Ferdinand Beaman Capstone Project April 2024

My students often ask what I'm doing "in school", and more than once I've gotten this response when I tell them:

"That sounds *boooring*"

But when I say "That's because you didn't try to predict stock prices yet",that always raises the eyebrows.

"Day trader" is a surprisingly common self-identifier, despite the fact that most people who try their hand at is are overwhelmingly bad at it. The stock market, on the whole, climbs up in value year after year. But, depending on the source, it's assumed that anywhere between 90 and 99% of people who take part in day trading lose money. My personal hypothesis? Human beings are overconfident pattern seekers and would benefit greatly from some scientific rigor. They learn trading "strategies" that involve reading what may amount to just tea leaves, and lose. Beyond the fallability of humans, there is also the theory of the "Efficient-market hypothesis" in the way.

However, neural networks may have the power to make this a viable path. While a NN may not have access to the latest news on Elon Musk or global pandemics, and thus have poor predictive powers long term, they may have the ability to sift through the chaos over shorter timescales and do well. After all, in any given hour it's unlikely that a CEO will get caught for insider trading or a boat will get stuck in a canal and shut down global trade for days.

> Here stands my business problem: Can I use new stock price information (up to an hour old) to predict impending prices for a client, live?

The data I'm using comes from https://firstratedata.com/, but in the interest of reprodicibility I should mention that their free samples seem to be tied to the day you request for them. My sample covered an 11 day span of minute-to-minute data beginning near the end of February 2024. (Due to this small sample, I didn't feel the need to account for any cyclical trends).

The seven stocks I chose were based on two criteria: 1) there needed to be enough volume to give the models the best chance and 2) they were in largely different fields.

I chose to use GRU as my NN based on this paper: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9141105/#:~:text=2.4.-,Recurrent%20Neural%20Networks%20(RNNs),time%20intervals%20or%20time%20steps. This ended up paying off quite a bit since some of my cells took over an hour to run. I would not want to rerun them with something more computationally demanding, like an LTSM.

I used a "walk forward" strategy to train my models, and used 5 different step-sizes: 25, 34, 45, 60, and 80 minutes (each step down is a 25% reduction). It quickly became apparent that the 80 minute models didn't have enough room to make enough steps for training and were scrapped midway through.

1.0 Data Exploration and Preprocessing

```
import pandas as pd
import matplotlib.pyplot as plt
# !pip install numpy==1.23.0
# One of the later cells refused to function with the newest numpy
import numpy as np

from sklearn.metrics import mean_squared_error as mse
from sklearn.preprocessing import StandardScaler

from datetime import datetime as dt

from keras.models import Sequential
from keras.layers import *
from keras.losses import MeanSquaredError
from keras.metrics import RootMeanSquaredError
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
from keras.layers import GRU
from keras.callbacks import ModelCheckpoint

import tensorflow as tf
import keras


print(np.__version__) # I found that it works up to ver 1.25.2
```

```
1.25.2
```

```
#American Airlines
df_aal = pd.read_csv("https://raw.githubusercontent.com/FerdinandBeaman/Capstone/main/1MinSamples/AAL_1min_sample.csv")
#Fed Ex
df_fdx = pd.read_csv("https://raw.githubusercontent.com/FerdinandBeaman/Capstone/main/1MinSamples/FDX_1min_sample.csv")
#Fidelity National
df_fis = pd.read_csv("https://raw.githubusercontent.com/FerdinandBeaman/Capstone/main/1MinSamples/FIS_1min_sample.csv")
#Macy's
df_mcy = pd.read_csv("https://raw.githubusercontent.com/FerdinandBeaman/Capstone/main/1MinSamples/M_1min_sample.csv")
#Sprint
df_spr = pd.read_csv("https://raw.githubusercontent.com/FerdinandBeaman/Capstone/main/1MinSamples/S_1min_sample.csv")
#Starbucks
df_sbx = pd.read_csv("https://raw.githubusercontent.com/FerdinandBeaman/Capstone/main/1MinSamples/SBUX_1min_sample.csv")
#Tesla
df_tsl = pd.read_csv("https://raw.githubusercontent.com/FerdinandBeaman/Capstone/main/1MinSamples/TSLA_1min_sample.csv")

all_dfs = [df_aal, df_fdx, df_fis, df_mcy, df_sbx, df_spr, df_tsl]


# Example data
all_dfs[0].head()
```

	timestamp	open	high	low	close	volume
0	2024-02-26 04:03:00	15.10	15.10	15.10	15.10	999
1	2024-02-26 04:04:00	15.10	15.11	15.10	15.11	200
2	2024-02-26 04:10:00	15.09	15.09	15.09	15.09	372
3	2024-02-26 04:13:00	15.09	15.09	15.09	15.09	100
4	2024-02-26 04:30:00	15.09	15.09	15.09	15.09	142

How many data points do I have?

```
for df in all_dfs:
    print(len(df))
```

```
5700
4231
4396
5595
4776
5050
10005
```

Tesla is a fairly popular name in the Zeitgeist here in 2024, so it's no surprise that it shows more movement than anyone else.

```
# Converting the dfs into datetime data
for df in all_dfs:
    df['timestamp'] = pd.to_datetime(df['timestamp'])


#Checking for null entries
for df in all_dfs:
  print(df.isnull().sum())
  print("\n")
```

```
volume       0
dtype: int64


timestamp    0
open         0
high         0
low          0
close        0
volume       0
dtype: int64
```

```
close        0
volume       0
dtype: int64


timestamp    0
open         0
high         0
low          0
close        0
volume       0
dtype: int64


timestamp    0
open         0
high         0
low          0
close        0
volume       0
dtype: int64


timestamp    0
open         0
high         0
low          0
close        0
volume       0
dtype: int64


timestamp    0
open         0
high         0
low          0
close        0
volume       0
dtype: int64
```

Instead of having null entries, the less popular stocks just don't have rows where nothing happened. Later on, this is addressed by forward-filling.

Ahead, I found the most exclusive boundaries (the latest starting time and the earliest ending time) so I could make all of my data uniform in length.

```
for df in all_dfs:
    print(df["timestamp"][0])
```

```
2024-02-26 04:03:00
2024-02-26 06:09:00
2024-02-26 06:06:00
2024-02-26 04:41:00
2024-02-26 08:00:00
2024-02-26 04:00:00
2024-02-26 04:00:00
```

```
for df in all_dfs:
    print(df["timestamp"].iloc[-1])
```

```
2024-03-11 19:44:00
2024-03-11 18:11:00
2024-03-11 16:00:00
2024-03-11 19:39:00
2024-03-11 19:04:00
2024-03-11 19:38:00
2024-03-11 19:54:00
```

8am on the 26th and 4pm on the 11th.

```
for df in all_dfs:
    df.set_index('timestamp', inplace=True)
```

```
#Finally ffilling the dfs, the last serious precursor to concatenation
for i, df in enumerate(all_dfs):
    all_dfs[i] = df.resample("1min").asfreq().ffill()


for i, df in enumerate(all_dfs):
    all_dfs[i] = df['2024-02-26 08:00' : "2024-03-11 16:00" ]


# Just removing superfluous columns
for i, df in enumerate(all_dfs):
    all_dfs[i].drop(["high", "low", "close"], axis = 1, inplace = True)


seven_dfs = pd.concat(all_dfs, axis=1)
```

