TR-1007

This is TR-1007 in its current state.

First we need to import the libraries in use:

In [1]:
```
import os
import datetime

import numpy as np
import pandas as pd
import tensorflow as tf
import elasticsearch as es
import elasticsearch.helpers
```

Access The Full Timeseries Data

Next we read the pickle back in (but since we already have it, this will be the first real step the code takes:

"df" is data frame, it's the only data frame in use in this code.

The pickle contained thousands of tickers from NASDAQ trader's FTP which give daily open/close for NASDAQ-traded ETFs over about 10 years.

In [2]:
```
df = pd.read_pickle("./ELASTIC.pickle")
```

The following code prepares the "column_indices" listing for use by the window generator -- necessary for processing this as time-series data (see time-series forecasting tutorial)

In [3]:
```
column_indices = {name: i for i, name in enumerate(df.columns)}
```

The following performs three tasks:

- Determine the amount of data
- Split the data up into a training, validation, and finally test set for evaluating performance of our predictions
- Create a variable to determine the number of ticker symbols
- (Currently disabled) normalize the data about 1, which should improve training performance but is disabled to evaluate performance in simpler terms.

Experiment: Introduce Random Fourier Features

The following experiment adds fourier features (map different ranges of each Close to a n-vector)

See 10 minutes to pandas for how this works

See also this visual

Results:

- ✔ After training the CNN with RFFs added, it is observed that there is considerably more curvature in high-epoch training scenarios.
- ✔ Tuning these also can further improve results -- but increases training performance
- ✘ VOLATILITY_COST seems only to effect learning-rate with SGD
- 🤔 Does adjusting RFF_MAX down so that more lower frequencies are represented help? What is the optimal here?

Brief explanation:

Read this as follows: If DOF = 8, MAX = 20, COST = 50 this means the following:

- each closing price is having it's range broken up into 8 vectors,
  - each with a maximum of 20 segments (but random number of segments),
- and the default assumption when passing to the network (the cost) is $50
  - this may effect how weighting happens since the data is not normalized.

In [4]:

```
# RFF-generator
RFF_DOF = 16
RFF_SMALLNESS_PREFERENCE = 3 # a larger value means a preference for lower frequencies
RFF_MAX = 4
features = (RFF_MAX * np.random.rand(1, RFF_DOF) ** RFF_SMALLNESS_PREFERENCE).tolist()[

# Badly understood dollar value multiplier -- may enable better adherance to long-term
VOLATILITY_COST = 4

olddf = df.copy()

for col in olddf:
    if(col[0:6] == "Close_"):
        for iteration, w in enumerate(features):
            new_col = olddf[col].transform(lambda x: VOLATILITY_COST * np.sin(w * x))
            new_col.name = col + "_fourier_s_" + str(iteration)
            df[new_col.name] = new_col
            new_col = olddf[col].transform(lambda x: VOLATILITY_COST * np.cos(w * x))
            new_col.name = col + "_fourier_c_" + str(iteration)
            df[new_col.name] = new_col

df = df.copy()
```

```
C:\Users\thoma\AppData\Local\Temp/ipykernel_4352/125773879.py:20: PerformanceWarning: Da
taFrame is highly fragmented.  This is usually the result of calling `frame.insert` many
times, which has poor performance.  Consider joining all columns at once using pd.concat
(axis=1) instead.  To get a de-fragmented frame, use `newframe = frame.copy()`
  df[new_col.name] = new_col
C:\Users\thoma\AppData\Local\Temp/ipykernel_4352/125773879.py:17: PerformanceWarning: Da
taFrame is highly fragmented.  This is usually the result of calling `frame.insert` many
times, which has poor performance.  Consider joining all columns at once using pd.concat
(axis=1) instead.  To get a de-fragmented frame, use `newframe = frame.copy()`
  df[new_col.name] = new_col
```

In [5]:
```
df.describe()
```

Out[5]:

	Close	Close_AAIT	Close_AAXJ	Close_ACT	Close_ACTX	Close_ACWI	Close_ACWX	C
count	3236.000000	3236.000000	3236.000000	3236.000000	3236.000000	3236.000000	3236.000000	3
mean	279.813324	5.201730	56.421497	5.357838	1.102755	54.207947	41.577602	
std	3513.418457	11.694008	16.901571	10.176934	3.799600	15.469787	8.590531	
min	59.270000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	108.990000	0.000000	53.639999	0.000000	0.000000	44.907500	39.520000	
50%	247.680000	0.000000	59.400002	0.000000	0.000000	55.895000	43.164999	
75%	302.824989	0.000000	65.265001	0.000000	0.000000	65.445002	46.279999	
max	199999.984375	37.610001	83.500000	27.270000	15.980000	84.010002	56.430000	

8 rows × 18093 columns

Split up data-sets

This code splits up the datasets into:

- Training: the actual set that is used to choose biases for the network
- Validation: a small set of data used to approximate generalization-quality using "early-stopping"
- Test: a larger set of data to actually confirm how the network performs over completely un-seen data

In [6]:
```
n = len(df)

train_df = df[0:int(n*0.85)]
val_df = df[int(n*0.85):int(n*0.9)]
test_df = df[int(n*0.9):]

num_features = df.shape[1]
```

Normalize data

This is disabled for two reasons:

- A: It's broken, enabling this breaks everything for some reason
- B: Using the training mean over the whole data set leaks future information
- 3: I don't know how to read the predictions

Possible solutions

- Switch to percent change
  - The data is pickled and no-longer malleable though

In [7]:
```
#train_mean = train_df.mean()
#train_std = train_df.std()

#train_df = (train_df - train_mean) / train_std
#val_df = (val_df - train_mean) / train_std
#test_df = (test_df - train_mean) / train_std
```

Verify datasets under operation

This code displays some basic info about the dataset:

