Data Science and ML (Part 32): Keeping your AI models updated, Online Learning

MetaTrader 5 — Statistics and analysis | 21 November 2024, 09:46

Omega J Msigwa

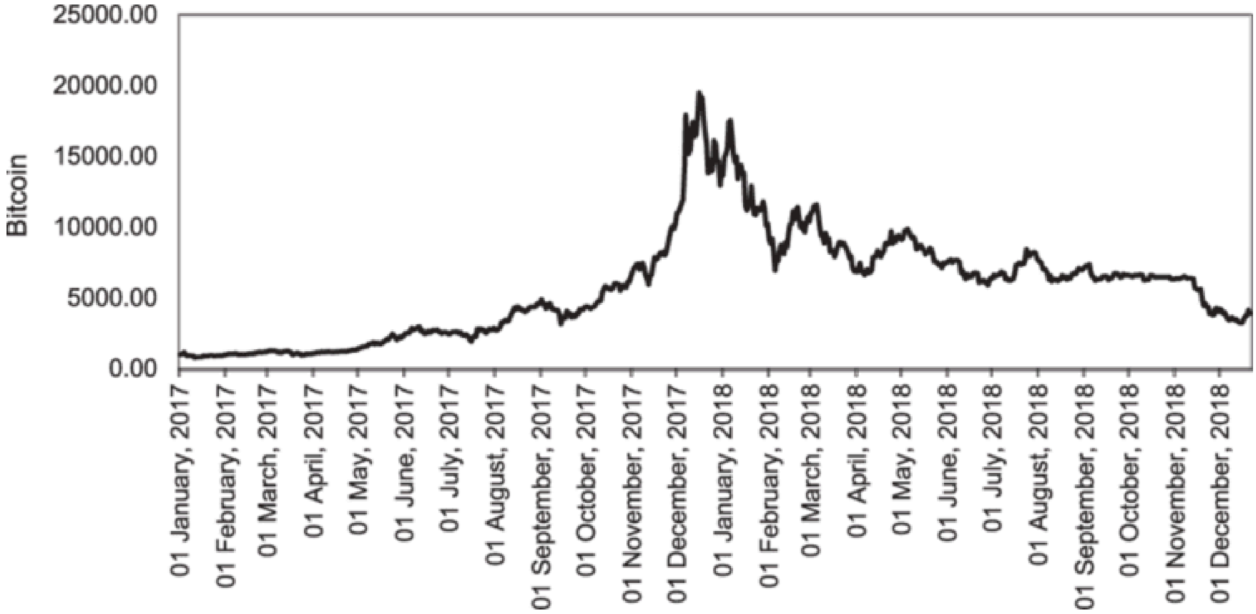
Contents

What is Online Learning?

Online machine learning is a machine learning method in which the model incrementally learns from a stream of data points in real time. It's a dynamic process that adapts its predictive algorithm over time, allowing the model to change as new data arrives. This method is incredibly significant in rapidly evolving data-rich environments such as in trading data as it can provide timely and accurate predictions.

While working with the trading data, it is always hard to determine the right time to update your models and how often for instance, if you have AI models trained on Bitcoin for the last year, the recent information might turn out to be **outliers** for a machine learning model considering this cryptocurrency just hit the new highest price last week.

Unlike forex instruments which usually go up and down within specific ranges historically, instruments like NASDAQ 100, S&P 500 and others of their kind and stocks usually tends to increase and hit new peak values.



Online learning is not only for the fear of the old training information becoming obsolete but also for the sake of keeping the model updated with recent information which might have some impact on what's currently happening in the market.

Benefits of Online Learning

- **Adaptability**

  Just like the cyclists learning as they go, online machine learning can adapt to new patterns in the data potentially improving its performance over time.

- **Scalability**

  Some online learning methods for some models processes data one at a time. This makes this technique safer for tight computational resources that most of us have, this can finally help in scaling models that depend on big data.

- **Real-time predictions**

  Unlike batch learning which might be outdated by the time it's implemented, online learning provides real-time insights which can be critical in many trading applications.

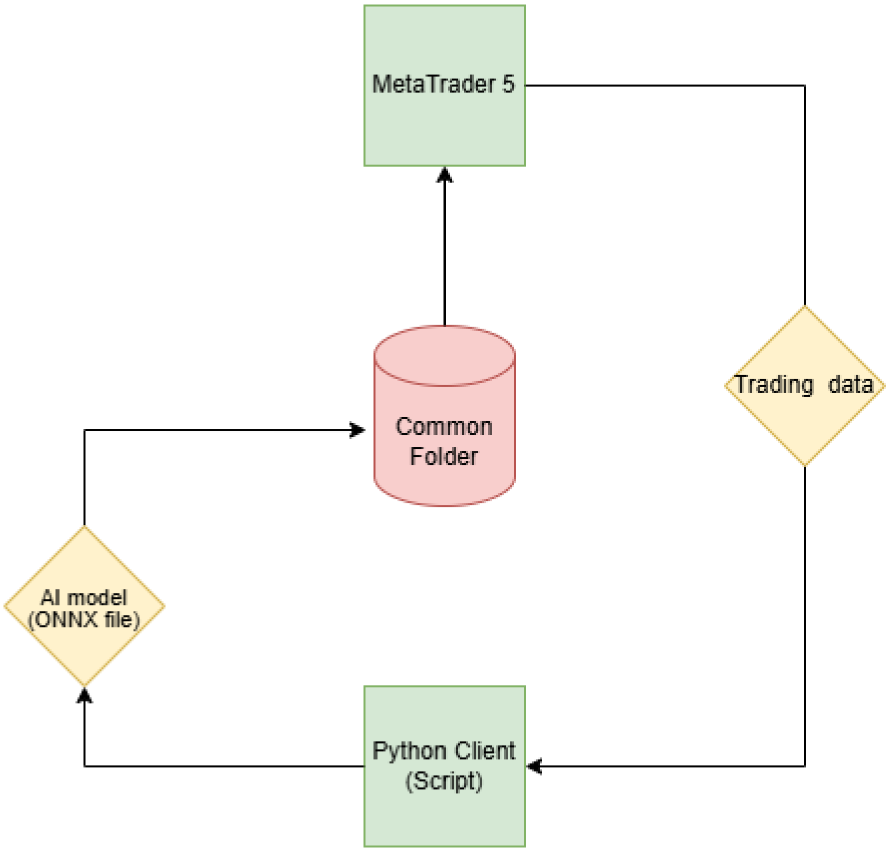
- **Efficiency**

  Incremental machine learning allows for continuous learning and updating of models, which can lead to a faster and more cost-efficient training process.

