Mémoire d'iniciation à la recherche
Deep Reinforcement Learning for Financial Markets

Hugo ABREU
supervised by
Eric Benhamou and David Saltiel

1/07/2020

Contents

1 Introduction

In this memoir, I worked on the implementation of Reinforcement Learning (RL) techniques to financial trading problems. The two topics were new to me, so I will be explaining them before describing my implementation.

Trading is a very researched topic, that encompasses fields such as mathematics, economics, and computer science. I will present the theoretical basis of trading in financial markets, the most used technical indicators, as well as some established approaches.

Reinforcement Learning are machine learning methods that aim to solve complex problems, where there isn't a clear description of a solution (as opposed to supervised learning approach in artificial intelligence). I will be looking at the theoretical background of RL, and more specifically Deep RL, and discussing its implementation

I try to implemented the approach described in the article *Deep Reinforcement Learning for Trading*[ZZR19]. I will discuss my interpretation of the article and how I translated it to a python implementation.

Implementing the reinforcement part was difficult because it was a new topic for me - I still don't have a working code. I will finish to implement it in a later date.

2 Financial Trading

Financial trading involves buying and selling 'financial instruments' with goal of getting some profit. The "financial instruments" can be shares, forex (currency), or derivatives, such as CFDs, futures, and options. I will be focusing on Futures.

The implicit objective of trading is, with a given capital, choose a set of actions (buying or selling stock) that maximizes the return on that initial capital.

While there exists traders that work by themselves, manually inputting what trade to execute, most trading is made by algorithms, nowadays. Since the objective is to minimize risk, clearly defined strategies can make more profits in the long run.

We can formulate trading as a Markov Decision Process. An agent takes actions in an environment. Those actions are buying, holding and selling assets. In the case of trading, the environment corresponds to the market: the prices of the assets. We can build more complex environments, giving the agent more data (technical indicators that are correlated to the evolution of the market, such as volatility), so that it can perform better.

When the agent performs an action, he can receive a reward. The objective is, with the reward function, to find a set of actions that maximizes the return.

This problem formulation is good for reinforcement learning, but many other types of algorithms exist. Just using technical indicators to advise actions, for instance, can provide good results.

2.1 Some technical indicators

- *Relative Strength Index*: The RSI is a technical indicator that charts the current and historical strength or weakness of a stock or market, based on the closing prices of a recent trading period.
- *Moving Average Convergence Divergence*: MACD is a trend-following momentum indicator, that connects two moving averages of an asset's price. The MACD is calculated by subtracting the 26-period Exponential Moving Average (EMA) from the 12-period EMA.

3 Deep Reinforcement Learning

3.1 Idea

In a Reinforcement Learning problem, there is a learner and a decision maker. The learner is the agent, that interacts with an environment. The environment, in return, provides rewards and a new state, taking into account the actions of the agent. So, in reinforcement learning, we do not teach an agent how it should do something - but present it with rewards based on the consequences of its actions.

3.2 Main Notions

Policies:

- *Policy*: The function that allows us to compute the next action for a particular state.
- An *optimal Policy* is a policy that maximizes the expected reward/reinforcement/feedback of a state.

Thus, the task of RL is to use observed rewards to find an optimal policy for the environment.
Reinforcement Learning can also have different mnodes of learning:

- *Passive Learning*: Agents policy is fixed and our task is to learn how good the policy is.
- *Active Learning*: Agents must learn what actions to take.
- *Off-policy learning*: learn the value of the optimal policy independently of the agent's actions.
- *On-policy learning*: learn the value of the policy the agent actually follows.

3.3 Algorithms

Many forms of reinforcement learning exist. We will be looking at deep reinforcement learning algorithms. When the decisions to become too complex for the reinforced learning approach - because the algorithm can't learn from all states and determine the reward path - we can use Deep Reinforcement learning. The 'deep' portion refers to the application of a neural network to estimate the states instead of having to map every solution, creating a more manageable solution space in the decision process.

Let's explore one reinforcement learning algorithm: Q-Learning. It's an off-policy learning process, where the environment is typically formulated as a Markov Decision Process (MDP). The MDP is formulated as follows:

- Finite sets of states $S = \{s_0, \ldots, s_n\}$ and actions $A = \{a_0, \ldots, a_m\}$.
- Probabilities $P_a(s, s')$ for transitions from state s to s' with action a.
- Reward function R that adjusts probabilities over time.
- Goal is to learn an optimal policy function $Q^*(s, a)$.