In [8]:
```
print("~~~~~~~~~~~~~~~~~~~~~~~~~TRAINING DATAFRAME:~~~~~~~~~~~~~~~~~~~~~")
print(train_df)
print(train_df.describe())
#print("~~~~~~~~~~~~~~~~~~~~~~~~~VALIDATION DATAFRAME:~~~~~~~~~~~~~~~~~~~~~")
#print(val_df)
#print(val_df.describe())
#print("~~~~~~~~~~~~~~~~~~~~~~~~~TESTING DATAFRAME:~~~~~~~~~~~~~~~~~~~~~")
#print(test_df)
#print(test_df.describe())
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~TRAINING DATAFRAME:~~~~~~~~~~~~~~~~~~~~~
                    Close  Close_AAIT  Close_AAXJ  Close_ACT  Close_ACTX  \
Last Trade Date
2007-11-21      100.680000         0.0    0.000000   0.000000         0.0
2007-11-23      101.849998         0.0    0.000000   0.000000         0.0
2007-11-28      103.919998         0.0    0.000000   0.000000         0.0
2007-11-29      104.720001         0.0    0.000000   0.000000         0.0
2007-11-30      104.160004         0.0    0.000000   0.000000         0.0
...                    ...         ...         ...        ...         ...
2018-10-22      292.829987         0.0   64.730003  26.049999         0.0
2018-10-23      291.350006         0.0   63.810001  25.969999         0.0
2018-10-24      279.089996         0.0   61.770000  25.299999         0.0
2018-10-25      287.019989         0.0   62.939999  25.430000         0.0
2018-10-26      280.609985         0.0   62.000000  25.330000         0.0

                Close_ACWI  Close_ACWX  Close_ADRA  Close_ADRD  Close_ADRE  \
Last Trade Date
2007-11-21        0.000000    0.000000   33.189999   31.500000   51.410000
2007-11-23        0.000000    0.000000   34.029999   32.250000   52.570000
2007-11-28        0.000000    0.000000   35.669998   33.080002   54.799999
2007-11-29        0.000000    0.000000   35.730000   32.840000   54.470001
2007-11-30        0.000000    0.000000   35.959999   33.090000   55.419998
...                    ...         ...         ...         ...         ...
2018-10-22       69.889999   44.599998   30.330000   21.170000   38.630001
2018-10-23       69.349998   44.180000   29.469999   21.160000   38.049999
2018-10-24       67.430000   42.959999    0.000000   20.940001   36.919998
2018-10-25       68.389999   43.540001    0.000000   20.590000   37.415001
2018-10-26       67.470001   43.209999   28.980000   20.480000   36.740002

                ...  Close_ZMLP_fourier_s_11  Close_ZMLP_fourier_c_11  \
Last Trade Date  ...
2007-11-21      ...                      0.0                      4.0
2007-11-23      ...                      0.0                      4.0
2007-11-28      ...                      0.0                      4.0
2007-11-29      ...                      0.0                      4.0
2007-11-30      ...                      0.0                      4.0
...             ...                      ...                      ...
```

```
2018-10-22       ...                    0.0                    4.0
2018-10-23       ...                    0.0                    4.0
2018-10-24       ...                    0.0                    4.0
2018-10-25       ...                    0.0                    4.0
2018-10-26       ...                    0.0                    4.0

                Close_ZMLP_fourier_s_12  Close_ZMLP_fourier_c_12  \
Last Trade Date
2007-11-21                          0.0                      4.0
2007-11-23                          0.0                      4.0
2007-11-28                          0.0                      4.0
2007-11-29                          0.0                      4.0
2007-11-30                          0.0                      4.0
...                                 ...                      ...
2018-10-22                          0.0                      4.0
2018-10-23                          0.0                      4.0
2018-10-24                          0.0                      4.0
2018-10-25                          0.0                      4.0
2018-10-26                          0.0                      4.0

                Close_ZMLP_fourier_s_13  Close_ZMLP_fourier_c_13  \
Last Trade Date
2007-11-21                          0.0                      4.0
2007-11-23                          0.0                      4.0
2007-11-28                          0.0                      4.0
2007-11-29                          0.0                      4.0
2007-11-30                          0.0                      4.0
...                                 ...                      ...
2018-10-22                          0.0                      4.0
2018-10-23                          0.0                      4.0
2018-10-24                          0.0                      4.0
2018-10-25                          0.0                      4.0
2018-10-26                          0.0                      4.0

                Close_ZMLP_fourier_s_14  Close_ZMLP_fourier_c_14  \
Last Trade Date
2007-11-21                          0.0                      4.0
2007-11-23                          0.0                      4.0
2007-11-28                          0.0                      4.0
2007-11-29                          0.0                      4.0
2007-11-30                          0.0                      4.0
...                                 ...                      ...
2018-10-22                          0.0                      4.0
2018-10-23                          0.0                      4.0
2018-10-24                          0.0                      4.0
2018-10-25                          0.0                      4.0
2018-10-26                          0.0                      4.0

                Close_ZMLP_fourier_s_15  Close_ZMLP_fourier_c_15
Last Trade Date
2007-11-21                          0.0                      4.0
2007-11-23                          0.0                      4.0
2007-11-28                          0.0                      4.0
2007-11-29                          0.0                      4.0
2007-11-30                          0.0                      4.0
...                                 ...                      ...
2018-10-22                          0.0                      4.0
2018-10-23                          0.0                      4.0
2018-10-24                          0.0                      4.0
2018-10-25                          0.0                      4.0
2018-10-26                          0.0                      4.0
```

```
[2750 rows x 18093 columns]
              Close   Close_AAIT   Close_AAXJ    Close_ACT   Close_ACTX  \
count    2750.000000  2750.000000  2750.000000  2750.000000  2750.000000
mean      270.316010     6.121018    54.220905     2.063489     1.297642
std      3811.226807    12.461823    17.322287     7.001296     4.091003
min        59.270000     0.000000     0.000000     0.000000     0.000000
25%       106.574999     0.000000    52.525001     0.000000     0.000000
50%       175.820000     0.000000    57.575001     0.000000     0.000000
75%       287.655006     0.000000    62.607500     0.000000     0.000000
max    199999.984375    37.610001    83.500000    27.270000    15.980000

         Close_ACWI   Close_ACWX   Close_ADRA   Close_ADRD   Close_ADRE  ...  \
count  2750.000000  2750.000000  2750.000000  2750.000000  2750.000000  ...
mean     50.863556    40.950325    22.538748    20.555838    39.034904  ...
std      14.225308     9.086999    11.078646     5.876845     6.359543  ...
min       0.000000     0.000000     0.000000     0.000000     0.000000  ...
25%      43.302500    39.052500    22.292500    19.532501    35.362500  ...
50%      51.990002    42.385000    26.495000    21.485000    38.910000  ...
75%      59.447500    45.889999    29.200001    23.219999    43.060001  ...
max      77.540001    56.430000    37.709999    33.830002    57.840000  ...

       Close_ZMLP_fourier_s_11  Close_ZMLP_fourier_c_11  \
count                   2750.0                   2750.0
mean                       0.0                      4.0
std                        0.0                      0.0
min                        0.0                      4.0
25%                        0.0                      4.0
50%                        0.0                      4.0
75%                        0.0                      4.0
max                        0.0                      4.0

       Close_ZMLP_fourier_s_12  Close_ZMLP_fourier_c_12  \
count                   2750.0                   2750.0
mean                       0.0                      4.0
std                        0.0                      0.0
min                        0.0                      4.0
25%                        0.0                      4.0
50%                        0.0                      4.0
75%                        0.0                      4.0
max                        0.0                      4.0

       Close_ZMLP_fourier_s_13  Close_ZMLP_fourier_c_13  \
count                   2750.0                   2750.0
mean                       0.0                      4.0
std                        0.0                      0.0
min                        0.0                      4.0
25%                        0.0                      4.0
50%                        0.0                      4.0
75%                        0.0                      4.0
max                        0.0                      4.0

       Close_ZMLP_fourier_s_14  Close_ZMLP_fourier_c_14  \
count                   2750.0                   2750.0
mean                       0.0                      4.0
std                        0.0                      0.0
min                        0.0                      4.0
25%                        0.0                      4.0
50%                        0.0                      4.0
75%                        0.0                      4.0
max                        0.0                      4.0
```

```
       Close_ZMLP_fourier_s_15  Close_ZMLP_fourier_c_15
count                   2750.0                   2750.0
mean                       0.0                      4.0
std                        0.0                      0.0
min                        0.0                      4.0
25%                        0.0                      4.0
50%                        0.0                      4.0
75%                        0.0                      4.0
max                        0.0                      4.0

[8 rows x 18093 columns]
```

The WindowGenerator

The following code is pulled verbatim from the tutorial

But in brief, it provides functions for pulling specified lengths of time (windows) from the time-series data. So basically, if we have:

Date	WOOD Close	A Close	AQ Close
01/01/2001	21.5	0.02	0.04
01/02/2001	21.8	0.02	0.03
01/03/2001	21.3	0.05	0.02
01/04/2001	19.4	0.05	0.08

The WindowGenerator can give us windows like, 01/01 - 01/02, 01/02 - 01/03, 01/03 - 01/04 ...

or alternatively, for windows of length 3, 01/01 - 01/03, 01/02 - 01/04 ...

and so on, as needed for training the neural networks later on this time-series data.