Woops, didn't realize that the columns would all now have the same names.

```
## There appears to be an update to the "set_axis" method, and now it no longer
## accepts the "inplace" parameter. So I had to switch to using .iloc() instead

# cols = ["open_1", "volume_1", "open_2", "volume_2", "open_3",
#                     "volume_3", "open_4", "volume_4", "open_5", "volume_5",
#                     "open_6", "volume_6", "open_7", "volume_7"]

# seven_dfs.set_axis(cols, axis = 1, inplace = True)
```

To prevent long stretches of time where the price doesn't change by much from ruining my experiment, I removed all of the after-hours data.

```
seven_dfs["hour"] = np.nan
for i in range(len(seven_dfs)):
  seven_dfs["hour"][i] = seven_dfs.index[i].hour

seven_dfs["day"] = np.nan
for i in range(len(seven_dfs)):
  seven_dfs["day"][i] = seven_dfs.index[i].dayofweek


seven_dfs.drop(seven_dfs[(seven_dfs["hour"] < 8) |
                         (seven_dfs["hour"] > 15)].index, inplace = True)
seven_dfs.drop(seven_dfs[seven_dfs["day"] > 4].index, inplace = True)
```

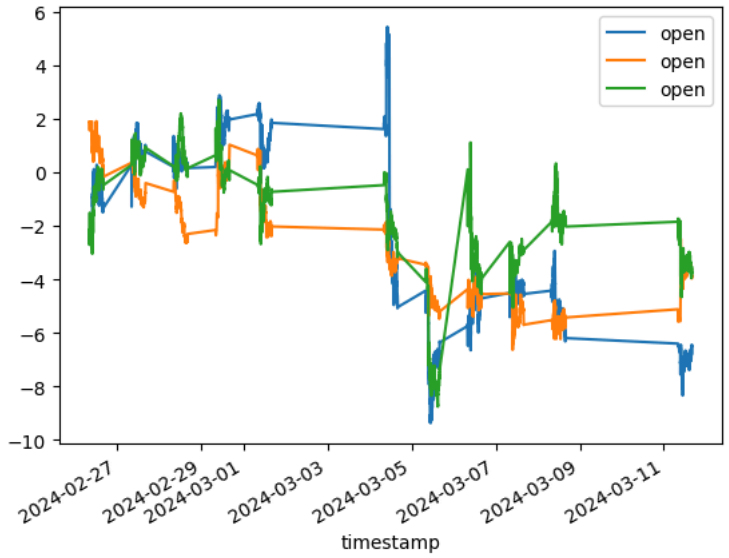
I only used the first day and a half's worth of data to scale everything else.

```
# Getting the first one and a half days of data for the initial training set
# to scale my data. If I only used the first few hours, there probably would not
# be enough variance in that small of a pool for the StD to be sensible.

prices = [0, 2, 4, 6, 8, 10, 12] # the location of the "open" columns

train_36_hrs = seven_dfs['2024-02-26 08:00' : "2024-02-28 12:00"].copy()

scaler = StandardScaler()

train_36_hrs.iloc[:, prices] = scaler.fit(train_36_hrs.iloc[:,prices])
seven_dfs.iloc[:, prices] = scaler.transform(seven_dfs.iloc[:,prices])
```

So what does this all look like now? Here's a sample:

```
# Some arbitrary columns
seven_dfs.iloc[:, [0, 8, 10]].plot()
```

```
<Axes: xlabel='timestamp'>
```



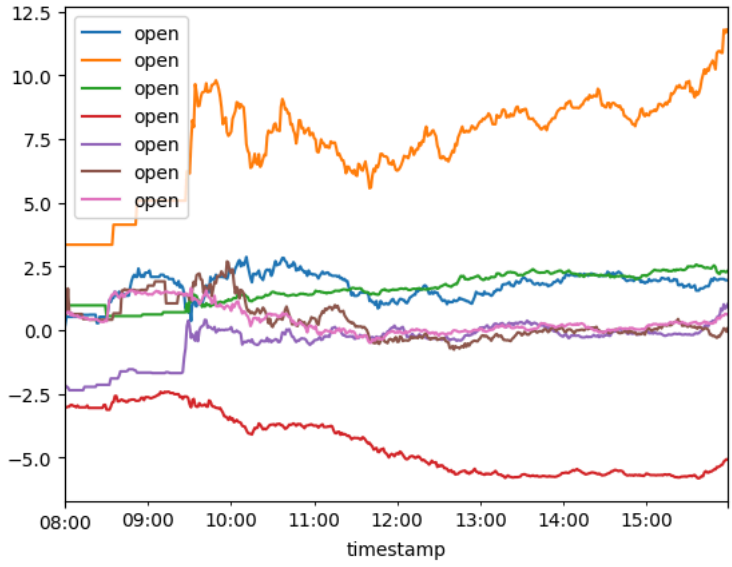
Those long horizontal lines frightened me half to death when I first saw them, but it turns out to not be bad at all (except visually). The graph still includes those after-hours times in the x-axis even though there's no price there. This means that in reality those long bars are just illustrating the difference between two adjacent data points which happen to be far apart in real time.

This was good to see, as there's no obvious pattern to what happens to prices during those times.

How much does a stock's price tend to move through the day? I should know what I'm working with to make sure that the predictions I'm trying to make have the potential to be meaningful.

```
# Over the course of one day:
seven_dfs.iloc[:,prices][1440:1920].plot()
```

```
<Axes: xlabel='timestamp'>
```

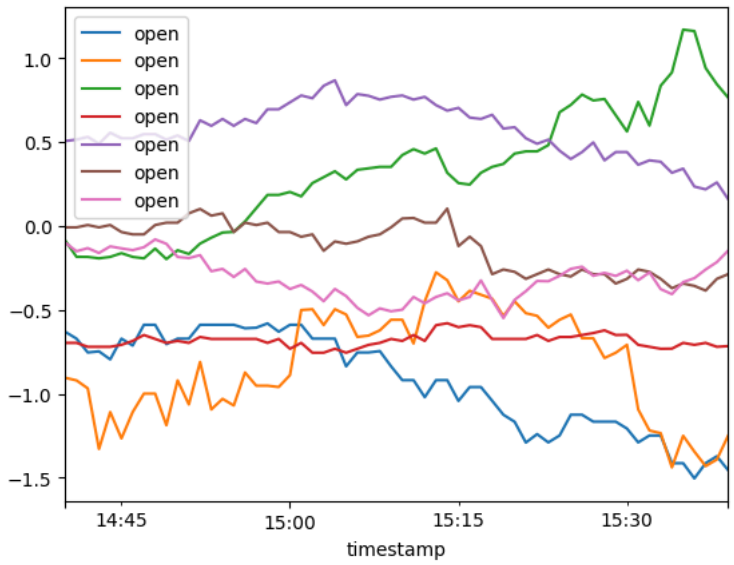


There is a lack of normal activity in the first hour or so of my current window. I set my early boundary to be 9am after looking at a few more slices.

