Ether Price Prediction - Deep Learning With Pytorch

This notebook will look at predicting Ether prices using GRU.

The following topics are covered in the notebook:

- Problem Statement.
- Exploratory Data Analysis.
- Feature Engineering.
- Modelling GRU.
- Best Model.
- Summary & Future References.

Downloading & Importing The Required Libraries.

In [2]:
```
!pip install numpy pandas matplotlib seaborn plotly --quiet
```

In [3]:
```
!pip install jovian opendatasets torch scikit-learn --upgrade --quiet
```

```
     |████████████████████████████████| 22.3 MB 1.6 MB/s
```

In [31]:
```
import opendatasets as od
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import plotly.express as px
import matplotlib
import jovian
import os
import torch
import torch.nn as nn
import torch.functional as F
import time
import math
from torchvision.datasets.utils import download_url
from torch.utils.data import DataLoader, TensorDataset, random_split
%matplotlib inline

pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 150)
sns.set_style('darkgrid')
matplotlib.rcParams['font.size'] = 14
matplotlib.rcParams['figure.figsize'] = (10, 6)
matplotlib.rcParams['figure.facecolor'] = '#00000000'
```

Downloading The Required Dataset

Source: https://www.kaggle.com/prasoonkottarathil/ethereum-historical-dataset?select=ETH_day.csv

In [5]:
```
od.download('https://www.kaggle.com/prasoonkottarathil/ethereum-historical-dataset?select=ETH_day.csv')
```

```
Please provide your Kaggle credentials to download this dataset. Learn more: http://bit.ly/kaggle-creds
Your Kaggle username: prachin
Your Kaggle Key: ··········
 42%|████▏     | 9.00M/21.4M [00:00<00:00, 57.2MB/s]
Downloading ethereum-historical-dataset.zip to ./ethereum-historical-dataset
100%|██████████| 21.4M/21.4M [00:00<00:00, 101MB/s]
```

In [6]:
```
os.listdir('ethereum-historical-dataset')
```

Out[6]:
```
['ETH_1min.csv', 'ETH_1H.csv', 'ETH_day.csv']
```

Problem Statement

We are trying to predict the trading prices of Ethereum using Gated Recurrent Units.

The dataset contains historical data from 2016-05-09 to 2016-05-09 of open, high, close, low, and volume of Ether, thereby the data is in a tabular form.

Hence this is a regression problem.

Now let's read and display the dataset using pandas.

In [7]:
```
raw_df = pd.read_csv('ethereum-historical-dataset/ETH_day.csv')
```

In [8]:
```
raw_df
```

Out[8]:

	Date	Symbol	Open	High	Low	Close	Volume ETH	Volume USD
0	2020-04-15	ETHUSD	158.61	158.61	158.61	158.61	0.00	0.00
1	2020-04-14	ETHUSD	156.97	162.15	155.74	158.61	18061.58	2872210.44
2	2020-04-13	ETHUSD	158.56	159.51	150.12	156.97	15698.32	2416728.28
3	2020-04-12	ETHUSD	158.66	165.37	155.21	158.56	12877.33	2082804.05
4	2020-04-11	ETHUSD	158.26	161.49	154.25	158.66	13761.72	2172914.57
...	...	...	...	...	...	...	...	...
1433	2016-05-13	ETHUSD	10.20	11.59	10.20	10.69	1769.71	18923.55
1434	2016-05-12	ETHUSD	10.43	12.00	9.92	10.20	2072.56	22183.39
1435	2016-05-11	ETHUSD	9.68	10.47	9.68	10.43	3052.51	30978.11
1436	2016-05-10	ETHUSD	9.98	9.98	9.36	9.68	672.06	6578.20
1437	2016-05-09	ETHUSD	12.00	12.00	9.36	9.98	1317.90	12885.06

1438 rows × 8 columns

General Statistics About The Dataset

In [9]:
```
raw_df.shape
```

Out[9]:
```
(1438, 8)
```

In [10]:
```
raw_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1438 entries, 0 to 1437
Data columns (total 8 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   Date        1438 non-null   object 
 1   Symbol      1438 non-null   object 
 2   Open        1438 non-null   float64
 3   High        1438 non-null   float64
 4   Low         1438 non-null   float64
 5   Close       1438 non-null   float64
 6   Volume ETH  1438 non-null   float64
 7   Volume USD  1438 non-null   float64
dtypes: float64(6), object(2)
memory usage: 90.0+ KB
```