In [9]:

```
class WindowGenerator():
    def __init__(self, input_width, label_width, shift,
                 train_df=train_df, val_df=val_df, test_df=test_df,
                 label_columns=None):
        #Store raw data
        self.train_df = train_df
        self.val_df = val_df
        self.test_df = test_df

        # Work out the label column indices
        self.label_columns = label_columns
        if label_columns is not None:
            self.label_columns_indices = {name: i for i, name in
                                          enumerate(label_columns)}

        self.column_indices = {name: i for i, name in
                               enumerate(train_df.columns)}

        #Work out the window parameters
        self.input_width = input_width
        self.label_width = label_width
```

```
        self.shift = shift

        self.total_window_size = input_width + shift

        self.input_slice = slice(0, input_width)
        self.input_indices = np.arange(self.total_window_size)[self.input_slice]

        self.label_start = self.total_window_size - self.label_width
        self.labels_slice = slice(self.label_start, None)
        self.label_indices = np.arange(self.total_window_size)[self.labels_slice]

    def __repr__(self):
        return 'n'.join([
            f'Total window size: {self.total_window_size}',
            f'Input indices: {self.input_indices}',
            f'Label indices: {self.label_indices}',
            f'Label column name(s): {self.label_columns}'])

    def split_window(self, features):
        inputs = features[:, self.input_slice, :]
        labels = features[:, self.labels_slice, :]
        if self.label_columns is not None:
            labels = tf.stack(
                [labels[:, :, self.column_indices[name]] for name in self.label_columns

        #repair shape information
        inputs.set_shape([None, self.input_width, None])
        labels.set_shape([None, self.label_width, None])

        return inputs, labels

    def make_dataset(self, data):
        data = np.array(data, dtype=np.float32)
        ds = tf.keras.preprocessing.timeseries_dataset_from_array(
            data=data,
            targets=None,
            sequence_length=self.total_window_size,
            sequence_stride=1,
            shuffle=True,
            batch_size=32,)

        ds = ds.map(self.split_window)

        return ds

    def plot(self, model=None, plot_col='T (degC)', max_subplots=3):
        inputs, labels = self.example
        plt.figure(figsize=(12, 8))
        plot_col_index = self.column_indices[plot_col]
        max_n = min(max_subplots, len(inputs))
        for n in range(max_n):
            plt.subplot(max_n, 1, n+1)
            plt.ylabel(f'{plot_col} [normed]')
            plt.plot(self.input_indices, inputs[n, :, plot_col_index],
                     label='Inputs', marker='.', zorder=-10)

            if self.label_columns:
              label_col_index = self.label_columns_indices.get(plot_col, None)
            else:
              label_col_index = plot_col_index
```

```
            if label_col_index is None:
              continue

            plt.scatter(self.label_indices, labels[n, :, label_col_index],
                        edgecolors='k', label='Labels', c='#2ca02c', s=64)
            if model is not None:
              predictions = model(inputs)
              plt.scatter(self.label_indices, predictions[n, :, label_col_index],
                          marker='X', edgecolors='k', label='Predictions',
                          c='#ff7f0e', s=64)

            if n == 0:
              plt.legend()

        plt.xlabel('Time [h]')

    @property
    def train(self):
        return self.make_dataset(self.train_df)

    @property
    def val(self):
        return self.make_dataset(self.val_df)

    @property
    def test(self):
        return self.make_dataset(self.test_df)

    @property
    def example(self):
        """Get and cache an example batch of 'inputs, labels' for plotting."""
        result = getattr(self, '_example', None)
        if result is None:
            # No example batch was found, so get from .train dataset
            result = next(iter(self.train))
            # and cache it
            self._example = result

        return result
```

Target Ticker Symbol

In [10]:

```
TARGET_TICKER = 'Close_VTWO'
```

Simple Model

There's 5 different types of models that are defined and trained, see the tutorial for more details, but initially the baseline model tests the assumption that basically the **label** (aka, the **target**, Close_WOOD) stays the same from one time-step to the next -- giving us a baseline accuracy

In [11]:

```
single_step_window = WindowGenerator(
    input_width=24, label_width=1, shift=24,
    label_columns=[TARGET_TICKER])
```

```
class Baseline(tf.keras.Model):
    def __init__(self, label_index=None):
        super().__init__()
        self.label_index = label_index

    def call(self, inputs):
        if self.label_index is None:
            return inputs
        result = inputs[:, :, self.label_index]
        return result[:, :, tf.newaxis]

baseline = Baseline(label_index=column_indices[TARGET_TICKER])

baseline.compile(loss=tf.losses.MeanSquaredError(),
                 metrics=[tf.metrics.MeanAbsoluteError()])

val_performance = {}
performance = {}

val_performance['Baseline'] = baseline.evaluate(single_step_window.val)
performance['Baseline'] = baseline.evaluate(single_step_window.test,
                                            verbose=0)
```

```
4/4 [==============================] - 0s 24ms/step - loss: 57.7025 - mean_absolute_erro
r: 5.6837
```

Models in total

- Trivial model / constant assumption
- Linear model / average assumption
- Neural Network / single hidden layer, learns interrelationships
- Dense multi-step model
- Convolutional Neural network

This function provides a way to configure how each model is trained:

- Early stopping means that if the error "jumps back up" multiple times, it will give up further training.
- Loss, Optimizer, works, are all interesting targets for improving performance.

Experiments

- Modify loss function of the model
  - Can this benefit training?
  - Note 1: The AbsoluteError may have better results for gaussian (see normalized) data
- Modify metrics of the model
  - Make sure that units are sensible

In [12]:

```
MAX_EPOCHS = 1000
def compile_and_fit(model, window, patience=60):
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
```

```
                                                    patience=patience,
                                                    mode='min')
    model.compile(loss=tf.losses.MeanSquaredError(),
                  #optimizer=tf.optimizers.SGD(momentum=0.004, learning_rate=0.0000005)
                  optimizer=tf.optimizers.SGD(momentum=0.002, learning_rate=0.000001),
                  metrics=[tf.metrics.MeanAbsoluteError()])

    history = model.fit(window.train, epochs=MAX_EPOCHS,
                        validation_data=window.val, workers=4,
                        callbacks=[early_stopping])

    return history
```

Convolution Width

This number determines the number of days the multi-day models will "look back" on

In [13]:
```
CONV_WIDTH = 64
```

Convolutional Neural Network

Note in the modeling it was useful for understanding appropriate geometry for a CNN by using this part of this lecture

In [14]:
```
conv_window = WindowGenerator(
    input_width=CONV_WIDTH,
    label_width=1,
    shift=1,
    label_columns=[TARGET_TICKER])

conv_model = tf.keras.Sequential([
        tf.keras.layers.GaussianNoise(stddev=1),
        tf.keras.layers.Conv1D(filters=64,
                               kernel_size=(2,),
                               activation='relu',
                                bias_initializer=tf.keras.initializers.RandomUniform()),
        tf.keras.layers.Conv1D(filters=256,
                               kernel_size=(4 ,),
                               activation='relu',
                               bias_initializer=tf.keras.initializers.RandomUniform()),
            tf.keras.layers.Conv1D(filters=4096,
                               kernel_size=(8,),
                               activation='relu',
                               bias_initializer=tf.keras.initializers.RandomUniform()),
    tf.keras.layers.Dense(units=1, activation='relu', bias_initializer=tf.keras.initial
    ])

history = compile_and_fit(conv_model, conv_window)

val_performance['Conv'] = conv_model.evaluate(conv_window.val)
performance['Conv'] = conv_model.evaluate(conv_window.test, verbose=0)
```

[truncated]

Display performance metrics - If the model performs better than the baseline, information is leaking -- **we know the future** (to some extent).

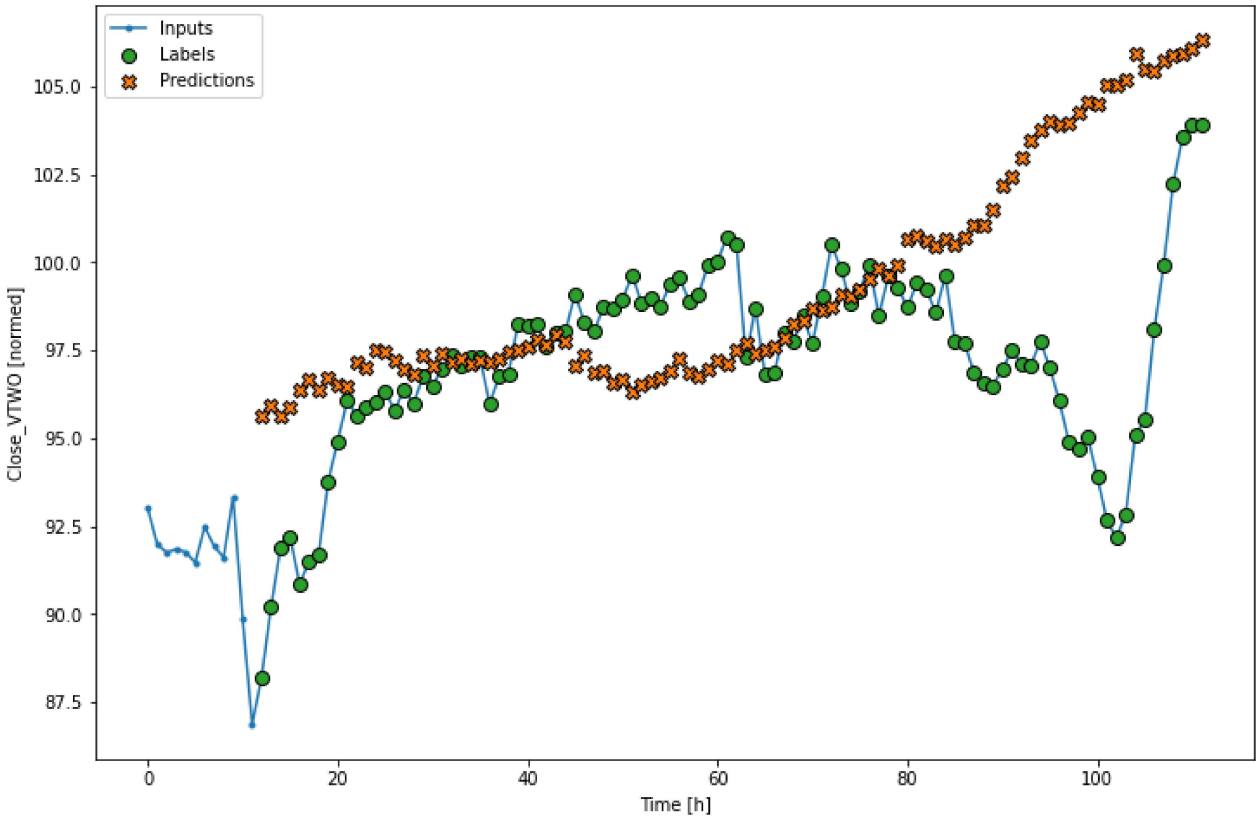
In [15]:
```
print("[total aggregate error, average error per guess (USD)]")
for s in performance:
    print(s, performance[s])
```

```
[total aggregate error, average error per guess (USD)]
Baseline [271.53192138671875, 11.343183517456055]
Conv [321.6611328125, 12.98602294921875]
```