In the meantime, I have yet to see if it's even meaningful to guess how much

```
seven_dfs.iloc[:,prices][400:460].plot()
```

```
<Axes: xlabel='timestamp'>
```

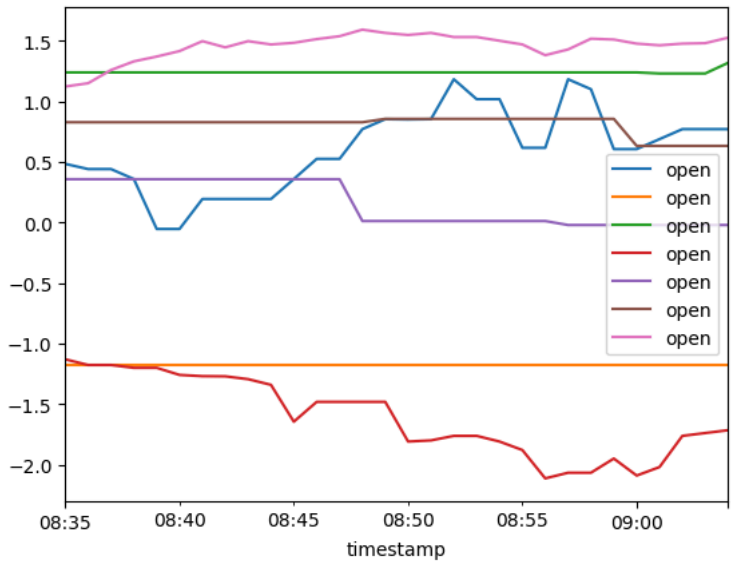


Sometimes, quite a bit! The price tracked by either the green or the blue line changed by an entire standard deviation in about 50 minutes.

What about in an arbitrary half-hour?

```
seven_dfs.iloc[:,prices][515:545].plot()
```

```
<Axes: xlabel='timestamp'>
```



This time, it's the red and orange lines that are most on the move.

That was all promising. And before I had a chance to forget, I removed the first hour's prices from the list.

```
seven_dfs.drop(seven_dfs[seven_dfs["hour"] < 9].index, inplace = True)
```

Which left me with this many data points for prices:

```
len(seven_dfs)*7
```

```
32340
```

2.0 Building the Model

```
# Code repurposed from Greg Hogg: https://www.youtube.com/watch?v=c0k-YLQGKjY
def df_to_Xy(df, window):
  df_np = df.to_numpy()
  X = []
  y = []
  for i in range(0, len(df)-window, window):
    row = [a for a in df_np[i:i+window]]
    X.append(row)
    y.append(df_np[i+window][[0,2,4,6,8,10,12]]) # y is just the 7 price cols
  return np.array(X), np.array(y,dtype=np.float32)


X25, y25 = df_to_Xy(seven_dfs, 25)
X34, y34 = df_to_Xy(seven_dfs, 34)
X45, y45 = df_to_Xy(seven_dfs, 45)
X60, y60 = df_to_Xy(seven_dfs, 60)


X_train25, y_train25 = X25[:129], y25[:129] #Just over 70% of the data
X_val25, y_val25 = X25[129:157], y25[129:157]
X_test25, y_test25 = X25[157:], y25[157:]

X_train34, y_train34 = X34[:95], y34[:95]
X_val34, y_val34 = X34[95:115], y34[95:115]
X_test34, y_test34 = X34[115:], y34[115:]

X_train45, y_train45 = X45[:72], y45[:72]
X_val45, y_val45 = X45[72:87], y45[72:87]
X_test45, y_test45 = X45[87:], y45[87:]

X_train60, y_train60 = X60[:55], y60[:55]
X_val60, y_val60 = X60[55:65], y60[55:65]
X_test60, y_test60 = X60[65:], y60[65:]
```

Below are just the functions used for graphing. In order, they will: show a particular prediction and price in dollars; show all of a model's predictions and prices but scaled; and show the training/validation loss curves.

```
def pred_plot_real(model, X, y, col):
  preds = scaler.inverse_transform(model.predict(X))[:,col]
  actuals = scaler.inverse_transform(y)[:,col]
  df = pd.DataFrame(data={"Predictions":preds, "Actuals":actuals})
  plt.plot(df["Predictions"][:], label = "Predictions")
  plt.plot(df["Actuals"][:], label = "Actuals")
  plt.legend()
  plt.xlabel("Step Number")
  plt.ylabel("Price in Dollars")
  plt.show()

def pred_plot_all(model, X, y):
  fig, ((ax1, ax2), (ax3, ax4), (ax5, ax6), (ax7, ax8)) = plt.subplots(4, 2)
  fig.set_figheight(22)
  fig.set_figwidth(15)
  axes = [ax1, ax2, ax3, ax4, ax5, ax6, ax7]
  for i, ax in enumerate(axes):
    actual = y[:,i].flatten()
    preds = model.predict(X)[:,i].flatten()
    ax = ax
    ax.plot(preds)
    ax.plot(actual)
    ax.legend(["Prediction", "Actual"])
    plt.legend()
  plt.show()

def plot_error(history):
  hist_dict = history.history
  rmse = hist_dict["root_mean_squared_error"]
  v_rmse = hist_dict["val_root_mean_squared_error"]
  df = pd.DataFrame(data={"Train_error":rmse, "Val_Error":v_rmse})
  plt.plot(df["Train_error"][:], label = "Train error")
  plt.plot(df["Val_Error"][:], label = "Val Error")
  plt.title('Training and Validation Loss')
  plt.xlabel('Epochs')
  plt.ylabel('Loss')
  plt.legend()
```

I chose two variables to adjust, and each one has two states:

Learning rate, which is either Default (0.0001) or Fast (0.01). Number of hidden layers: Short (2) vs Long (4)

```
# All of the training and validation datasets
the_X_trains = [X_train25, X_train34, X_train45, X_train60]
the_y_trains = [y_train25, y_train34, y_train45, y_train60]

the_X_vals = [X_val25, X_val34, X_val45, X_val60]
the_y_vals = [y_val25, y_val34, y_val45, y_val60]


## Instantiating models, checkpoints, and what will be their histories,
## then putting all of them in lists

# Default/Short
mod_25_DeSh = Sequential()
mod_34_DeSh = Sequential()
mod_45_DeSh = Sequential()
mod_60_DeSh = Sequential()

cp_25_DeSh = ModelCheckpoint("model_25_DeSh/", save_best_only=True)
cp_34_DeSh = ModelCheckpoint("model_34_DeSh/", save_best_only=True)
cp_45_DeSh = ModelCheckpoint("model_45_DeSh/", save_best_only=True)
cp_60_DeSh = ModelCheckpoint("model_60_DeSh/", save_best_only=True)

hist_25_DeSh = None
hist_34_DeSh = None
hist_45_DeSh = None
hist_60_DeSh = None

mods_DeSh = [mod_25_DeSh, mod_34_DeSh, mod_45_DeSh, mod_60_DeSh]

cps_DeSh = [cp_25_DeSh, cp_34_DeSh, cp_45_DeSh, cp_60_DeSh]

hists_DeSh = [hist_25_DeSh, hist_34_DeSh, hist_45_DeSh, hist_60_DeSh]

# Fast/Short
mod_25_FaSh = Sequential()
mod_34_FaSh = Sequential()
mod_45_FaSh = Sequential()
mod_60_FaSh = Sequential()

cp_25_FaSh = ModelCheckpoint("model_25_FaSh/", save_best_only=True)
cp_34_FaSh = ModelCheckpoint("model_34_FaSh/", save_best_only=True)
cp_45_FaSh = ModelCheckpoint("model_45_FaSh/", save_best_only=True)
cp_60_FaSh = ModelCheckpoint("model_60_FaSh/", save_best_only=True)

hist_25_FaSh = None
hist_34_FaSh = None
hist_45_FaSh = None
hist_60_FaSh = None

mods_FaSh = [mod_25_FaSh, mod_34_FaSh, mod_45_FaSh, mod_60_FaSh]

cps_FaSh = [cp_25_FaSh, cp_34_FaSh, cp_45_FaSh, cp_60_FaSh]

hists_FaSh = [hist_25_FaSh, hist_34_FaSh, hist_45_FaSh, hist_60_FaSh]

# Default/Long
mod_25_DeLo = Sequential()
mod_34_DeLo = Sequential()
mod_45_DeLo = Sequential()
mod_60_DeLo = Sequential()

cp_25_DeLo = ModelCheckpoint("model_25_DeLo/", save_best_only=True)
cp_34_DeLo = ModelCheckpoint("model_34_DeLo/", save_best_only=True)
cp_45_DeLo = ModelCheckpoint("model_45_DeLo/", save_best_only=True)
cp_60_DeLo = ModelCheckpoint("model_60_DeLo/", save_best_only=True)

hist_25_DeLo = None
hist_34_DeLo = None
hist_45_DeLo = None
hist_60_DeLo = None

mods_DeLo = [mod_25_DeLo, mod_34_DeLo, mod_45_DeLo, mod_60_DeLo]

cps_DeLo = [cp_25_DeLo, cp_34_DeLo, cp_45_DeLo, cp_60_DeLo]

hists_DeLo = [hist_25_DeLo, hist_34_DeLo, hist_45_DeLo, hist_60_DeLo]
```

```
# Fast/Long
mod_25_FaLo = Sequential()
mod_34_FaLo = Sequential()
mod_45_FaLo = Sequential()
mod_60_FaLo = Sequential()

cp_25_FaLo = ModelCheckpoint("model_25_FaLo/", save_best_only=True)
cp_34_FaLo = ModelCheckpoint("model_34_FaLo/", save_best_only=True)
cp_45_FaLo = ModelCheckpoint("model_45_FaLo/", save_best_only=True)
cp_60_FaLo = ModelCheckpoint("model_60_FaLo/", save_best_only=True)

hist_25_FaLo = None
hist_34_FaLo = None
hist_45_FaLo = None
hist_60_FaLo = None

mods_FaLo = [mod_25_FaLo, mod_34_FaLo, mod_45_FaLo, mod_60_FaLo]

cps_FaLo = [cp_25_FaLo, cp_34_FaLo, cp_45_FaLo, cp_60_FaLo]

hists_FaLo = [hist_25_FaLo, hist_34_FaLo, hist_45_FaLo, hist_60_FaLo]
```

I gave models with a fast learning rate a maximum of 30 epochs to train, but few if any needed that long. The default learning rate-models were given 300 epochs, and many of them could have used even *more* time.

Model type 1: Default LR, Shorter Network

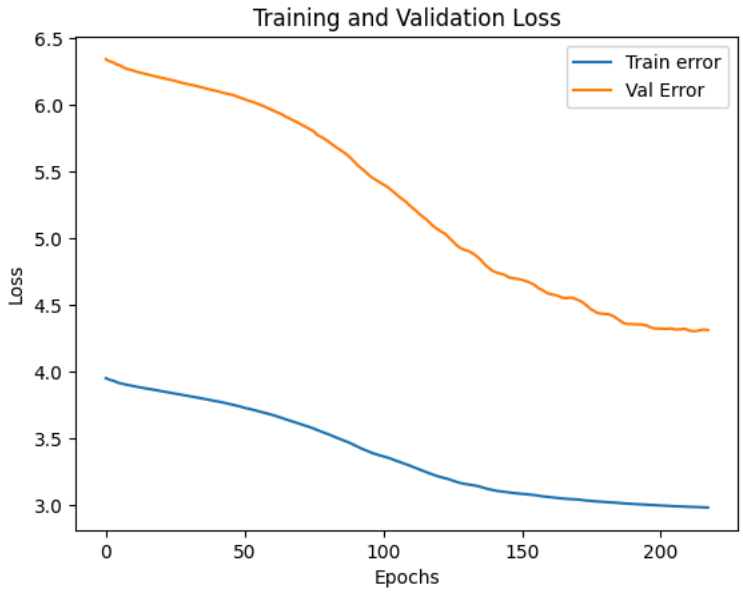
```
# Default and Short models
for i, n in enumerate([25, 34, 45, 60]):
  mods_DeSh[i].add(InputLayer((n,16)))
  mods_DeSh[i].add(GRU(64))
  mods_DeSh[i].add(Dense(16, "relu"))
  mods_DeSh[i].add(Dense(14, "relu"))
  mods_DeSh[i].add(Dense(7, "linear"))

  mods_DeSh[i].compile(loss=MeanSquaredError(),
                       optimizer=Adam(learning_rate=.0001),
                       metrics=[RootMeanSquaredError()])

  print("Default and Short, samples = " + str(n))
  hists_DeSh[i] = mods_DeSh[i].fit(the_X_trains[i], the_y_trains[i],
      validation_data=(the_X_vals[i], the_y_vals[i]), epochs = 300,
      callbacks = [cps_DeSh[i], EarlyStopping(patience=5, start_from_epoch=10)])

  print("\n")
  print("\n")
```

```
Epoch 288/300
2/2 [==============================] - 3s 3s/step - loss: 10.4262 - root_mean_squared_error: 3.2290 - val_loss: 23.0244 - va
Epoch 289/300
2/2 [==============================] - 5s 5s/step - loss: 10.4072 - root_mean_squared_error: 3.2260 - val_loss: 22.9632 - va
Epoch 290/300
2/2 [==============================] - 3s 3s/step - loss: 10.3887 - root_mean_squared_error: 3.2232 - val_loss: 22.9018 - va
Epoch 291/300
2/2 [==============================] - 3s 3s/step - loss: 10.3716 - root_mean_squared_error: 3.2205 - val_loss: 22.8385 - va
Epoch 292/300
2/2 [==============================] - 4s 4s/step - loss: 10.3526 - root_mean_squared_error: 3.2175 - val_loss: 22.7773 - va
Epoch 293/300
2/2 [==============================] - 4s 4s/step - loss: 10.3337 - root_mean_squared_error: 3.2146 - val_loss: 22.7177 - va
Epoch 294/300
2/2 [==============================] - 3s 3s/step - loss: 10.3158 - root_mean_squared_error: 3.2118 - val_loss: 22.6574 - va
Epoch 295/300
2/2 [==============================] - 4s 4s/step - loss: 10.2998 - root_mean_squared_error: 3.2093 - val_loss: 22.5964 - va
Epoch 296/300
2/2 [==============================] - 3s 3s/step - loss: 10.2791 - root_mean_squared_error: 3.2061 - val_loss: 22.5394 - va
Epoch 297/300
2/2 [==============================] - 4s 4s/step - loss: 10.2618 - root_mean_squared_error: 3.2034 - val_loss: 22.4819 - va
Epoch 298/300
2/2 [==============================] - 4s 4s/step - loss: 10.2437 - root_mean_squared_error: 3.2006 - val_loss: 22.4230 - va
Epoch 299/300
2/2 [==============================] - 3s 3s/step - loss: 10.2268 - root_mean_squared_error: 3.1979 - val_loss: 22.3632 - va
Epoch 300/300
2/2 [==============================] - 4s 4s/step - loss: 10.2072 - root_mean_squared_error: 3.1949 - val_loss: 22.3058 - va
```

```
error_list = [] #for storing and retrieving the best model


for i, n in enumerate([25, 34, 45, 60]):
  score = min(hists_DeSh[i].history['val_root_mean_squared_error'])

  print(str(score) + " = Default/Short best Val RMSE with samples sized " + str(n))

  error_list.append(score)

  print("\n")
```

```
4.304657459259033 = Default/Short best Val RMSE with samples sized 25


4.188783645629883 = Default/Short best Val RMSE with samples sized 34


4.100379943847656 = Default/Short best Val RMSE with samples sized 45


4.722903251647949 = Default/Short best Val RMSE with samples sized 60
```