Now that we understand several benefits of this technique, Let's see the infrastructure it takes to make effective online learning in MetaTrader 5.

Online Learning Infrastructure for MetaTrader 5

Since our final goal is to make AI models useful for trading purposes in MetaTrader 5, it takes a different Online learning infrastructure than that usually seen in Python-based applications.



Step 01: Python Client

Inside a Python client (script) is where we want to build AI models based on the trading data received from MetaTrader 5.

Using MetaTrader 5 (python library), we start by initializing the platform.

```
import pandas as pd
import numpy as np
import MetaTrader5 as mt5
from datetime import datetime

if not mt5.initialize(): # Initialize the MetaTrader 5 platform
    print("initialize() failed")
    mt5.shutdown()
```

After MetaTrader 5 platform initialization, we can obtain trading information from it using the copy_rates_from_pos method.

```
def getData(start = 1, bars = 1000):

    rates = mt5.copy_rates_from_pos("EURUSD", mt5.TIMEFRAME_H1, start, bars)

   if len(rates) < bars: # if the received information is less than specified
        print("Failed to copy rates from MetaTrader 5, error = ",mt5.last_error())

    # create a pnadas DataFrame out of the obtained data

    df_rates = pd.DataFrame(rates)

    return df_rates
```

We can print to see the obtained information.

```
print("Trading info:\n",getData(1, 100)) # get 100 bars starting at the recent closed ba
```

Ouputs

```
          time     open     high      low    close  tick_volume  spread  real_volume
0   1731351600  1.06520  1.06564  1.06451  1.06491         1688       0            0
1   1731355200  1.06491  1.06519  1.06460  1.06505         1607       0            0
2   1731358800  1.06505  1.06573  1.06495  1.06512         1157       0            0
3   1731362400  1.06512  1.06564  1.06512  1.06557         1112       0            0
4   1731366000  1.06557  1.06579  1.06553  1.06557          776       0            0
..         ...      ...      ...      ...      ...          ...     ...          ...
95  1731693600  1.05354  1.05516  1.05333  1.05513         5125       0            0
96  1731697200  1.05513  1.05600  1.05472  1.05486         3966       0            0
97  1731700800  1.05487  1.05547  1.05386  1.05515         2919       0            0
98  1731704400  1.05515  1.05522  1.05359  1.05372         2651       0            0
99  1731708000  1.05372  1.05379  1.05164  1.05279         2977       0            0

[100 rows x 8 columns]
```

We use the **copy_rates_from_pos** method as it allows us to access the recenlyt closed bar placed at the index of 1, this is very useful compared to accessing using dates that are fixed.

We can always be confident that by copying from the bar located at the index of 1, we always get the information starting at the recently closed bar all the way to some specified number of bars we want in.

After receiving this information, we can do the typical machine learning stuff for this data.

We create a separate file for our model, by putting each model in its separate file, we make it easy to call these models in the "**main.py**" file where all the key processes and functions are deployed.

File catboost_models.py

```
from catboost import CatBoostClassifier
from sklearn.metrics import accuracy_score

from onnx.helper import get_attribute_value
from skl2onnx import convert_sklearn, update_registered_converter
from sklearn.pipeline import Pipeline
from skl2onnx.common.shape_calculator import (
    calculate_linear_classifier_output_shapes,
)  # noqa
from skl2onnx.common.data_types import (
    FloatTensorType,
    Int64TensorType,
    guess_tensor_type,
)
from skl2onnx._parse import _apply_zipmap, _get_sklearn_operator_name
from catboost.utils import convert_to_onnx_object

# Example initial data (X_initial, y_initial are your initial feature matrix and target)

class CatBoostClassifierModel():
    def __init__(self, X_train, X_test, y_train, y_test):

        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test
        self.model = None

    def train(self, iterations=100, depth=6, learning_rate=0.1, loss_function="CrossEnt
```

```
        # Initialize the CatBoost model

        params = {
            "iterations": iterations,
            "depth": depth,
            "learning_rate": learning_rate,
            "loss_function": loss_function,
            "use_best_model": use_best_model
        }

        self.model = Pipeline([ # wrap a catboost classifier in sklearn pipeline | good
            ("catboost", CatBoostClassifier(**params))
        ])

        # Testing the model

        self.model.fit(X=self.X_train, y=self.y_train, catboost__eval_set=(self.X_test,

        y_pred = self.model.predict(self.X_test)
        print("Model's accuracy on out-of-sample data = ",accuracy_score(self.y_test, y_

    # a function for saving the trained CatBoost model to ONNX format

    def to_onnx(self, model_name):

        update_registered_converter(
            CatBoostClassifier,
            "CatBoostCatBoostClassifier",
            calculate_linear_classifier_output_shapes,
            self.skl2onnx_convert_catboost,
            parser=self.skl2onnx_parser_castboost_classifier,
            options={"nocl": [True, False], "zipmap": [True, False, "columns"]},
        )

        model_onnx = convert_sklearn(
            self.model,
            "pipeline_catboost",
            [("input", FloatTensorType([None, self.X_train.shape[1]]))],
            target_opset={"": 12, "ai.onnx.ml": 2},
        )

        # And save.
        with open(model_name, "wb") as f:
            f.write(model_onnx.SerializeToString())
```

For more information about this CatBoost model deployed, kindly refer to this article. *I have used the CatBoost model as an example, feel free to use any of your preferred models.*

Now that we have this class to help us with initializing, training, and saving the catboost model. Let us deploy this model in the "**main.py**" file.