4 An implementation of Zhang et al.'s "Deep Reinforcement Learning for financial Trading"

We will be analysing and implementing Zhang et al.'s approach to Financial Trading, described in "Deep Reinforcement Learning for financial Trading". Hereinafter, this paper will be simply referred to as "the paper".

4.1 Analysis and Interpretation

In the paper, the authors propose a reinforcement learning approach to Financial Trading. Contrary to classical trading approaches, no forecasting step takes place: the models directly output trade positions.

As previously seen, trading can be formalized as a Markov Decision Process - which can be addressed with a Deep Reinforcement Learning framework. In this model, an agent interacts with its environment at discrete time steps.

4.1.1 State Space

The authors introduce 4 features:

- Normalised close price series

  We normalize the close prices according to the entire range. With this information we define a scaler. If we are testing, we will use the training scaler.

- Past returns, for 1 month, 2 months, 3 months and 1 year

  These are normalized by the square root of the number of days in the period and the daily volatility, which corresponds to the exponentially weighted moving standard deviation (EMSD) in the last 60 days.

- Moving Average Convergence Divergence (MACD)

  It's a measure that is more sensitive to recent changes (as defined previously).

  But in this paper, the author propose a variation: They normalize the MACD histogram by the standard deviation of MACD for the previous year. We implemented this in our environment function.

  I only found one other author that used a similar approach. [MK19]

- Relative Strength Index

  Their implementation of RSI is the classical one. Refer to the first par of the report.

4.1.2 Action Space

We consider a discrete action space:

- Buy: close a short position and open a long position. If there is no long position, do nothing
- Sell: close a short position and open the short position if having a short position. If not, do nothing

4.1.3 Reward

In the paper, the reward function is defined as follows:

$$R_t = \mu \left(A_{t-1} \frac{\sigma_{tgt}}{\sigma_{t-1}} r_t - bp\, p_{t-1} \left| \frac{\sigma_{tgt}}{\sigma_{t-1}} A_{t-1} - \frac{\sigma_{tgt}}{\sigma_{t-2}} A_{t-2} \right| \right) \tag{1}$$

Let's define and interpret each term:

- A_t corresponds to the action selected at time t. Since we use a discrete action space, A_t can take three values:
  - $A_t = 1$: the agent buys a stock at time t, so it corresponds to a buy position
  - $A_t = 0$: the agent does nothing at time t, it corresponds to a hold position
  - $A_t = -1$: the agent buys a stock at time t, it corresponds to a sell position

Note that R_t, the reward at time t, is independent of the action A_t at time t: it only uses the previous two actions - A_{t-1} and A_{t-2}

- μ depends on the assets being traded. For Futures, it corresponds to the number of currencies that can be purchased - it is set to 1 as detailed in the paper.
- σ_t corresponds to the volatility of close prices at time t.
- σ_{tgt}: I didn't understand the function of this term - according to the paper, it's value corresponds to a percentage of the value of return for a particular period of time. We are going to attribute a constant value to it.
- p_t corresponds to the price at time t
- r_t: returns for time t, $= p_t - p_{t-1}$
- bp corresponds to the *basis point* - it's the percentage of transaction fees for each transaction. It's a constant, defined to 0.0001 in the paper.

The purpose of the reward function is to learn appropriate actions, corresponding to input features from the results of transaction in the training data.

We can interpret the function as follows:

The expression in brackets corresponds to the change in holding status of a long or short position, for on currency bought and sold one episode ago. μ corresponds to the number of currencies bought and sold.

The first item inside this expression, evaluates the action one episode earlier: it takes a positive value if the price has gone up and a negative value if the price has gone down. As such, $A_{t-1}r_t$ takes a positive value if we buy, and the price has gone up - or if we sell, and the price has gone down. If we hold our position, the value is 0. In short, it represents whether the trade was correct or not.

$\frac{\sigma_{tgt}}{\sigma_{t-1}}$ is a coefficient for scaling the previous item: in situations of low volatility, the value will be higher. In situations of high volatility, the value will be lower. It makes the agent avoid situations of high volatility.

$-bp\,p_{t-1}$ represents the transaction cost.

4.2 Implementation

We created two distinct python programs: one for the environment and one for the agent. The agent program will call the environment program.

4.2.1 Environment

We create a class Environment that takes care of creating the features representing the a given state, and that feeds them to the Agent. It is also in Environment that we compute the reward, and create the training/testing separation.