In [11]:
```
raw_df.describe()
```

Out[11]:

	Open	High	Low	Close	Volume ETH	Volume USD
count	1438.000000	1438.000000	1438.000000	1438.000000	1.438000e+03	1.438000e+03
mean	239.397149	248.919200	227.681446	239.468011	3.720638e+04	1.139557e+07
std	237.662224	248.677428	222.794938	237.606382	6.908336e+04	2.143780e+07
min	6.770000	7.290000	5.990000	6.770000	0.000000e+00	0.000000e+00
25%	79.782500	84.875000	74.677500	80.732500	7.020215e+03	7.541171e+05
50%	181.430000	187.020000	175.850000	181.430000	1.780439e+04	3.221372e+06
75%	297.735000	306.015000	287.427500	297.502500	4.204451e+04	1.204918e+07
max	1381.850000	1420.010000	1270.000000	1381.850000	1.827755e+06	2.221193e+08

Data Preparation & Cleaning

In [12]:
```
# Sort Date In Ascending Order
raw_df.sort_values(by=['Date'], inplace=True)
# Setting Date As An Index
raw_df.set_index('Date', inplace=True)
# Dropping Columns: Symbol, Volume ETH
raw_df.drop(['Symbol', 'Volume ETH'], axis=1, inplace=True)
```

In [13]:
```
# Rename Column: Volume USD
raw_df.rename({'Volume USD':'Volume_USD'}, axis=1, inplace=True)
raw_df
```

Out[13]:

	Open	High	Low	Close	Volume_USD
Date					
2016-05-09	12.00	12.00	9.36	9.98	12885.06
2016-05-10	9.98	9.98	9.36	9.68	6578.20
2016-05-11	9.68	10.47	9.68	10.43	30978.11
2016-05-12	10.43	12.00	9.92	10.20	22183.39
2016-05-13	10.20	11.59	10.20	10.69	18923.55
...	...	...	...	...	...
2020-04-11	158.26	161.49	154.25	158.66	2172914.57
2020-04-12	158.66	165.37	155.21	158.56	2082804.05
2020-04-13	158.56	159.51	150.12	156.97	2416728.28
2020-04-14	156.97	162.15	155.74	158.61	2872210.44
2020-04-15	158.61	158.61	158.61	158.61	0.00

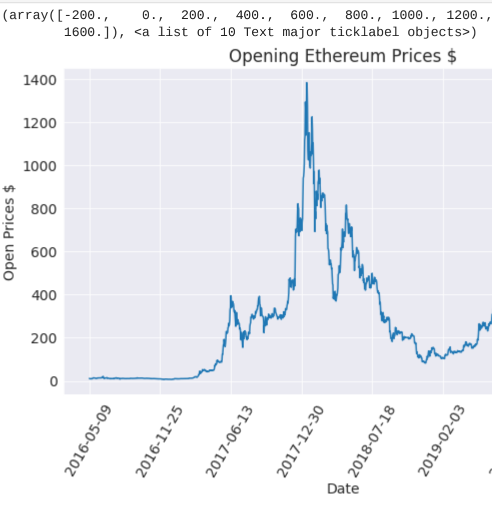
1438 rows × 5 columns

Exploratory Data Analysis

Open Prices

In [14]:
```
raw_df.Open.plot()
plt.xlabel('Date')
plt.ylabel('Open Prices $')
plt.title('Opening Ethereum Prices $')
plt.xticks(rotation=60)
```

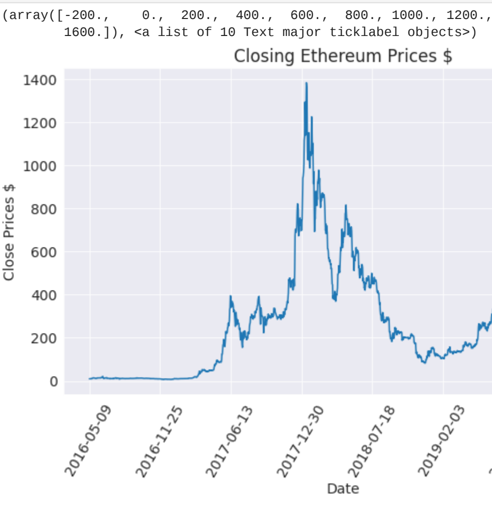
Out[14]:
```
(array([-200.,    0.,  200.,  400.,  600.,  800., 1000., 1200., 1400.,
        1600.]), <a list of 10 Text major ticklabel objects>)
```