File: main.py

Again, we start by receiving the data from the MetaTrader 5 desktop app.

```
data = getData(start=1, bars=1000)
```

If you look closely at the CatBoost model, you'll see that it is a classifier model. We are yet to have the target variable for this classifier, let's make one.

```
# Preparing the target variable

data["future_open"] = data["open"].shift(-1) # shift one bar into the future
data["future_close"] = data["close"].shift(-1)
```

```
target = []
for row in range(data.shape[0]):
    if data["future_close"].iloc[row] > data["future_open"].iloc[row]: # bullish signal
        target.append(1)
    else: # bearish signal
        target.append(0)

data["target"] = target # add the target variable to the dataframe

data = data.dropna() # drop empty rows
```

We can drop all future variables and other features with plenty of zero values from the X 2D array, and assign the "target" variable to the y 1D array.

```
X = data.drop(columns = ["spread","real_volume","future_close","future_open","target"])
y = data["target"]
```

We then split the information into training and validation samples, initialize the CatBoost model with the data from the market, and train it.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_state=

catboost_model = catboost_models.CatBoostClassifierModel(X_train, X_test, y_train, y_te
catboost_model.train()
```

Finally, we save this model in ONNX format in a Common MetaTrader 5 directory.

Step 02: Common Folder

Using the MetaTrader 5 Python, we can get the information on the common path.

```
terminal_info_dict = mt5.terminal_info()._asdict()
common_path = terminal_info_dict["commondata_path"]
```

This is where we want to save all trained AI models from this Python client we have.

> When accessing the common folder using MQL5, it usually refers to a "**Files**" sub-folder found under the common folder, To make it easier to access these files from an MQL5 standpoint, we have to save the models in that subfolder.
>
> ```
> # Save models in a specific location under the common parent folder
>
> models_path = os.path.join(common_path, "Files")
>
> if not os.path.exists(models_path): #if the folder exists
>     os.makedirs(models_path) # Create the folder if it doesn't exist
>
> catboost_model.to_onnx(model_name=os.path.join(models_path, "catboost.H1.onnx"))
> ```

Finally, we have to wrap all these lines of code in a single function to make it easier to carry out all these different processes whenever we want.

```
def trainAndSaveCatBoost():

    data = getData(start=1, bars=1000)

    # Check if we were able to receive some data

    if (len(data)<=0):
```

```
        print("Failed to obtain data from Metatrader5, error = ",mt5.last_error())
        mt5.shutdown()

    # Preparing the target variable

    data["future_open"] = data["open"].shift(-1) # shift one bar into the future
    data["future_close"] = data["close"].shift(-1)

    target = []
    for row in range(data.shape[0]):
        if data["future_close"].iloc[row] > data["future_open"].iloc[row]: # bullish si
            target.append(1)
        else: # bearish signal
            target.append(0)

    data["target"] = target # add the target variable to the dataframe

    data = data.dropna() # drop empty rows

    X = data.drop(columns = ["spread","real_volume","future_close","future_open","targe
    y = data["target"]

    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, random_st

    catboost_model = catboost_models.CatBoostClassifierModel(X_train, X_test, y_train, y
    catboost_model.train()

    # Save models in a specific location under the common parent folder

    models_path = os.path.join(common_path, "Files")

    if not os.path.exists(models_path): #if the folder exists
        os.makedirs(models_path) # Create the folder if it doesn't exist

    catboost_model.to_onnx(model_name=os.path.join(models_path, "catboost.H1.onnx"))
```

Let's call this function then and see what it does.

```
trainAndSaveCatBoost()
exit() # stop the script
```

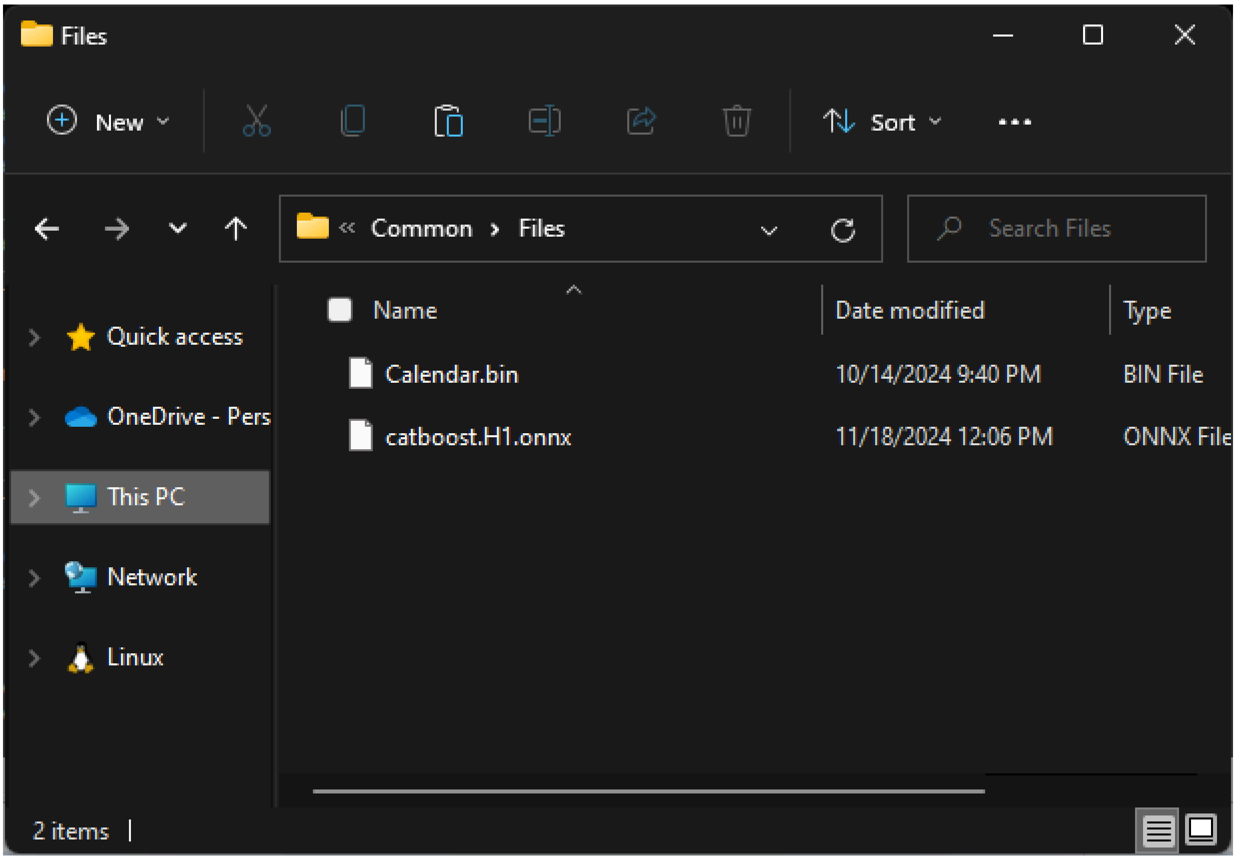
Outcome