4.2.1.1 Data retrieval

We use cls files - which are the standard format for market price history - that contain 5 seperate indicators, for each time step t: the date, the high price, the low price, the close price and the daily returns (in percentage).

We first normalize all the data. It's important that we don't normalize testing data with a min-max approach, so when testing, we will use python's standard scaler. In the paper, only the close price are said to be normalized.

Input program 1: *Data extraction*

```python
class Data(object):

    def __init__(self, path, file, thousands = ','):
        self.file = file
        self.full = pd.read_csv(path + file, thousands =
        ↪  thousands)
        self.ohlc = self.full[['Date', 'Open', 'High',
                               'Low', 'Price', 'Change
                               ↪  %']][::-1]

        self.ohlc['Change %'] = self.ohlc['Change
        ↪  %'].str.rstrip('%').astype('float') / 100.0

        self.ohlc.columns = ['Date', 'Open', 'High',
                             'Low', 'Close', 'Change']

    def preprocess_data(self, scaler=None):
        if scaler == None:
            scaler = StandardScaler()
            scaler.fit(data)

        scaled_data = scaler.transform(self.raw_data)
        self.norm_close_price = scaled_data
```

4.2.1.2 State computation

For each state, we need to provide the past 60 observations of each feature. So we dynamically access the raw data to be able to compute each indicator.

The features for a given time t will be in the form of a pandas DataFrame.

Input program 2: *Compute features for a given state*

```python
def compute_state(self):
    # 60 observations indexes
    indexes = [index for index in range(self.step - 59, self.step + 1)]

    # feature dictionary init
    features = {
        'close price' : self.norm_close_price,
        '1 month returns' : [],
        '2 month returns' : [],
        '3 month returns' : [],
        '1 year returns'  : [],
        'macd'            : self.compute_macd(),
        'rsi'             : self.compute_rsi()
    }

    # retrieve observations for the current step
    for index in indexes:
        print("in index", index)
        # if observation doesn't exist, compute it
        if index not in self.observations:
            self.observations[index] = self.compute_observation(index)

        # add observation to feature dictionary
        feature_list = self.observations[index]

        features['close price'].append(feature_list[0])
        features['1 month returns'].append(feature_list[1])
        features['2 month returns'].append(feature_list[2])
        features['3 month returns'].append(feature_list[3])
        features['1 year returns'].append(feature_list[4])

    # compute normalised close price series from close month price series

    features['normalised close price series'] = cp_norm
    del features['close price']

    features_df = pd.DataFrame(data=features, index =
    ↪  indexes)

    self.current_state = features_df
```

4.2.2 Feature computation

I built functions in the Environment class to compute the features:

Input program 3: *MACD*

```
def compute_macd(self):
    def compute_qt(self, index):
        print("in compute_qt")
        obs_63 = self.raw_data[index - 62 : index + 1]

        m_S = obs_63['Close'].ewm(span=self.short_window,
        ↪  adjust=False).mean()
        m_L = obs_63['Close'].ewm(span=self.long_window,
        ↪  adjust=False).mean()

        rolling_std = obs_63['Close'].std()

        return ((m_S - m_L) / rolling_std).values.tolist()

    def compute_qt_list(self):
        qt_list = []
        for index in range(self.step - 251, self.step + 1):
            # compute qt
            qt_list.append(compute_qt(self, index))
        return np.array(qt_list)

    qt = compute_qt(self, self.step)[3:]

    qt_std = np.std(compute_qt_list(self), axis=0).mean()
    print(qt_std)

    return [x / qt_std for x in qt]
```

