Machine Learning Engineer Nanodegree

Capstone Proposal

October 13th, 2018

Proposal

Domain Background

The Financial Market is a complex and dynamical system, and is influenced by many factors that are subject to uncertainty. Therefore, it is a difficult task to forecast stock price movements. Machine Learning aims to automatically learn and recognize patterns in large data sets. The self organizing characteristics of Machine Learning algorithms suggest that such algorithms might be effective to tackle the task of predicting stock price fluctuations, and in developing automated trading strategies based on these predictions

Financial markets are now extremely efficient, nevertheless there are still many investment funds that generate systematically beating markets' return benchmarks. that continuously consumes streams of data across multiple diverse markets. We demonstrate a simple scalable trading model that learns to generate profit from multiple inter-market price predictions and markets' correlation structure.

In this Section , I would like to explain that training Reinforcement Learning agents to trade in the financial markets can be an extremely interesting research problem. I believe that now days lot of start-up companies are trying to build the strong solution, but it has not received enough attention from the research community . I think it has the potential to push in this related fields. introduce the basic concepts of RL and present two learning algorithms that allow two determine an approximation for the optimal policy of a sequential decision problem.

Problem Statement

I am going to use the reinforcement Learning to predict the prices /strategy of Two stocks based on 5 years of Historical Data that I have downloaded from Kaggle. After that I have to set up the MDP parameters (State , Reward , Gamma , State Transition , Action) and Agent will use the defined optional policy to make it Trend and Will change the reward function. The goal of the agent is to learn by trial-and-error which actions maximize his long-run rewards

Datasets and Inputs

Dataset is gathered from Kaggle:

https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs/home

Content

The data is presented in CSV format as follows: Date, Open, High, Low, Close, Volume, OpenInt.

I will use following columns from data set: Date, Open, Close and Volume

The data is for a period of more than 5 years (various stocks have various data points).

I will be planning to use 2 equities from the data set- say for e.g Google (googl.us.txt) and Unilever (un.us.txt)

When I decide on the number of epochs to use (say 1000), I will divide the dataset in chronological order from a start date D. So:

- Training data will be from Day D to Day D+1000.
- Then I will do a validation from day D+1001 to D+1200
- Test will be D+1201 to D+1400

Solution Statement

I will use Reinforcement Learning (RL) as general class of algorithms in the field of Machine Learning (ML) that allows an agent to learn how to behave in a stochastic and possibly unknown environment, where the only feedback consists of a scalar reward signal . The goal of the agent is to learn by trial-and-error which actions maximize his long-run rewards. However, since the environment evolves stochastically and may be influenced by the actions chosen, the agent must balance his desire to obtain a large immediate reward by acting greedily and the opportunities that will be available in the future. Thus, RL algorithms can be seen as computational methods to solve sequential decision problems by directly interacting with the environment. In my Capstone project .. I am going to use only two Equities for buying and selling .. I will simply take a single position at time .. Agent will sell the stocks ..if Stocks are physically available in my bucket. Also buy the stocks based on amount availability .

As stated above, to make the model easy, my agent will only trade in 2 equities, say A and B. It can take either long or short position in either or both of the 2 stocks.

- It can only buy a stock (either A or B or both) if it has sufficient amount to buy the stock, so no leverage
- It can only sell the stock (A or B or both) if it has these stocks in its portfolio

Using reinforcement learning, the agent will decide when is it right moment to buy, hold or sell stock A or Stock B.

Benchmark Model

Benchmark model is model defined in the paper:
http://www1.mate.polimi.it/~forma/Didattica/ProgettiPacs/BrambillaNecchi15-16/PACS_Report_Pierpaolo_Necchi.pdf

Benchmark model can be a model where the agent for the training period:

For every period of 1 month:

- Buys stocks A and Stock B, with half amount invested in each stock
- Holds these stocks for 1 month
- Sells these stocks at the end of 1 month

Continue till end of training period

Evaluation Metrics

Here are a few of the most basic metrics that traders are using. I will use the same for my evaluation:

Net PnL (Net Profit and Loss)

Simply how much money an algorithm makes (positive) or loses (negative) over some period of time, minus the trading fees.

