# financial_analyzer

May 2, 2024

# 1 CNIT 484 Final Project

## 1.1 Financial Analysis

**Libraries and GPU testing** Add more as needed. Make sure to add to requirements.txt as well.

[ ]:
```
%pip install -r requirements.txt
```

[90]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import torch
import torch.nn as nn
import math
from sklearn.model_selection import train_test_split
```

[91]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(torch.version.cuda)
print(torch.cuda.is_available())
print("Num GPUs Available: ", torch.cuda.device_count())
print(device)
```

```
12.1
True
Num GPUs Available:  1
cuda
```

## 1.2 Loading Data

[92]:
```
raw_data = pd.read_csv("../Datasets/1_ETHUSDT_1.1.2018-1.2.2024_1hour.csv",␣
  ↪header=None)
print(raw_data.head())
print(raw_data.shape)
```

```
                     0       1       2       3       4           5  \
0  2017-12-31 19:00:00  733.01  734.52  720.03  727.62  2105.90100
1  2017-12-31 20:00:00  727.01  732.00  716.80  717.97  2305.97086
```

```
2  2017-12-31 21:00:00  717.67  725.75  717.59  724.05  2166.45725
3  2017-12-31 22:00:00  723.95  737.99  722.70  734.50  2160.90450
4  2017-12-31 23:00:00  734.99  744.98  730.01  744.82  2335.33705

               6             7     8           9              10  11
0  1514768399999  1.528559e+06  3114  1275.23271  925445.068280   0
1  1514771999999  1.675753e+06  2875  1035.33513  753211.787422   0
2  1514775599999  1.564151e+06  2957  1179.82843  851942.067625   0
3  1514779199999  1.582200e+06  3647  1095.63271  801470.072777   0
4  1514782799999  1.724788e+06  3512  1313.03081  970430.311743   0
(53207, 12)
```

