Precious metals price predictor

Pavel Dostál

September 2020

1 Introduction

Goal of this project was to explore possible application of artificial intelligence in such seemingly chaotic and random setting as stock exchange. More precisely to be able to predict the future price in certain time horizon for silver from data set containing previous data not only of precious metals, but other commodities, FOREX, indexes and stocks.

2 Existing solutions

Existing solutions deals primarily with stock prediction, prediction of commodities can be found marginally in some commercial software. However stock prediction has similar core as commodity prediction therefore we decided to mention here stock prediction solutions.

2.1 Commercial software

Stock market prediction is area in which commercial companies pursue for very long time, long before invention of computers or artificial intelligence. However with possibilities of machine learning they started to develop commercial software used internally or as paid product (iknowfirst, finbrain). To development of the commercial software went amount of money and time and it is kept in secret. In the public is not known not even what data they use nor used methods or algorithms.

2.2 Non-commercial software

Usually to this area belongs quite simple software from tutorials which is not actually useful in market prediction (if it were very useful, creator would use it to generate great sums of money without any effort) but it serves to learning purposes. Some examples are (clickable links):

- Stock Market Predictions with LSTM in Python
- Stock Price Prediction Using Python & Machine Learning

- Stock Prices Prediction Using Machine Learning and Deep Learning Techniques
- Stock Prediction in Python
- Stock prediction using recurrent neural networks

3 Implementation and algorithms

3.1 Data

In this project two datasets were used, both attached to this document. The first one is dataset containing daily prices of stocks, commodities, indexes and FOREXes, total 40 values daily for 12 years (01.01.2008-01.01.2020) loaded from Yahoo Finance through their DataReader and their API.

The second dataset is downloaded from Stooq and contains hourly information about prices of circa 1 000 stocks, commodities, indexes and FOREXes for half year, with total amount of more than 2 000 000 entries.

Both dataset were divided into train section (first 70% of the whole), validation part (20% of the whole) and testing part (last 10%).

3.2 Neural networks

We will use TensorFlow library to create in total 7 types of neural networks which we first explain them and then show how they are implemented. In Section subsection 3.4 we compare how they fared. The time horizon of prediction is set to be 24 hours in future for hourly data and 14 day in future for daily data.

Dense model

The dense model is the simplest trainable model imaginable, using inputs from the last time step to predict all future steps independently as can be seen in Figure 1. In the code shown below we just make one Dense layer and under that another dense layer with shape so that we have number of future steps in f_record with n_col columns of data. Dense layer is a simple neuron layer where each neuron is connected to each neuron in the next layer.

```
def gen_dense_model(f_record, n_col, dropout, units):
  return tf.keras.Sequential([
      tf.keras.layers.Dense(units, activation='relu'),
      tf.keras.layers.Dropout(dropout),
      tf.keras.layers.Dense(f_record*n_col, kernel_initializer=tf.initializers.z
      tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),
      tf.keras.layers.Reshape([f_record, n_col])
    ])
```

GRU/LSTM model

Both GRU (Gated Recurrent Unit) and LSTM (Long Short Term Memory) belong to type called RNN (Recurrent Neural Network) which is charakterised

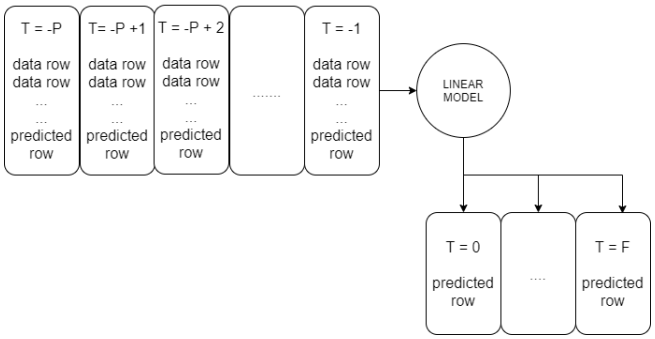

Figure 1: Dense model

by the fact that output from previous step is fed directly to the next step as can be seen in Figure 2. GRU and LSTM tried to asnwer vanishing gradient problem, which caused that for the earliest layers grandient computed by loss function was insignificant and layers were not able to learn in longer sequences. Both GRU and LSTM propagate gradients to further layers, making it more suitable for longer time sequence. For further details look at Cho et al., 2014 for GRU and LSTM at Hochreiter & Schmidhuber, 1997.