```
0:      learn: 0.6916088        test: 0.6934968 best: 0.6934968 (0)     total: 163ms
1:      learn: 0.6901684        test: 0.6936087 best: 0.6934968 (0)     total: 168ms
2:      learn: 0.6888965        test: 0.6931576 best: 0.6931576 (2)     total: 175ms
3:      learn: 0.6856524        test: 0.6927187 best: 0.6927187 (3)     total: 184ms
4:      learn: 0.6843646        test: 0.6927737 best: 0.6927187 (3)     total: 196ms
...
...
...
96:     learn: 0.5992419        test: 0.6995323 best: 0.6927187 (3)     total: 915ms
97:     learn: 0.5985751        test: 0.7002011 best: 0.6927187 (3)     total: 924ms
98:     learn: 0.5978617        test: 0.7003299 best: 0.6927187 (3)     total: 928ms
99:     learn: 0.5968786        test: 0.7010596 best: 0.6927187 (3)     total: 932ms

bestTest = 0.6927187021
bestIteration = 3

Shrink model to first 4 iterations.
Model's accuracy on out-of-sample data =  0.5
```

The .onnx file can be seen under Common\Files.



Step 03: MetaTrader 5

Now in MetaTrader 5, we have to load this model saved in ONNX format.

We start by importing the library to help us with this task.

Inside "Online Learning Catboost.mq5"

```
#include <CatBoost.mqh>
CCatBoost *catboost;

input string model_name = "catboost.H1.onnx";
input string symbol = "EURUSD";
input ENUM_TIMEFRAMES timeframe = PERIOD_H1;

string common_path;
```

The first thing we want to do inside the Oninit function is to check whether the file exists in the common folder, if it doesn't exist, this could indicate that the model wasn't trained.

After that, we initialize the ONNX model by passing the [ONNX_COMMON_FOLDER] flag to explicitly load the model from the "Common folder".

```
int OnInit()
  {
//--- Check if the model file exists

   if (!FileIsExist(model_name, FILE_COMMON))
     {
       printf("%s Onnx file doesn't exist",__FUNCTION__);
       return INIT_FAILED;
     }

//--- Initialize a catboost model

    catboost = new CCatBoost();
    if (!catboost.Init(model_name, ONNX_COMMON_FOLDER))
      {
```

```
        printf("%s failed to initialize the catboost model, error = %d",__FUNCTION__,GetLa
        return INIT_FAILED;
     }

//---
}
```

To use this loaded model to make predictions, we can go back to the Python script and check what features were used for training after some were dropped.

The same features and in the same order must be collected in MQL5.

Python code "main.py" file.

```
X = data.drop(columns = ["spread","real_volume","future_close","future_open","target"])
y = data["target"]

print(X.head())
```

Outcome

```
         time     open     high      low    close  tick_volume
0  1726772400  1.11469  1.11584  1.11453  1.11556         3315
1  1726776000  1.11556  1.11615  1.11525  1.11606         2812
2  1726779600  1.11606  1.11680  1.11606  1.11656         2309
3  1726783200  1.11656  1.11668  1.11590  1.11622         2667
4  1726786800  1.11622  1.11644  1.11605  1.11615         1166
```

Now, let's obtain this information inside the OnTick function and call the **predict_bin** function which predicts classes.

This function will predict two classes that were seen in the target variable we prepared in the Python client. 0 (bullish), 1 (bearish).

```
void OnTick()
  {
//---
     MqlRates rates[];
     CopyRates(symbol, timeframe, 1, 1, rates); //copy the recent closed bar informatio

     vector x = {
                  (double)rates[0].time,
                  rates[0].open,
                  rates[0].high,
                  rates[0].low,
                  rates[0].close,
                  (double)rates[0].tick_volume};

     Comment(TimeCurrent(),"\nPredicted signal: ",catboost.predict_bin(x)==0?"Bearish":
  }
```

Outcome

Automating the Training and Deployment Process

We were able to train and deploy the model in MetaTrader 5 but, this isn't what we want, our main goal is to automate the entire process.

Inside the Python virtual environment, we have to install the schedule library.

```
$ pip install schedule
```

This small module can help in scheduling when we want a specific function to be executed. Since we already wrapped the code for collecting data, training, and saving the model in one function, let's schedule this function to be called after every (one) minute.