In [16]:
```
import IPython
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt
import math
```

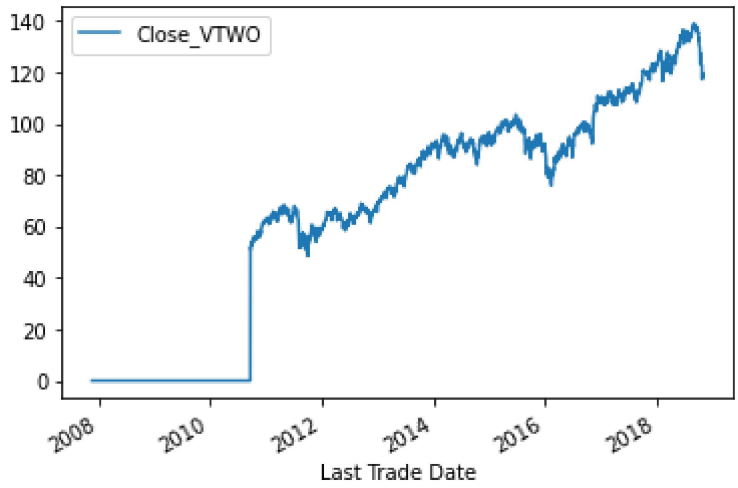
In [17]:
```
LABEL_WIDTH = 100
INPUT_WIDTH = LABEL_WIDTH + 11
wide_conv_window = WindowGenerator(
    input_width=INPUT_WIDTH, label_width=LABEL_WIDTH, shift=1,
    label_columns=[TARGET_TICKER])

print("Wide conv window")
print('Input shape:', wide_conv_window.example[0].shape)
print('Labels shape:', wide_conv_window.example[1].shape)
print('Output shape:', conv_model(wide_conv_window.example[0]).shape)

wide_conv_window.plot(conv_model, plot_col=TARGET_TICKER, max_subplots=1)
```

```
Wide conv window
Input shape: (32, 111, 18093)
Labels shape: (32, 100, 1)
Output shape: (32, 100, 1)
```



In [18]:

```
plt_cols = [TARGET_TICKER]
plot_features = train_df[plt_cols]
plot_features.index = train_df.index
_ = plot_features.plot(subplots=True)
```



Experiment: Part 1: Find the most predictable ticker symbol

We choose the top-50 oldest NASDAQ tickers, and see which of them are most predictable by training a model for each.

In [19]:

```
good_dict = {}
```

```
for col in olddf:
    non_zero_count = 0
    if(col[0:6] == "Close_"):
        for x in olddf[col].array:
            if(x > 0.0):
                non_zero_count += 1
    good_dict[col] = non_zero_count

good_dict = sorted(good_dict.items(), key=lambda x: x[1], reverse=True)

interesting_tickers = []

for i, item in enumerate(good_dict):
    if(i < 50):
        interesting_tickers.append(item[0])
    else:
        break

print(interesting_tickers)
```

```
['Close_PRFZ', 'Close_ADRE', 'Close_IFGL', 'Close_ONEQ', 'Close_QQEW', 'Close_QTEC', 'Cl
ose_QCLN', 'Close_ACWI', 'Close_ACWX', 'Close_QQXT', 'Close_ICLN', 'Close_WOOD', 'Close_
AAXJ', 'Close_IBB', 'Close_PNQI', 'Close_IGOV', 'Close_ISHG', 'Close_ADRD', 'Close_GUL
F', 'Close_EMIF', 'Close_IFEU', 'Close_QABA', 'Close_INDY', 'Close_VCIT', 'Close_VCSH',
'Close_VGSH', 'Close_VCLT', 'Close_VMBS', 'Close_VGIT', 'Close_VGLT', 'Close_SQQQ', 'Clo
se_TQQQ', 'Close_EUFN', 'Close_PSAU', 'Close_BIB', 'Close_VTWO', 'Close_VTWG', 'Close_SO
XX', 'Close_VONE', 'Close_VNQI', 'Close_VONG', 'Close_VONV', 'Close_BIS', 'Close_ADRA',
'Close_VXUS', 'Close_VTWV', 'Close_VTHR', 'Close_PSCH', 'Close_QQQ', 'Close_PSCT']
```

Experiment Part 2: Pick most predictable tickers

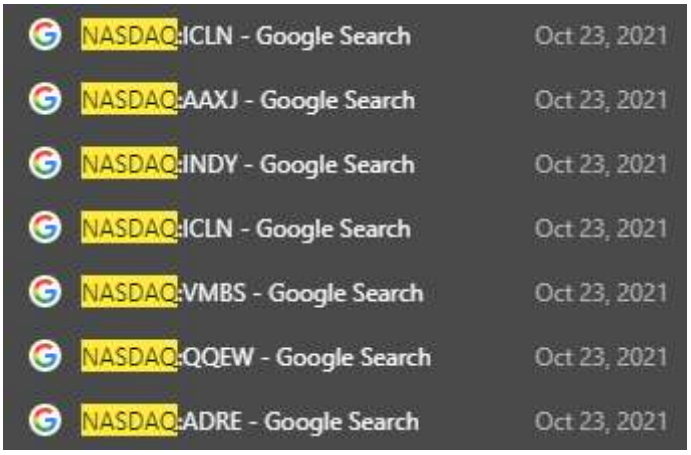
... continued

This builds a dictionary describing the performance of a model across the top 50 "interesting_tickers" identified in Part 1

Note in the modeling it was useful for understanding appropriate geometry for a CNN by using this part of this lecture

Note to self top 2 were:

NASDAQ: IGOV NASDAQ: ICLN



In [20]:

```
# Loop over each ticker identified previously
for TARGET_TICKER in interesting_tickers:
    # First run the baseline
    baseline = Baseline(label_index=column_indices[TARGET_TICKER])

    baseline.compile(loss=tf.losses.MeanSquaredError(),
                     metrics=[tf.metrics.MeanAbsoluteError()])

    val_performance[TARGET_TICKER+'-Baseline'] = baseline.evaluate(single_step_window.v
    performance[TARGET_TICKER+'-Baseline'] = baseline.evaluate(single_step_window.test,
                                            verbose=0)

    # Next run the convolution
    conv_window = WindowGenerator(
        input_width=CONV_WIDTH,
        label_width=1,
        shift=1,
        label_columns=[TARGET_TICKER])
    # https://www.youtube.com/watch?v=bNb2fEVKeEo&?t=26m23s
    conv_model = tf.keras.Sequential([
        tf.keras.layers.GaussianNoise(stddev=1),
        tf.keras.layers.Conv1D(filters=64,
                               kernel_size=(2,),
                               activation='relu',
                                bias_initializer=tf.keras.initializers.RandomUniform()),
        tf.keras.layers.Conv1D(filters=256,
                               kernel_size=(2,),
                               activation='relu',
                               bias_initializer=tf.keras.initializers.RandomUniform()),
            tf.keras.layers.Conv1D(filters=4096,
                               kernel_size=(8,),
                               activation='relu',
                               bias_initializer=tf.keras.initializers.RandomUniform()),
    tf.keras.layers.Dense(units=1, activation='relu', bias_initializer=tf.keras.initial
    ])

    history = compile_and_fit(conv_model, conv_window)

    val_performance[TARGET_TICKER+'-Conv'] = conv_model.evaluate(conv_window.val)
    performance[TARGET_TICKER+'-Conv'] = conv_model.evaluate(conv_window.test, verbose=
```

```
[EXTREMELY TRUNCATED TRAINING OUTPUT]
```

