EE 689 - Deep Learning Neural Networks

Investigating exchange rate prediction models

Mert Sarıkaya
Emre Kara
M. Fikret Yalçınbaş

1. Problem Definition

Forecasting of currency rates is an important task in the finance community. Investment decisions and portfolio managements are highly guided by forecasting studies. Graham Elliott [1] defines ”a good forecast” as a forecast which yields ”low expected loss” in decision making processes. Therefore a good forecast is the foundation of accurate economic policies.

In our study, we analyzed the EUR/USD exchange rate from January 2017 to December 2020 [2] and tried to make predictions based on our analysis. The EUR/USD exchange rate was chosen due to its relatively stable nature compared to overall exchange rates. Also these two currencies are ones which have most transaction volumes and therefore we considered EUR/USD exchange rate as a good source to have a generalized model.

In the beginning of the study, we obtained several statistical models to use as a baseline. Later, we built dense neural network (DNN) and recurrent neural network (RNN) models using Keras packages. And finally, we utilized more advanced algorithms like DeepAR and Neural Prophet using GluonTS library of Amazon [3]. The study has been an overall comparison of different Deep Learning approaches on EUR/USD and evaluating performances of them.

2. Descriptive Analysis and Statistical Models

As our data is of time series, we first examined auto-correlation and partial auto-correlation plots. As expected, auto-correlation values of the variables are significantly large in all visible lag values in a decreasing order. We also checked histogram of the series and we have seen that they were distributed close to normal.

Our initial baseline models were AR and Auto ARIMA models [4]. We have obtained 0.006 RMSE and 0.004 MAPE for AR model and 0.008 RMSE and 0.006 MAPE for ARIMA model. We used these models to evaluate performance of our deep learning models.

3. ArNet with Keras

We created a dense neural network (DNN) model using lag values of the data inspiring from AR-Net algorithm [5] and tried to predict next seven business days. Models with different depths, widths and input sizes were tried. Optimum model obtained processes variables from lag1 to lag7 which has 3 layers with 150, 250 and 300 units. Model is trained with 512 batch size, 0.01 learning rate and 200 epochs. Training and test predictions can be seen in the figure. The performance of the model on test set is:

- RMSE: 0.03077
- MAPE: 1.20083

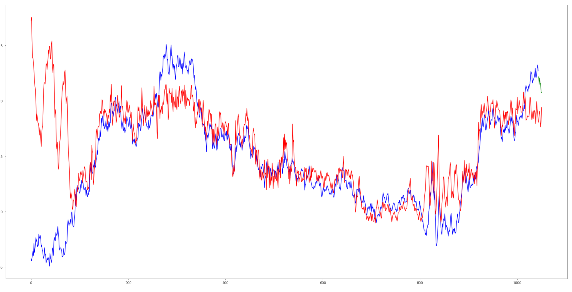


Fig. 1. Predictions of AR-Net from Scratch Model

4. LSTM

We used an LSTM based network to compute the final price of stock given the highs, lows, and opening prices [6]. This is not prediction in the sense that it requires data for a day in order to yield a result, but it does show that the daily fluctuation does follow a pattern that can be used to determine the final price. Scores for LSTM: Training 0.33 RMSE, Testing 0.30 RMSE.

Structure: Input $\rightarrow LSTM \rightarrow Dropout \rightarrow LSTM \rightarrow Dropout \rightarrow Fully-Connected(Dense)$

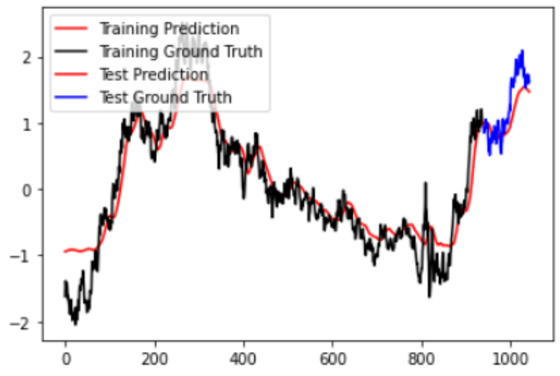


Fig. 2. Full prediction

5. GluonTS Models

GluonTS is a probabilistic time series modelling, built around Apache MXNet. Using Amazon's GluonTS toolkit, we tried four different models [3]. First, as a quick baseline model, we built a simple feed forward model (simple FF). It generates a forecast with RMSE of 0.03, it is worse than our baseline forecasts but a good start.

DeepAR model is based on training an auto-regressive recurrent network model on a large number of related time series. It tries to learn a sparse auto regressive coefficients with a globally shared parameter. This model is implemented in GluonTS and we train this

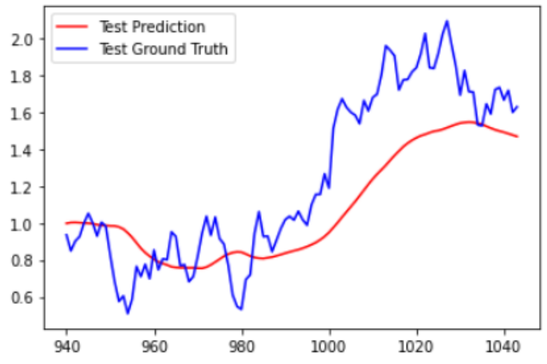


Fig. 3. Training and test predictions

model by giving EUR/USD time series only, and it actually gives the best result among all the other methods for selected test period.

The other method is Deep State Space model, shortly Deep State, this model is developed by Amazon and implemented in this GluonTS library. The model is based on classic state space models, and tries to find the state space parameters. According to our observations, if the hyperparameters of the deep state models did not select wisely, this model results in very poor forecasts. But after some iterations, we managed to make it work.

Our next method is Deep Factor and it is an implementation of Deep Factors for Forecasting [7] Final method is an implementation of AR-Net, the model is implemented in Neural Prophet library, and we trained a network to learn AR-Net parameters with an additive seasonality. The accuracies of the all the models can be seen in the scores table.

6. Results and Comparison

Type	AR	ARIMA	ARnet Keras	Simple FF	DeepAR
MAPE	0.004	0.006	0.008674	0.021455	0.003445
RMSE	0.006	0.008	0.012147	0.035463	0.004410
Type	DeepState	DeepFactor	NeuralProphet	LSTM	
MAPE	0.034916	0.059644	0.016897	-	
RMSE	0.043060	0.074684	0.022691	0.30	

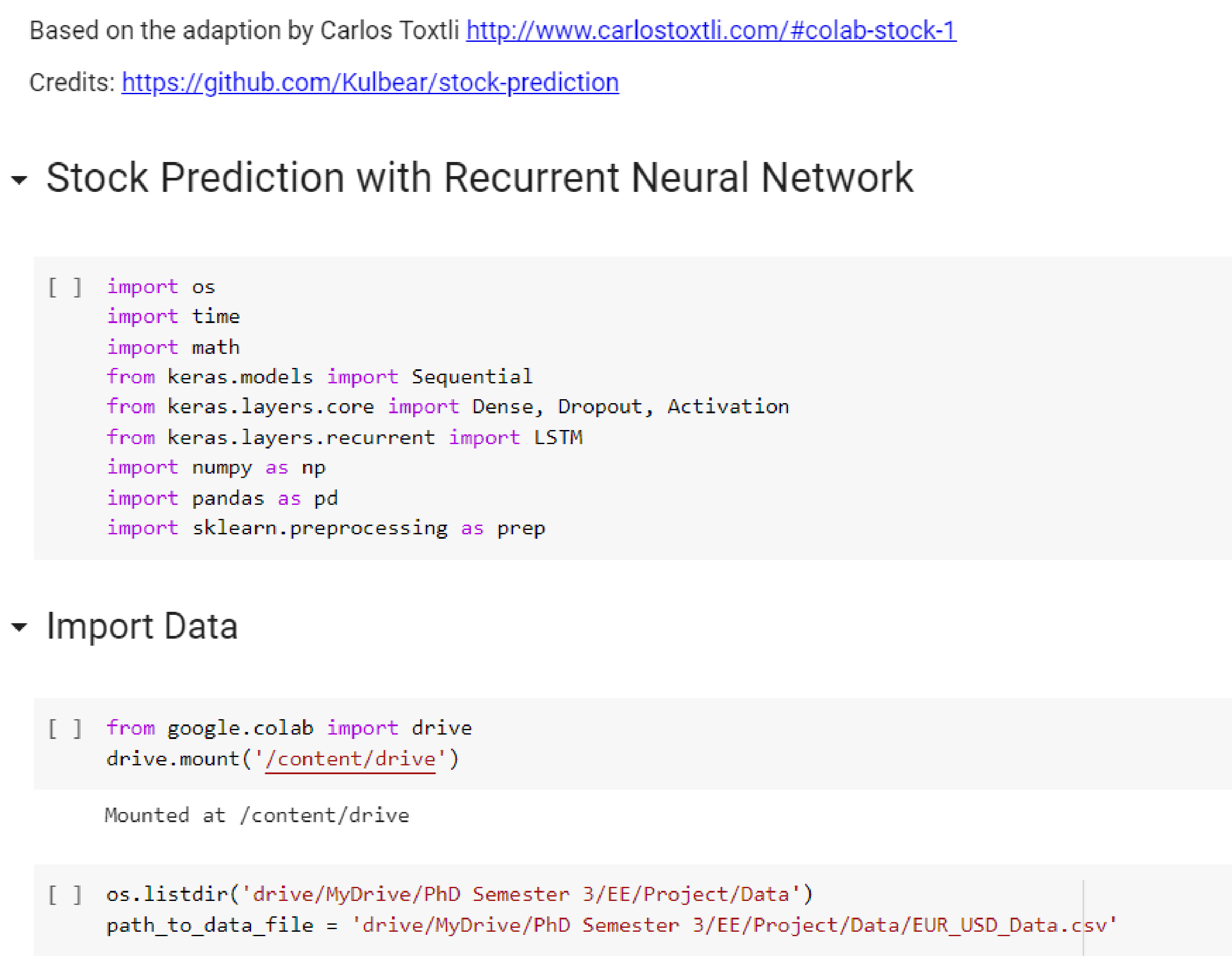
Table 1. Scores

Deep learning models of the study have shown different performances. We have obtained best performing model using DeepAR algorithm which resulted in 0.004 root mean squared error. Other than DeepAR model, ARNet-Keras and DeepState models performed close to statistical models however they did not produce test results better than AR and ARIMA models either. The experiments have shown that DeepAR model has a certain capacity in explaining time series financial assets and performs better on this data than other algorithms investigated.