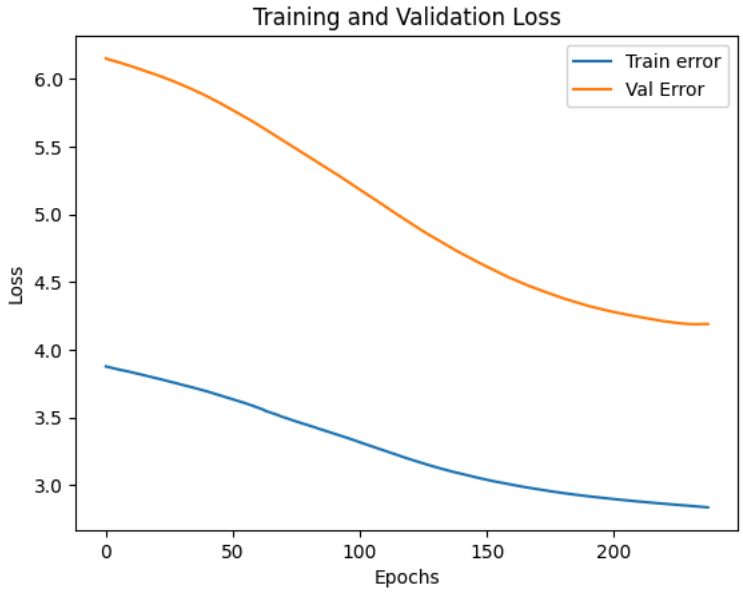
```
plot_error(hists_DeSh[0])
```



It appears that while the two curves haven't really converged for this model (the gap has only shrunk by about a third), we probably just passed the inflection point where the rate of the decrease is itself decreasing. The return on investment timewise in pushing through more epochs is very unappealing.

And it's only worse for the other ones...

```
#..like this one, which stopped making meaningful progress a hundred epochs ago.
plot_error(hists_DeSh[1])
```



These models were ultimately unlikely to lead anywhere good, and their progress ultimately resulted in the below graphs.

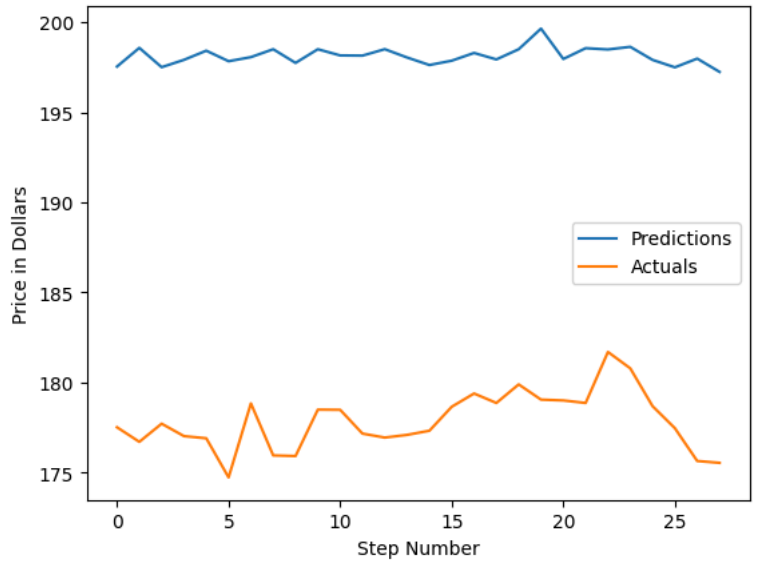
```
# 25 min steps, Default Learning Rate + Short Network
pred_plot_all(mod_25_DeSh, X_val25, y_val25)
```

```
1/1 [==============================] - 0s 52ms/step
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note t
```

Bear with me a moment while I zoom in on just the last graph here, the one for Tesla.

```
pred_plot_real(mod_25_DeSh, X_val25, y_val25, 6)
```

```
1/1 [==============================] - 0s 26ms/step
```



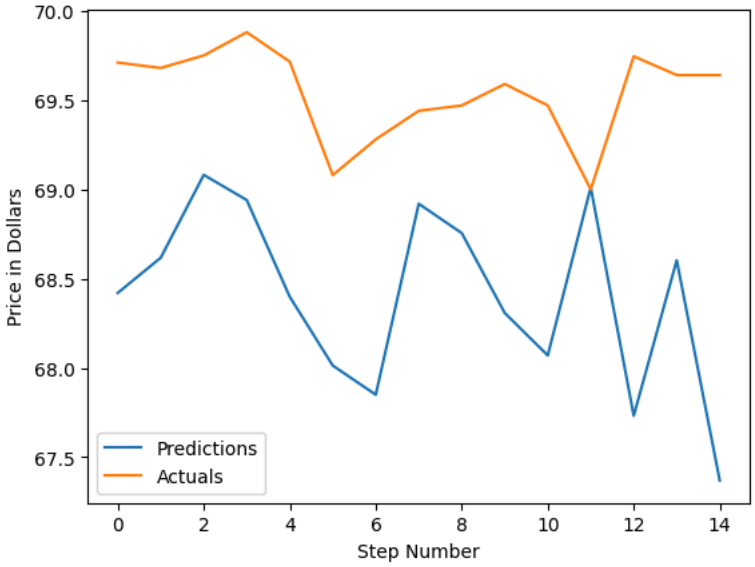
Why is there so much bias here? At first I thought "If *I'm* smart enough to realize it should add 15 dollars to the price, why isn't the model?"

Well, if you look at the other graphs they show a similar bias, but it's mixed between being consistently overshooting and undershooting. I assume that it's this mixture, born from training on multiple stocks, which explains it away. If I trained on Tesla alone, perhaps we'd see something different.