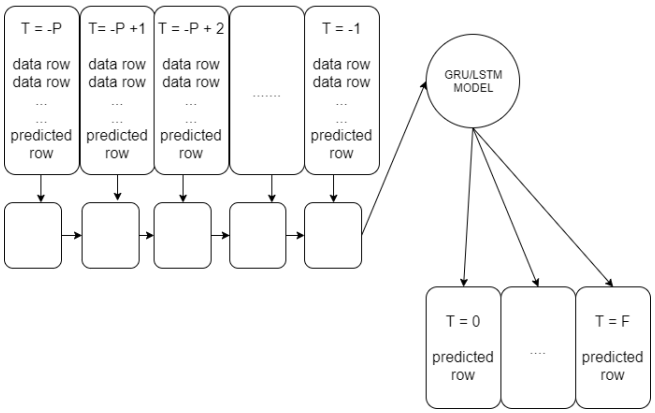

Figure 2: RNN model

The code is similar for both GRU and LSTM model, on the top of the output dense layer we put layer specific to the model.

```
def gen_gru_model(f_record, n_col, dropout, units):
  return tf.keras.Sequential([
      tf.keras.layers.GRU(units, return_sequences=False, activation='relu', drop

      tf.keras.layers.Dense(f_record*n_col, kernel_initializer=tf.initializers.z
      # tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),
```

```
    tf.keras.layers.Reshape([f_record, n_col])
  ])

def gen_lstm_model(f_record, n_col, dropout, units):
  return tf.keras.Sequential([
    tf.keras.layers.LSTM(units, return_sequences=False, dropout=dropout),

    tf.keras.layers.Dense(f_record*n_col, kernel_initializer=tf.initializers.zer
    tf.keras.layers.Reshape([f_record, n_col])
  ])
```

Autoregressive RNN (simple RNN, GRU and LSTM)

Recurrent Neural Network was already mentioned in previous chapter, however all models up to this one had one serious disadvantage, they generated output steps independently. However it may be useful use prediction for step n to help predicting step $n+1$ what is exactly what autoregressive RNN does as can be seen in Figure 6. Model predict value for step 0 and uses predicted value to predict step 1. That's why there are two arrows from models ("white boxes") on prediction steps.

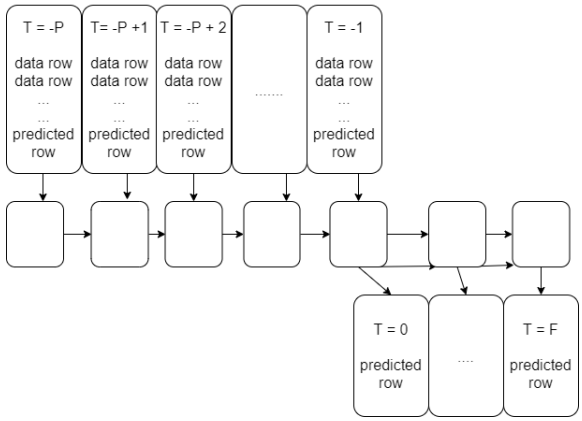

Figure 3: Autoregressive model

The code is encapsulated in the class *Arr_Rnn*. Function *warmup* is used to initialize the model which return single step prediction and state of RNN. Function *call* then can give prediction in future using state of RNN and value given by RNN.

```
class Arr_Rnn(tf.keras.Model):
  def __init__(self, f_records):
    super().__init__()
    self.f_records = f_records

  def warmup(self, inputs):
```

```
    x, *state = self.rnn(inputs)
    prediction = self.dense(x)
    return prediction, state

  def call(self, inputs, training=None):
    predictions = []
    prediction, state = self.warmup(inputs)
    predictions.append(prediction)

    for n in range(1, self.f_records):
      x, state = self.neuron_cell(prediction, states=state,
                                  training=training)
      prediction = self.dense(x)
      predictions.append(prediction)

    predictions = tf.stack(predictions)
    predictions = tf.transpose(predictions, [1, 0, 2])
    return predictions
```

However you might noticed that in previous code was not mentioned which type of RNN is used and how it is added. In the class we have several functions which can add simple RNN, stacked RNN, LSTM and GRU layers as they were dealt in previous chapter. Difference is that here they give not only resulting value, but also their state. Below example for LSTM, other three can be found in the code itself.

```
def put_lstm(self, units, dropout):
  self.neuron_cell = tf.keras.layers.LSTMCell(units, recurrent_dropout=dropout)
  self.rnn = tf.keras.layers.RNN(self.neuron_cell, return_state=True)
  self.dense = tf.keras.layers.Dense(num_features)
```

3.3 Bayesian optimization

Setting of neural network is dependent on many factors, such as dropout and learning rate, number of neurons on each layer or number of layers. Since generating and training one model might be time consuming dependent on variables mentioned above and dataset Bayesian optimization was chosen to compute the best combination as possible.