7. Future Work

To bring further the study, more parameter tuning work can be done on the models. As the effort to obtain better performing hyper-parameters accumulates, chance to develop more accurate algorithms will increase. Also, extending the study period could yield better results as with more data, algorithms learn more about the nature of the process. Finally, the problem can be analyzed as a classification problem as a second approach. The values of change percentages of the exchange rate can be categorized into some number of bins and algorithms can be used to predict classes of the incoming values.

8. Notebooks



Based on the adaption by Carlos Toxtli http://www.carlostoxtli.com/#colab-stock-1

Credits: https://github.com/Kulbear/stock-prediction

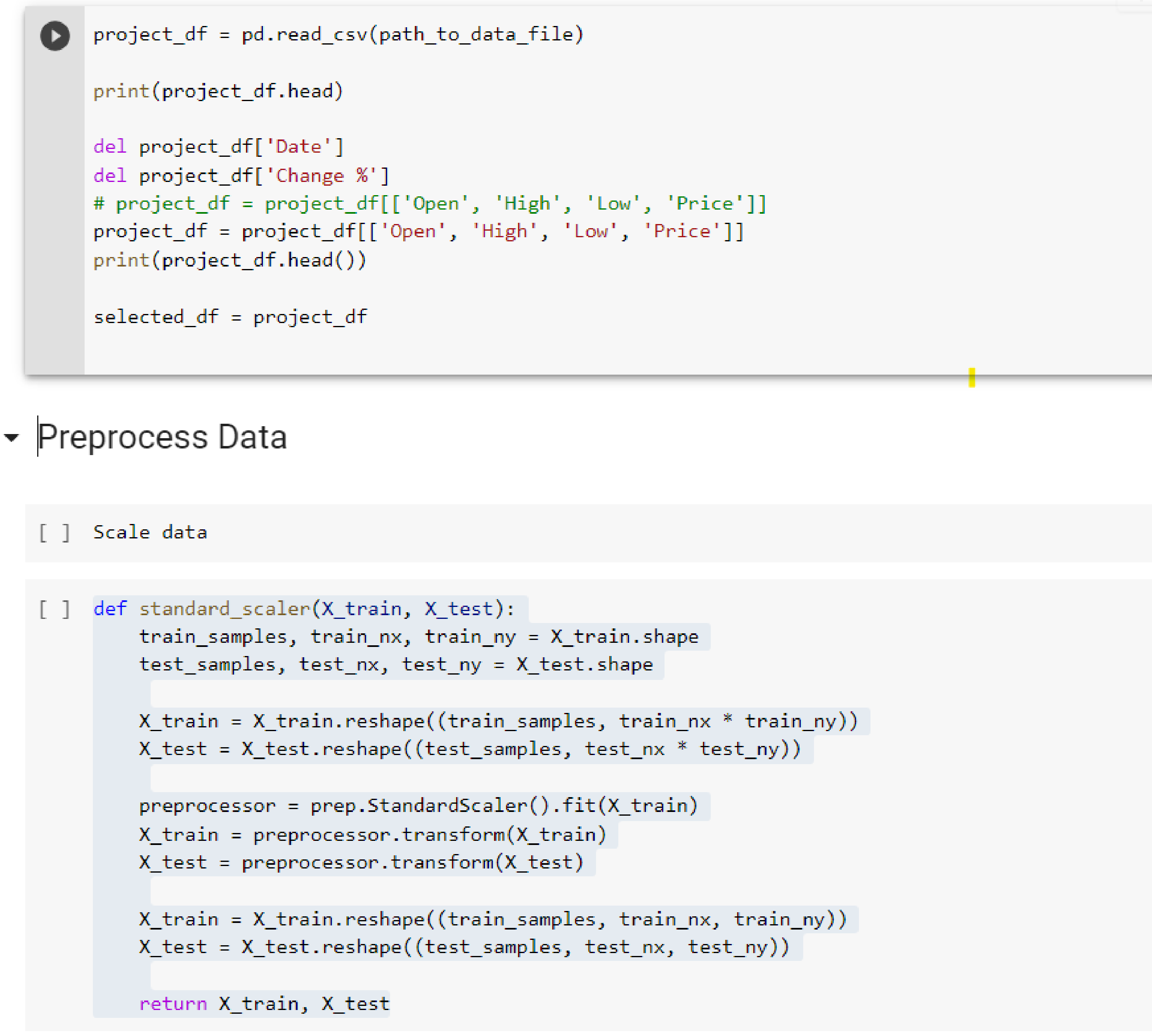
Stock Prediction with Recurrent Neural Network

```
[ ] import os
    import time
    import math
    from keras.models import Sequential
    from keras.layers.core import Dense, Dropout, Activation
    from keras.layers.recurrent import LSTM
    import numpy as np
    import pandas as pd
    import sklearn.preprocessing as prep
```

Import Data

```
[ ] from google.colab import drive
    drive.mount('/content/drive')

    Mounted at /content/drive
```

```
[ ] os.listdir('drive/MyDrive/PhD Semester 3/EE/Project/Data')
    path_to_data_file = 'drive/MyDrive/PhD Semester 3/EE/Project/Data/EUR_USD_Data.csv'
```



```
project_df = pd.read_csv(path_to_data_file)

print(project_df.head)

del project_df['Date']
del project_df['Change %']
# project_df = project_df[['Open', 'High', 'Low', 'Price']]
project_df = project_df[['Open', 'High', 'Low', 'Price']]
print(project_df.head())

selected_df = project_df
```

Preprocess Data

```
[ ]  Scale data
```

```
[ ]  def standard_scaler(X_train, X_test):
         train_samples, train_nx, train_ny = X_train.shape
         test_samples, test_nx, test_ny = X_test.shape

         X_train = X_train.reshape((train_samples, train_nx * train_ny))
         X_test = X_test.reshape((test_samples, test_nx * test_ny))

         preprocessor = prep.StandardScaler().fit(X_train)
         X_train = preprocessor.transform(X_train)
         X_test = preprocessor.transform(X_test)

         X_train = X_train.reshape((train_samples, train_nx, train_ny))
         X_test = X_test.reshape((test_samples, test_nx, test_ny))

         return X_train, X_test
```

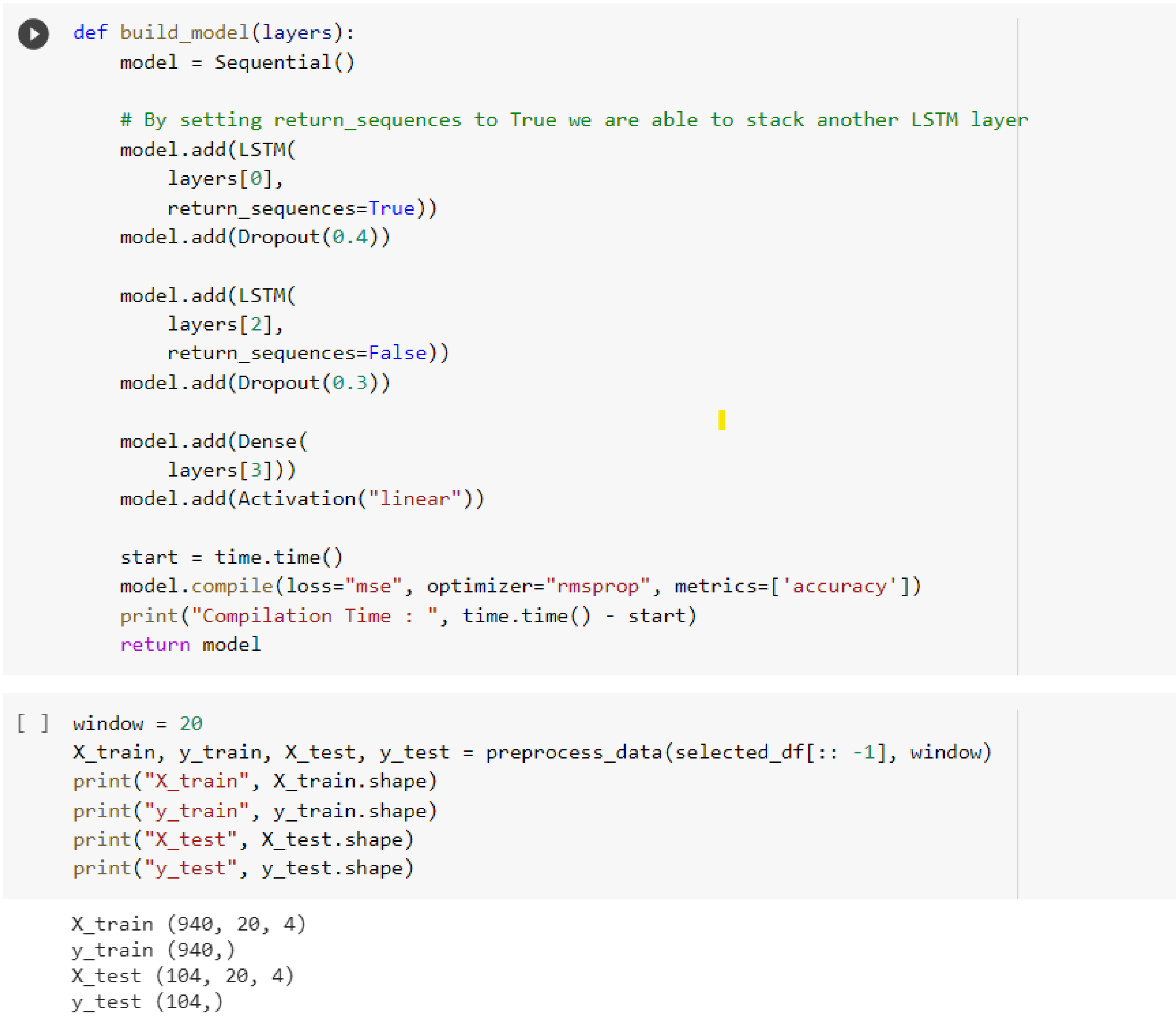
Split data

```
def preprocess_data(stock, seq_len):
    amount_of_features = len(stock.columns)
    data = stock.values

    sequence_length = seq_len + 1
    result = []
    for index in range(len(data) - sequence_length):
        result.append(data[index : index + sequence_length])

    result = np.array(result)
    row = round(0.9 * result.shape[0])
    train = result[: int(row), :]

    train, result = standard_scaler(train, result)

    X_train = train[:, : -1]
    y_train = train[:, -1][: ,-1]
    X_test = result[int(row) :, : -1]
    y_test = result[int(row) :, -1][ : ,-1]

    X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], amount_of_features))
    X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], amount_of_features))

    return [X_train, y_train, X_test, y_test]
```