Closing Prices

In [15]:
```
raw_df.Close.plot()
plt.xlabel('Date')
plt.ylabel('Close Prices $')
plt.title('Closing Ethereum Prices $')
plt.xticks(rotation=60)
```

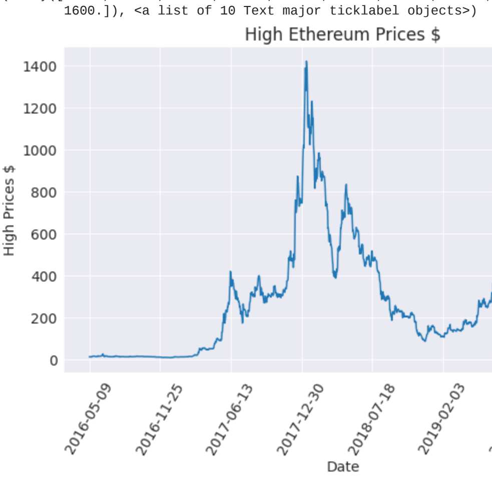
Out[15]:
```
(array([-200.,    0.,  200.,  400.,  600.,  800., 1000., 1200., 1400.,
        1600.]), <a list of 10 Text major ticklabel objects>)
```



Highs Made

In [16]:
```
raw_df.High.plot()
plt.xlabel('Date')
plt.ylabel('High Prices $')
plt.title('High Ethereum Prices $')
plt.xticks(rotation=60)
```

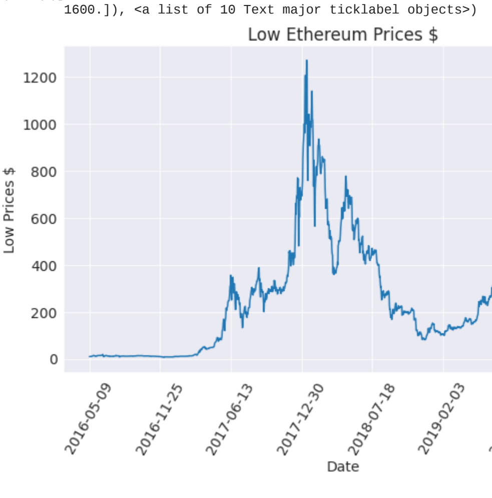
Out[16]:
```
(array([-200.,    0.,  200.,  400.,  600.,  800., 1000., 1200., 1400.,
        1600.]), <a list of 10 Text major ticklabel objects>)
```



Lows Made

In [17]:
```
raw_df.Low.plot()
plt.xlabel('Date')
plt.ylabel('Low Prices $')
plt.title('Low Ethereum Prices $')
plt.xticks(rotation=60)
```

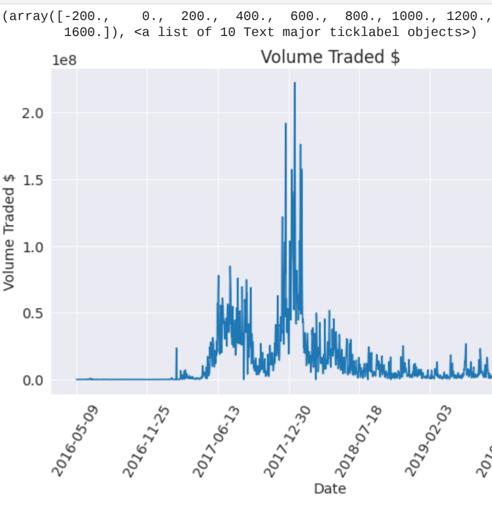
Out[17]:
```
(array([-200.,    0.,  200.,  400.,  600.,  800., 1000., 1200., 1400.,
        1600.]), <a list of 10 Text major ticklabel objects>)
```



Trading Volume

In [18]:
```
raw_df.Volume_USD.plot()
plt.xlabel('Date')
plt.ylabel('Volume Traded $')
plt.title('Volume Traded $')
plt.xticks(rotation=60)
```

Out[18]:
```
(array([-200.,    0.,  200.,  400.,  600.,  800., 1000., 1200., 1400.,
        1600.]), <a list of 10 Text major ticklabel objects>)
```



Preprocessing & Feature Engineering

Target Column

We will be predicting the closing price.