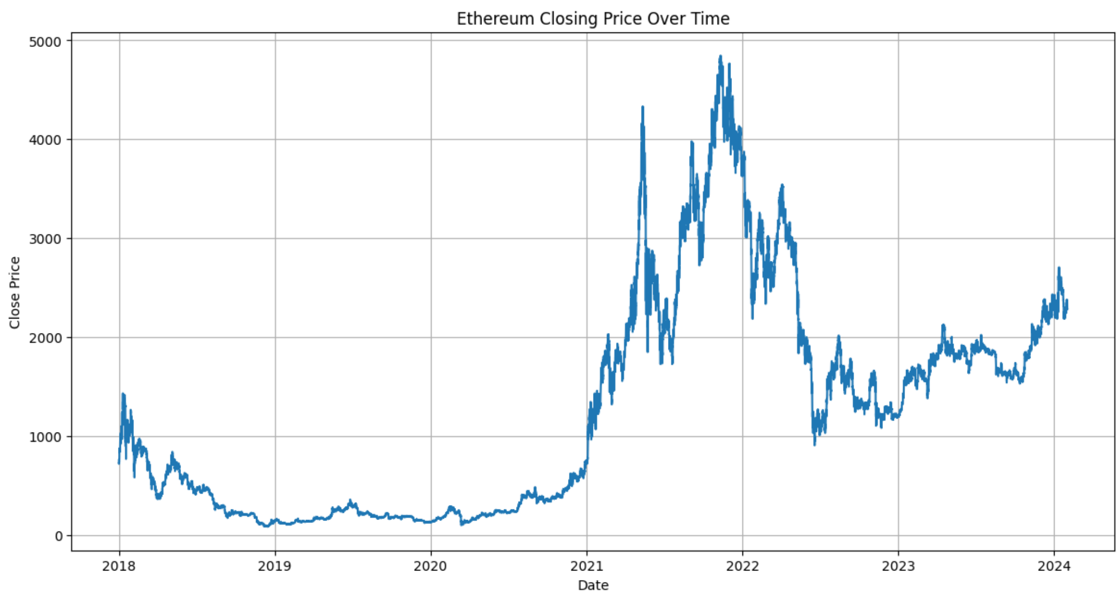
### 1.3 Preprocessing

**Data Modification**

[93]:
```
#Label the important columns and drop the rest
raw_data.columns = ['date', 'open', 'high', 'low', 'close', 'volume'] +␣
 ↪list(raw_data.columns[6:])
raw_data.drop(raw_data.columns[6:], axis=1, inplace=True)

# Convert the 'date' column to datetime and make it the index
raw_data['date'] = pd.to_datetime(raw_data['date'])
raw_data.set_index('date', inplace=True)

# create next close column which will be used for prediction
raw_data['next_close'] = raw_data['close'].shift(-1)
#drop last value since itll be NaN
cleaned_data = raw_data.dropna().copy()

# Calculate the 10-day simple moving average
cleaned_data['sma_10'] = cleaned_data['close'].rolling(window=10).mean()

# Calculate the 10-day exponential moving average
cleaned_data['ema_10'] = cleaned_data['close'].ewm(span=10, adjust=False).mean()
cleaned_data.dropna(inplace=True)

print(cleaned_data.head())
print(cleaned_data.shape)

plt.figure(figsize=(14, 7))
plt.plot(raw_data['close'])
plt.title('Ethereum Closing Price Over Time')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.grid(True)
plt.show()
```

```
                       open    high     low   close      volume  next_close  \
```

```
date
2018-01-01 04:00:00  744.08  755.00  744.08  753.21  1817.60549      757.10
2018-01-01 05:00:00  754.22  759.00  750.00  757.10  1619.87095      741.01
2018-01-01 06:00:00  757.07  758.00  730.58  741.01  3117.42838      745.00
2018-01-01 07:00:00  741.01  748.80  737.74  745.00  2263.78790      738.93
2018-01-01 08:00:00  746.52  747.06  729.79  738.93  2714.39907      742.74

                      sma_10      ema_10
date
2018-01-01 04:00:00  741.033  744.017304
2018-01-01 05:00:00  743.981  746.395976
2018-01-01 06:00:00  746.285  745.416708
2018-01-01 07:00:00  748.380  745.340943
2018-01-01 08:00:00  748.823  744.175317
(53197, 8)
```




**Sequencing**

[94]:
```
# takes sequences of inputs and targets, and a sequence length as input.
# Each sample it returns consists of a sequence of inputs and a sequence of
 ↪targets.
class TimeSeriesDataset(torch.utils.data.Dataset):
    def __init__(self, sequences, targets, seq_length):
        self.sequences = sequences
        self.targets = targets
        self.seq_length = seq_length

    def __len__(self):
```

```
        return self.sequences.shape[0] - self.seq_length - 1  # Adjusted for 2␣
↪future 'close' values

    def __getitem__(self, index):
        sequence = self.sequences[index:index+self.seq_length] # get the␣
↪sequence which is the current index to index + seq_length
        target = self.targets[index+(self.seq_length-1):index+(self.
↪seq_length-1)+2] # the targets are the next 2 values (index+(self.
↪seq_length-1)+10) after the last index (index+(self.seq_length-1))
        return sequence, target
```

### 1.3.1 Define Model

Sequence to vector model

[95]:
```
class PositionalEncoding(nn.Module): # Positional encoding
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.
↪log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        if d_model % 2 == 0:  # If d_model is even
            pe[:, 1::2] = torch.cos(position * div_term)
        else:  # If d_model is odd
            pe[:, 1::2] = torch.cos(position * div_term)[:,:-1]  # Exclude the␣
↪last column
        pe = pe.unsqueeze(0).transpose(0, 1)
        self.register_buffer('pe', pe)

    def forward(self, x) -> torch.Tensor:
        x = x + self.pe[:x.size(0), :]
        return self.dropout(x)

class TransformerModel(nn.Module):
    def __init__(self, ninp, nhead, nhid, nlayers, n_output, dropout=0.2 ):
        super(TransformerModel, self).__init__()
        self.model_type = 'Transformer'
        self.pos_encoder = PositionalEncoding(ninp, dropout, max_len=1000)
        self.encoder_layer = nn.TransformerEncoderLayer(d_model=ninp,␣
↪nhead=nhead, dim_feedforward=nhid, dropout=dropout)
        self.transformer_encoder = nn.TransformerEncoder(self.encoder_layer,␣
↪num_layers=nlayers)
```

```
        self.decoder = nn.Linear(ninp, ninp)
        self.final_layer = nn.Linear(ninp, n_output) # 2 outputs
        self.init_weights() # Initialize the weights

    def init_weights(self): # Initialize the weights
        initrange = 0.1
        self.decoder.bias.data.zero_()
        self.decoder.weight.data.uniform_(-initrange, initrange)
        self.final_layer.bias.data.zero_()
        self.final_layer.weight.data.uniform_(-initrange, initrange)

    def generate_look_ahead_mask(self, size): # masking
        mask = torch.triu(torch.ones(size, size), diagonal=1)
        return mask.masked_fill(mask==1, float('-inf'))

    def forward(self, src):
        src = self.pos_encoder(src) # Positional encoding (only 1 layer)
        mask = self.generate_look_ahead_mask(src.size(0)).to(src.device) #␣
↪masking
        output = self.transformer_encoder(src, mask=mask)
        output = self.decoder(output)
        output = self.final_layer(output[:,-1])
        return output
```