Build the LSTM Network

Here we will build a simple RNN with 2 LSTM layers. The architecture is:

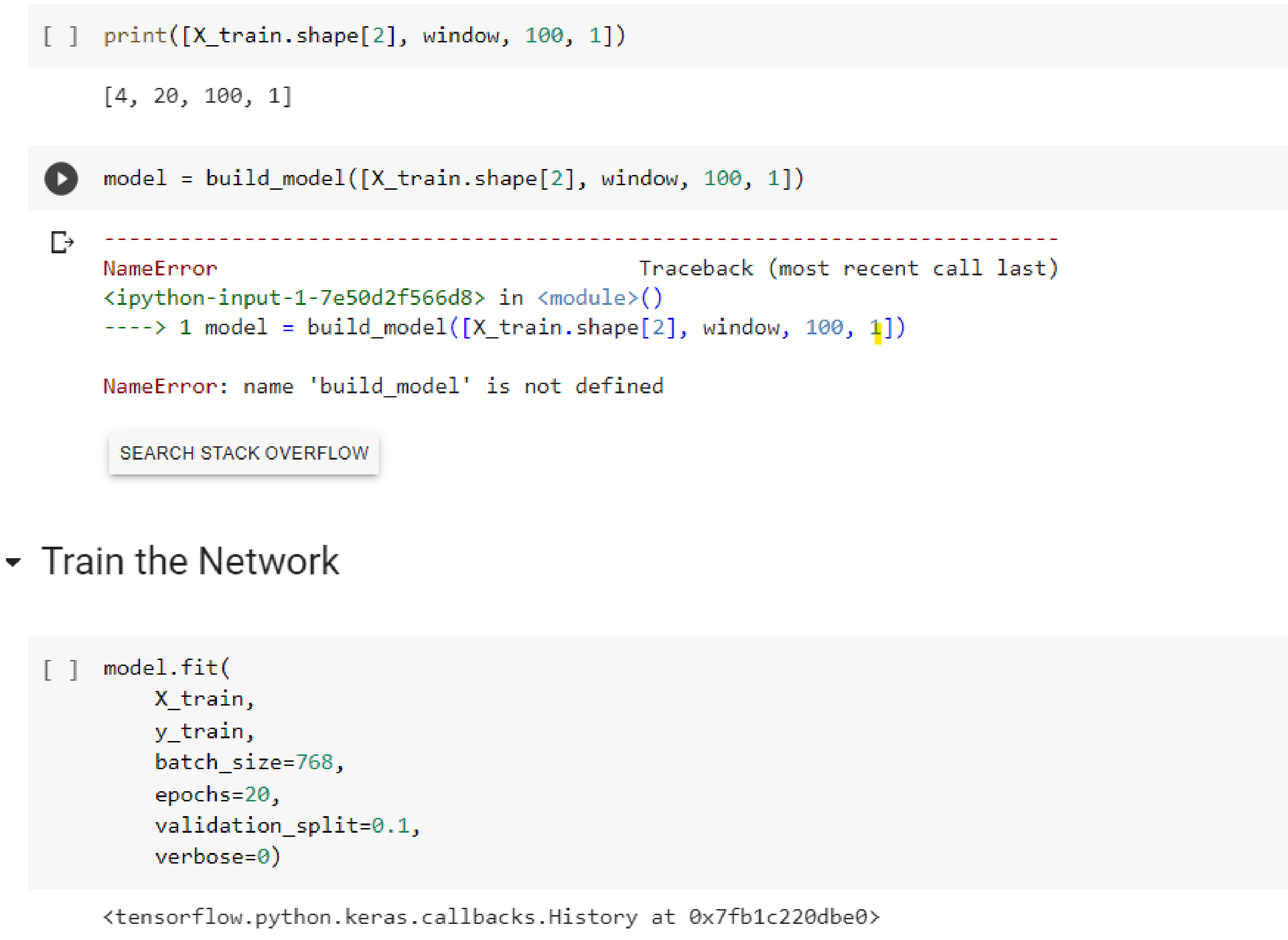
```
LSTM --> Dropout --> LSTM --> Dropout --> Fully-Connected(Dense)
```



```
def build_model(layers):
    model = Sequential()

    # By setting return_sequences to True we are able to stack another LSTM layer
    model.add(LSTM(
        layers[0],
        return_sequences=True))
    model.add(Dropout(0.4))

    model.add(LSTM(
        layers[2],
        return_sequences=False))
    model.add(Dropout(0.3))

    model.add(Dense(
        layers[3]))
    model.add(Activation("linear"))

    start = time.time()
    model.compile(loss="mse", optimizer="rmsprop", metrics=['accuracy'])
    print("Compilation Time : ", time.time() - start)
    return model
```

```
[ ] window = 20
    X_train, y_train, X_test, y_test = preprocess_data(selected_df[:: -1], window)
    print("X_train", X_train.shape)
    print("y_train", y_train.shape)
    print("X_test", X_test.shape)
    print("y_test", y_test.shape)
```

```
X_train (940, 20, 4)
y_train (940,)
X_test (104, 20, 4)
y_test (104,)
```



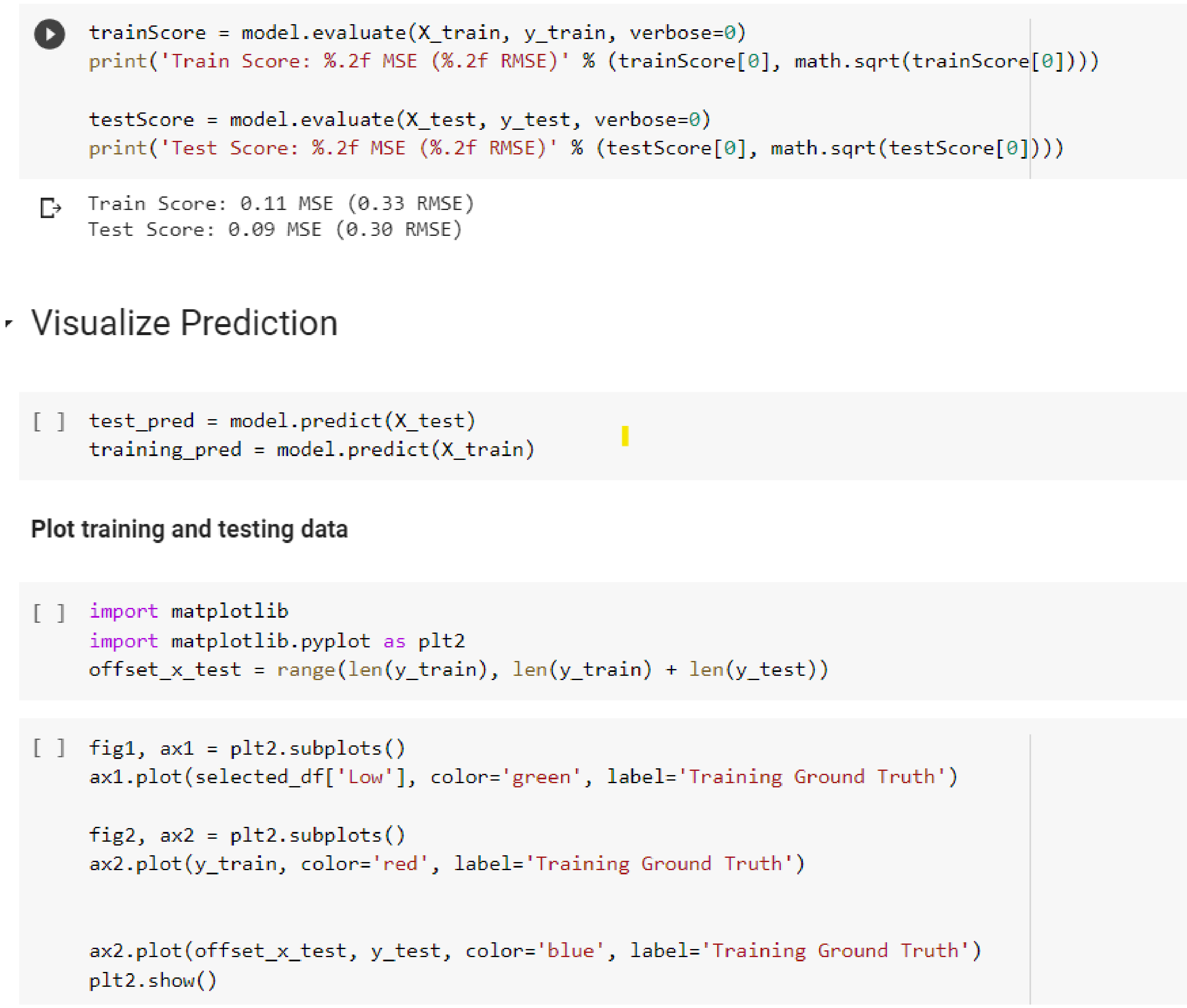
```
[ ] print([X_train.shape[2], window, 100, 1])

    [4, 20, 100, 1]
```

```
model = build_model([X_train.shape[2], window, 100, 1])

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-7e50d2f566d8> in <module>()
----> 1 model = build_model([X_train.shape[2], window, 100, 1])

NameError: name 'build_model' is not defined
```