Alpha and Beta

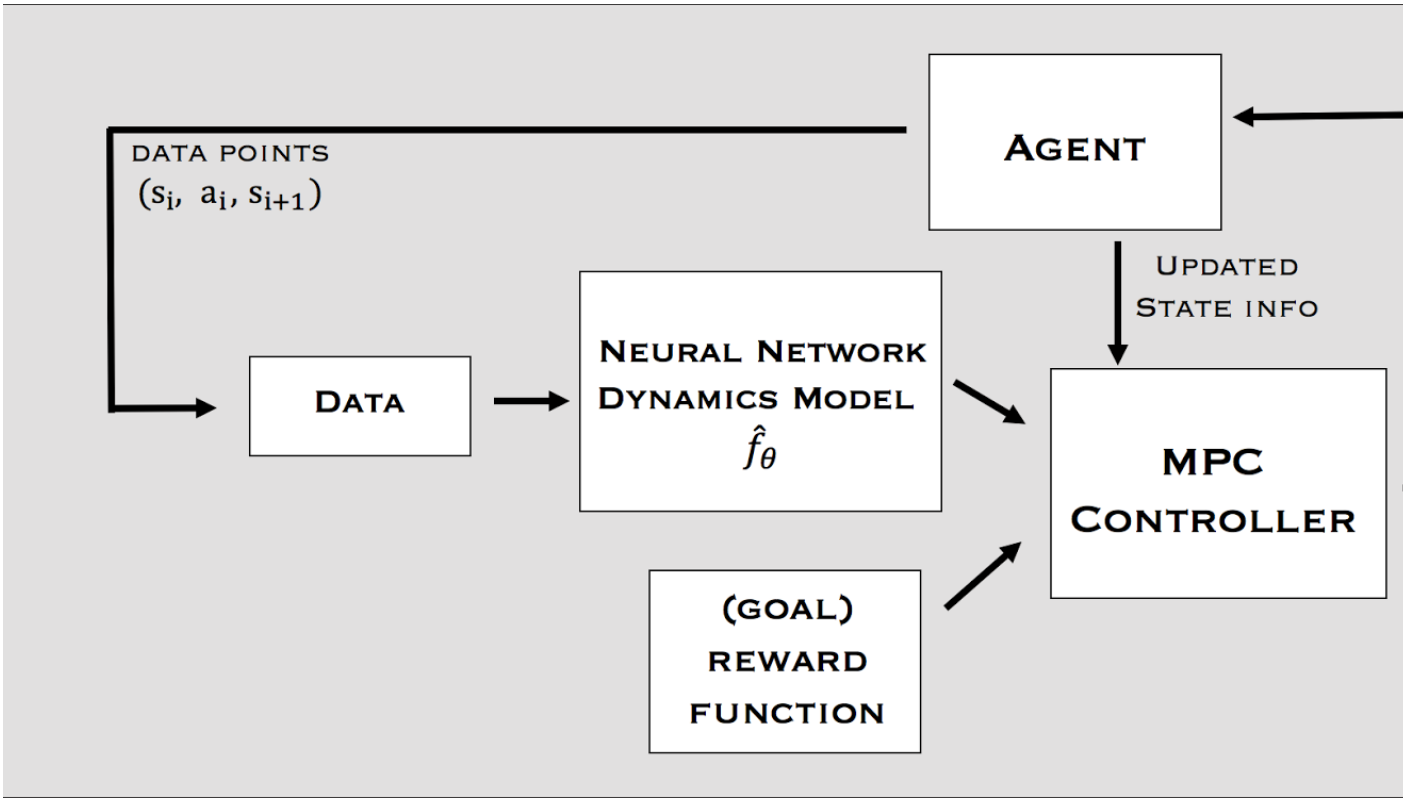
Alpha defines how much better, in terms of profit, your strategy is when compared to an alternative, relatively risk-free, investment, like a government bond. Even if your strategy is profitable, you could be better off investing in a risk-free alternative. Beta is closely related, and tells you how volatile your strategy is compared to the market. For example, a beta of 0.5 means that your investment moves $1 when the market moves $2.

Sharpe Ratio

The Sharpe Ratio measures the excess return per unit of risk you are taking. It's basically your return on capital over the standard deviation, adjusted for risk. Thus, the higher the better. It considers both the volatility of your strategy, as well as an alternative risk-free investment.

Sharpe ratio = (Mean portfolio return − Risk-free rate)/Standard deviation of portfolio return

Project Design



Please find the below steps in sequence to build the algorithm by reinforcement learning.

1. Data Analysis and visualization: Download the Dataset from Kaggle and do visualization and pre-analysis of data
2. Select 2 stocks as input for the model
3. Define Markov Decision process parameters:
    a. State (2 stocks, and their Prices),
    b. Actions (Buy, Hold, Sell stocks),
    c. Reward Function (This is described in next point)
    d. State transition matrix
    e. Decay factor: Defined between 0 and 1
4. Reward Function: This can be simple PnL, profit realized in a day or can be complex based on Sharpe ratio. It is very important to have right reward function
5. Agent can perform tasks buy, hold or sell stocks so that long term reward is maximised.
6. Constraint: Agent has to buy only with amount of money it has and sell only the stocks it has (no short selling).