In [21]:

```
print("[total aggregate error, average error per guess (USD)]")
```

```
for s in performance:
    print(s, performance[s])

for s in val_performance:
    print(s, performance[s])
```

```
[total aggregate error, average error per guess (USD)]
Baseline [271.53192138671875, 11.343183517456055]
Conv [321.6611328125, 12.98602294921875]
Close_PRFZ-Baseline [310.10858154296875, 11.910513877868652]
Close_PRFZ-Conv [655.6212768554688, 20.068035125732422]
Close_ADRE-Baseline [6406.072265625, 78.97486877441406]
Close_ADRE-Conv [15.765191078186035, 3.027269124984741]
Close_IFGL-Baseline [8589.80859375, 91.80224609375]
Close_IFGL-Conv [9.006390571594238, 2.6109883785247803]
Close_ONEQ-Baseline [52653.69140625, 226.23898315429688]
Close_ONEQ-Conv [5758.70654296875, 66.14673614501953]
Close_QQEW-Baseline [2438.078857421875, 47.15165328979492]
Close_QQEW-Conv [174.42620849609375, 11.767197608947754]
Close_QTEC-Baseline [798.3406982421875, 25.299665451049805]
Close_QTEC-Conv [811.5105590820312, 25.720561981201172]
Close_QCLN-Baseline [9002.2021484375, 93.79124450683594]
Close_QCLN-Conv [96.72685241699219, 7.4777140617370605]
Close_ACWI-Baseline [2207.299560546875, 45.07790756225586]
Close_ACWI-Conv [41.14529800415039, 5.24696683883667]
Close_ACWX-Baseline [5643.3388671875, 74.06620788574219]
Close_ACWX-Conv [17.38245964050293, 3.5001652240753174]
Close_QQXT-Baseline [4318.61328125, 63.33123779296875]
Close_QQXT-Conv [193.19911193847656, 11.908103942871094]
Close_ICLN-Baseline [11689.6748046875, 107.31248474121094]
Close_ICLN-Conv [14.040369033813477, 3.184950590133667]
Close_WOOD-Baseline [3863.22021484375, 60.76024627685547]
Close_WOOD-Conv [198.5597381591797, 12.633749961853027]
Close_AAXJ-Baseline [2721.39306640625, 50.498191833496094]
Close_AAXJ-Conv [60.83486557006836, 6.616479396820068]
Close_IBB-Baseline [356.5406494140625, 16.93366050720215]
Close_IBB-Conv [215.9587860107422, 12.836917877197266]
Close_PNQI-Baseline [1373.2921142578125, 28.40145492553711]
Close_PNQI-Conv [2026.7447509765625, 36.05331039428711]
Close_IGOV-Baseline [4850.7646484375, 68.47388458251953]
Close_IGOV-Conv [104.44621276855469, 9.919407844543457]
Close_ISHG-Baseline [1943.7364501953125, 41.27352523803711]
Close_ISHG-Conv [105.41488647460938, 7.874069690704346]
Close_ADRD-Baseline [12057.1591796875, 108.80537414550781]
Close_ADRD-Conv [192.66273498535156, 12.479622840881348]
Close_GULF-Baseline [11092.7265625, 104.1585922241211]
Close_GULF-Conv [115.01811981201172, 7.329125881195068]
Close_EMIF-Baseline [8998.423828125, 93.781982421875]
Close_EMIF-Conv [32.27143478393555, 4.560852527618408]
Close_IFEU-Baseline [7292.8330078125, 84.02949523925781]
Close_IFEU-Conv [244.73086547851562, 9.082401275634766]
Close_QABA-Baseline [5977.9892578125, 76.0903091430664]
Close_QABA-Conv [132.90821838378906, 8.886602401733398]
Close_INDY-Baseline [7386.85546875, 84.99234771728516]
Close_INDY-Conv [26.508075714111328, 4.112894535064697]
Close_VCIT-Baseline [907.5214233398438, 27.88328742980957]
Close_VCIT-Conv [50.71247863769531, 6.394037246704102]
Close_VCSH-Baseline [1592.8514404296875, 37.82890319824219]
Close_VCSH-Conv [47.69023132324219, 5.420071125030518]
Close_VGSH-Baseline [3475.66015625, 57.48871994018555]
Close_VGSH-Conv [8.752913475036621, 2.3433570861816406]
```

```
Close_VCLT-Baseline [475.8887023925781, 19.941631317138672]
Close_VCLT-Conv [257.8767395019531, 15.525931358337402]
Close_VMBS-Baseline [4433.3056640625, 65.29832458496094]
Close_VMBS-Conv [10.532672882080078, 2.4868953227996826]
Close_VGIT-Baseline [2766.785888671875, 50.842987060546875]
Close_VGIT-Conv [34.53764724731445, 5.352859020233154]
Close_VGLT-Baseline [994.7594604492188, 27.25550651550293]
Close_VGLT-Conv [574.6965942382812, 22.51681900024414]
Close_SQQQ-Baseline [9742.6015625, 97.66260528564453]
Close_SQQQ-Conv [115.01455688476562, 8.94639778137207]
Close_TQQQ-Baseline [2313.72265625, 43.38814163208008]
Close_TQQQ-Conv [1336.3128662109375, 28.253543853759766]
Close_EUFN-Baseline [10686.1982421875, 102.5885009765625]
Close_EUFN-Conv [24.91954231262207, 4.01464319229126]
Close_PSAU-Baseline [14310.40234375, 118.92787170410156]
Close_PSAU-Conv [184.7440643310547, 13.549272537231445]
Close_BIB-Baseline [4095.812744140625, 61.6713752746582]
Close_BIB-Conv [121.0992202758789, 8.603809356689453]
Close_VTWO-Baseline [271.5318908691406, 11.343182563781738]
Close_VTWO-Conv [240.0479278564453, 10.958015441894531]
Close_VTWG-Baseline [963.0525512695312, 25.776948928833008]
Close_VTWG-Conv [232.15353393554688, 10.60171890258789]
Close_SOXX-Baseline [14732.041015625, 117.23997497558594]
Close_SOXX-Conv [63892.67578125, 250.79296875]
Close_VONE-Baseline [658.3538208007812, 20.113431930541992]
Close_VONE-Conv [299.09130859375, 15.908491134643555]
Close_VNQI-Baseline [4435.1904296875, 65.33409118652344]
Close_VNQI-Conv [128.8120880126953, 9.051608085632324]
Close_VONG-Baseline [3957.080810546875, 58.96491241455078]
Close_VONG-Conv [1593.6839599609375, 35.10276794433594]
Close_VONV-Baseline [362.4069519042969, 17.619646072387695]
Close_VONV-Conv [93.04931640625, 7.152625560760498]
Close_BIS-Baseline [11183.4873046875, 104.9464340209961]
Close_BIS-Conv [55.45903015136719, 5.960760116577148]
Close_ADRA-Baseline [11424.8251953125, 105.20769500732422]
Close_ADRA-Conv [369.44744873046875, 18.83836555480957]
Close_VXUS-Baseline [4817.75537109375, 68.24903869628906]
Close_VXUS-Conv [54.04290008544922, 5.286073684692383]
Close_VTWV-Baseline [814.1851806640625, 26.649803161621094]
Close_VTWV-Conv [370.4032897949219, 14.452120780944824]
Close_VTHR-Baseline [598.492919921875, 18.66722869873047]
Close_VTHR-Conv [259.28485107421875, 14.616181373596191]
Close_PSCH-Baseline [278.5491638183594, 11.74344253540039]
Close_PSCH-Conv [218.06594848632812, 10.179407119750977]
Close_QQQ-Baseline [9707.1708984375, 94.01348876953125]
Close_QQQ-Conv [3438.685791015625, 50.583221435546875]
Close_PSCT-Baseline [1250.4144287109375, 33.583038330078125]
Close_PSCT-Conv [151.78903198242188, 10.880789756774902]
Baseline [271.53192138671875, 11.343183517456055]
Conv [321.6611328125, 12.98602294921875]
Close_PRFZ-Baseline [310.10858154296875, 11.910513877868652]
Close_PRFZ-Conv [655.6212768554688, 20.068035125732422]
Close_ADRE-Baseline [6406.072265625, 78.97486877441406]
Close_ADRE-Conv [15.765191078186035, 3.027269124984741]
Close_IFGL-Baseline [8589.80859375, 91.80224609375]
Close_IFGL-Conv [9.006390571594238, 2.6109883785247803]
Close_ONEQ-Baseline [52653.69140625, 226.23898315429688]
Close_ONEQ-Conv [5758.70654296875, 66.14673614501953]
Close_QQEW-Baseline [2438.078857421875, 47.15165328979492]
Close_QQEW-Conv [174.42620849609375, 11.767197608947754]
Close_QTEC-Baseline [798.3406982421875, 25.299665451049805]
```

```
Close_QTEC-Conv [811.5105590820312, 25.720561981201172]
Close_QCLN-Baseline [9002.2021484375, 93.79124450683594]
Close_QCLN-Conv [96.72685241699219, 7.4777140617370605]
Close_ACWI-Baseline [2207.299560546875, 45.07790756225586]
Close_ACWI-Conv [41.14529800415039, 5.24696683883667]
Close_ACWX-Baseline [5643.3388671875, 74.06620788574219]
Close_ACWX-Conv [17.38245964050293, 3.5001652240753174]
Close_QQXT-Baseline [4318.61328125, 63.33123779296875]
Close_QQXT-Conv [193.19911193847656, 11.908103942871094]
Close_ICLN-Baseline [11689.6748046875, 107.31248474121094]
Close_ICLN-Conv [14.040369033813477, 3.184950590133667]
Close_WOOD-Baseline [3863.22021484375, 60.76024627685547]
Close_WOOD-Conv [198.5597381591797, 12.633749961853027]
Close_AAXJ-Baseline [2721.39306640625, 50.498191833496094]
Close_AAXJ-Conv [60.83486557006836, 6.616479396820068]
Close_IBB-Baseline [356.5406494140625, 16.93366050720215]
Close_IBB-Conv [215.9587860107422, 12.836917877197266]
Close_PNQI-Baseline [1373.2921142578125, 28.40145492553711]
Close_PNQI-Conv [2026.7447509765625, 36.05331039428711]
Close_IGOV-Baseline [4850.7646484375, 68.47388458251953]
Close_IGOV-Conv [104.44621276855469, 9.919407844543457]
Close_ISHG-Baseline [1943.7364501953125, 41.27352523803711]
Close_ISHG-Conv [105.41488647460938, 7.874069690704346]
Close_ADRD-Baseline [12057.1591796875, 108.80537414550781]
Close_ADRD-Conv [192.66273498535156, 12.479622840881348]
Close_GULF-Baseline [11092.7265625, 104.1585922241211]
Close_GULF-Conv [115.01811981201172, 7.329125881195068]
Close_EMIF-Baseline [8998.423828125, 93.781982421875]
Close_EMIF-Conv [32.27143478393555, 4.560852527618408]
Close_IFEU-Baseline [7292.8330078125, 84.02949523925781]
Close_IFEU-Conv [244.73086547851562, 9.082401275634766]
Close_QABA-Baseline [5977.9892578125, 76.0903091430664]
Close_QABA-Conv [132.90821838378906, 8.886602401733398]
Close_INDY-Baseline [7386.85546875, 84.99234771728516]
Close_INDY-Conv [26.508075714111328, 4.112894535064697]
Close_VCIT-Baseline [907.5214233398438, 27.88328742980957]
Close_VCIT-Conv [50.71247863769531, 6.394037246704102]
Close_VCSH-Baseline [1592.8514404296875, 37.82890319824219]
Close_VCSH-Conv [47.69023132324219, 5.420071125030518]
Close_VGSH-Baseline [3475.66015625, 57.48871994018555]
Close_VGSH-Conv [8.752913475036621, 2.3433570861816406]
Close_VCLT-Baseline [475.8887023925781, 19.941631317138672]
Close_VCLT-Conv [257.8767395019531, 15.525931358337402]
Close_VMBS-Baseline [4433.3056640625, 65.29832458496094]
Close_VMBS-Conv [10.532672882080078, 2.4868953227996826]
Close_VGIT-Baseline [2766.785888671875, 50.842987060546875]
Close_VGIT-Conv [34.53764724731445, 5.352859020233154]
Close_VGLT-Baseline [994.7594604492188, 27.25550651550293]
Close_VGLT-Conv [574.6965942382812, 22.51681900024414]
Close_SQQQ-Baseline [9742.6015625, 97.66260528564453]
Close_SQQQ-Conv [115.01455688476562, 8.94639778137207]
Close_TQQQ-Baseline [2313.72265625, 43.38814163208008]
Close_TQQQ-Conv [1336.3128662109375, 28.253543853759766]
Close_EUFN-Baseline [10686.1982421875, 102.5885009765625]
Close_EUFN-Conv [24.91954231262207, 4.01464319229126]
Close_PSAU-Baseline [14310.40234375, 118.92787170410156]
Close_PSAU-Conv [184.7440643310547, 13.549272537231445]
Close_BIB-Baseline [4095.812744140625, 61.6713752746582]
Close_BIB-Conv [121.0992202758789, 8.603809356689453]
Close_VTWO-Baseline [271.5318908691406, 11.343182563781738]
Close_VTWO-Conv [240.0479278564453, 10.958015441894531]
```

```
Close_VTWG-Baseline [963.0525512695312, 25.776948928833008]
Close_VTWG-Conv [232.15353393554688, 10.60171890258789]
Close_SOXX-Baseline [14732.041015625, 117.23997497558594]
Close_SOXX-Conv [63892.67578125, 250.79296875]
Close_VONE-Baseline [658.3538208007812, 20.113431930541992]
Close_VONE-Conv [299.09130859375, 15.908491134643555]
Close_VNQI-Baseline [4435.1904296875, 65.33409118652344]
Close_VNQI-Conv [128.8120880126953, 9.051608085632324]
Close_VONG-Baseline [3957.080810546875, 58.96491241455078]
Close_VONG-Conv [1593.6839599609375, 35.10276794433594]
Close_VONV-Baseline [362.4069519042969, 17.619646072387695]
Close_VONV-Conv [93.04931640625, 7.152625560760498]
Close_BIS-Baseline [11183.4873046875, 104.9464340209961]
Close_BIS-Conv [55.45903015136719, 5.960760116577148]
Close_ADRA-Baseline [11424.8251953125, 105.20769500732422]
Close_ADRA-Conv [369.44744873046875, 18.83836555480957]
Close_VXUS-Baseline [4817.75537109375, 68.24903869628906]
Close_VXUS-Conv [54.04290008544922, 5.286073684692383]
Close_VTWV-Baseline [814.1851806640625, 26.649803161621094]
Close_VTWV-Conv [370.4032897949219, 14.452120780944824]
Close_VTHR-Baseline [598.492919921875, 18.66722869873047]
Close_VTHR-Conv [259.28485107421875, 14.616181373596191]
Close_PSCH-Baseline [278.5491638183594, 11.74344253540039]
Close_PSCH-Conv [218.06594848632812, 10.179407119750977]
Close_QQQ-Baseline [9707.1708984375, 94.01348876953125]
Close_QQQ-Conv [3438.685791015625, 50.583221435546875]
Close_PSCT-Baseline [1250.4144287109375, 33.583038330078125]
Close_PSCT-Conv [151.78903198242188, 10.880789756774902]
```