SEARCH STACK OVERFLOW

Train the Network

```
[ ] model.fit(
        X_train,
        y_train,
        batch_size=768,
        epochs=20,
        validation_split=0.1,
        verbose=0)

    <tensorflow.python.keras.callbacks.History at 0x7fb1c220dbe0>
```



```
trainScore = model.evaluate(X_train, y_train, verbose=0)
print('Train Score: %.2f MSE (%.2f RMSE)' % (trainScore[0], math.sqrt(trainScore[0])))

testScore = model.evaluate(X_test, y_test, verbose=0)
print('Test Score: %.2f MSE (%.2f RMSE)' % (testScore[0], math.sqrt(testScore[0])))
```

```
Train Score: 0.11 MSE (0.33 RMSE)
Test Score: 0.09 MSE (0.30 RMSE)
```

Visualize Prediction

```
test_pred = model.predict(X_test)
training_pred = model.predict(X_train)
```

Plot training and testing data

```
import matplotlib
import matplotlib.pyplot as plt2
offset_x_test = range(len(y_train), len(y_train) + len(y_test))
```

```
fig1, ax1 = plt2.subplots()
ax1.plot(selected_df['Low'], color='green', label='Training Ground Truth')

fig2, ax2 = plt2.subplots()
ax2.plot(y_train, color='red', label='Training Ground Truth')


ax2.plot(offset_x_test, y_test, color='blue', label='Training Ground Truth')
plt2.show()
```

8.1 ArNet

Open in Colab

```
if 'google.colab' in str(get_ipython()):
    !pip install git+https://github.com/ourownstory/neural_prophet.git
if 'google.colab' in str(get_ipython()):
    !pip uninstall statsmodels -y
    !pip install statsmodels pmdarima yfinance
```

```
if 'google.colab' in str(get_ipython()):
  from google.colab import drive
  drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.m
```

```
import yfinance as yf
import pandas as pd
from statsmodels.tsa.ar_model import AutoReg
from statsmodels.tsa.arima.model import ARIMA

from math import sqrt
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import numpy as np
import random
import pmdarima as pm
```

Data Prep

Data Load

[ ] ↳ *4 hücre gizli*

Create Time Series

[ ] ↳ *9 hücre gizli*

Train-Valid-Test Split

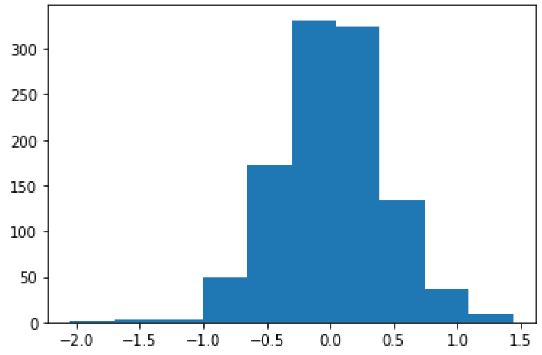
[ ] ↳ *1 hücre gizli*

Labeled Data

Histogram

```
plt.hist(inv_data["Change %"] ,   bins = 10)
```

```
(array([  2.,   3.,   3.,  49., 173., 331., 324., 134.,  36.,  10.]),
 array([-2.05 , -1.701, -1.352, -1.003, -0.654, -0.305,  0.044,  0.393,
         0.742,  1.091,  1.44 ]),
 <a list of 10 Patch objects>)
```



Split 1

[ ] ↳ *4 hücre gizli*

Split 2

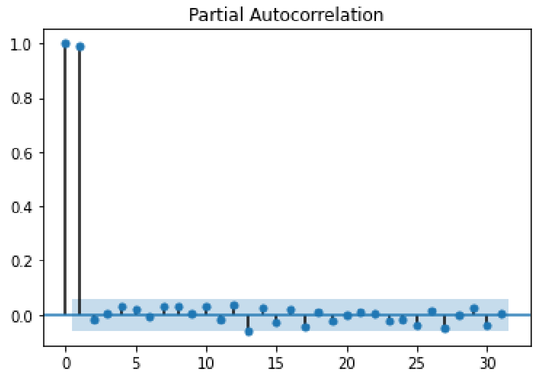
[ ] ↳ *5 hücre gizli*

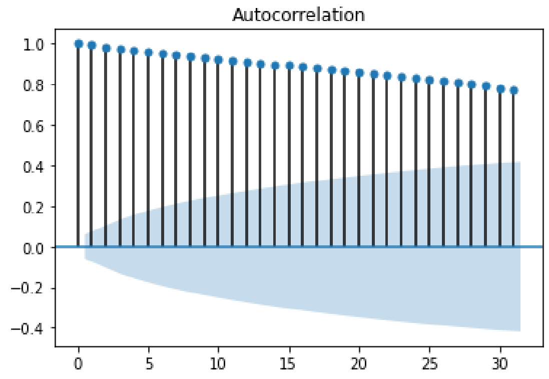
Plotting Values

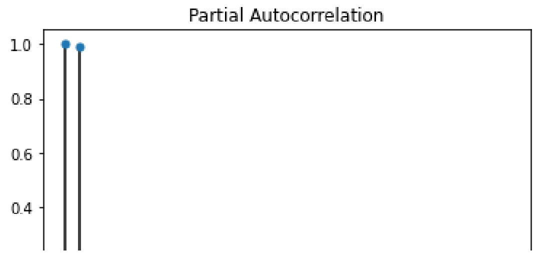
[ ] ↳ *1 hücre gizli*

ACF - PACF

```
plot_acf(inv_data.iloc[:,0]);
plot_pacf(inv_data.iloc[:,0])
```







AR Model

```
modelAR = AutoReg(train.Price , lags=14)
modelAR_fit = modelAR.fit()
print('Coefficients: %s' % modelAR_fit.params)
```

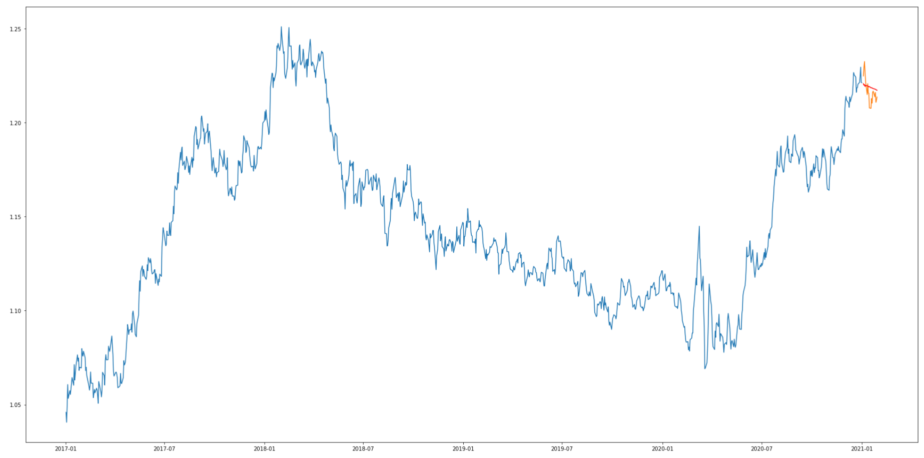
```
     Coefficients: intercept    0.006011
     Price.L1     1.012650
     Price.L2     0.005454
     Price.L3    -0.041303
     Price.L4    -0.042735
     Price.L5     0.065430
     Price.L6    -0.060324
     Price.L7     0.035583
     Price.L8     0.014944
     Price.L9    -0.010185
     Price.L10    0.020188
     Price.L11   -0.039037
     Price.L12    0.076462
     Price.L13   -0.078260
     Price.L14    0.036033
     dtype: float64
     /usr/local/lib/python3.6/dist-packages/statsmodels/tsa/ar_model.py:252: FutureWarning
       FutureWarning,
```

```
predictions = modelAR_fit.predict(start=len(train), end=len(train)+len(test)-1, dynamic=Fa
```

```
rmse = sqrt(mean_squared_error(test.Price, predictions))
mape = abs(predictions/test.Price - 1).mean()
print('Test RMSE: %.3f' % rmse)
print('Test MAPE: %.3f' % mape)
```

```
     Test RMSE: 0.006
     Test MAPE: 0.004
```

```
plt.figure(figsize=(30, 15))
# plt.plot(train.iloc[-75:])
plt.plot(train.Price)
plt.plot(test.Price)
plt.plot(predictions, color='red')
plt.show()
```