All of the other De/Sh models were about as bad. I'll spare you the gory details of those graphs, except for one glimmer here:

```
print("predicted price:")
print(scaler.inverse_transform(mod_45_DeSh.predict(X_val45))[:,2][11])
print("actual price:")
print(scaler.inverse_transform(y_val45)[:,2][11])
pred_plot_real(mod_45_DeSh, X_val45, y_val45, 2)
```

```
1/1 [==============================] - 0s 72ms/step
69.01133
69.0
1/1 [==============================] - 0s 72ms/step
```



There, at the 11th prediction, it got within about a penny's worth of being exactly correct. Cause for optimism, or random noise?

In any case, let's skip to the second model.

Model Type 2: Default LR and a Longer Network

These models were marginally better than those before, which is to say not very good either.

```
# Default and Long models
for i, n in enumerate([25, 34, 45, 60]):
  mods_DeLo[i].add(InputLayer((n,16)))
  mods_DeLo[i].add(GRU(64))
  mods_DeLo[i].add(Dense(16, "relu"))
  mods_DeLo[i].add(Dense(16, "relu"))
  mods_DeLo[i].add(Dense(15, "relu"))
  mods_DeLo[i].add(Dense(14, "relu"))
  mods_DeLo[i].add(Dense(7, "linear"))

  mods_DeLo[i].compile(loss=MeanSquaredError(),
                       optimizer=Adam(learning_rate=.0001),
                       metrics=[RootMeanSquaredError()])

  print("Default and Long, samples = " + str(n))
  hists_DeLo[i] = mods_DeLo[i].fit(the_X_trains[i], the_y_trains[i],
      validation_data=(the_X_vals[i], the_y_vals[i]), epochs = 300,
      callbacks = [cps_DeLo[i], EarlyStopping(patience=5, start_from_epoch=10)])


  print("\n")
  print("\n")
```

```
2/2 [==============================] - 5s 5s/step - loss: 11.0004 - root_mean_squared_error: 3.3167 - val_loss: 27.8083 - va
Epoch 282/300
2/2 [==============================] - 5s 4s/step - loss: 10.9517 - root_mean_squared_error: 3.3093 - val_loss: 27.6354 - va
Epoch 283/300
2/2 [==============================] - 4s 4s/step - loss: 10.8981 - root_mean_squared_error: 3.3012 - val_loss: 27.4640 - va
Epoch 284/300
2/2 [==============================] - 4s 4s/step - loss: 10.8470 - root_mean_squared_error: 3.2935 - val_loss: 27.2922 - va
Epoch 285/300
2/2 [==============================] - 5s 5s/step - loss: 10.8005 - root_mean_squared_error: 3.2864 - val_loss: 27.1192 - va
Epoch 286/300
2/2 [==============================] - 4s 4s/step - loss: 10.7429 - root_mean_squared_error: 3.2776 - val_loss: 26.9505 - va
Epoch 287/300
2/2 [==============================] - 4s 4s/step - loss: 10.7009 - root_mean_squared_error: 3.2712 - val_loss: 26.7771 - va
Epoch 288/300
2/2 [==============================] - 5s 5s/step - loss: 10.6471 - root_mean_squared_error: 3.2630 - val_loss: 26.6072 - va
Epoch 289/300
2/2 [==============================] - 5s 5s/step - loss: 10.6026 - root_mean_squared_error: 3.2562 - val_loss: 26.4352 - va
Epoch 290/300
2/2 [==============================] - 4s 4s/step - loss: 10.5463 - root_mean_squared_error: 3.2475 - val_loss: 26.2684 - va
Epoch 291/300
2/2 [==============================] - 4s 4s/step - loss: 10.5050 - root_mean_squared_error: 3.2411 - val_loss: 26.0964 - va
Epoch 292/300
2/2 [==============================] - 4s 4s/step - loss: 10.4599 - root_mean_squared_error: 3.2342 - val_loss: 25.9247 - va
Epoch 293/300
2/2 [==============================] - 5s 5s/step - loss: 10.4092 - root_mean_squared_error: 3.2263 - val_loss: 25.7562 - va
Epoch 294/300
2/2 [==============================] - 4s 4s/step - loss: 10.3638 - root_mean_squared_error: 3.2193 - val_loss: 25.5876 - va
Epoch 295/300
2/2 [==============================] - 4s 4s/step - loss: 10.3187 - root_mean_squared_error: 3.2123 - val_loss: 25.4188 - va
Epoch 296/300
2/2 [==============================] - 4s 4s/step - loss: 10.2772 - root_mean_squared_error: 3.2058 - val_loss: 25.2482 - va
Epoch 297/300
2/2 [==============================] - 5s 5s/step - loss: 10.2283 - root_mean_squared_error: 3.1982 - val_loss: 25.0817 - va
Epoch 298/300
2/2 [==============================] - 4s 4s/step - loss: 10.1805 - root_mean_squared_error: 3.1907 - val_loss: 24.9165 - va
Epoch 299/300
2/2 [==============================] - 4s 4s/step - loss: 10.1457 - root_mean_squared_error: 3.1852 - val_loss: 24.7452 - va
Epoch 300/300
2/2 [==============================] - 4s 4s/step - loss: 10.1003 - root_mean_squared_error: 3.1781 - val_loss: 24.5777 - va
```

```
for i, n in enumerate([25, 34, 45, 60]):
  score = min(hists_DeLo[i].history['val_root_mean_squared_error'])

  print(str(score) + " = Default/Long best Val RMSE with samples sized " + str(n))

  error_list.append(score)

  print("\n")
```

```
4.2636590003967285 = Default/Long best Val RMSE with samples sized 25


4.011013031005859 = Default/Long best Val RMSE with samples sized 34


3.8015613555908203 = Default/Long best Val RMSE with samples sized 45


4.957592964172363 = Default/Long best Val RMSE with samples sized 60
```

Once again, let's look only at a sample of what's here. The best result, which shows us another reason to be somewhat hopeful, is for the third stock below:

```
pred_plot_all(mod_45_DeLo, X_val45, y_val45)
```

```
1/1 [==============================] - 0s 52ms/step
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note t
```

Model Type 3: Fast LR, Short Network

I expected to see a dramatic difference between the fast and slow models. While I did see them in some senses, where I did not see such differences was in the RMSE scores.

```
# Fast and Short models
for i, n in enumerate([25, 34, 45, 60]):
  mods_FaSh[i].add(InputLayer((n,16)))
  mods_FaSh[i].add(GRU(64))
  mods_FaSh[i].add(Dense(16, "relu"))
  mods_FaSh[i].add(Dense(14, "relu"))
  mods_FaSh[i].add(Dense(7, "linear"))

  mods_FaSh[i].compile(loss=MeanSquaredError(),
                       optimizer=Adam(learning_rate=.01),
                       metrics=[RootMeanSquaredError()])

  print("Default and Long, samples = " + str(n))
  hists_FaSh[i] = mods_FaSh[i].fit(the_X_trains[i], the_y_trains[i],
  validation_data=(the_X_vals[i], the_y_vals[i]), epochs = 30,
      callbacks = [cps_FaSh[i], EarlyStopping(patience=4, start_from_epoch=6)])



  print("\n")
  print("\n")
```

```
   Default and Long, samples = 25
   Epoch 1/30
   5/5 [==============================] - 7s 988ms/step - loss: 15.6977 - root_mean_squared_error: 3.9620 - val_loss: 34.1873 -
   Epoch 2/30
   5/5 [==============================] - 3s 864ms/step - loss: 13.0734 - root_mean_squared_error: 3.6157 - val_loss: 26.1225 -
   Epoch 3/30
   5/5 [==============================] - 4s 1s/step - loss: 11.1026 - root_mean_squared_error: 3.3321 - val_loss: 17.5957 - va
   Epoch 4/30
   5/5 [==============================] - 4s 1s/step - loss: 10.2534 - root_mean_squared_error: 3.2021 - val_loss: 14.2027 - va
   Epoch 5/30
   5/5 [==============================] - 0s 25ms/step - loss: 9.1015 - root_mean_squared_error: 3.0169 - val_loss: 17.8214 - v
   Epoch 6/30
   5/5 [==============================] - 0s 24ms/step - loss: 8.8687 - root_mean_squared_error: 2.9780 - val_loss: 18.4424 - v
   Epoch 7/30
   5/5 [==============================] - 0s 29ms/step - loss: 8.6304 - root_mean_squared_error: 2.9378 - val_loss: 16.7116 - v
   Epoch 8/30
   5/5 [==============================] - 0s 25ms/step - loss: 8.4519 - root_mean_squared_error: 2.9072 - val_loss: 14.2056 - v
   Epoch 9/30
```

