1)(x)
    x = Dense(64, activation='relu')(x)
    x = Dropout(0.1)(x)
```

```
out = Dense(1, activation='linear')(x)

model = Model(inputs=in_seq, outputs=out)
return model
```

- Training configurations
    1. Loss

    We used *Huber* as our loss function. Huber loss is a variant of MSE and MAE that is more robust to the outliers. More clearly, it behaves like MSE near the origin (0) and behaves like MAE anywhere else.



    Why did we choose this loss?
    The most appropriate loss for the task of real value estimation is MSE but our special Input/Output normalization produce many outliers when a feature of the first day in the window is zero (ver near zero) thus, we should have selected a loss function that is more robust to the outliers.

    Why is this loss more robust to the outliers than MSE?
    Because if we look at the figure above, when numbers go to infinity (whether positive or negative) the MSE goes much larger than the MAE and since Huber acts like MAE for large numbers, it is more robust.

    Also, when the numbers are small it acts like MSE and does not bother the training landscape.

    2. Optimizer

    We used Adam optimizer with its default settings: learning rate = 1e-3
    Batch Size = 128
    Epochs = 10
    Number of attention heads = 3

3. Metrics

   We used MAE as our metric.

These configurations were defined in *main.py*:

```
model.compile(loss="huber", optimizer="adam", metrics=['mae'])