AutoARIMA

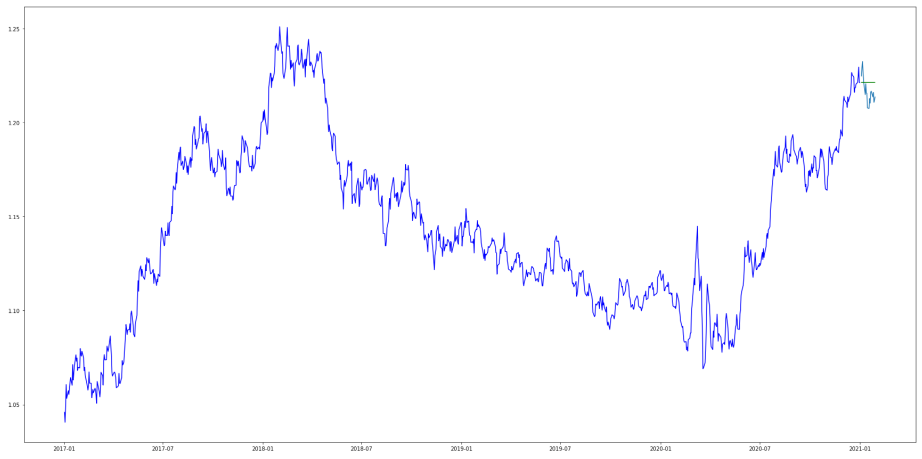
```
model = pm.auto_arima(train.Price)


forecasts = model.predict(test.shape[0])  # predict N steps into the future
predictions = pd.Series(forecasts, index=test.index)


rmse = sqrt(mean_squared_error(test.Price, predictions))
mape = abs(predictions/test.Price- 1).mean()
print('Test RMSE: %.3f' % rmse)
print('Test MAPE: %.3f' % mape)
```

```
     Test RMSE: 0.008
     Test MAPE: 0.006
```

```
model
```

```
     ARIMA(maxiter=50, method='lbfgs', order=(0, 1, 0), out_of_sample_size=0,
           scoring='mse', scoring_args={}, seasonal_order=(0, 0, 0, 0),
           start_params=None, suppress_warnings=True, trend=None,
           with_intercept=False)
```

```
# Visualize the forecasts (blue=train, green=forecasts)
x = np.arange(train.shape[0] + test.shape[0])
plt.figure(figsize=(30, 15))
plt.plot(train.Price, c='blue')
plt.plot(test.Price)
plt.plot(predictions, c='green')
plt.show()
```



AR-Net with Keras

```
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Activation
from keras.layers import Conv2D, MaxPooling2D, BatchNormalization
from keras import backend as K
from keras import initializers
from keras import activations
```

Hyperparameters

```
lr = 0.01
batch_size = 512
epochs = 200
lag = 7
step = 0
```

Preparing AR-Net Data

[ ] ↳ *8 hücre gizli*

AR-Net Model

```
del mArnet
```

```
np.random.seed(46)

mArnet = Sequential()
mArnet.add(BatchNormalization())
mArnet.add(Dense(150, kernel_initializer = initializers.GlorotUniform(), bias_initializer=
mArnet.add(Activation(activations.relu))
mArnet.add(BatchNormalization())
mArnet.add(Dense(250, kernel_initializer = initializers.GlorotUniform() , bias_initializer
mArnet.add(Activation(activations.relu))
mArnet.add(BatchNormalization())
mArnet.add(Dense(300, kernel_initializer = initializers.GlorotUniform() , bias_initializer
mArnet.add(Activation(activations.relu))
mArnet.add(BatchNormalization())
mArnet.add(Dense(1,activation= "linear", kernel_initializer=initializers.GlorotUniform(),
```

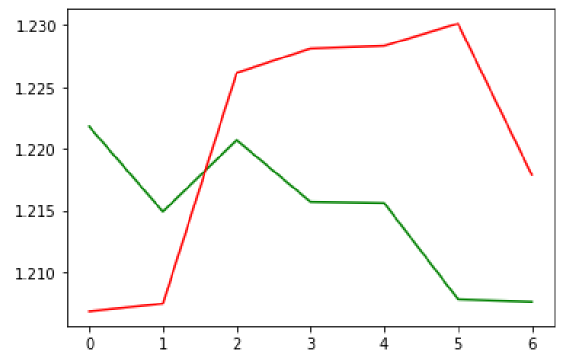
```
mArnet.compile(loss=keras.losses.mean_squared_error ,
              optimizer=keras.optimizers.Adam( learning_rate= lr),
              metrics=['mse'])
```

```
history = mArnet.fit(xtrain, ytrain,
                          batch_size=batch_size,
                          epochs=epochs,
                          verbose=0, shuffle = False,
                          validation_data=(xvalid, yvalid ))
```

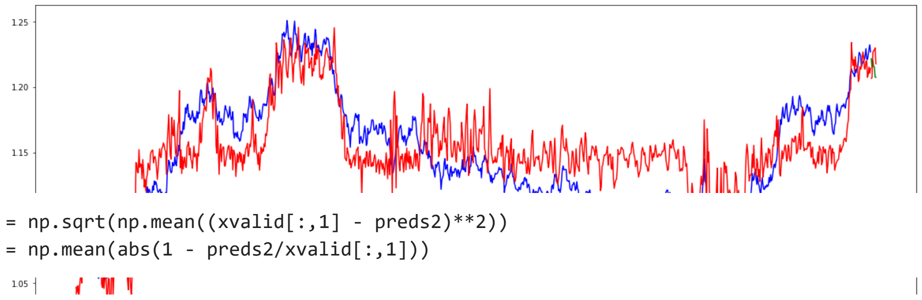
```
preds1 = mArnet.predict(xtrain)
```

```
preds2 = mArnet.predict(xvalid );preds2

     array([[1.2068133],
            [1.2074323],
            [1.2261554],
            [1.2281225],
            [1.2283404],
            [1.2301403],
            [1.2178925]], dtype=float32)
```

```
plt.plot(xvalid[:,1], c = 'green')
plt.plot(preds2, c='red')

     [<matplotlib.lines.Line2D at 0x7f26dcf2eda0>]
```



```
x = np.arange(xtrain.shape[0], xtrain.shape[0] + xvalid.shape[0])
plt.figure(figsize=(20, 8))
plt.plot(xtrain[:,1], c='blue')
plt.plot(preds1, c='red')
plt.plot(x ,xvalid[:,1], c = 'green')
plt.plot(x ,preds2, c='red')
plt.show()
```



```
RMSE = np.sqrt(np.mean((xvalid[:,1] - preds2)**2))
MAPE = np.mean(abs(1 - preds2/xvalid[:,1]))

print(RMSE)
print(MAPE)
```

```
    0.01214757831201916
    0.008674241571832806
```

8.2 Sample subsection

I implemented some of models we mentioned, namely **DeepAR**, **DeepState**, **DeepFactor** (we did not mention this, I found them on the documentation) and a **simple feed forward network**. I finished with the comparison of the methods with the selected metrics, MAPE and RMSE - can be easily increased -

Other available models can be found here: https://ts.gluon.ai/api/gluonts/gluonts.model.html

Notes:

- Some models like deepstate can work with multiple items at the same time, so maybe we can add pound/usd or jpy/eur etc. to see whether they improve the performance or not.
- I played with the hyperparameters but I think we need to analyze them more. And to keep the running fast, I selected small numbers of epochs. I think we have to increase them before the report.
- We can run classic time series model that are implemented in R from gluonts library, but I couldn't succeed.

References:

- https://aws.amazon.com/tr/blogs/machine-learning/creating-neural-time-series-models-with-gluon-time-series/
- https://ts.gluon.ai/#get-started-a-quick-example

```
!pip install mxnet gluonts
!pip uninstall statsmodels -y && pip install statsmodels

# installing R packages, requires for GluonTS R wrapper, but I couldn't make it work, so for
# !pip install 'rpy2>=2.9.*,<3.*'
# !R -e 'install.packages(c("forecast", "nnfor"), repos="https://cloud.r-project.org")'


!pip install git+https://github.com/ourownstory/neural_prophet.git
```

Data Preparation

```
import mxnet as mx
from mxnet import gluon
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import json

from gluonts.dataset.repository.datasets import get_dataset, dataset_recipes
from gluonts.dataset.util import to_pandas
```

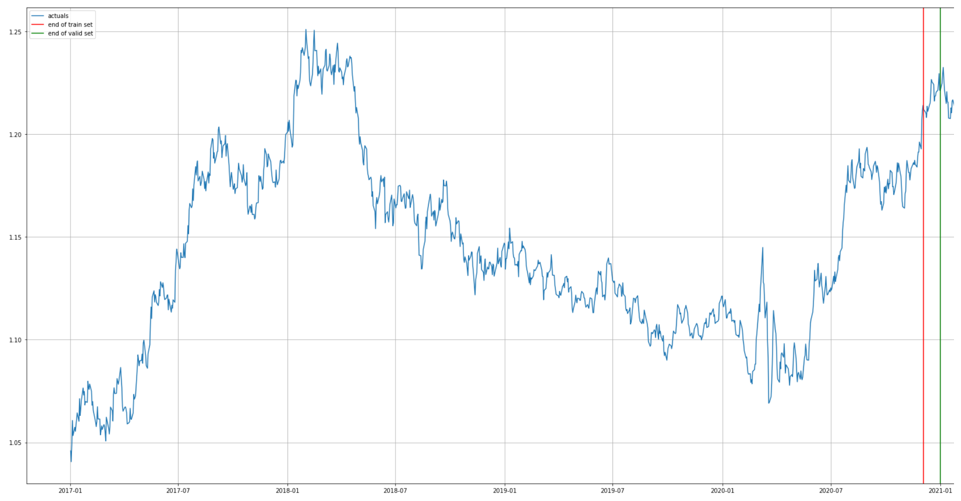
```
# I uploaded csv file to colab
inv_data = pd.read_csv("EUR_USD_Data.csv",
                       low_memory=False,
                       usecols=["Date", "Price"],
                       parse_dates=["Date"],
                       index_col="Date")
inv_data = inv_data.sort_index()
inv_data.index.freq = "1B"


# ensure that we only have weekdays
inv_data.index.dayofweek.value_counts()
```

```
4    213
3    213
2    213
1    213
0    213
Name: Date, dtype: int64
```

```
test_length, valid_length = 20, 20

train = inv_data.iloc[:-(test_length + valid_length)]
valid = inv_data.iloc[-(test_length + valid_length):-valid_length]
test = inv_data.iloc[-valid_length:]
train_valid = pd.concat([train, valid]) # use this if you won't need a validation set

print("train start-end dates:", train.index[[0, -1]].values)
print("valid start-end dates:", valid.index[[0, -1]].values)
print("test start-end dates:", test.index[[0, -1]].values)
```

```
train start-end dates: ['2017-01-02T00:00:00.000000000' '2020-12-04T00:00:00.000000000'
valid start-end dates: ['2020-12-07T00:00:00.000000000' '2021-01-01T00:00:00.000000000'
test start-end dates: ['2021-01-04T00:00:00.000000000' '2021-01-29T00:00:00.000000000']
```

```
plt.figure(figsize=(30, 15))
plt.plot(inv_data)
plt.axvline(train.index[-1], color='r') # end of train dataset
plt.axvline(valid.index[-1], color='g') # end of valid dataset
plt.grid(which="both")
plt.legend(["actuals", "end of train set", "end of valid set"], loc="upper left")
plt.show()
```