In [22]:

```
print("validation performance ratio; lower is better -- convolution outperforms assumpt
perf_dict_val = {}
for s in val_performance:
    if(s.find("Baseline") != -1 and s.find("Close") != -1):
        print(s.replace("-Baseline",""), val_performance[s.replace("Baseline","Conv")][
        perf_dict_val[s.replace("-Baseline","")] = val_performance[s.replace("Baseline"

print("test performance ratio; lower is better -- convolution outperforms assumption")
print("NOTE: This data was NOT utilized for termination conditions and should be consid
perf_dict_test = {}
for s in performance:
    if(s.find("Baseline") != -1 and s.find("Close") != -1):
        print(s.replace("-Baseline",""), performance[s.replace("Baseline","Conv")][1] /
        perf_dict_test[s.replace("-Baseline","")] = performance[s.replace("Baseline","C
```

```
validation performance ratio; lower is better -- convolution outperforms assumption
Close_PRFZ 0.9838947723963345
Close_ADRE 0.03079740220508842
Close_IFGL 0.030391666686311977
Close_ONEQ 0.1284384106330237
Close_QQEW 0.08580943511024851
Close_QTEC 0.22060529345578644
Close_QCLN 0.017389843297132668
Close_ACWI 0.04182557695346402
Close_ACWX 0.0318573556245935
Close_QQXT 0.10024232944689015
Close_ICLN 0.010351885221059301
Close_WOOD 0.11705212137803861
Close_AAXJ 0.0929426441290251
Close_IBB 0.3293740520408639
Close_PNQI 1.5670362303216643
```

```
Close_IGOV 0.04505675901185494
Close_ISHG 0.13856584559900895
Close_ADRD 0.054721981852768074
Close_GULF 0.025823133308490124
Close_EMIF 0.07897477092566843
Close_IFEU 0.06225427080994323
Close_QABA 0.021094457417456237
Close_INDY 0.044989879558688085
Close_VCIT 0.12622285908800585
Close_VCSH 0.05426719460103995
Close_VGSH 0.05970668511114414
Close_VCLT 0.24373122863579758
Close_VMBS 0.06634588683712644
Close_VGIT 0.061174525007341975
Close_VGLT 0.12588652174957782
Close_SQQQ 0.07229545788023654
Close_TQQQ 0.09627782557264222
Close_EUFN 0.009501663516309483
Close_PSAU 0.11709135347059794
Close_BIB 0.11893864134774866
Close_VTWO 1.0197652887689108
Close_VTWG 0.3478294682432173
Close_SOXX 3.4700136369754775
Close_VONE 1.5506618943794344
Close_VNQI 0.07023890845199934
Close_VONG 0.4106862203305916
Close_VONV 0.32169994620301307
Close_BIS 0.012464676003416284
Close_ADRA 0.13597394013497416
Close_VXUS 0.035044603036190024
Close_VTWV 0.1813822774650577
Close_VTHR 1.190716819791917
Close_PSCH 0.9500585539868441
Close_QQQ 0.23461454009734875
Close_PSCT 0.11980700433548763
test performance ratio; lower is better -- convolution outperforms assumption
NOTE: This data was NOT utilized for termination conditions and should be considered a b
etter evaluation of performance.
Close_PRFZ 1.684900864187023
Close_ADRE 0.038332056411919016
Close_IFGL 0.028441443315650412
Close_ONEQ 0.2923755014400277
Close_QQEW 0.2495606577488671
Close_QTEC 1.016636446476564
Close_QCLN 0.07972720802518032
Close_ACWI 0.11639774609303301
Close_ACWX 0.04725724893968957
Close_QQXT 0.18802891523767395
Close_ICLN 0.02967921764009401
Close_WOOD 0.207927892594231
Close_AAXJ 0.1310240853501466
Close_IBB 0.7580710544974919
Close_PNQI 1.2694177283808743
Close_IGOV 0.14486410264323968
Close_ISHG 0.19077773573476375
Close_ADRD 0.11469675040308279
Close_GULF 0.07036506278257652
Close_EMIF 0.04863250285221708
Close_IFEU 0.1080858721068641
Close_QABA 0.1167901997220782
Close_INDY 0.048391351051339945
```

```
Close_VCIT 0.22931432539293556
Close_VCSH 0.1432785692100736
Close_VGSH 0.04076203277129495
Close_VCLT 0.7785687695967866
Close_VMBS 0.03808513217768603
Close_VGIT 0.10528215059155846
Close_VGLT 0.8261383433632604
Close_SQQQ 0.09160515178973118
Close_TQQQ 0.6511812396424423
Close_EUFN 0.03913346187998643
Close_PSAU 0.11392848743600413
Close_BIB 0.13951058036847286
Close_VTWO 0.9660441750168927
Close_VTWG 0.41128680247836685
Close_SOXX 2.1391421211257096
Close_VONE 0.7909386717085667
Close_VNQI 0.13854341464383013
Close_VONG 0.5953162059759733
Close_VONV 0.40594604065115725
Close_BIS 0.056798119652017996
Close_ADRA 0.17905881840200097
Close_VXUS 0.07745272000409589
Close_VTWV 0.542297468138812
Close_VTHR 0.7829861416220939
Close_PSCH 0.8668162754716381
Close_QQQ 0.5380421692417859
Close_PSCT 0.32399658571183215
```

In [28]:

```
#perf_dict_val = sorted(perf_dict_val.items(), key=lambda x: x[1], reverse=False)
#perf_dict_test = sorted(perf_dict_test.items(), key=lambda x: x[1], reverse=False)

print("Most trainable tickers (in validation):")
for i, item in enumerate(perf_dict_val):
    if(i < 20):
        print(item)
    else:
        break

print("*****Most trainable tickers (in testing):")
for i, item in enumerate(perf_dict_test):
    if(i < 20):
        print(item)
    else:
        break
```

```
Most trainable tickers (in validation):
('Close_EUFN', 0.009501663516309483)
('Close_ICLN', 0.010351885221059301)
('Close_BIS', 0.012464676003416284)
('Close_QCLN', 0.017389843297132668)
('Close_QABA', 0.021094457417456237)
('Close_GULF', 0.025823133308490124)
('Close_IFGL', 0.030391666686311977)
('Close_ADRE', 0.03079740220508842)
('Close_ACWX', 0.0318573556245935)
('Close_VXUS', 0.035044603036190024)
('Close_ACWI', 0.04182557695346402)
('Close_INDY', 0.044989879558688085)
('Close_IGOV', 0.04505675901185494)
```

```
('Close_VCSH', 0.05426719460103995)
('Close_ADRD', 0.054721981852768074)
('Close_VGSH', 0.05970668511114414)
('Close_VGIT', 0.061174525007341975)
('Close_IFEU', 0.06225427080994323)
('Close_VMBS', 0.06634588683712644)
('Close_VNQI', 0.07023890845199934)
*****Most trainable tickers (in testing):
('Close_IFGL', 0.028441443315650412)
('Close_ICLN', 0.02967921764009401)
('Close_VMBS', 0.03808513217768603)
('Close_ADRE', 0.038332056411919016)
('Close_EUFN', 0.03913346187998643)
('Close_VGSH', 0.04076203277129495)
('Close_ACWX', 0.04725724893968957)
('Close_INDY', 0.048391351051339945)
('Close_EMIF', 0.04863250285221708)
('Close_BIS', 0.056798119652017996)
('Close_GULF', 0.07036506278257652)
('Close_VXUS', 0.07745272000409589)
('Close_QCLN', 0.07972720802518032)
('Close_SQQQ', 0.09160515178973118)
('Close_VGIT', 0.10528215059155846)
('Close_IFEU', 0.1080858721068641)
('Close_PSAU', 0.11392848743600413)
('Close_ADRD', 0.11469675040308279)
('Close_ACWI', 0.11639774609303301)
('Close_QABA', 0.1167901997220782)
```

DEAD CODE: Interaction with Elasticsearch Data Lake

The following code is not active, since it is surrounded as a comment, but the gist of it is to access data scalably -- since we may not want to create a PICKLE for each dataset pulled in, nor use the entire dataset for training.