**Hyperparameters**

[96]:
```
ninp = 7  # Number of expected features in the input
nhead = 7  # Number of heads in the multiheadattention models
nhid = 200  # Dimension of hidden layers
nlayers = 3 # Number of encoder layers
dropout = 0.2  # Dropout
noutput = 2  # Number of outputs
```

[97]:
```
model = TransformerModel(ninp, nhead, nhid, nlayers, noutput, dropout ).
 ↪to(device)
criterion = torch.nn.MSELoss() # loss function
optimizer = torch.optim.Adam(model.parameters(), lr=0.001) # optimizer and␣
 ↪learning rate
```

```
c:\Users\drejc\OneDrive - purdue.edu\CNIT484\CNIT484\.conda\lib\site-
packages\torch\nn\modules\transformer.py:306: UserWarning: enable_nested_tensor
is True, but self.use_nested_tensor is False because
encoder_layer.self_attn.batch_first was not True(use batch_first for better
inference performance)
  warnings.warn(f"enable_nested_tensor is True, but self.use_nested_tensor is
False because {why_not_sparsity_fast_path}")
```

### 1.3.2 Data Prep for Transformer

**Loading and Splitting Data**

[98]:
```
features = cleaned_data[['open', 'high', 'low', 'close', 'volume', 'sma_10',␣
    ↪'ema_10']] #inputs
targets = cleaned_data['next_close'] # targets

# Define sequence length
seq_length = 12 # set the number of hours/steps for each sequence
```

[99]:
```
train_size = int(len(features) * 0.7)
val_size = int(len(features) * 0.15)
test_size = len(features) - train_size - val_size

# Split features
features_train = features[:train_size].values
features_val = features[train_size:train_size+val_size].values
features_test = features[train_size+val_size:].values

# Split targets and reshape to 2D array
targets_train = targets[:train_size].values.reshape(-1, 1)
targets_val = targets[train_size:train_size+val_size].values.reshape(-1, 1)
targets_test = targets[train_size+val_size:].values.reshape(-1, 1)
feature_scaler = MinMaxScaler()

# Scale the features
features_train = feature_scaler.fit_transform(features_train)
features_val = feature_scaler.transform(features_val)
features_test = feature_scaler.transform(features_test)

# Scale the targets
targets_scaler = MinMaxScaler()
targets_train = targets_scaler.fit_transform(targets_train)
targets_val = targets_scaler.transform(targets_val)
targets_test = targets_scaler.transform(targets_test)

# Create datasets
train_dataset = TimeSeriesDataset(torch.FloatTensor(features_train), torch.
    ↪FloatTensor(targets_train), seq_length)
val_dataset = TimeSeriesDataset(torch.FloatTensor(features_val), torch.
    ↪FloatTensor(targets_val), seq_length)
test_dataset = TimeSeriesDataset(torch.FloatTensor(features_test), torch.
    ↪FloatTensor(targets_test), seq_length)
```

[100]:
```
print(f"Shape of sequences: {train_dataset[0][0].shape}")
print(f"Number of sequences in train dataset: {len(train_dataset)}")
print(f"Number of sequences in test dataset: {len(test_dataset)}")
```

```
print(f"Number of sequences in val dataset: {len(val_dataset)}")
```

```
Shape of sequences: torch.Size([12, 7])
Number of sequences in train dataset: 37224
Number of sequences in test dataset: 7968
Number of sequences in val dataset: 7966
```