GluonTS Helpers

```
from gluonts.dataset.common import ListDataset


train_df = ListDataset([{'target': train_valid.Price.values,
                         'start': train_valid.index[0]}],
                       freq=train_valid.index.freq)

test_df = ListDataset([{'target': inv_data.Price.values,
                        'start': inv_data.index[0]}],
                      freq=inv_data.index.freq)


from gluonts.evaluation import Evaluator
from gluonts.trainer import Trainer
from gluonts.evaluation.backtest import make_evaluation_predictions

from math import sqrt
from sklearn.metrics import mean_squared_error
```

```
def plot_prob_forecasts(ts_entry, forecast_entry):
    plot_length = 150
    prediction_intervals = [50.0, 90.0]
    legend = ["observations", "median prediction"] + [f"{k}% prediction interval" for k in pr

    fig, ax = plt.subplots(1, 1, figsize=(30, 15))
    ts_entry[-plot_length:].plot(ax=ax)  # plot the time series
    forecast_entry.plot(prediction_intervals=prediction_intervals, color='g')
    plt.grid(which="both")
    plt.legend(legend, loc="upper left")
    plt.show()

def gluonts_wrapper(estimator, num_samples=100, verbose=True):
    """
    estimator: GluonTS estimator
    num_samples: number of sample paths we want for evaluation
    """
    predictor = estimator.train(train_df)
    forecast_it, ts_it = make_evaluation_predictions(
      dataset=test_df,
      predictor=predictor,
      num_samples=num_samples,
    )
    forecasts = list(forecast_it)
    tss = list(ts_it)
    forecast_entry = forecasts[0]
    ts_entry = tss[0]

    if verbose:
        print(f"Number of sample paths: {forecast_entry.num_samples}")
        print(f"Dimension of samples: {forecast_entry.samples.shape}")
        print(f"Start date of the forecast window: {forecast_entry.start_date}")
        print(f"Frequency of the time series: {forecast_entry.freq}")

        print(f"Mean of the future window:\n {forecast_entry.mean}")
        print(f"0.5-quantile (median) of the future window:\n {forecast_entry.quantile(0.5)}"

    plot_prob_forecasts(ts_entry, forecast_entry)

    evaluator = Evaluator()
    agg_metrics, item_metrics = evaluator(iter(tss), iter(forecasts), num_series=len(test_df)
    return agg_metrics

def calculate_accuracy(actual, pred):
    rmse = sqrt(mean_squared_error(actual, pred))
    mape = abs(pred/actual - 1).mean()
    return {"MAPE": mape, "RMSE": rmse}
```

```
 /usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:2: DeprecationWarning: gluc
```

Simple Feed Forward

```
from gluonts.model.simple_feedforward import SimpleFeedForwardEstimator

ff_estimator = SimpleFeedForwardEstimator(
    num_hidden_dimensions=[10],
    prediction_length=test_length,
    context_length=100,
    freq="1B",
    trainer=Trainer(ctx="cpu",
                    epochs=20,
                    learning_rate=1e-3,
                    num_batches_per_epoch=100)
)
```

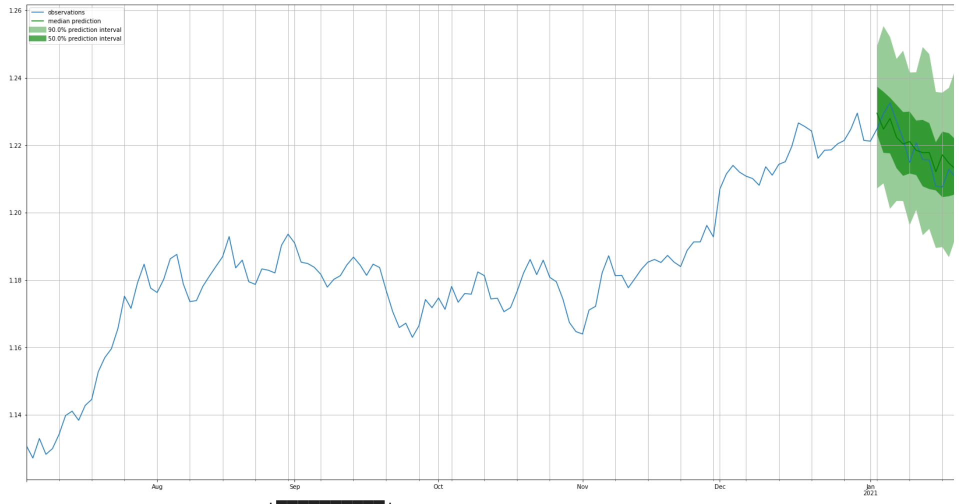
```
agg_metrics_simple_ff = gluonts_wrapper(ff_estimator)
```

DeepAR

```
from gluonts.model.deepar import DeepAREstimator

deepar_estimator = DeepAREstimator(
    freq="1B",
    prediction_length=test_length,
    trainer=Trainer(epochs=10)
)
```

```
agg_metrics_deepar = gluonts_wrapper(deepar_estimator)
```

```
  0%|          | 0/50 [00:00<?, ?it/s]learning rate from ``lr_scheduler`` has been overw
100%|██████████| 50/50 [00:04<00:00, 12.01it/s, epoch=1/10, avg_epoch_loss=-.279]
100%|██████████| 50/50 [00:03<00:00, 12.66it/s, epoch=2/10, avg_epoch_loss=-2.02]
100%|██████████| 50/50 [00:03<00:00, 12.70it/s, epoch=3/10, avg_epoch_loss=-2.32]
100%|██████████| 50/50 [00:03<00:00, 12.66it/s, epoch=4/10, avg_epoch_loss=-2.5]
100%|██████████| 50/50 [00:03<00:00, 12.78it/s, epoch=5/10, avg_epoch_loss=-2.33]
100%|██████████| 50/50 [00:03<00:00, 12.78it/s, epoch=6/10, avg_epoch_loss=-2.55]
100%|██████████| 50/50 [00:03<00:00, 12.67it/s, epoch=7/10, avg_epoch_loss=-2.53]
100%|██████████| 50/50 [00:03<00:00, 12.80it/s, epoch=8/10, avg_epoch_loss=-2.53]
100%|██████████| 50/50 [00:03<00:00, 12.79it/s, epoch=9/10, avg_epoch_loss=-2.57]
100%|██████████| 50/50 [00:03<00:00, 12.81it/s, epoch=10/10, avg_epoch_loss=-2.68]
Number of sample paths: 100
Dimension of samples: (100, 20)
Start date of the forecast window: 2021-01-04 00:00:00
Frequency of the time series: 1B
Mean of the future window:
 [1.2291464 1.2279739 1.2273409 1.2228351 1.2219054 1.220582  1.2202641
 1.2179749 1.2175559 1.2134295 1.214394  1.2148519 1.2144507 1.2102237
 1.2110016 1.2098941 1.2075766 1.2089115 1.2080461 1.2103679]
0.5-quantile (median) of the future window:
 [1.2294904 1.2247252 1.2279203 1.2223179 1.2203554 1.2210736 1.2184645
 1.2177619 1.2178606 1.2120941 1.2171524 1.2146477 1.2129328 1.2091175
 1.2097648 1.2096568 1.208508  1.2101005 1.2081398 1.2110069]
```



```
Running evaluation: 100%|██████████| 1/1 [00:00<00:00, 22.69it/s]
```

Deep State

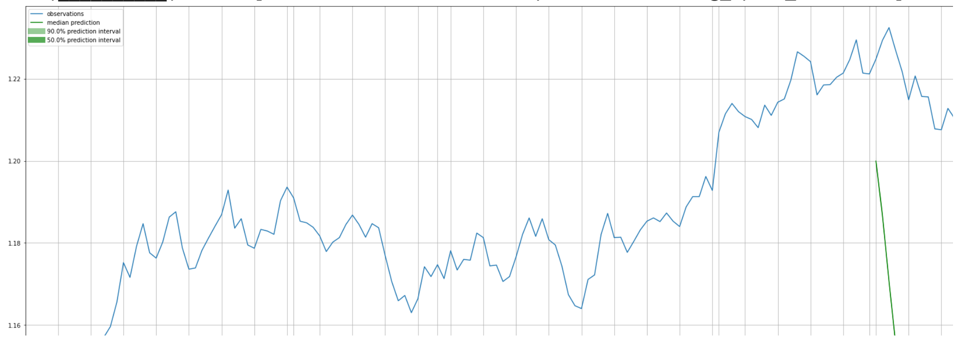
```
from gluonts.model.deepstate import DeepStateEstimator

deepstate_estimator = DeepStateEstimator(
    prediction_length=test_length,
    cardinality=[1],
    freq="1B",
    use_feat_static_cat=False,
    num_cells=50,
    num_layers=2,
    cell_type="lstm",
    past_length=500,
    trainer=Trainer(epochs=50))

agg_metrics_deepstate = gluonts_wrapper(deepstate_estimator, verbose=False)
```