```
schedule.every(1).minute.do(trainAndSaveCatBoost) #schedule catboost training

# Keep the script running to execute the scheduled tasks
while True:
    schedule.run_pending()
    time.sleep(60)  # Wait for 1 minute before checking again
```

This scheduling thing works like a charm :)

In our main Expert Advisor, we also schedule when and how often our EA should load the model from the common directory, in doing so, we effectively update the model for our trading robot.

We can use the OnTimer function which also works like a charm :)

```
int OnInit()
  {
//--- Check if the model file exists

  ....

//--- Initialize a catboost model

....

//---
```

```
    if (!EventSetTimer(60)) //Execute the OnTimer function after every 60 seconds
      {
        printf("%s failed to set the event timer, error = %d",__FUNCTION__,GetLastError(
        return INIT_FAILED;
      }


//---
    return(INIT_SUCCEEDED);
  }
//+------------------------------------------------------------------+
//| Expert deinitialization function                                 |
//+------------------------------------------------------------------+
void OnDeinit(const int reason)
  {
//---
     if (CheckPointer(catboost) != POINTER_INVALID)
       delete catboost;
  }
//+------------------------------------------------------------------+
//| Expert tick function                                             |
//+------------------------------------------------------------------+
void OnTick()
  {
//---
      ....
  }
//+------------------------------------------------------------------+
//|                                                                  |
//+------------------------------------------------------------------+
void OnTimer(void)
  {
     if (CheckPointer(catboost) != POINTER_INVALID)
       delete catboost;

//--- Load the new model after deleting the prior one from memory

      catboost = new CCatBoost();
      if (!catboost.Init(model_name, ONNX_COMMON_FOLDER))
        {
          printf("%s failed to initialize the catboost model, error = %d",__FUNCTION__,Ge
          return;
        }

      printf("%s New model loaded",TimeToString(TimeCurrent(), TIME_DATE|TIME_MINUTES));
  }
```

Outcome

```
HO      0       13:14:00.648    Online Learning Catboost (EURUSD,D1)    2024.11.18 12:1
FK      0       13:15:55.388    Online Learning Catboost (GBPUSD,H1)    2024.11.18 12:1
JG      0       13:16:55.380    Online Learning Catboost (GBPUSD,H1)    2024.11.18 12:1
MP      0       13:17:55.376    Online Learning Catboost (GBPUSD,H1)    2024.11.18 12:1
JM      0       13:18:55.377    Online Learning Catboost (GBPUSD,H1)    2024.11.18 12:1
PF      0       13:19:55.368    Online Learning Catboost (GBPUSD,H1)    2024.11.18 12:1
CR      0       13:20:55.387    Online Learning Catboost (GBPUSD,H1)    2024.11.18 12:2
NO      0       13:21:55.377    Online Learning Catboost (GBPUSD,H1)    2024.11.18 12:2
LH      0       13:22:55.379    Online Learning Catboost (GBPUSD,H1)    2024.11.18 12:2
```

Now that we have seen how you can schedule the training process and keep the new models in sync with the ExpertAdvisor in MetaTrader 5. While the process is easy to implement for most machine learning techniques, it could be a challenging process when working with deep learning models such as Recurrent Neural Networks(RNNs), which can't be contained inside the Sklearn pipeline which makes our life easier when working with various machine learning models.

Let us see how you can apply this technique when working with a Gated Recurrent Unit(GRU) which is a special form of a recurrent neural network.

Online Learning for Deep Learning AI models

In Python Client

We apply the typical machine learning stuff inside the GRUClassifier class. For more information on GRU kindly refer to this article.

After training the model we save it to ONNX, **this time we also save the StandardScaler's information in binary files**, this will help us later in similarly normalizing the new data in MQL5 as it currently stands in Python.

File gru_models.py

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dense, Input, Dropout
from keras.callbacks import EarlyStopping
from keras.optimizers import Adam
import tf2onnx


class GRUClassifier():
    def __init__(self, time_step, X_train, X_test, y_train, y_test):

        self.X_train = X_train
        self.X_test = X_test
        self.y_train = y_train
        self.y_test = y_test
        self.model = None
        self.time_step = time_step
        self.classes_in_y = np.unique(self.y_train)


    def train(self, learning_rate=0.001, layers=2, neurons = 50, activation="relu", batc

        self.model = Sequential()
        self.model.add(Input(shape=(self.time_step, self.X_train.shape[2])))
        self.model.add(GRU(units=neurons, activation=activation)) # input layer


        for layer in range(layers): # dynamically adjusting the number of hidden layers

            self.model.add(Dense(units=neurons, activation=activation))
            self.model.add(Dropout(0.5))

        self.model.add(Dense(units=len(self.classes_in_y), activation='softmax', name='

        # Compile the model
        adam_optimizer = Adam(learning_rate=learning_rate)
        self.model.compile(optimizer=adam_optimizer, loss=loss, metrics=['accuracy'])


        early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_we

        history = self.model.fit(self.X_train, self.y_train, epochs=epochs, batch_size=b
                                 validation_data=(self.X_test, self.y_test),
                                 callbacks=[early_stopping], verbose=verbose)

        val_loss, val_accuracy = self.model.evaluate(self.X_test, self.y_test, verbose=v
```

```
        print("Gru accuracy on validation sample = ",val_accuracy)


    def to_onnx(self, model_name, standard_scaler):

        # Convert the Keras model to ONNX
        spec = (tf.TensorSpec((None, self.time_step, self.X_train.shape[2]), tf.float16
        self.model.output_names = ['outputs']

        onnx_model, _ = tf2onnx.convert.from_keras(self.model, input_signature=spec, op

        # Save the ONNX model to a file
        with open(model_name, "wb") as f:
            f.write(onnx_model.SerializeToString())

        # Save the mean and scale parameters to binary files
        standard_scaler.mean_.tofile(f"{model_name.replace('.onnx','')}.standard_scaler
        standard_scaler.scale_.tofile(f"{model_name.replace('.onnx','')}.standard_scale
```