[101]:
```
print(f"Example sequence:\n {train_dataset[0][0]}\n")
print(f"Example target:\n {train_dataset[0][1]}")
```

```
Example sequence:
 tensor([[0.1389, 0.1404, 0.1394, 0.1408, 0.0037, 0.1394, 0.1401],
        [0.1410, 0.1413, 0.1406, 0.1417, 0.0033, 0.1400, 0.1406],
        [0.1416, 0.1411, 0.1365, 0.1383, 0.0063, 0.1405, 0.1404],
        [0.1383, 0.1392, 0.1381, 0.1391, 0.0046, 0.1409, 0.1404],
        [0.1394, 0.1388, 0.1364, 0.1378, 0.0055, 0.1410, 0.1401],
        [0.1379, 0.1383, 0.1352, 0.1386, 0.0076, 0.1410, 0.1401],
        [0.1386, 0.1399, 0.1382, 0.1399, 0.0051, 0.1410, 0.1403],
        [0.1398, 0.1392, 0.1371, 0.1379, 0.0038, 0.1405, 0.1400],
        [0.1379, 0.1388, 0.1376, 0.1374, 0.0046, 0.1402, 0.1398],
        [0.1374, 0.1381, 0.1373, 0.1387, 0.0033, 0.1401, 0.1398],
        [0.1387, 0.1394, 0.1389, 0.1394, 0.0040, 0.1400, 0.1399],
        [0.1394, 0.1411, 0.1395, 0.1417, 0.0040, 0.1400, 0.1405]])

Example target:
 tensor([[0.1421],
        [0.1411]])
```

**Dataloaders**

[102]:
```
from torch.utils.data import DataLoader

batch_size = 8  # batch size (8 produced the best outcome we found)

train_dataloader = DataLoader(train_dataset, batch_size=batch_size,␣
 ↪shuffle=True)
val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

### 1.3.3 Training

[103]:
```
def train(model, train_loader, criterion, optimizer, device):
    model.train()  # training mode
    total_loss = 0

    for i, (sequence, target) in enumerate(train_loader):
        sequence, target = sequence.to(device), target.to(device)  # Move␣
 ↪sequence and target to the GPU
        optimizer.zero_grad()  # Clear gradients
```

```
        output = model(sequence)
        target = target.squeeze(-1)
        loss = criterion(output, target)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.5)
        optimizer.step()  # Update the weights
        total_loss += loss.item()
    return total_loss / len(train_loader)

def validate(model, val_loader, criterion, device):
    model.eval()  # evaluation mode
    total_loss = 0

    with torch.no_grad():  # no need to calculate gradients
        for i, (sequence, target) in enumerate(val_loader):
            sequence, target = sequence.to(device), target.to(device)
            output = model(sequence)
            target = target.squeeze(-1)
            loss = criterion(output, target)
            total_loss += loss.item()
    return total_loss / len(val_loader)
```

[104]:
```
import os
from tempfile import TemporaryDirectory

best_val_loss = float('inf')
epochs = 5 # Number of epochs

with TemporaryDirectory() as tempdir:
    best_model_params_path= os.path.join(tempdir, "best_model_params.pt")

    for epoch in range(epochs):  # Number of epochs
        train_loss = train(model, train_dataloader, criterion, optimizer,␣
↪device)
        val_loss = validate(model, val_dataloader, criterion, device)
        print(f"Epoch: {epoch}, Train Loss: {train_loss}, Validation Loss:␣
↪{val_loss}")

        # Checkpointing to find best model
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            print("new best model found, saving...")
            torch.save(model.state_dict(), best_model_params_path)

    # Load the best model parameters after training
    model.load_state_dict(torch.load(best_model_params_path))
```

```
Epoch: 0, Train Loss: 0.01990302643550191, Validation Loss: 0.008817339685886265
```