Input program 4: *MACD*

```python
def normalised_returns(self, index):
    # periods
    period_1m  = self.raw_data[index - self.nb_days_month +
    ↪  1:
                               index + 1]
    period_2m  = self.raw_data[index - 2 *
    ↪  self.nb_days_month + 1:
                               index + 1]
    period_3m  = self.raw_data[index - 3 *
    ↪  self.nb_days_month + 1:
                               index + 1]
    period_1y  = self.raw_data[index - self.nb_days_year +
    ↪  1:
                               index + 1]

    # daily_volatility
    daily_volatility_1y = period_1y['Change'].ewm(span =
    ↪  60).std().mean()
    daily_volatility_3m = period_3m['Change'].ewm(span =
    ↪  20).std().mean()
    daily_volatility_2m = period_2m['Change'].ewm(span =
    ↪  10).std().mean()
    daily_volatility_1m = period_1m['Change'].ewm(span =
    ↪  5).std().mean()


    # normalised returns
    returns_1y_norm = ((period_1y['Close'][-1:].item() -
    ↪  period_1y['Close'][:1].item())
                       / (period_1y['Close'][:1].item() *
                       ↪  daily_volatility_1y *
                       ↪  np.sqrt(self.nb_days_year)))
    returns_3m_norm = ((period_3m['Close'][-1:].item() -
    ↪  period_3m['Close'][:1].item())
                       / (period_3m['Close'][:1].item() *
                       ↪  daily_volatility_3m *
                       ↪  np.sqrt(3*self.nb_days_month)))
    returns_2m_norm = ((period_2m['Close'][-1:].item() -
    ↪  period_2m['Close'][:1].item())
```

```
                        / (period_2m['Close'][:1].item() *
                        ↪  daily_volatility_2m *
                        ↪  np.sqrt(2*self.nb_days_month)))
    returns_1m_norm = ((period_1m['Close'][-1:].item() -
    ↪  period_1m['Close'][:1].item())
                        / (period_1m['Close'][:1].item() *
                        ↪  daily_volatility_1m *
                        ↪  np.sqrt(self.nb_days_month)))

    return returns_1m_norm, returns_2m_norm,
    ↪  returns_3m_norm, returns_1y_norm
```

Input program 5: *RSI*

```
def compute_rsi(self, price_arr, cur_pos, period = 30):
    if self.step <= 60:
        return 0
    else:
        s = cur_pos - 61
    tmp_arr = self.raw_dats[s :self.step]['Close']
    prices = np.array(tmp_arr, dtype=float)

    # we can use the ta-lib library to compute RSI
    rsi_val = ta.RSI(prices, timeperiod = 60)[-1]
    return rsi_val
```

4.2.3 Agent

I made several attempts to make an Agent class, but didn't manage to have a full working code.

5 Concluding Remarks

I discovered two new topics that were unknown to me: Reinforcement Learning and Financial Markets. I got a good understanding of how to represent a market, the tools that we use to modelize its trends and make intelligent choices.

I implemented the representation of the features, which seem to describe well the trend of the price.

I haven't reached the memoir's objectives. Nevertheless, I learned a lot about financial trading and Reinforcement Learning. I liked this subject a lot, so I will be working on it to arrive at the goals that were set. The Agent part still needs to be developped, which I want to do with more time this summer. When I have a working reinforcement learning implementation, I can see if I can develop an extrinsic reward signal - as is described in [PAED17].

References

[BGH+15] Jamil Baz, Nicolas M Granger, Campbell R. Harvey, Nicolas Le Roux, and Sandy Rattray. Dissecting Investment Strategies in the Cross Section and Time Series. *SSRN Electronic Journal*, 2015.

[BK12] Akindynos-Nikolaos Baltas and Robert Kosowski. Improving Time-Series Momentum Strategies: The Role of Trading Signals and Volatility Estimators. *SSRN Electronic Journal*, 2012.

[Kla19] Jannes Klaas. Machine Learning for finance: The practical guide to using data-driven algorithms in banking, insurance, and investments, 2019. OCLC: 1137786464.

[LZR19] Bryan Lim, Stefan Zohren, and Stephen Roberts. Enhancing Time Series Momentum Strategies Using Deep Neural Networks. *SSRN Electronic Journal*, 2019.

[Mak06] Don K. Mak. *Mathematical techniques in financial market trading.* World Scientific, Hackensack, N.J, 2006. OCLC: ocm63117060.

[MK19] Terry Lingze Meng and Matloob Khushi. Reinforcement Learning in Financial Markets. *Data*, 4(3):110, July 2019.

[MOP12] Tobias J. Moskowitz, Yao Hua Ooi, and Lasse Heje Pedersen. Time series momentum. *Journal of Financial Economics*, 104(2):228–250, May 2012.

[PAED17] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-Driven Exploration by Self-Supervised Prediction. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 488–489, Honolulu, HI, USA, July 2017. IEEE.

[Sch17] Jack D. Schwager. *A complete guide to the futures market: technical analysis and trading systems, fundamental analysis, options, spreads, and trading principles.* Wiley trading series. Wiley, Hoboken, New Jersey, second edition edition, 2017.

[ZZR19] Zihao Zhang, Stefan Zohren, and Stephen Roberts. Deep Reinforcement Learning for Trading. *arXiv:1911.10107 [cs, q-fin]*, November 2019. arXiv: 1911.10107.