In [19]:
```
price = raw_df[['Close']]
price.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1438 entries, 2016-05-09 to 2020-04-15
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Close   1438 non-null   float64
dtypes: float64(1)
memory usage: 22.5+ KB
```

Scaling Numeric Data

In [20]:
```
from sklearn.preprocessing import MinMaxScaler
```

In [21]:
```
scaler = MinMaxScaler(feature_range=(-1, 1))
price['Close'] = scaler.fit_transform(price['Close'].values.reshape(-1,1))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

Now let's create a helper function to use the sliding window method.

Source:https://www.mathworks.com/help/dsp/ug/sliding-window-method-and-exponential-weighting-method.html

In [47]:
```
def split_data(crypto, lookback):
    data_raw = crypto.to_numpy()
    data = []

    # create all possible sequences of length seq_len
    for index in range(len(data_raw) - lookback):
        data.append(data_raw[index: index + lookback])

    data = np.array(data);
    test_df_size = int(np.round(0.2*data.shape[0]));
    train_df_size = data.shape[0] - (test_df_size);

    x_train = data[:train_df_size,:-1,:]
    y_train = data[:train_df_size,-1,:]

    x_test = data[train_df_size:,:-1]
    y_test = data[train_df_size:,-1,:]

    return [x_train, y_train, x_test, y_test]
```

Now let's split the raw data using our helper function.

In [48]:
```
lookback = 20
x_train, y_train, x_test, y_test = split_data(price, lookback)
print('x_train.shape = ',x_train.shape)
print('y_train.shape = ',y_train.shape)
print('x_test.shape = ',x_test.shape)
print('y_test.shape = ',y_test.shape)
```

```
x_train.shape =  (1134, 19, 1)
y_train.shape =  (1134, 1)
x_test.shape =  (284, 19, 1)
y_test.shape =  (284, 1)
```

Converting Numpy Arrays To Pytorch Tensors

In [49]:
```
x_train = torch.from_numpy(x_train).type(torch.Tensor)
x_test = torch.from_numpy(x_test).type(torch.Tensor)
y_train_gru = torch.from_numpy(y_train).type(torch.Tensor)
y_test_gru = torch.from_numpy(y_test).type(torch.Tensor)
```

Modelling

Let's set our model's layer's configuration

In [50]:
```
input_dim = 1
hidden_dim = 32
num_layers = 2
output_dim = 1
num_epochs = 100
```

GRU Model

In [51]:
```
class GRU(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers, output_dim):
        super(GRU, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        self.gru = nn.GRU(input_dim, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        j0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).requires_grad_()
        out, (hn) = self.gru(x, (j0.detach()))
        out = self.fc(out[:, -1, :])
        return out
```

Training The Model & Setting The Loss Function & Optimizer

In [52]:
```
model = GRU(input_dim=input_dim, hidden_dim=hidden_dim, output_dim=output_dim, num_layers=num_layers)
loss_function = torch.nn.MSELoss(reduction='mean')
optimiser = torch.optim.Adam(model.parameters(), lr=0.01)
```

Calculating MSE Loss

In [53]:
```
hist = np.zeros(num_epochs)
start_time = time.time()
gru = []

for t in range(num_epochs):
    y_train_pred = model(x_train)

    loss = loss_function(y_train_pred, y_train_gru)
    print("Epoch ", t, "MSE: ", loss.item())
    hist[t] = loss.item()

    optimiser.zero_grad()
    loss.backward()
    optimiser.step()

training_time = time.time()-start_time
print("Training time: {}".format(training_time))
```