Inside the "main.py" file, we create a function responsible for everything we want to happen with the GRU model.

```
def trainAndSaveGRU():

    data = getData(start=1, bars=1000)

    # Preparing the target variable

    data["future_open"] = data["open"].shift(-1)
    data["future_close"] = data["close"].shift(-1)

    target = []
    for row in range(data.shape[0]):
        if data["future_close"].iloc[row] > data["future_open"].iloc[row]:
            target.append(1)
        else:
            target.append(0)

    data["target"] = target

    data = data.dropna()

    # Check if we were able to receive some data

    if (len(data)<=0):
        print("Failed to obtain data from Metatrader5, error = ",mt5.last_error())
        mt5.shutdown()

    X = data.drop(columns = ["spread","real_volume","future_close","future_open","targe
    y = data["target"]

    X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7, shuffle=F

    ########### Preparing data for timeseries forecasting ###############

    time_step = 10

    scaler = StandardScaler()

    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    x_train_seq, y_train_seq = create_sequences(X_train, y_train, time_step)
    x_test_seq, y_test_seq = create_sequences(X_test, y_test, time_step)

    ###### One HOt encoding #######
```

```
    y_train_encoded = to_categorical(y_train_seq)
    y_test_encoded = to_categorical(y_test_seq)

    gru = gru_models.GRUClassifier(time_step=time_step,
                                    X_train= x_train_seq,
                                    y_train= y_train_encoded,
                                    X_test= x_test_seq,
                                    y_test= y_test_encoded
                                    )

    gru.train(
        batch_size=64,
        learning_rate=0.001,
        activation = "relu",
        epochs=1000,
        loss="binary_crossentropy",
        layers = 2,
        neurons = 50,
        verbose=1
        )

    # Save models in a specific location under the common parent folder

    models_path = os.path.join(common_path, "Files")

    if not os.path.exists(models_path): #if the folder exists
        os.makedirs(models_path) # Create the folder if it doesn't exist

    gru_to_onnx(model_name=os.path.join(models_path, "gru.H1.onnx"), standard_scaler=sc
```

Finally, we can schedule how often this **trainAndSaveGRU** function should be called into action, similarly to how we scheduled for the CatBoost function.

```
schedule.every(1).minute.do(trainAndSaveGRU) #scheduled GRU training
```

Outcome

```
Epoch 1/1000
11/11 ━━━━━━━━━━━━━━━━━━━━ 7s 87ms/step - accuracy: 0.4930 - loss: 0.6985
Epoch 2/1000
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.4847 - loss: 0.6957
Epoch 3/1000
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.5500 - loss: 0.6915
Epoch 4/1000
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.4910 - loss: 0.6923
Epoch 5/1000
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.5538 - loss: 0.6910
Epoch 6/1000
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - accuracy: 0.5037 - loss: 0.6953
Epoch 7/1000
...
...
...
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.4964 - loss: 0.6952
Epoch 20/1000
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.5285 - loss: 0.6914
Epoch 21/1000
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.5224 - loss: 0.6935
Epoch 22/1000
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.5009 - loss: 0.6936
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.4925 - loss: 0.6938
Gru accuracy on validation sample =  0.5103448033332825
```

In MetaTrader 5

We start by loading the libraries to help us with the task of loading the GRU model **and the standard scaler.**

```
#include <preprocessing.mqh>
#include <GRU.mqh>

CGRU *gru;
StandardizationScaler *scaler;

//--- Arrays for temporary storage of the scaler values
double scaler_mean[], scaler_std[];

input string model_name = "gru.H1.onnx";

string mean_file;
string std_file;
```

The first thing we want to do in the OnInit function is to get the names of the scaler binary files, *we applied this same principle when creating these files.*

```
 string base_name__ = model_name;

 if (StringReplace(base_name__,".onnx","")<0)
   {
     printf("%s Failed to obtain the parent name for the scaler files, error = %d",__FUI
     return INIT_FAILED;
   }

  mean_file = base_name__ + ".standard_scaler_mean.bin";
  std_file = base_name__ + ".standard_scaler_scale.bin";
```

Finally, we proceed to load the GRU model in ONNX format from the common folder, we also read the scaler files in binary format by assigning their values in the **scaler_mean** and **scaler_std** arrays.

```
int OnInit()
  {

   string base_name__ = model_name;

   if (StringReplace(base_name__,".onnx","")<0) //we followed this same file patterns wl
     {
       printf("%s Failed to obtain the parent name for the scaler files, error = %d",__I
       return INIT_FAILED;
     }

   mean_file = base_name__ + ".standard_scaler_mean.bin";
   std_file = base_name__ + ".standard_scaler_scale.bin";

//--- Check if the model file exists

   if (!FileIsExist(model_name, FILE_COMMON))
     {
       printf("%s Onnx file doesn't exist",__FUNCTION__);
       return INIT_FAILED;
     }

//--- Initialize the GRU model from the common folder

     gru = new CGRU();
     if (!gru.Init(model_name, ONNX_COMMON_FOLDER))
       {
         printf("%s failed to initialize the gru model, error = %d",__FUNCTION__,GetLas
         return INIT_FAILED;
```

```
        }

//--- Read the scaler files

   if (!readArray(mean_file, scaler_mean) || !readArray(std_file, scaler_std))
     {
       printf("%s failed to read scaler information",__FUNCTION__);
       return INIT_FAILED;
     }

   scaler = new StandardizationScaler(scaler_mean, scaler_std); //Load the scaler class

//--- Set the timer

   if (!EventSetTimer(60))
     {
       printf("%s failed to set the event timer, error = %d",__FUNCTION__,GetLastError()
       return INIT_FAILED;
     }

//---
   return(INIT_SUCCEEDED);
  }
//+------------------------------------------------------------------+
//| Expert deinitialization function                                 |
//+------------------------------------------------------------------+
void OnDeinit(const int reason)
  {
//---

    if (CheckPointer(gru) != POINTER_INVALID)
      delete gru;
    if (CheckPointer(scaler) != POINTER_INVALID)
      delete scaler;
  }
```