```
new best model found, saving…
Epoch: 1, Train Loss: 0.010422340971165154, Validation Loss:
0.0077916644501334325
new best model found, saving…
Epoch: 2, Train Loss: 0.008580279465587098, Validation Loss:
0.005328899700105532
new best model found, saving…
Epoch: 3, Train Loss: 0.007776351408154659, Validation Loss:
0.005794815292622124
Epoch: 4, Train Loss: 0.007460626744989057, Validation Loss:
0.008111444525149497
```

### 1.3.4 Evaluation

[105]:

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
def evaluate_and_get_predictions(model, test_loader, criterion, device):
    model.eval()  # evaluation mode
    total_loss = 0
    all_outputs = []
    all_targets = []
    with torch.no_grad():
        for i, (sequence, target) in enumerate(test_loader):
            sequence, target = sequence.to(device), target.to(device)  # Move␣
 ↪sequence and target to the GPU
            output = model(sequence)
            target = target.squeeze(-1)
            loss = criterion(output, target)
            total_loss += loss.item()
            all_outputs.extend(output.cpu().numpy())
            all_targets.extend(target.cpu().numpy())
    return total_loss / len(test_loader), np.array(all_targets), np.
 ↪array(all_outputs)

test_loss, y_true, y_pred = evaluate_and_get_predictions(model,␣
 ↪test_dataloader, criterion, device)
# Generate 5 random indices from the range of y_true's length
random_indices = np.random.choice(range(len(y_true)), size=5, replace=False)

# Use these indices to select values from y_true and y_pred
random_y_true = y_true[random_indices]
random_y_pred = y_pred[random_indices]

print('=' * 89)
print(f'| End of training | test loss (MSE) {test_loss} | ')
print('=' * 89)
print('=' * 89)
print(f'5 Actual: {random_y_true}')
```

```python
print('=' * 89)
print(f'5 Predicted: {random_y_pred}')
print('=' * 89)
```

```
================================================================================
=========
| End of training | test loss (MSE) 0.0016879842776279658 |
================================================================================
=========
================================================================================
=========
5 Actual: [[0.33109176 0.33039075]
 [0.31411007 0.3142276 ]
 [0.49217135 0.49280727]
 [0.36599967 0.36619484]
 [0.3848913  0.39062113]]
================================================================================
=========
5 Predicted: [[0.34672743 0.34710562]
 [0.30741027 0.30787024]
 [0.43855017 0.44047818]
 [0.36653107 0.3670891 ]
 [0.38521188 0.38603643]]
================================================================================
=========
```

### 1.3.5 Conclusion

The goal of this project was to try to see if a transformer would be able to perform as good as or better (or worse) than a LSTM which is usually the standard for time-series data. Instead of going for the conventional single step prediction for stock data, we instead took it one step further and tried to implement a multi-step (2-step) prediction of close price.

Due to this being time series data, its important to keep the order of data the same and also for the training data to come BEFORE the validation and testing data (to prevent look-ahead bias). With this, as you can see with the visualization of the data set in the beginning, the dataset was not particularly friendly for the train, test, and validation split. The data the model was trained on (low close prices and not many fluctuations) differed drastically from the data the test and validation sets were comprised of (lots of fluctuations). This made predicting on testing data particularly difficult for the transformer.

Regardless, we were still able to get a decent loss and predictions from the model. As shown in the comparision between the actual and predicted values, while our model is not spot on, its in the ball park.

Lastly, it's important to note that stock price prediction is an extremely challenging problem due to the stochastic nature of financial markets. Factors outside the scope of historical price data can have significant impacts on future prices. After all, transformers were reliable and accurate at doing this, these models would be EXTREMELY valuable. However, due to these conditions that weren't captured in our data, our model did not perform the best. However, for future developments, it likely would be crucial to implement other contexts and/or more data for the model to be trained on.

### 1.3.6 References


Aslan, K. (2024, January 29). Time series forecasting with a basic transformer model in pytorch. Medium. https://medium.com/@mkaanaslan99/time-series-forecasting-with-a-basic-transformer-model-in-pytorch-650f116a1018

Intel. (2024, March 11). How to apply transformers to time series models. Medium. https://medium.com/intel-tech/how-to-apply-transformers-to-time-series-models-spacetimeformer-e452f2825d2e

Ludvigsen, K. G. A. (2022, October 27). How to make a pytorch transformer for time series forecasting. Medium. https://towardsdatascience.com/how-to-make-a-pytorch-transformer-for-time-series-forecasting-69e073d4061e

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … & Polosukhin, I. (2017). Attention is all you need. Advances in neural information processing systems, 30.