Variables which are subject to Bayesian optimization are dropout and learning rate, which specify how quickly the model learns and how quickly it discards pattern from learning history. Variable *units* gives how many units of neurons there are, from 10 to 1000.

Optimization score can be evaluated according to many metrics such as Euclidean distance, log loss or binary cross-entropy. We evaluated Bayesian optization according to mean square error (MSE).

We use library *BayesianOptimization* and it is enough to specify the function which result should be optimized and bounds between which can be arguments of function as shown in the code below.

```
pbounds_m = {'units': (0.1, 0.99),
             'learning_rate': (0.01, 0.299),
             'dropout': (0.01, 0.299)
            }

dense_optimizer = BayesianOptimization(
    f=eval_dense_model,
    pbounds=pbounds_m,
    verbose=1,
    random_state=1,
)
```

3.4 Evaluation

To evaluate different types of neural network we decided to test them on data described in Data section.

3.5 Test daily

First we launched evaluation tests on daily data from Yahoo Finance. For all 7 models were variables set to 14 days, number of units 64, dropout 0.1. We predicted for each step in data the next 14 days and this value we compared to the actual values.

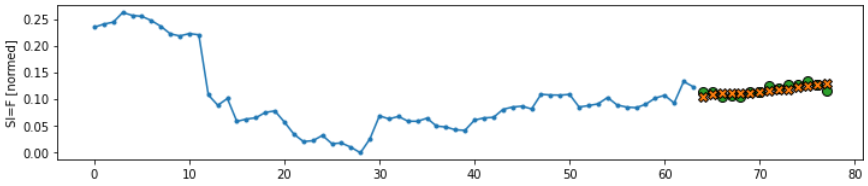

Figure 4: Example of 14 days prediction, on x axis days from the start, on y normalised price of silver bullion SI=F

In the table below we can see first test results from all 7 models, the names and order are as were listed in this documentation. The best results (smallest mean square error) we obtained from Autoregressive GRU and GRU, which both had the smallest MSE.

	loss:	mean_absolute_error
Dense	0.0465	0.1648
GRU	0.0476	0.1597
LSTM	0.0687	0.1984
AR LSTM	0.0570	0.1711
AR GRU	0.0457	0.1592
AR SiRNN	0.1309	0.2697
AR StRNN	0.0626	0.1833

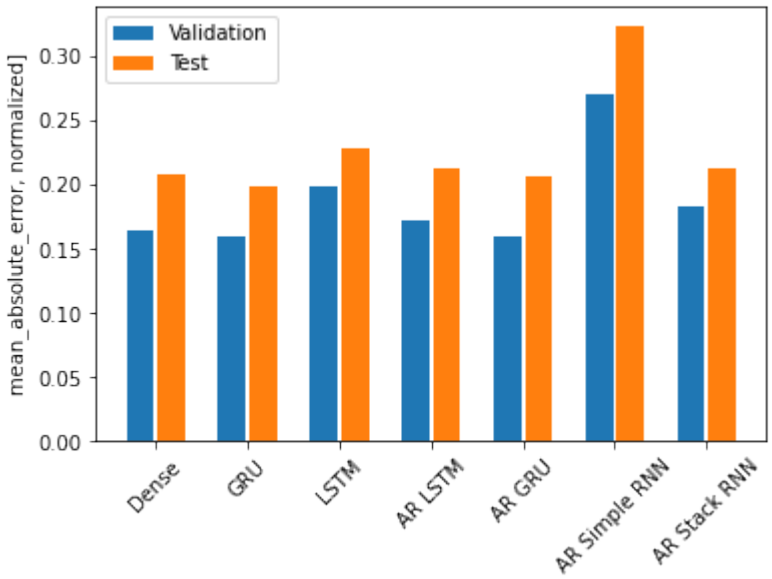

Figure 5: Comparison of mean square error for various types of model

3.6 Hourly test

In the second test we made on hourly data, only difference on variables was that we predicted 24 hours in future, not 14 days as in previous example.