```

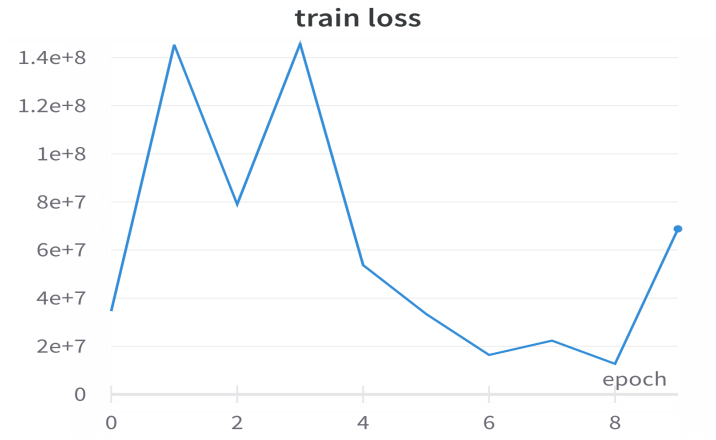
- Scientific Framework

In all of our experiments, we used the TensorFlow framework to implement different Neural Network's related settings.

- Logging

  To log different metrics, artifacts etc, we have used WandB's api.

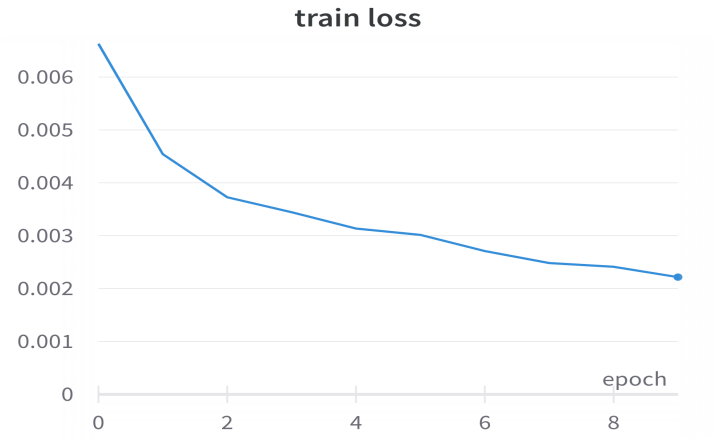
- Sample Training Plots (Ethereum)
  1. CNN

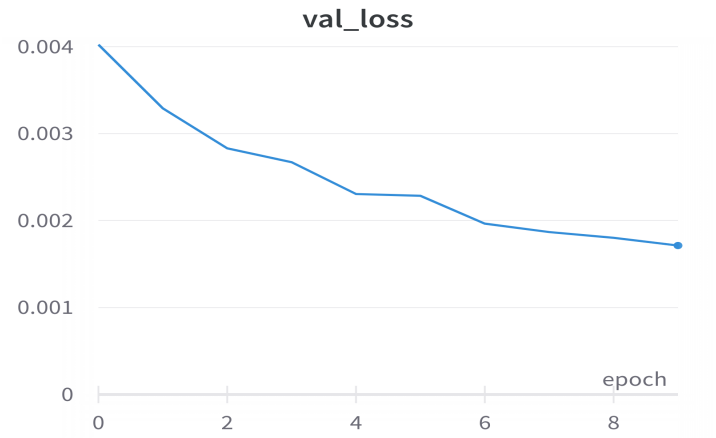


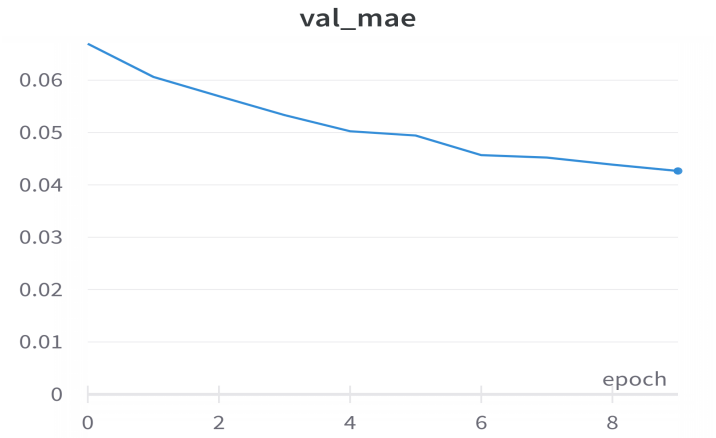

val_loss
0.05
0.04
0.03