7. Select optimal policy for sequential decision problem using value iteration or Policy iteration
8. Agent will use Neural network and we will run epochs with 2 to 3 years (out of data from 2005 to 2017) of data and we will check the performance
9. Based on agent's performance change the reward function (if needed) and fine-tune hyper-parameters of the neural network
10. Check the model performance on data (where Epochs were not earlier run on) and visualize performance of agent

Environment : most important part is to design the environment. The environment class should implement the following attributes / methods based on the OpenAI / gymconvention:

Init : For initialization of the environment at the beginning of the episode.

State: Holds the price of A and B at any given time = t.

Step: The change in environment after one time step. With each call to this method, the environment returns 4 values described below:

a) *next_state*: The state as a result of the action performed by the agent. In our case, it will always be the Price of A and B at t = t + 1

b) *reward*: Gives the reward associated with the action performed by the Agent.

c) *done*: whether we have reached the end of the episode.

d) *info*: Contains diagnostic information.

Reset: To reset the environment after every episode of training. In this case, it restores the prices of both A and B to their respective means and simulates new price path.

Its a good practice to keep the environment code separate from that of the agent. Doing so, will make it easier to modify the environment's behavior and training the agent on the fly. I wrote a Python class called *market_env* to implement its behavior.

A sample path of 500 time steps for the two assets generated by the environment with A(blue): mean = 100.0, vol = 10% and B(green): mean = 100.0, vol = 20% using the Ornstein–Uhlenbeck process (plotted using *python/matplotlib*) is shown below. As you can see that the two processes cross each other many times exhibiting a co-integration property, an ideal ground to train the agent for a long-short strategy.

AGENT

The agent is a **MLP** *(Multi Layer Perceptron)* multi-class classifier neural network taking in two inputs from the environment: Price of A and B resulting in actions : (0) Long A, Short B (1) Short A, Long B (2) Do nothing, subject to

maximizing the overall reward in every step. After every action, it receives the next observation (state) and the reward associated with its previous action. Since the environment is stochastic in nature, the agent operates through a **MDP** (M*arkov Decision Process*) i.e. the next action is entirely based on the current state and not on the history of prices/states/actions and it discounts the future reward(s) with a certain measure (gamma). The score is calculated with every step and saved in the Agent's memory along with the action, current state and the next state. The cumulative reward per episode is the sum of all the individual scores in the lifetime of an episode and will eventually judge the performance of the agent over its training. The complete workflow diagram is shown below:

Why should this approach even work ? Since the spread of the two co-integrated processes exhibits a stationary property i.e. it has a constant mean and variance over time and can be thought of as having a normal distribution. The agent can identify this statistical behavior by buying and selling A and B simultaneously based on their price spread (= Price_A— Price_B) . For example, if the spread is negative it implies that A is cheap and B is expensive, the agent will figure the action would be to go long A and short B to attain the higher reward. The agent will try to approximate this through the *Q(s, a)*function where 's' is the state and 'a' is the optimal action associated with that state to maximize its returns over the lifetime of the episode. The policy for next action will be determined using Bellman Ford Algorithm as described by the equation below:

Through this mechanism, it will also appreciate the long term prospects than just immediate rewards by assigning different Q values to each action. This is the crux of *Reinforcement Learning*. Since the input space can be massively large, we will use a Deep Neural Network to approximate the *Q(s, a)* function through backward propagation. Over multiple iterations,

the *Q(s, a)* function will converge to find the optimal action in every possible state it has explored.

Speaking of the internal details, it has two major components:

1. *Memory*: Its a list of events. The Agent will store the information through iterations of *exploration and exploitation*. It contains a list of the format: (state, action, reward, next_state, message)

2. *Brain*: This is the Fully Connected, Feed-Forward Neural Net which will train from the memory i.e. past experiences. Given the current state as input, it will predict the next optimal action.

To train the agent, we need to build our Neural Network which will learn to classify actions based on the inputs it receives. (A simplified Image below. Of course the real neural net will be more complicated than this.).

In the above image,

Inputs(2): Price of A and B in green.

Hidden(2 layers): Denoted by 'H' nodes in blue.

Output(3): classes of actions in red.

For implementation, I am using Keras and Tensorflow both of which are free and open source python libraries.

The neural net is trained with an arbitrarily chosen sample size from its memory at the end of every episode in real-time hence after every episode the network collects more data and trains further from it. As a result of that, the *Q(s, a)* function would converge with more iterations and we will see the agent's performance increasing over time until it reaches a saturation point. The returns/rewards are scaled in the image below.