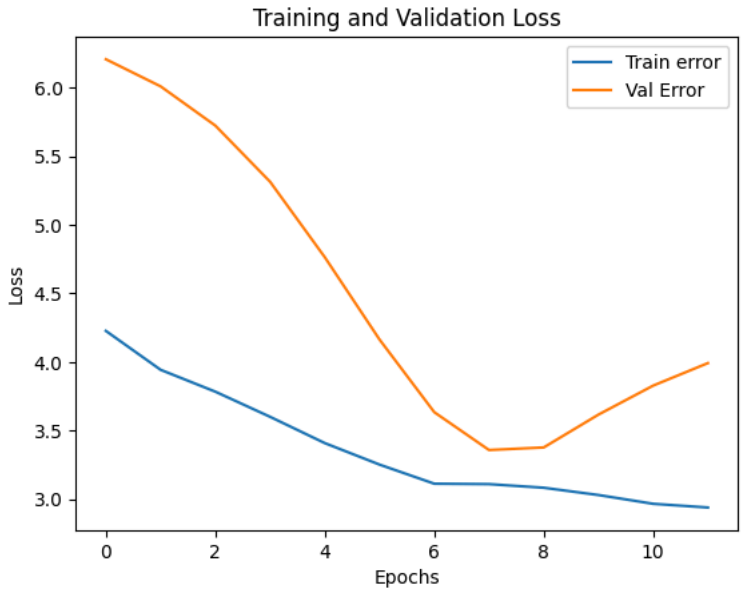
```
5/5 [==============================] - 4s 876ms/step - loss: 8.3862 - root_mean_squared_error: 2.8959 - val_loss: 12.8197 -
Epoch 10/30
5/5 [==============================] - 0s 34ms/step - loss: 8.3008 - root_mean_squared_error: 2.8811 - val_loss: 16.5127 - v
Epoch 11/30
5/5 [==============================] - 0s 33ms/step - loss: 8.5114 - root_mean_squared_error: 2.9174 - val_loss: 20.0733 - v
Epoch 12/30
5/5 [==============================] - 0s 36ms/step - loss: 8.5739 - root_mean_squared_error: 2.9281 - val_loss: 16.1745 - v
Epoch 13/30
5/5 [==============================] - 0s 39ms/step - loss: 8.1594 - root_mean_squared_error: 2.8565 - val_loss: 13.2894 - v
```

```
Default and Long, samples = 34
Epoch 1/30
3/3 [==============================] - 7s 2s/step - loss: 12.4585 - root_mean_squared_error: 3.5297 - val_loss: 20.0909 - va
Epoch 2/30
3/3 [==============================] - 4s 2s/step - loss: 9.4192 - root_mean_squared_error: 3.0691 - val_loss: 12.0567 - val
Epoch 3/30
3/3 [==============================] - 0s 45ms/step - loss: 8.6075 - root_mean_squared_error: 2.9338 - val_loss: 16.3971 - v
Epoch 4/30
3/3 [==============================] - 0s 46ms/step - loss: 8.5144 - root_mean_squared_error: 2.9179 - val_loss: 17.7315 - v
Epoch 5/30
3/3 [==============================] - 0s 50ms/step - loss: 7.8284 - root_mean_squared_error: 2.7979 - val_loss: 15.8729 - v
Epoch 6/30
3/3 [==============================] - 0s 52ms/step - loss: 7.5679 - root_mean_squared_error: 2.7510 - val_loss: 16.2280 - v
Epoch 7/30
3/3 [==============================] - 0s 44ms/step - loss: 7.5346 - root_mean_squared_error: 2.7449 - val_loss: 15.7279 - v
Epoch 8/30
3/3 [==============================] - 0s 45ms/step - loss: 7.0583 - root_mean_squared_error: 2.6567 - val_loss: 17.3987 - v
Epoch 9/30
3/3 [==============================] - 0s 46ms/step - loss: 6.8218 - root_mean_squared_error: 2.6119 - val_loss: 16.9792 - v
Epoch 10/30
3/3 [==============================] - 0s 48ms/step - loss: 6.8246 - root_mean_squared_error: 2.6124 - val_loss: 15.2904 - v
Epoch 11/30
3/3 [==============================] - 0s 55ms/step - loss: 6.6270 - root_mean_squared_error: 2.5743 - val_loss: 17.9071 - v
Epoch 12/30
3/3 [==============================] - 0s 52ms/step - loss: 6.4106 - root_mean_squared_error: 2.5319 - val_loss: 15.4234 - v
Epoch 13/30
3/3 [==============================] - 0s 44ms/step - loss: 6.1266 - root mean squared error: 2.4752 - val loss: 14.0054 - v
```

The validation loss curve showed pretty messy behavior in the fast models, either by bouncing away from a path of steady improvement repeatedly or by diverging sharply all at once.

```
# The early patience params I set allowed the model to try to improve
# despite finding a local minima very early on
plot_error(hists_FaSh[1])
```


```
plot_error(hists_FaSh[3])
```




```
for i, n in enumerate([25, 34, 45, 60]):
  score = min(hists_FaSh[i].history['val_root_mean_squared_error'])

  print(str(score) + " = Fast/Short best Val RMSE with samples sized " + str(n))

  error_list.append(score)

  print("\n")
```

```
    3.580456018447876 = Fast/Short best Val RMSE with samples sized 25


    3.1669328212738037 = Fast/Short best Val RMSE with samples sized 34


    3.7278451919555664 = Fast/Short best Val RMSE with samples sized 45


    3.358412981033325 = Fast/Short best Val RMSE with samples sized 60
```

These are still quite far away from being useful on the whole. Let's again look at the best of these, which in this case is the 34 minute model.

```
pred_plot_all(mod_34_FaSh, X_val34, y_val34)
```

```
1/1 [==============================] - 0s 93ms/step
WARNING:matplotlib.legend:No artists with labels found to put in legend.  Note t
```

There's something that's both troubling and funny in these graphs. Look at the predictions for the last three stocks: they appear to be identical! I was worried about the models getting lazy and trying to predict the prices to be the same as the last data point in X_val, but I didn't expect it to be lazy and plagairize itself!