0.02
0.01
0
epoch
0
2
4
6
8


val_mae
0.12
0.1
0.08
0.06
0.04
0.02
0
epoch
0
2
4
6
8

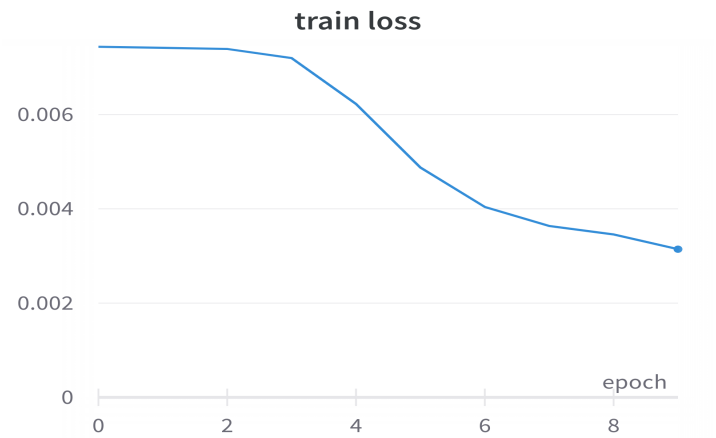
2. LSTM

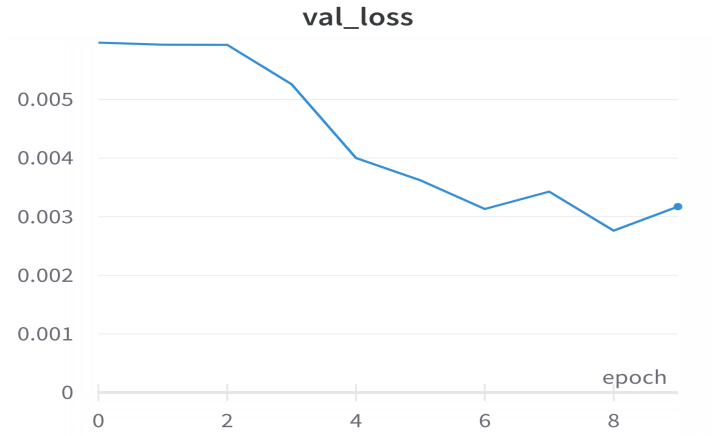

train loss
0.006
0.005
0.004
0.003
0.002
0.001
0
epoch
0
2
4
6
8





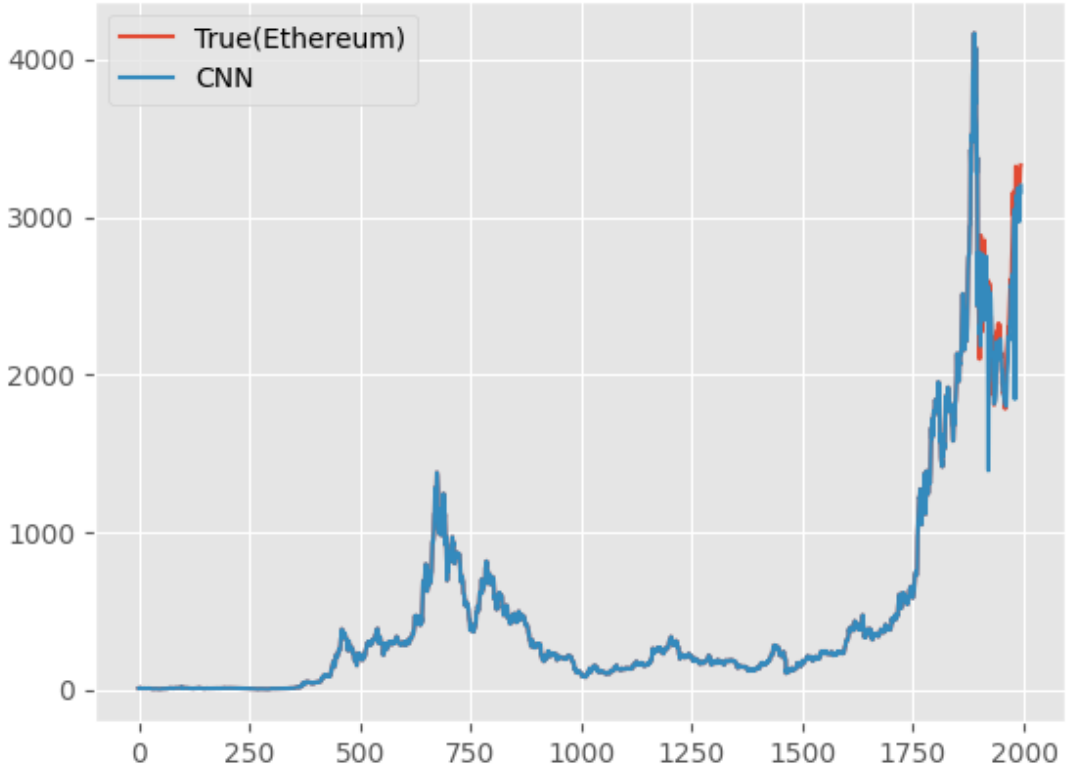
3. Transformer

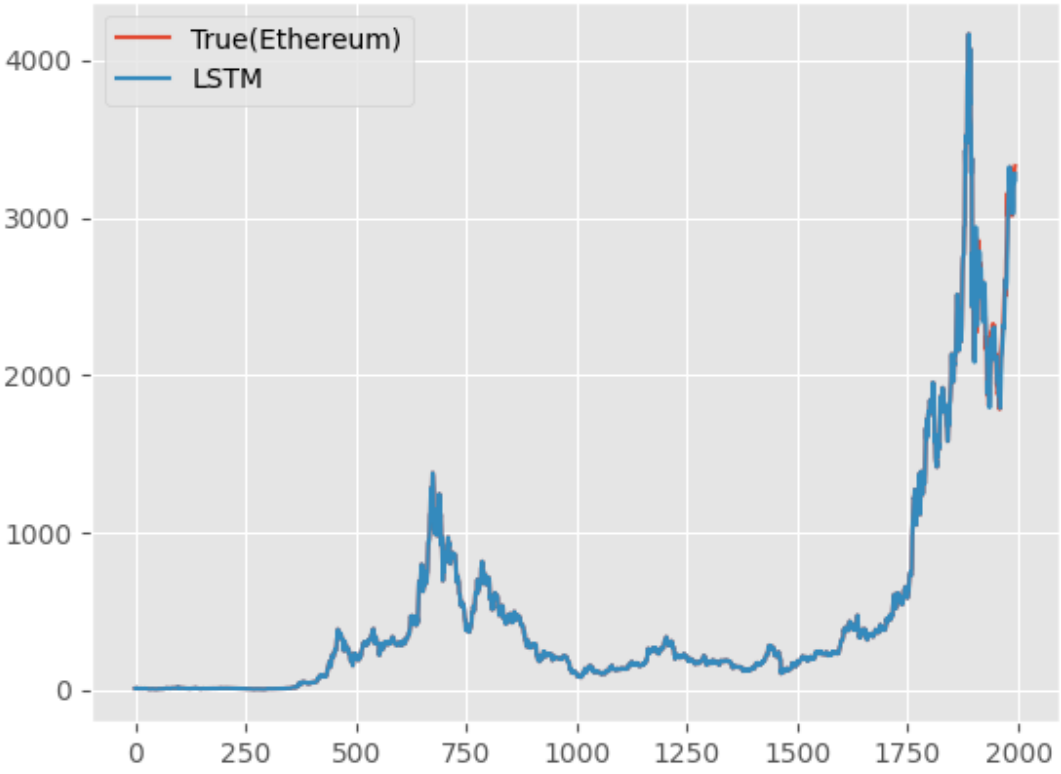


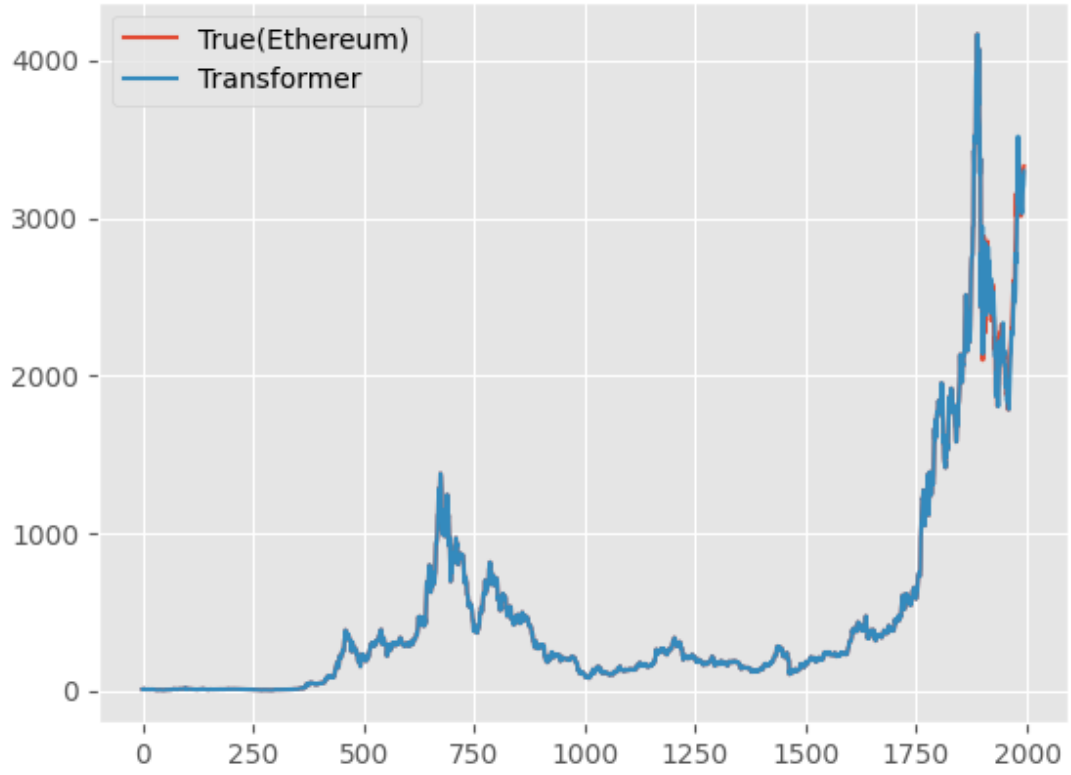


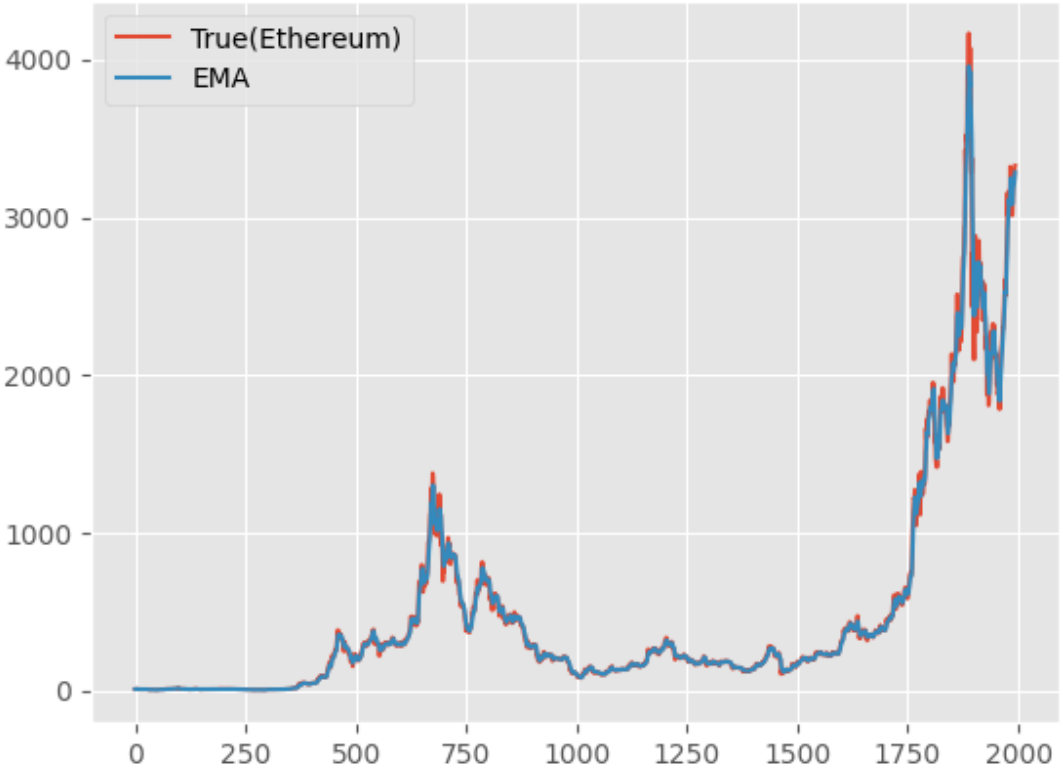
All of the models have achieved good values.