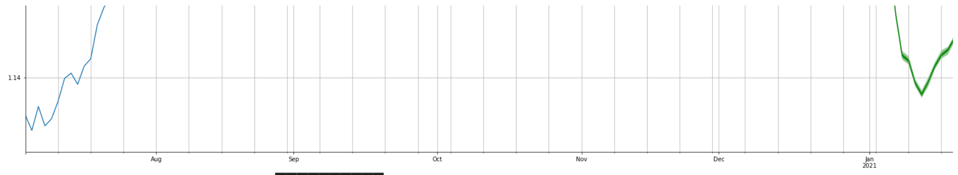
Deep Factor

```
from gluonts.model.deep_factor import DeepFactorEstimator

deepfactor_estimator = DeepFactorEstimator(
    freq="1B",
    prediction_length=test_length,
    num_factors=50, #10
    num_hidden_global=20, # 50
    num_layers_global=2, # 1
    num_hidden_local=5, # 5
    num_layers_local=1, # 1
    trainer=Trainer(epochs=100))

agg_metrics_deepfactor = gluonts_wrapper(deepfactor_estimator, verbose=False)
```

```
  0%|          | 0/50 [00:00<?, ?it/s]learning rate from ``lr_scheduler`` has been overw
100%|██████████| 50/50 [00:00<00:00, 64.53it/s, epoch=1/100, avg_epoch_loss=0.39]
100%|██████████| 50/50 [00:00<00:00, 68.43it/s, epoch=2/100, avg_epoch_loss=-.329]
100%|██████████| 50/50 [00:00<00:00, 70.91it/s, epoch=3/100, avg_epoch_loss=-1.14]
100%|██████████| 50/50 [00:00<00:00, 69.29it/s, epoch=4/100, avg_epoch_loss=-1.7]
100%|██████████| 50/50 [00:00<00:00, 73.13it/s, epoch=5/100, avg_epoch_loss=-2.16]
100%|██████████| 50/50 [00:00<00:00, 70.96it/s, epoch=6/100, avg_epoch_loss=-2.55]
100%|██████████| 50/50 [00:00<00:00, 71.61it/s, epoch=7/100, avg_epoch_loss=-2.91]
100%|██████████| 50/50 [00:00<00:00, 70.62it/s, epoch=8/100, avg_epoch_loss=-3.25]
100%|██████████| 50/50 [00:00<00:00, 68.86it/s, epoch=9/100, avg_epoch_loss=-3.57]
100%|██████████| 50/50 [00:00<00:00, 71.38it/s, epoch=10/100, avg_epoch_loss=-3.89]
100%|██████████| 50/50 [00:00<00:00, 69.37it/s, epoch=11/100, avg_epoch_loss=-4.19]
100%|██████████| 50/50 [00:00<00:00, 70.59it/s, epoch=12/100, avg_epoch_loss=-4.48]
100%|██████████| 50/50 [00:00<00:00, 69.97it/s, epoch=13/100, avg_epoch_loss=-4.76]
100%|██████████| 50/50 [00:00<00:00, 70.64it/s, epoch=14/100, avg_epoch_loss=-5.01]
100%|██████████| 50/50 [00:00<00:00, 69.13it/s, epoch=15/100, avg_epoch_loss=-5.21]
100%|██████████| 50/50 [00:00<00:00, 68.32it/s, epoch=16/100, avg_epoch_loss=-5.34]
100%|██████████| 50/50 [00:00<00:00, 69.09it/s, epoch=17/100, avg_epoch_loss=-5.4]
100%|██████████| 50/50 [00:00<00:00, 69.42it/s, epoch=18/100, avg_epoch_loss=-5.46]
100%|██████████| 50/50 [00:00<00:00, 68.46it/s, epoch=19/100, avg_epoch_loss=-5.5]
100%|██████████| 50/50 [00:00<00:00, 69.73it/s, epoch=20/100, avg_epoch_loss=-5.55]
100%|██████████| 50/50 [00:00<00:00, 67.07it/s, epoch=21/100, avg_epoch_loss=-5.59]
100%|██████████| 50/50 [00:00<00:00, 68.35it/s, epoch=22/100, avg_epoch_loss=-5.62]
100%|██████████| 50/50 [00:00<00:00, 67.24it/s, epoch=23/100, avg_epoch_loss=-5.66]
100%|██████████| 50/50 [00:00<00:00, 68.97it/s, epoch=24/100, avg_epoch_loss=-5.69]
100%|██████████| 50/50 [00:00<00:00, 68.23it/s, epoch=25/100, avg_epoch_loss=-5.71]
100%|██████████| 50/50 [00:00<00:00, 69.07it/s, epoch=26/100, avg_epoch_loss=-5.74]
100%|██████████| 50/50 [00:00<00:00, 69.58it/s, epoch=27/100, avg_epoch_loss=-5.75]
100%|██████████| 50/50 [00:00<00:00, 70.57it/s, epoch=28/100, avg_epoch_loss=-5.76]
100%|██████████| 50/50 [00:00<00:00, 66.73it/s, epoch=29/100, avg_epoch_loss=-5.75]
100%|██████████| 50/50 [00:00<00:00, 69.69it/s, epoch=30/100, avg_epoch_loss=-5.79]
100%|██████████| 50/50 [00:00<00:00, 67.55it/s, epoch=31/100, avg_epoch_loss=-5.8]
100%|██████████| 50/50 [00:00<00:00, 67.23it/s, epoch=32/100, avg_epoch_loss=-5.81]
100%|██████████| 50/50 [00:00<00:00, 68.89it/s, epoch=33/100, avg_epoch_loss=-5.81]
100%|██████████| 50/50 [00:00<00:00, 68.89it/s, epoch=34/100, avg_epoch_loss=-5.81]
100%|██████████| 50/50 [00:00<00:00, 68.06it/s, epoch=35/100, avg_epoch_loss=-5.82]
100%|██████████| 50/50 [00:00<00:00, 67.64it/s, epoch=36/100, avg_epoch_loss=-5.83]
100%|██████████| 50/50 [00:00<00:00, 68.94it/s, epoch=37/100, avg_epoch_loss=-5.84]
100%|██████████| 50/50 [00:00<00:00, 69.25it/s, epoch=38/100, avg_epoch_loss=-5.84]
100%|██████████| 50/50 [00:00<00:00, 69.49it/s, epoch=39/100, avg_epoch_loss=-5.84]
100%|██████████| 50/50 [00:00<00:00, 66.95it/s, epoch=40/100, avg_epoch_loss=-5.84]
100%|██████████| 50/50 [00:00<00:00, 66.93it/s, epoch=41/100, avg_epoch_loss=-5.82]
100%|██████████| 50/50 [00:00<00:00, 68.14it/s, epoch=42/100, avg_epoch_loss=-5.83]
100%|██████████| 50/50 [00:00<00:00, 69.51it/s, epoch=43/100, avg_epoch_loss=-5.85]
100%|██████████| 50/50 [00:00<00:00, 67.27it/s, epoch=44/100, avg_epoch_loss=-5.86]
100%|██████████| 50/50 [00:00<00:00, 67.86it/s, epoch=45/100, avg_epoch_loss=-5.86]
100%|██████████| 50/50 [00:00<00:00, 69.42it/s, epoch=46/100, avg_epoch_loss=-5.86]
100%|██████████| 50/50 [00:00<00:00, 69.89it/s, epoch=47/100, avg_epoch_loss=-5.87]
100%|██████████| 50/50 [00:00<00:00, 69.32it/s, epoch=48/100, avg_epoch_loss=-5.87]
100%|██████████| 50/50 [00:00<00:00, 68.17it/s, epoch=49/100, avg_epoch_loss=-5.84]
100%|██████████| 50/50 [00:00<00:00, 69.92it/s, epoch=50/100, avg_epoch_loss=-5.84]
100%|██████████| 50/50 [00:00<00:00, 71.13it/s, epoch=51/100, avg_epoch_loss=-5.82]
100%|██████████| 50/50 [00:00<00:00, 68.67it/s, epoch=52/100, avg_epoch_loss=-5.85]
100%|██████████| 50/50 [00:00<00:00, 68.27it/s, epoch=53/100, avg_epoch_loss=-5.86]
100%|██████████| 50/50 [00:00<00:00, 69.04it/s, epoch=54/100, avg_epoch_loss=-5.87]
100%|██████████| 50/50 [00:00<00:00, 70.40it/s, epoch=55/100, avg_epoch_loss=-5.87]
100%|██████████| 50/50 [00:00<00:00, 68.95it/s, epoch=56/100, avg_epoch_loss=-5.88]
```

```
100%|██████████| 50/50 [00:00<00:00, 67.21it/s, epoch=57/100, avg_epoch_loss=-5.88]
100%|██████████| 50/50 [00:00<00:00, 68.48it/s, epoch=58/100, avg_epoch_loss=-5.88]
100%|██████████| 50/50 [00:00<00:00, 67.37it/s, epoch=59/100, avg_epoch_loss=-5.88]
100%|██████████| 50/50 [00:00<00:00, 68.68it/s, epoch=60/100, avg_epoch_loss=-5.88]
100%|██████████| 50/50 [00:00<00:00, 67.18it/s, epoch=61/100, avg_epoch_loss=-5.89]
100%|██████████| 50/50 [00:00<00:00, 69.06it/s, epoch=62/100, avg_epoch_loss=-5.89]
100%|██████████| 50/50 [00:00<00:00, 67.84it/s, epoch=63/100, avg_epoch_loss=-5.88]
100%|██████████| 50/50 [00:00<00:00, 68.86it/s, epoch=64/100, avg_epoch_loss=-5.88]
100%|██████████| 50/50 [00:00<00:00, 68.68it/s, epoch=65/100, avg_epoch_loss=-5.89]
100%|██████████| 50/50 [00:00<00:00, 68.30it/s, epoch=66/100, avg_epoch_loss=-5.89]
100%|██████████| 50/50 [00:00<00:00, 68.00it/s, epoch=67/100, avg_epoch_loss=-5.88]
100%|██████████| 50/50 [00:00<00:00, 68.79it/s, epoch=68/100, avg_epoch_loss=-5.87]
100%|██████████| 50/50 [00:00<00:00, 68.71it/s, epoch=69/100, avg_epoch_loss=-5.88]
100%|██████████| 50/50 [00:00<00:00, 68.44it/s, epoch=70/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 67.76it/s, epoch=71/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 69.36it/s, epoch=72/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 70.15it/s, epoch=73/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 69.97it/s, epoch=74/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 70.33it/s, epoch=75/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 69.87it/s, epoch=76/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 69.61it/s, epoch=77/100, avg_epoch_loss=-5.87]
100%|██████████| 50/50 [00:00<00:00, 72.10it/s, epoch=78/100, avg_epoch_loss=-5.88]
100%|██████████| 50/50 [00:00<00:00, 70.95it/s, epoch=79/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 68.74it/s, epoch=80/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 65.95it/s, epoch=81/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 64.80it/s, epoch=82/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 67.55it/s, epoch=83/100, avg_epoch_loss=-5.91]
100%|██████████| 50/50 [00:00<00:00, 66.53it/s, epoch=84/100, avg_epoch_loss=-5.91]
100%|██████████| 50/50 [00:00<00:00, 67.94it/s, epoch=85/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 67.38it/s, epoch=86/100, avg_epoch_loss=-5.86]
100%|██████████| 50/50 [00:00<00:00, 68.27it/s, epoch=87/100, avg_epoch_loss=-5.89]
100%|██████████| 50/50 [00:00<00:00, 67.91it/s, epoch=88/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 69.15it/s, epoch=89/100, avg_epoch_loss=-5.91]
100%|██████████| 50/50 [00:00<00:00, 70.27it/s, epoch=90/100, avg_epoch_loss=-5.91]
100%|██████████| 50/50 [00:00<00:00, 69.93it/s, epoch=91/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 70.63it/s, epoch=92/100, avg_epoch_loss=-5.9]
100%|██████████| 50/50 [00:00<00:00, 69.91it/s, epoch=93/100, avg_epoch_loss=-5.91]
100%|██████████| 50/50 [00:00<00:00, 69.93it/s, epoch=94/100, avg_epoch_loss=-4.52]
100%|██████████| 50/50 [00:00<00:00, 69.90it/s, epoch=95/100, avg_epoch_loss=-5.87]
100%|██████████| 50/50 [00:00<00:00, 68.59it/s, epoch=96/100, avg_epoch_loss=-5.89]
100%|██████████| 50/50 [00:00<00:00, 68.88it/s, epoch=97/100, avg_epoch_loss=-5.89]
100%|██████████| 50/50 [00:00<00:00, 71.49it/s, epoch=98/100, avg_epoch_loss=-5.89]
100%|██████████| 50/50 [00:00<00:00, 69.88it/s, epoch=99/100, avg_epoch_loss=-5.89]
100%|██████████| 50/50 [00:00<00:00, 72.18it/s, epoch=100/100, avg_epoch_loss=-5.9]
```





```
Running evaluation: 100%|██████████| 1/1 [00:00<00:00,  9.64it/s]
```