We schedule the process of reading the scaler and model files from the common folder in the OnTimer function.

```
void OnTimer(void)
  {
//--- Delete the existing pointers in memory as the new ones are about to be created

    if (CheckPointer(gru) != POINTER_INVALID)
      delete gru;
    if (CheckPointer(scaler) != POINTER_INVALID)
      delete scaler;

//---

   if (!readArray(mean_file, scaler_mean) || !readArray(std_file, scaler_std))
     {
       printf("%s failed to read scaler information",__FUNCTION__);
       return;
     }

   scaler = new StandardizationScaler(scaler_mean, scaler_std);

     gru = new CGRU();
     if (!gru.Init(model_name, ONNX_COMMON_FOLDER))
       {
         printf("%s failed to initialize the gru model, error = %d",__FUNCTION__,GetLas
         return;

       }
```

```
    printf("%s New model loaded",TimeToString(TimeCurrent(), TIME_DATE|TIME_MINUTES));
  }
```

Outcome

```
 II      0       14:49:35.920    Online Learning GRU (GBPUSD,H1) 2024.11.18 13:49 New mo
 QP      0       14:50:35.886    Online Learning GRU (GBPUSD,H1) Initilaizing ONNX model
 MF      0       14:50:35.919    Online Learning GRU (GBPUSD,H1) ONNX model Initialized
 IJ      0       14:50:35.919    Online Learning GRU (GBPUSD,H1) 2024.11.18 13:50 New mo
 EN      0       14:51:35.894    Online Learning GRU (GBPUSD,H1) Initilaizing ONNX model
 JD      0       14:51:35.913    Online Learning GRU (GBPUSD,H1) ONNX model Initialized
 EL      0       14:51:35.913    Online Learning GRU (GBPUSD,H1) 2024.11.18 13:51 New mo
 NM      0       14:52:35.885    Online Learning GRU (GBPUSD,H1) Initilaizing ONNX model
 KK      0       14:52:35.915    Online Learning GRU (GBPUSD,H1) ONNX model Initialized
 QQ      0       14:52:35.915    Online Learning GRU (GBPUSD,H1) 2024.11.18 13:52 New mo
 DK      0       14:53:35.899    Online Learning GRU (GBPUSD,H1) Initilaizing ONNX model
 HI      0       14:53:35.935    Online Learning GRU (GBPUSD,H1) ONNX model Initialized
 MS      0       14:53:35.935    Online Learning GRU (GBPUSD,H1) 2024.11.18 13:53 New mo
 DI      0       14:54:35.885    Online Learning GRU (GBPUSD,H1) Initilaizing ONNX model
 IL      0       14:54:35.908    Online Learning GRU (GBPUSD,H1) ONNX model Initialized
 QE      0       14:54:35.908    Online Learning GRU (GBPUSD,H1) 2024.11.18 13:54 New mo
```

To receive the predictions from the GRU model, we have to consider the **timestep value** which helps Recurrent Neural Networks(RNNs) understand temporal dependencies in the data.

We used a timestep value of ten(10) inside the function "**trainAndSaveGRU**".

```
def trainAndSaveGRU():

    data = getData(start=1, bars=1000)

     ....
     ....

    time_step = 10
```

Let's collect the last 10 bars (timesteps) from history starting from the recently closed bar in MQL5. *(this is how it is supposed to be)*

```
input int time_step = 10;
```

```
void OnTick()
  {
//---
     MqlRates rates[];
     CopyRates(symbol, timeframe, 1, time_step, rates); //copy the recent closed bar in

     vector classes = {0,1}; //Beware of how classes are organized in the target variab

     matrix X = matrix::Zeros(time_step, 6); // 6 columns
     for (int i=0; i<time_step; i++)
       {
         vector row = {
                   (double)rates[i].time,
                   rates[i].open,
                   rates[i].high,
                   rates[i].low,
                   rates[i].close,
                   (double)rates[i].tick_volume};

         X.Row(row, i);
       }
```

```
        X = scaler.transform(X); //it's important to normalize the data
        Comment(TimeCurrent(),"\nPredicted signal: ",gru.predict_bin(X, classes)==0?"Beari
     }
```

Outcome

Incremental Machine Learning

Some models are more proficient and robust than others when it comes to the training methods. When you are searching for "Online machine learning" on the internet, most folks say that it is a process through which small batches of data are retrained back to the model for the bigger training goal.

The problem with this is that many models do not support or work well when given a small sample of data.