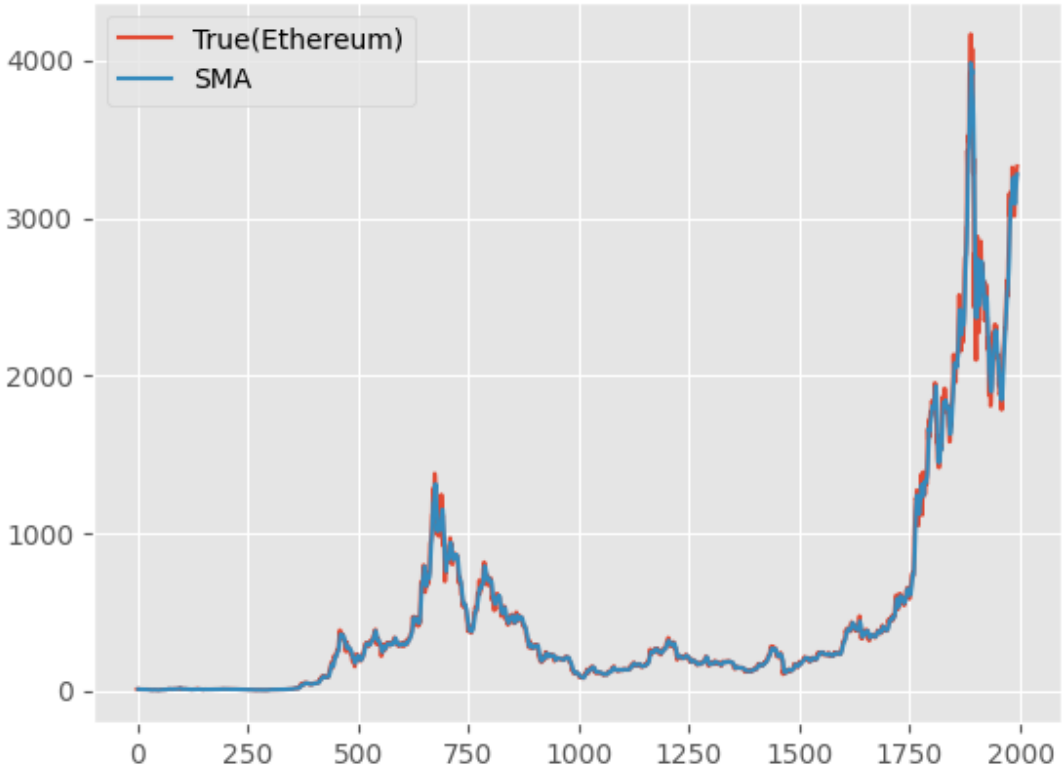
- Sample Results (Ethereum)





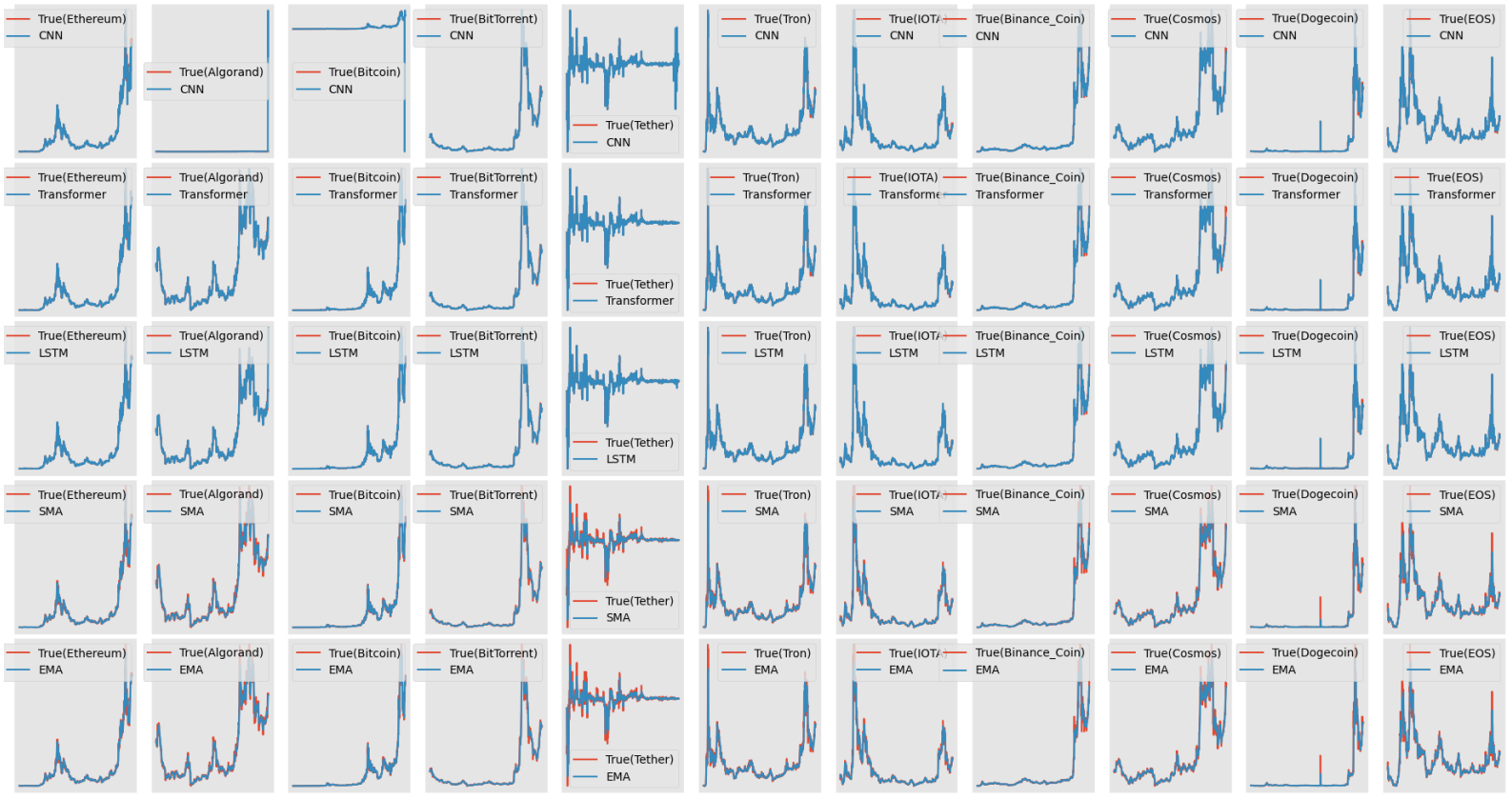

True(Ethereum)
Transformer
4000
3000
2000
1000
0
0
250
500
750
1000
1250
1500
1750
2000


True(Ethereum)
EMA
4000
3000
2000
1000
0
0
250
500
750
1000
1250
1500
1750
2000



The transformer and LSTM models have achieved the best performance.

- All Results



As depicted above, Deep Learning methods have performed well on all tasks and usually the transformer is the best among the methods and LSTM and CNN models are inferior in decreasing order.

If you zoom in for the Algorand CNN plot, you find it diverged, which shows that this sort of architecture may become susceptible to outliers and hyper-parameter tuning compared to other architectures in the task of series forecasting.

- Table of Results

Test MAE Metric

Coin	CNN	LSTM	Transformer	EMA	SMA
Ethereum	0.69	0.05	0.054	15.95	19.41
Algorand	32.77	0.107	0.068	0.2	0.025
Bitcoin	0.109	0.037	0.039	133.019	163.219
BitTorrent	0.064	0.05	0.097	0	0
Tether	0.006	0.001	0.00	0.001	0.002
IOTA	0.066	0.053	0.052	0.033	0.04
Binance Coin	0.043	0.042	0.04	2.42	3.03
Cosmos	0.08	0.067	0.077	0.288	0.36
Dogecoin	0.078	0.058	0.77	0.002	0.002
EOS	0.057	0.046	0.048	0.179	0.22

- Conclusion:

  First of all, we should emphasize that no paper was found that has used it explicitly and codes on Kaggle weren't focused on predicting future prices and weren't representable therefore, our conclusion is only based on the result that we ourselves could obtain.

  1. When local short-term violation in the dataset is not catastrophic, simple models that may not be on Deep Learning's basis can achieve very good results. It is based on the observation that statistical methods like SMA

and EMA that do not contain any learning phase, could perform well on some tasks.

2. Transformers and LSTMs were performing equally on this dataset but the Transformer is more preferable because it can be executed in parallel and further, it has much fewer parameters (6K) compared to LSTM (200K). So, it is less prone to overfitting.

3. We showed the applicability of CNN architectures in series-forecasting problems. Thus, they can be a very appealing option because of their desirable characteristics (like sparsity of connections) and also, because there is rapid and wild-spread research on improving them.