ETS - Holt's Linear Trend

```
from statsmodels.tsa.api import ExponentialSmoothing, SimpleExpSmoothing, Holt
```

```
holt = Holt(train_valid).fit(smoothing_level=0.8, smoothing_trend=0.2, optimized=False, )
fcast1 = holt.forecast(test_length).rename("Holt's linear trend")
```

```
    /usr/local/lib/python3.6/dist-packages/statsmodels/tsa/holtwinters/model.py:429: FutureW
      FutureWarning,
```

```
agg_metrics_holt = calculate_accuracy(test["Price"], fcast1)
```

```
    {'MAPE': 0.00524936379507589, 'RMSE': 0.007246046580491728}
    {'MAPE': 0.005300547581983272, 'RMSE': 0.007304239661073174}
    {'MAPE': 0.005245474635773156, 'RMSE': 0.0071967360532264645}
    {'MAPE': 0.006308269955262147, 'RMSE': 0.0085247289260802}
```

Neural Prophet

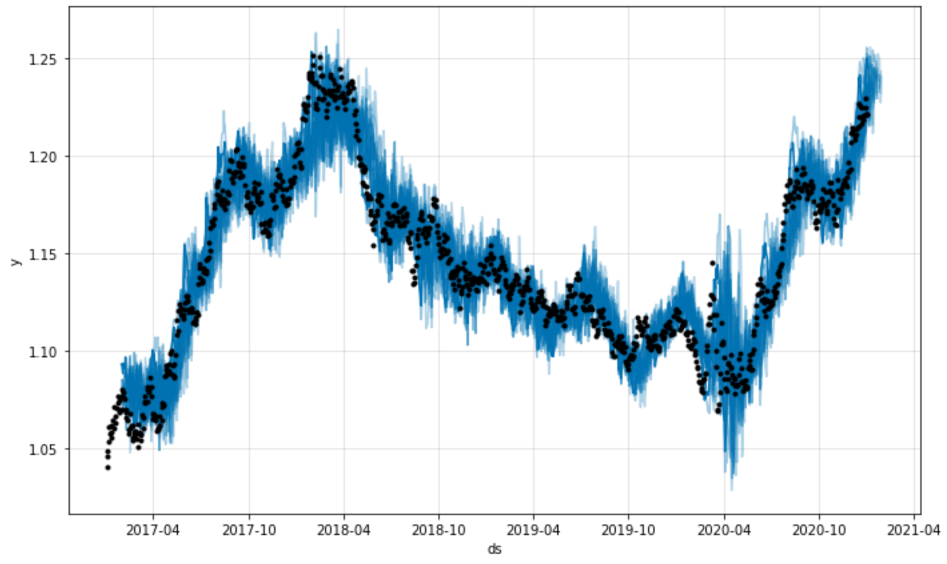
```
from neuralprophet import NeuralProphet

prophet_df = train_valid.reset_index()
prophet_df.columns = ["ds", "y"]
```

```
m = NeuralProphet(
    n_lags=test_length,
    n_forecasts=test_length,
    batch_size=64,
    epochs=50,
    learning_rate=0.1,
    ar_sparsity=0.2,
)
metrics = m.fit(prophet_df, freq='B')
```

```
    INFO - (NP.utils.set_auto_seasonalities) - Disabling daily seasonality. Run NeuralProphe
    INFO:NP.utils:Disabling daily seasonality. Run NeuralProphet with daily_seasonality=True
```

```
Epoch[50/50]: 100%|██████████| 50/50 [00:04<00:00, 11.77it/s, SmoothL1Loss=0.00344, MAE=
```

```
future = m.make_future_dataframe(prophet_df, n_historic_predictions=True)
forecast = m.predict(future)
fig = m.plot(forecast)
```



```
# fig_comp = m.plot_components(forecast)
# m = m.highlight_nth_step_ahead_of_each_forecast(1) # temporary workaround to plot actual AR
# fig_param = m.plot_parameters()
```

```
prophet_output = forecast[["ds", f"yhat{test_length}"]].iloc[-test_length:]
prophet_output.columns = ["Date", "Prediction"]
prophet_output = prophet_output.set_index("Date")
```

```
agg_metrics_prophet = calculate_accuracy(test["Price"], prophet_output["Prediction"])
```

Comparison

```
df_metrics = pd.concat(
    [
        pd.DataFrame.from_dict(agg_metrics_simple_ff, orient='index').rename(columns={0: "Sim
        pd.DataFrame.from_dict(agg_metrics_deepar, orient='index').rename(columns={0: "DeepAR
        pd.DataFrame.from_dict(agg_metrics_deepstate, orient='index').rename(columns={0: "Dee
        pd.DataFrame.from_dict(agg_metrics_deepfactor, orient='index').rename(columns={0: "De
        pd.DataFrame.from_dict(agg_metrics_prophet, orient='index').rename(columns={0: "Neura
        pd.DataFrame.from_dict(agg_metrics_holt, orient='index').rename(columns={0: "Holt's L
    ], axis=1)
df_metrics.loc[["MAPE", "RMSE"]]
```

	Simple FF	DeepAR	DeepState	DeepFactor	NeuralProphet
MAPE	0.021455	0.003445	0.164916	0.059644	0.016897
RMSE	0.035463	0.004410	0.203060	0.074684	0.022691

References

[1] Elliott G, Timmermann A (2016) Forecasting in economics and finance. *Annual Review of Economics* 8:81–110.

[2] Eur usd historical data. Available at https://www.investing.com/currencies/eur-usd-historical-data.

[3] Alexandrov A, Benidis K, Bohlke-Schneider M, Flunkert V, Gasthaus J, Januschowski T, Maddix DC, Rangapuram S, Salinas D, Schulz J, et al. (2019) Gluonts: Probabilistic time series models in python. *arXiv preprint arXiv:190605264* .

[4] Matyjaszek M, Fernández PR, Krzemień A, Wodarski K, Valverde GF (2019) Forecasting coking coal prices by means of arima models and neural networks, considering the transgenic time series theory. *Resources Policy* 61:283–292.

[5] Triebe O, Laptev N, Rajagopal R (2019) Ar-net: A simple auto-regressive neural network for time-series. *arXiv preprint arXiv:191112436* .

[6] Lai G, Chang WC, Yang Y, Liu H (2018) Modeling long-and short-term temporal patterns with deep neural networks. *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*, , pp 95–104.

[7] Wang Y, Smola A, Maddix D, Gasthaus J, Foster D, Januschowski T (2019) Deep factors for forecasting. *International Conference on Machine Learning* (PMLR), , pp 6607–6617.