Modern machine learning techniques like CatBoost comes with incremental learning in mind. This training method can be used for Online learning and it can help save a lot of memory when working with big data, as data can be split into small chunks which can be re-trained back to the initial model.

```
def getData(start = 1, bars = 1000):

    rates = mt5.copy_rates_from_pos("EURUSD", mt5.TIMEFRAME_H1, start, bars)

    df_rates = pd.DataFrame(rates)

    return df_rates

def trainIncrementally():

    # CatBoost model
    clf = CatBoostClassifier(
        task_type="CPU",
        iterations=2000,
        learning_rate=0.2,
        max_depth=1,
        verbose=0,
    )

    # Get big data
    big_data = getData(1, 10000)

    # Split into chunks of 1000 samples
    chunk_size = 1000
    chunks = [big_data[i:i + chunk_size].copy() for i in range(0, len(big_data), chunk_s
```

```
    for i, chunk in enumerate(chunks):

        # Preparing the target variable

        chunk["future_open"] = chunk["open"].shift(-1)
        chunk["future_close"] = chunk["close"].shift(-1)

        target = []
        for row in range(chunk.shape[0]):
            if chunk["future_close"].iloc[row] > chunk["future_open"].iloc[row]:
                target.append(1)
            else:
                target.append(0)

        chunk["target"] = target

        chunk = chunk.dropna()

        # Check if we were able to receive some data

        if (len(chunk)<=0):
            print("Failed to obtain chunk from Metatrader5, error = ",mt5.last_error())
            mt5.shutdown()

        X = chunk.drop(columns = ["spread","real_volume","future_close","future_open","
        y = chunk["target"]

        X_train, X_val, y_train, y_val = train_test_split(X, y, train_size=0.8, random_

        if i == 0:
            # Initial training, training the model for the first time
            clf.fit(X_train, y_train, eval_set=(X_val, y_val))

            y_pred = clf.predict(X_val)
            print(f"---> Acc score: {accuracy_score(y_pred=y_pred, y_true=y_val)}")
        else:
            # Incremental training by using the intial trained model
            clf.fit(X_train, y_train, init_model="model.cbm", eval_set=(X_val, y_val))

            y_pred = clf.predict(X_val)
            print(f"---> Acc score: {accuracy_score(y_pred=y_pred, y_true=y_val)}")

        # Save the model
        clf.save_model("model.cbm")
        print(f"Chunk {i + 1}/{len(chunks)} processed and model saved.")
```

Outcome

```
---> Acc score: 0.555
Chunk 1/10 processed and model saved.
---> Acc score: 0.505
Chunk 2/10 processed and model saved.
---> Acc score: 0.55
Chunk 3/10 processed and model saved.
---> Acc score: 0.565
Chunk 4/10 processed and model saved.
---> Acc score: 0.495
Chunk 5/10 processed and model saved.
---> Acc score: 0.55
Chunk 6/10 processed and model saved.
---> Acc score: 0.555
Chunk 7/10 processed and model saved.
---> Acc score: 0.52
Chunk 8/10 processed and model saved.
---> Acc score: 0.455
```

```
Chunk 9/10 processed and model saved.
---> Acc score: 0.535
Chunk 10/10 processed and model saved.
```

You can follow the same online learning architecture while building the model incrementally and save the final model in ONNX format in the "Common Folder" for MetaTrader 5 usage.

Final Thoughts

Online learning is an excellent approach for keeping models continuously updated with minimal manual intervention. By implementing this infrastructure, you can be sure that your models stay aligned with the latest market trends and are adapting quickly to new information. However, it is important to note that online learning can sometimes make models highly sensitive to the order in which data is processed very often human oversight may be necessary to verify that the model and training information makes logical sense from a human perspective.

You need to find the right balance between automating the learning process and periodic evaluation of your models to ensure everything goes as expected.

Attachments Table

Infrastructure (Folders)	Files	Description & Usage
Python Client	- catboost_models.py - gru_models.py - main.py - incremental_learning.py	- A CatBoost model can be found in this file - A GRU model can be found in this file - The main python file for putting it all together - Incremental learning for CatBoost model is deployed in this file
Common Folder	- catboost.H1.onnx - gru.H1.onnx - gru.H1.standard_scaler_mean.bin - gru.H1.standard_scaler_scale.bin	All the AI models in ONNX format and the scaler files in binary formats can be found in this folder
MetaTrader 5 (MQL5)	- Experts\Online Learning Catboost.mq5 - Experts\Online Learning GRU.mq5 - Include\CatBoost.mqh - Include\GRU.mqh - Include\preprocessing.mqh	- Deploys a CatBoost model in MQL5 - Deploys a GRU model in MQL5 - A library file for initializing and deploying a CatBoost model in ONNX format - A library file for initializing and deploying a GRU model in ONNX format - A library file which contains the StandardScaler for normalizing the data for ML model usage

Attached files | Download ZIP
Attachments.zip (475 KB)

Other articles by this author

- Data Science and ML (Part 42): Forex Time series Forecasting using ARIMA in Python, Everything you need to Know

- Building MQL5-Like Trade Classes in Python for MetaTrader 5
- Data Science and ML (Part 41): Forex and Stock Markets Pattern Detection using YOLOv8
- Data Science and ML (Part 40): Using Fibonacci Retracements in Machine Learning data
- Data Science and ML (Part 39): News + Artificial Intelligence, Would You Bet on it?
- Data Science and ML (Part 38): AI Transfer Learning in Forex Markets
- Data Science and ML (Part 37): Using Candlestick patterns and AI to beat the market

Mutual information as criteria for Stepwise Feature Selection

In this article, we present an MQL5 implementation of Stepwise Feature Selection based on the mutual information between an optimal predictor set and a target variable.

Neural Networks Made Easy (Part 93): Adaptive Forecasting in Frequency and Time Domains (Final Part)

In this article, we continue the implementation of the approaches of the ATFNet model, which adaptively combines the results of 2 blocks (frequency and time) within time series forecasting.

Price Action Analysis Toolkit Development (Part 2): Analytical Comment Script

Aligned with our vision of simplifying price action, we are pleased to introduce another tool that can significantly enhance your market analysis and help you make well-informed decisions. This tool displays key technical indicators such as previous day's prices, significant support and resistance levels, and trading volume, while automatically generating visual cues on the chart.

Automating Trading Strategies in MQL5 (Part 1): The Profitunity System (Trading Chaos by Bill Williams)

In this article, we examine the Profitunity System by Bill Williams, breaking down its core components and unique approach to trading within market chaos. We guide readers through implementing the system in MQL5, focusing on automating key indicators and entry/exit signals. Finally, we test and optimize the strategy, providing insights into its performance across various market scenarios.

Install Platform

Uninstall Platform

articles and
CodeBase updates

Not a broker, no real trading accounts

35 Dodekanisou str, Germasogeia, 4043, Limassol, Cyprus