It performs data cleaning for nasdaq data as it was pulled on AWS.

In [24]:
```
"""
# Connect to ES Cluster
brain = es.Elasticsearch(host="172.16.1.39")

# Establish ES scanner to extract all stored data, specific columns
brain_scan = elasticsearch.helpers.scan(brain,
                                        index='nasdaq-traders-test-pull',
                                        query={
                                        "_source" : ["Close", "Symbol",
                                        "Last Trade Date"]
                                        })

# Establish an empty list
db_list = []

print("Querying data from elasticsearch instance...")

# Pull all queried data into empty list
```

```
for res in brain_scan:
    db_list.append(res['_source'])

print("pulled ", len(db_list), " records from ES")

df = pd.DataFrame(db_list, copy=True)

df['Close'] = df['Close'].astype(np.float32)

df = pd.get_dummies(df, prefix='Close_', prefix_sep='', columns=['Symbol'], dtype=np.fl
df = df.mask( cond = df == 1.0, other = df['Close'], axis=0 )

# This stupid shit that doesn't work right
df = df.groupby('Last Trade Date').agg([np.max], dtype=np.float32)

# Probably a fucking workaround for "This stupid shit that doesn't work right"
df.columns = df.columns.get_level_values(0)
df.index = pd.DatetimeIndex(df.index)
df = df.sort_index()
df = df[ df['Close'] >= 0 ]

# Verify the data looks as expected
print(df.describe())

#
date_time = pd.to_datetime(df.index, format='%d.%m.%Y %H:%M:%S')

print(df.head())
print(df.describe().transpose())

timestamp_s = date_time.map(datetime.datetime.timestamp)

day = 24*60*60
month = (30.42)*day
year = (265.2425)*day

# Append consistent features representing periodicity for learning
df['Day sin'] = np.sin(timestamp_s * (2 * np.pi / day))
df['Day cos'] = np.cos(timestamp_s * (2 * np.pi / day))

df['Month sin'] = np.sin(timestamp_s * (2 * np.pi / month))
df['Month cos'] = np.cos(timestamp_s * (2 * np.pi / month))

df['Quarterly sin'] = np.sin(timestamp_s * (2 * np.pi / month))
df['Quarterly cos'] = np.cos(timestamp_s * (2 * np.pi / month))

df['Year sin'] = np.sin(timestamp_s * (2 * np.pi / year))
df['Year cos'] = np.cos(timestamp_s * (2 * np.pi / year))


df.to_pickle("./ELASTIC.pickle")

print("elastic data pickled")
"""
```

Out[24]:

```
'\n# Connect to ES Cluster\nbrain = es.Elasticsearch(host="172.16.1.39")\n\n# Establish
ES scanner to extract all stored data, specific columns\nbrain_scan = elasticsearch.help
ers.scan(brain, \n                                          index=\'nasdaq-traders-test-pu
ll\', \n                                    query={ \n
"_source" : ["Close", "Symbol",\n                                                "Last Trade Dat
e"] \n                                    })\n\n# Establish an empty list\ndb_list =
```

```
[]\n\nprint("Querying data from elasticsearch instance...")\n\n# Pull all queried data i
nto empty list\nfor res in brain_scan:\n    db_list.append(res[\'_source\'])\n\nprint("p
ulled ", len(db_list), " records from ES")\n\ndf = pd.DataFrame(db_list, copy=True)\n\nd
f[\'Close\'] = df[\'Close\'].astype(np.float32)\n\ndf = pd.get_dummies(df, prefix=\'Clos
e_\', prefix_sep=\'\', columns=[\'Symbol\'], dtype=np.float32)\ndf = df.mask( cond = df
== 1.0, other = df[\'Close\'], axis=0 )\n\n# This stupid shit that doesn\'t work right\n
df = df.groupby(\'Last Trade Date\').agg([np.max], dtype=np.float32)\n\n# Probably a fuc
king workaround for "This stupid shit that doesn\'t work right"\ndf.columns = df.column
s.get_level_values(0)\ndf.index = pd.DatetimeIndex(df.index)\ndf = df.sort_index()\ndf =
df[ df[\'Close\'] >= 0 ]\n\n# Verify the data looks as expected\nprint(df.describe())\n
\n# \ndate_time = pd.to_datetime(df.index, format=\'%d.%m.%Y %H:%M:%S\')\n\nprint(df.hea
d())\nprint(df.describe().transpose())\n\ntimestamp_s = date_time.map(datetime.datetime.
timestamp)\n\nday = 24*60*60\nmonth = (30.42)*day\nyear = (265.2425)*day\n\n# Append con
sistent features representing periodicity for learning\ndf[\'Day sin\'] = np.sin(timesta
mp_s * (2 * np.pi / day))\ndf[\'Day cos\'] = np.cos(timestamp_s * (2 * np.pi / day))\n\n
df[\'Month sin\'] = np.sin(timestamp_s * (2 * np.pi / month))\ndf[\'Month cos\'] = np.co
s(timestamp_s * (2 * np.pi / month))\n\ndf[\'Quarterly sin\'] = np.sin(timestamp_s * (2
* np.pi / month))\ndf[\'Quarterly cos\'] = np.cos(timestamp_s * (2 * np.pi / month))\n\n
df[\'Year sin\'] = np.sin(timestamp_s * (2 * np.pi / year))\ndf[\'Year cos\'] = np.cos(t
imestamp_s * (2 * np.pi / year))\n\n\ndf.to_pickle("./ELASTIC.pickle")\n\nprint("elastic
data pickled")\n'
```

Simple Linear Model

In [25]:

```
"""linear = tf.keras.Sequential([
    tf.keras.layers.Dense(units=1)
])
"""
#history = compile_and_fit(linear, single_step_window)

#val_performance['Linear'] = linear.evaluate(single_step_window.val)
#performance['Linear'] = linear.evaluate(single_step_window.test, verbose=0)
```

Out[25]:

```
'linear = tf.keras.Sequential([\n    tf.keras.layers.Dense(units=1)\n])\n'
```

Dense single-step model

In [26]:

```
"""dense = tf.keras.Sequential([
    tf.keras.layers.Dense(units=64, activation='relu'),
    tf.keras.layers.Dense(units=64, activation='relu'),
    tf.keras.layers.Dense(units=1)
    ])
"""
#history = compile_and_fit(dense, single_step_window)

#val_performance['Dense'] = dense.evaluate(single_step_window.val)
#performance['Dense'] = dense.evaluate(single_step_window.test, verbose=0)
```

Out[26]:

```
"dense = tf.keras.Sequential([\n    tf.keras.layers.Dense(units=64, activation='rel
u'),\n    tf.keras.layers.Dense(units=64, activation='relu'),\n    tf.keras.layers.Dense
(units=1)\n    ])\n"
```

Multi-step Dense

In [27]:
```
"""conv_window = WindowGenerator(
    input_width=CONV_WIDTH,
    label_width=1,
    shift=1,
    label_columns=[TARGET_TICKER])

multi_step_dense = tf.keras.Sequential([
    # Shape: (time, features) => (time*features)
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=256, activation='relu'),
    tf.keras.layers.Dense(units=128, activation='relu'),
    tf.keras.layers.Dense(units=1),
    #Add back time dimension
    tf.keras.layers.Reshape([1, -1]),
    ])
"""
#history = compile_and_fit(multi_step_dense, conv_window)

#val_performance['Multi step dense'] = multi_step_dense.evaluate(conv_window.val)
#performance['Multi step dense'] = multi_step_dense.evaluate(conv_window.test, verbose=
```

Out[27]:
```
"conv_window = WindowGenerator(\n    input_width=CONV_WIDTH,\n    label_width=1,\n    sh
ift=1,\n    label_columns=[TARGET_TICKER])\n\nmulti_step_dense = tf.keras.Sequential([\n
# Shape: (time, features) => (time*features)\n    tf.keras.layers.Flatten(),\n    tf.ker
as.layers.Dense(units=256, activation='relu'),\n    tf.keras.layers.Dense(units=128, act
ivation='relu'),\n    tf.keras.layers.Dense(units=1),\n    #Add back time dimension\n
tf.keras.layers.Reshape([1, -1]),\n    ])\n"
```