Here the results are more evenly distributed, Dense and GRU both have smallest MSE, other models fared worse.

	loss:	mean_absolute_error
Dense	1.0736	0.7837
GRU	1.1269	0.8089
LSTM	1.4180	0.9153
AR LSTM	1.4477	0.9246
AR GRU	1.4222	0.9157
AR SiRNN	1.4487	0.9265
AR StRNN	1.4344	0.9196

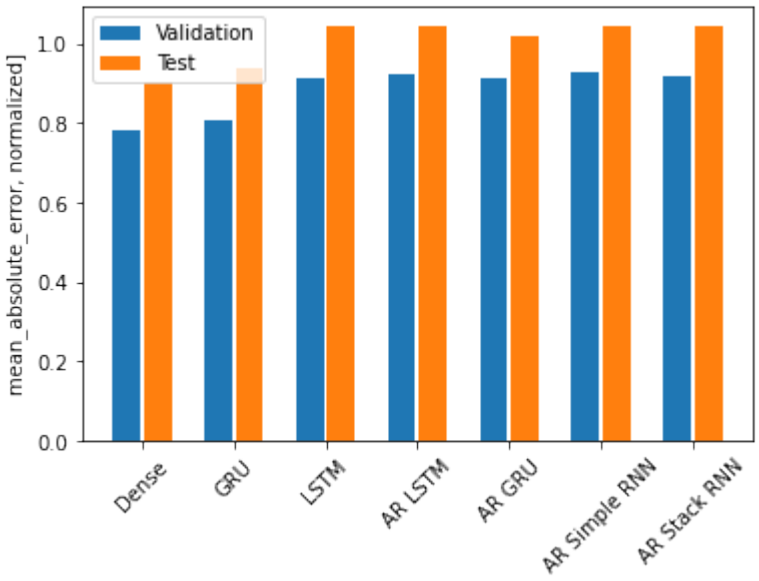

Figure 6: Comparison of mean square error for various types of model

3.7 Bayesian optimization

In the last testing section we launched Bayesian optimization on models shown in previous section, to tweak hyperparameters of each type of model to get the best results.

In following table there are results obtained from optimizing hyperparameters *units* (ranging from 10 to 1000), *learningrate* (0.01, 0.299) and *dropout* (0.01, 0.299) which change precision of the model. In the table we show only the final MSE for test part of each dataset.

	MSE - daily	MSE - hourly
Dense	0.1522	0.7403
GRU	0.1472	0.7312
LSTM	0.1488	0.8207
AR LSTM	0.1611	0.8233
AR GRU	0.1582	0.7858
AR SiRNN	0.1940	0.8087
AR StRNN	0.1833	0.8241

As one can see from the table, the least MSE for both datasets had GRU model, however the difference in MSE was not large for Dense and LSTM model. Autoregressive models fared worse, the best from AR models - AR GRU differed from the best model in total for $\approx 8\%$.

From these result we find important to mention and comment hyperparameters which Bayesian optimization found as the best in the range. For Dense, GRU and LSTM units were close to the number of columns in the dataset (for daily it was around 40 and for hourly it was maximal value 1000), learning rate from 0.09 and 0.105 and dropout around 0.1. Autoregressive models on the other hand worked better with higher number of units (120 for daily, 1000 for hourly), learning rate was also from 0.09 and 0.105 but dropout was set as the lower bound 0.01.

4 Conclusion

In this project we tried to predict prices of silver for several steps in future with several types of neural network and compare how they are useful among themselves and in absolute numbers. To improve precision we optimized hyperparameters in neural network by Bayesian optimization.

In our experiments the best results achieved model using GRU (Gated Recurrent Unit). We improved scores of all models by optimization. However the results also showed that the price of silver is so volatile that only with information from history, even though that we may take houndred of different values, is still very difficult to predict in the future. As we can seen in the example shown in Figure 7 the model can predict correctly the direction of the price (up or down) and sudden change in price which will came, however it cannot predict more precise values according to which would be possible to invest.

In conclusion we successfully showed how to use 7 different types of neural network to predict price of silver in the next x steps and how to improve quality of predicted solution by using Bayesian optimization and everything is available and able to launch completely online, therefore this document may serve as more advanced manual compared to tutorials we mentioned in section Existing solutions.

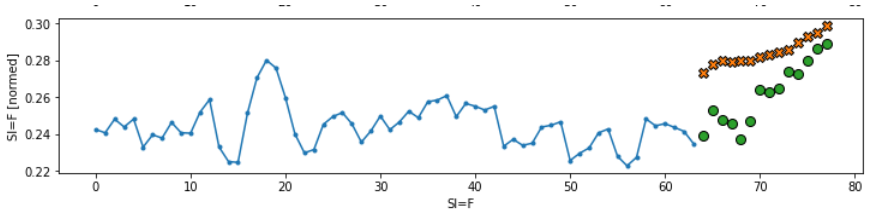

Figure 7: Prediction for 14 days for daily data, y is normalized silver price, x number of days

5 Install: how to?

The documented code and datasets are attached in the folder as this documentation. The program can be launched on system with working Python (3.7+) and libraries such as Pandas, Keras, Numpy, Sklearn, bayes_opt. But the easiest way is to run the code in online Python interpreter Google Colab.