```
Epoch  0 MSE:  0.9470643401145935
Epoch  1 MSE:  0.5071138739585876
Epoch  2 MSE:  0.2145431935787201
Epoch  3 MSE:  0.10997916758060455
Epoch  4 MSE:  0.2370074838399887
Epoch  5 MSE:  0.15443529188632965
Epoch  6 MSE:  0.08482107520103455
Epoch  7 MSE:  0.09094755351543427
Epoch  8 MSE:  0.11518192291259766
Epoch  9 MSE:  0.12017963826656342
Epoch  10 MSE:  0.1024416983127594
Epoch  11 MSE:  0.07344551384449005
Epoch  12 MSE:  0.04892092943191528
Epoch  13 MSE:  0.041651539504528046
Epoch  14 MSE:  0.050839509814977646
Epoch  15 MSE:  0.05786249041557312
Epoch  16 MSE:  0.048092618584632874
Epoch  17 MSE:  0.02863214537501335
Epoch  18 MSE:  0.014693562872707844
Epoch  19 MSE:  0.01256976742297411
Epoch  20 MSE:  0.01739439181983471
Epoch  21 MSE:  0.020397761836647987
Epoch  22 MSE:  0.016743674874305725
Epoch  23 MSE:  0.00898042693734169
Epoch  24 MSE:  0.004547950346022844
Epoch  25 MSE:  0.008165067061781883
Epoch  26 MSE:  0.014892244711518288
Epoch  27 MSE:  0.015663621947169304
Epoch  28 MSE:  0.010000087320804596
Epoch  29 MSE:  0.004782294854521751
Epoch  30 MSE:  0.004173374269157648
Epoch  31 MSE:  0.006352081894874573
Epoch  32 MSE:  0.00758962519466877
Epoch  33 MSE:  0.006321861408650875
Epoch  34 MSE:  0.003756007645279169
Epoch  35 MSE:  0.002137680072337389
Epoch  36 MSE:  0.0026182602159678936
Epoch  37 MSE:  0.004280545748770237
Epoch  38 MSE:  0.005201115272939205
Epoch  39 MSE:  0.004519056063145399
Epoch  40 MSE:  0.0031327311880886555
Epoch  41 MSE:  0.002415720373392105
Epoch  42 MSE:  0.0027589222881942987
Epoch  43 MSE:  0.0034444311168044806
Epoch  44 MSE:  0.0035676215775310993
Epoch  45 MSE:  0.0029016537591814995
Epoch  46 MSE:  0.0020063261035829782
Epoch  47 MSE:  0.0016163018299266696
Epoch  48 MSE:  0.001893785665743053
Epoch  49 MSE:  0.002293765777722001
Epoch  50 MSE:  0.0022326018661260605
Epoch  51 MSE:  0.001769408816471696
Epoch  52 MSE:  0.0014465347630903125
Epoch  53 MSE:  0.001574853085912764
Epoch  54 MSE:  0.001911554834805429
Epoch  55 MSE:  0.0020269514061510563
Epoch  56 MSE:  0.0018115798011422157
Epoch  57 MSE:  0.0015434208326041698
Epoch  58 MSE:  0.0015104388585314155
Epoch  59 MSE:  0.001668385579250753
Epoch  60 MSE:  0.001740537816658616
Epoch  61 MSE:  0.0015983132179826498
Epoch  62 MSE:  0.0014019868103787303
Epoch  63 MSE:  0.0013533326564356685
Epoch  64 MSE:  0.001443977002054453
Epoch  65 MSE:  0.001509471912868321
Epoch  66 MSE:  0.001455593854188919
Epoch  67 MSE:  0.0013554643373936415
Epoch  68 MSE:  0.0013308922061696649
Epoch  69 MSE:  0.0013937480980530381
Epoch  70 MSE:  0.0014456716598942876
Epoch  71 MSE:  0.0014175110263749957
Epoch  72 MSE:  0.0013516563922166824
Epoch  73 MSE:  0.0013283869484439492
Epoch  74 MSE:  0.0013571379240602255
Epoch  75 MSE:  0.001378125511109829
Epoch  76 MSE:  0.0013514285674318671
Epoch  77 MSE:  0.0013055112212896347
Epoch  78 MSE:  0.0012904464965686202
Epoch  79 MSE:  0.0013095607282593846
Epoch  80 MSE:  0.0013222221750766039
Epoch  81 MSE:  0.0013054219307377934
Epoch  82 MSE:  0.001282042358070612
Epoch  83 MSE:  0.0012813141802325845
Epoch  84 MSE:  0.0012970312964171171
Epoch  85 MSE:  0.0013016782468184829
Epoch  86 MSE:  0.0012874362291768193
Epoch  87 MSE:  0.0012733417097479105
Epoch  88 MSE:  0.0012742906110361218
Epoch  89 MSE:  0.0012809280306100845
Epoch  90 MSE:  0.0012771609472110868
Epoch  91 MSE:  0.0012645606184378266
Epoch  92 MSE:  0.0012573382118716836
Epoch  93 MSE:  0.0012599758338183165
Epoch  94 MSE:  0.0012627043761312962
Epoch  95 MSE:  0.0012580540496855974
Epoch  96 MSE:  0.0012510546948760748
Epoch  97 MSE:  0.001249635242857039
Epoch  98 MSE:  0.0012524256017059088
Epoch  99 MSE:  0.0012521670432761312
Training time: 9.657639026641846
```

Now let's compare the original prices and the predicted prices and visualize it.

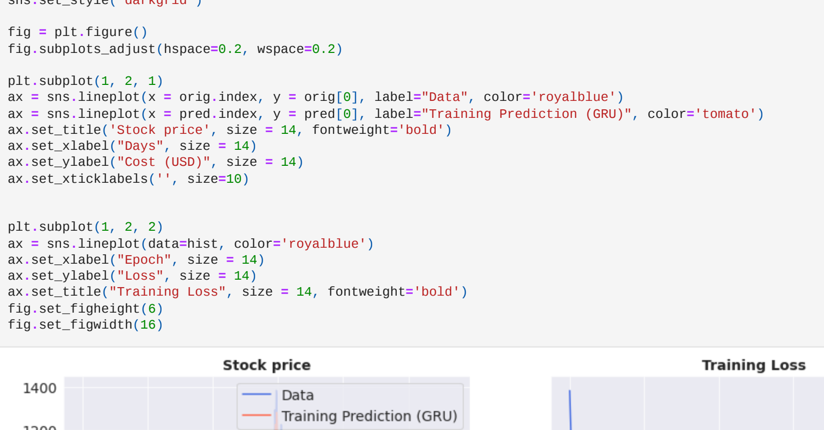
In [54]:
```
pred = pd.DataFrame(scaler.inverse_transform(y_train_pred.detach().numpy()))
orig = pd.DataFrame(scaler.inverse_transform(y_train_gru.detach().numpy()))
```

In [55]:
```
sns.set_style("darkgrid")