Model Type 4: Fast LR and Long Network

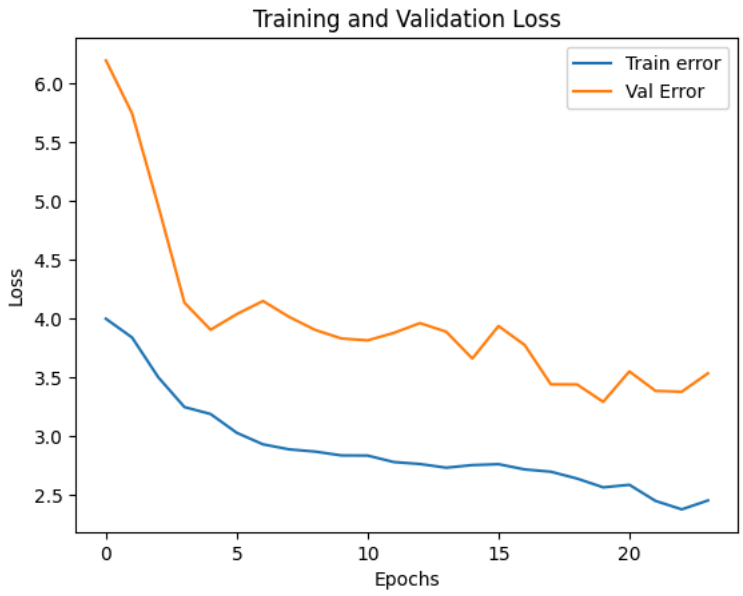
By now it should be clear that (at least on our scale) this is not a fruitful endeavor. But I may as well have that fruitlessness quantified.

```
# Fast and Long models
for i, n in enumerate([25, 34, 45, 60]):
  mods_FaLo[i].add(InputLayer((n,16)))
  mods_FaLo[i].add(GRU(64))
  mods_FaLo[i].add(Dense(16, "relu"))
  mods_FaLo[i].add(Dense(16, "relu"))
  mods_FaLo[i].add(Dense(15, "relu"))
  mods_FaLo[i].add(Dense(14, "relu"))
  mods_FaLo[i].add(Dense(7, "linear"))

  mods_FaLo[i].compile(loss=MeanSquaredError(),
                       optimizer=Adam(learning_rate=.01),
                       metrics=[RootMeanSquaredError()])

  print("Default and Long, samples = " + str(n))
  hists_FaLo[i] = mods_FaLo[i].fit(the_X_trains[i], the_y_trains[i],
  validation_data=(the_X_vals[i], the_y_vals[i]), epochs = 30,
      callbacks = [cps_FaLo[i], EarlyStopping(patience=4, start_from_epoch=6)])

  print("\n")
  print("\n")
```

```
Epoch 5/30
2/2 [==============================] - 4s 4s/step - loss: 10.1520 - root_mean_squared_error: 3.1862 - val_loss: 15.2350 - va
Epoch 6/30
2/2 [==============================] - 0s 76ms/step - loss: 9.1593 - root_mean_squared_error: 3.0264 - val_loss: 16.2915 - v
Epoch 7/30
2/2 [==============================] - 0s 67ms/step - loss: 8.5754 - root_mean_squared_error: 2.9284 - val_loss: 17.2070 - v
Epoch 8/30
2/2 [==============================] - 0s 88ms/step - loss: 8.3263 - root_mean_squared_error: 2.8855 - val_loss: 16.1044 - v
Epoch 9/30
2/2 [==============================] - 5s 5s/step - loss: 8.2165 - root_mean_squared_error: 2.8664 - val_loss: 15.2223 - val
Epoch 10/30
2/2 [==============================] - 4s 4s/step - loss: 8.0311 - root_mean_squared_error: 2.8339 - val_loss: 14.6617 - val
Epoch 11/30
2/2 [==============================] - 4s 4s/step - loss: 8.0241 - root_mean_squared_error: 2.8327 - val_loss: 14.5395 - val
Epoch 12/30
2/2 [==============================] - 0s 67ms/step - loss: 7.7150 - root_mean_squared_error: 2.7776 - val_loss: 15.0283 - v
Epoch 13/30
2/2 [==============================] - 0s 68ms/step - loss: 7.6219 - root_mean_squared_error: 2.7608 - val_loss: 15.6800 - v
Epoch 14/30
2/2 [==============================] - 0s 91ms/step - loss: 7.4508 - root_mean_squared_error: 2.7296 - val_loss: 15.1081 - v
Epoch 15/30
2/2 [==============================] - 4s 4s/step - loss: 7.5711 - root_mean_squared_error: 2.7516 - val_loss: 13.3870 - val
Epoch 16/30
2/2 [==============================] - 0s 79ms/step - loss: 7.6149 - root_mean_squared_error: 2.7595 - val_loss: 15.4858 - v
Epoch 17/30
2/2 [==============================] - 0s 81ms/step - loss: 7.3700 - root_mean_squared_error: 2.7148 - val_loss: 14.2401 - v
Epoch 18/30
2/2 [==============================] - 4s 4s/step - loss: 7.2648 - root_mean_squared_error: 2.6953 - val_loss: 11.8264 - val
Epoch 19/30
2/2 [==============================] - 6s 6s/step - loss: 6.9536 - root_mean_squared_error: 2.6370 - val_loss: 11.8205 - val
Epoch 20/30
2/2 [==============================] - 4s 4s/step - loss: 6.5654 - root_mean_squared_error: 2.5623 - val_loss: 10.8240 - val
Epoch 21/30
2/2 [==============================] - 0s 99ms/step - loss: 6.6766 - root_mean_squared_error: 2.5839 - val_loss: 12.5962 - v
Epoch 22/30
2/2 [==============================] - 0s 88ms/step - loss: 5.9855 - root_mean_squared_error: 2.4465 - val_loss: 11.4444 - v
Epoch 23/30
2/2 [==============================] - 0s 74ms/step - loss: 5.6404 - root_mean_squared_error: 2.3749 - val_loss: 11.3964 - v
Epoch 24/30
2/2 [==============================] - 0s 85ms/step - loss: 6.0042 - root_mean_squared_error: 2.4503 - val_loss: 12.4791 - v
```

The train-val loss curves have the wibbildy-wobbeldy quality like the previous batch. The learning rate is presumably jumping over the local minima, unable to converge.

```
plot_error(hists_FaLo[3])
```




```
plot_error(hists_FaLo[0])
```


```
for i, n in enumerate([25, 34, 45, 60]):
  score = min(hists_FaLo[i].history['val_root_mean_squared_error'])

  print(str(score) + " = Fast/Long best Val RMSE with samples sized " + str(n))

  error_list.append(score)

  print("\n")
```

```
    2.8924918174743652 = Fast/Long best Val RMSE with samples sized 25


    3.917736053466797 = Fast/Long best Val RMSE with samples sized 34


    2.942361831665039 = Fast/Long best Val RMSE with samples sized 45


    3.2899811267852783 = Fast/Long best Val RMSE with samples sized 60
```

On the bright side, this class was our first to have a RMSE in the 2s.

```
pred_plot_all(mod_25_FaLo, X_val25, y_val25)
```