fig = plt.figure()
fig.subplots_adjust(hspace=0.2, wspace=0.2)

plt.subplot(1, 2, 1)
ax = sns.lineplot(x = orig.index, y = orig[0], label="Data", color='royalblue')
ax = sns.lineplot(x = pred.index, y = pred[0], label="Training Prediction (GRU)", color='tomato')
ax.set_title('Stock price', size = 14, fontweight='bold')
ax.set_xlabel("Days", size = 14)
ax.set_ylabel("Cost (USD)", size = 14)
ax.set_xticklabels('', size=10)


plt.subplot(1, 2, 2)
ax = sns.lineplot(data=hist, color='royalblue')
ax.set_xlabel("Epoch", size = 14)
ax.set_ylabel("Loss", size = 14)
ax.set_title("Training Loss", size = 14, fontweight='bold')
fig.set_figheight(6)
fig.set_figwidth(16)
```



Now let's predict using our test dataset & compare the training & test RMSE loss.

In [56]:
```
from sklearn.metrics import mean_squared_error

y_test_pred = model(x_test)

y_train_pred = scaler.inverse_transform(y_train_pred.detach().numpy())
y_train = scaler.inverse_transform(y_train_gru.detach().numpy())
y_test_pred = scaler.inverse_transform(y_test_pred.detach().numpy())
y_test = scaler.inverse_transform(y_test_gru.detach().numpy())

train_loss = math.sqrt(mean_squared_error(y_train[:,0], y_train_pred[:,0]))
print('Test Loss: %.2f RMSE' % (train_loss))
test_loss = math.sqrt(mean_squared_error(y_test[:,0], y_test_pred[:,0]))
print('Train Loss: %.2f RMSE' % (test_loss))
gru.append(train_loss)
gru.append(test_loss)
gru.append(training_time)
```

```
Test Loss: 24.33 RMSE
Train Loss: 11.29 RMSE
```

Save & Record Weights & Metrics

In [60]:
```
jovian.reset()
jovian.log_hyperparams(epochs=num_epochs,
                       lr=0.01
                       )
```

```
[jovian] Hyperparams logged.
```

In [61]:
```
jovian.log_metrics(train_loss=train_loss,
                   test_loss=test_loss)
```

```
[jovian] Metrics logged.
```

Summary & Future Refrences

- We used the sliding window method to implement GRU to predict Ether prices.
- We can also predict prices using LSTM (Long-Short Temporary Memory) as another method and compare the training and validation losses.
- Other recurrent nueral network methods c

In [35]:
```
# Execute this to save new versions of the notebook
jovian.commit(project="my-dl-ethereum-price-prediction-course-project")
```

```
[jovian] Detected Colab notebook...
[jovian] Please enter your API key ( from https://jovian.ai/ ):
API KEY: ··········
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/prachin2002patel/my-dl-ethereum-price-prediction-course-project
```

Out[35]:
```
'https://jovian.ai/prachin2002patel/my-dl-ethereum-price-prediction-course-project'
```

In [ ]: