# Billionaire

Beat an AI and be the first billionaire!

– Course project for COMP3359 AI Applications

Group members:

Guo Shunhua (3035447635)

Chen Xin (3035233553)

## 1 Project Objectives

This project will explore the application of cutting-edge AI algorithms in the financial industry, specifically using Deep Reinforcement Learning. With an attractive competitive game implemented, we will demonstrate the capability of this robust investment AI.

## 2 Description

### 2.1 Application

As AI increasingly appears on the headlines of financial pages, e.g., "AI for selecting stocks" and "Robo-advisor for investment", people are not only interested in "which stock is the best for the year selected by AI" but also "when to buy and sell the particular stock by how many to maximize the return". This project will explore an application of Deep RL to train a learning agent on historical data to learn investment. This agent will be performing buy and sell actions given the historical data information prior to the trading time to maximize its reward, i.e., portfolio returns. The detailed setup of the model will be explained in following sections.

After successfully trained the investment AI, we will implement a small game for the human player to compete with this oriented investment AI on trading a particular stock. The game will provide buy and sell options for the player and calculate the portfolio value on every transaction. To simplify the model, a player will buy the stock at the Open price and sell it at the Close price. And players have no prior knowledge of any price data of the trading date, and in other words, players can only make decisions based on previous price data. We will provide a reasonable amount of money for both the AI and the human player to start their investment, and the goal is to be the first billionaire in the game.

Want to beat this AI to be the first billionaire? Come and play!

## 2.2 Dataset

The dataset this project will be using is from Kaggle, named Huge Stock Market Dataset. As introduced in the previous section, the investment AI will be trained on financial data and trade with it. We select this dataset with the full historical daily price and volume data for all US-based stocks and ETFs trading on the NYSE, NASDAQ, and NYSE MKT after a careful selection process. This daily historical data contains Date, Open, High, Low, Close, Volume and OpenInt, with the price adjusted for dividends distribution and stock splits.

# 3 Methodology

Our project uses a deep reinforcement learning (RL) algorithm, Deep-Q Network (DQN) [1], to train an investment AI. In this section, we will first introduce notations and relevant reinforcement learning frameworks, then elucidate how our stock trading game can be implemented using DQN.

## 3.1 Deep-Q Network

In standard RL settings, the environment $\varepsilon$ an agent interacts with usually involves a sequential decision making process and is often assumed to be stochastic. In this project, we consider a finite-horizon Markov Decision Process (MDP), defined by the tuple $(S, A, P, R)$, where $S$ is the state space, $A$ is a discrete action space, $P$ is the unknown state transition function $S \times A \to [0, \infty)$ that decides the probability density of the next state $s_{t+1}$, given current state and the action taken. $R$ is the reward $S \times A \to \mathbb{R}$ that the environment returns. We further denote the AI's policy as $\pi$, where it maps a state $s_t$ to a distribution over the action space.

The goal of an RL agent is to maximize its expected sum of reward $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where $T$ is the time step that a game terminates. In DQN's algorithm setting, the policy $\pi$ maps a given state $s$ to expected $Q$ values of each action in the action space. The optimal $Q$ value given state and action obeys the Bellman Equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim \varepsilon} \left[ r + \gamma \max_{a'} Q^*\left(s', a'\right) \mid s, a \right] \tag{1}$$

where $s'$ means the next state, and $a'$ stands for an action in the next state. It essentially means that the optimal strategy for selecting an action is to pick the one that maximizes the $r + \gamma \max_{a'} Q^*\left(s', a'\right)$ term.

Since the policy $\pi$ is a function approximator of the state-action value function $Q$, we can train it through enforcing it to obey the Bellman equation above. Therefore, the loss function at each iteration $i$ can be written as:

$$L_i\left(\theta_i\right) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[ \left( y_i - Q\left(s, a; \theta_i\right) \right)^2 \right] \tag{2}$$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} Q\left(s', a'; \theta_{i-1}\right) \mid s, a \right]$ is the target for iteration $i$. Since in a sequential decision making environment, states are connected sequentially and the state visitation is highly reliant on a neural network and its action selection, it contradicts with one basic and important assumption behind deep learning: The training data is Independent and identically distributed (IID). Therefore, the authors of the DQN algorithm further introduced a component called Randomized Experience Replay to stabilize the training process. This is especially critical when we use mini-batches for gradient updates as it prevents gradients from oscillating throughout the training process.

Another common practice in reinforcement learning is to balance the exploration-exploitation trade-off. While there are many methods explicitly encourage exploration, one simplest way to nudge exploration is to deploy the $\epsilon$-greedy strategy like in most $Q$-learning algorithms, or in the case of Policy Gradient methods, adding an entropy term into loss functions.

## 3.2 Trading Stocks with DQN

Stock trading can be framed as a sequential decision making problem throughout certain time windows. Therefore, components in investment can be integrated into defining an MDP. In this project, we set the state input $s_t$ as an $n$-dimension vector containing relevant data on date $t$ in the aforementioned dataset, including Open, High, Low, Close, Volume, and OpenInt, etc. The action space $A$ contains only two actions, buy and sell. The agent's policy $\pi$ is a Multi-Layer Perceptron (MLP) taking vector input $s_t$ and output a 2-dimension vector representing action values for each action. The agent's reward $r$ at each time step (i.e., day) is given by its profit (or loss) averaged over the course of time it holds the shares. We decide to average out the reward to each time step instead of having one reward at the end of the trajectory (when the agent sells the stock) because DQN is notoriously inefficient in solving environments with sparse reward. As with most model-free RL frameworks, we assume the transition probability $P$ is unknown and do not explicitly model such probability.

Throughout the course of our training and implementation, we will also consider deploying some common practices in stabilizing reinforcement learning agents, like decaying the $\epsilon$ value, using Huber loss [2], and learning rate decay.

Since the dataset contains information from thousands of stocks and ETFs, it is not directly deployable to our model. For data preprocessing, we plan to first select a subset of them to form an index, then generate relevant data for this index (Open, High, Low, Close, etc.). Unlike supervised learning where people usually split the dataset into train/validation/test sets, reinforcement learning agents learn and get tested in the same underlying environment. However, we are still interested in its performance in unseen time period. Therefore, we plan to use 80% of data for training the agent, and at test time only evaluate its performance on the remaining 20%. For example, if we have 5 years of data for this index throughout 2011-2015, we will use the 2011-2014 data for training and the 2015 data for testing.

## 4 Platform and tools

We will use PyTorch [3] as our primary framework for implementing the DQN algorithm. In data preprocessing, we will first load the csv dataset into Pandas [4] frames, or optionally an IterableDataset in PyTorch. To better monitor our development process, we might use Sacred [5] for storing experimental results in a clean way.

In hyperparameter tuning, we might use the Ray Tune [6] package to manage experiment runs and scikit-optimize [7] for finding an optimal set of hyperparameters. Throughout the tuning process, we will monitor loss curves and return curves to decide whether the training is successful or not. We plan to code the interactive game using functions provided by Jupyter notebook [8] with ipywidgets. The whole process of collaborative development will be coordinated through Git.

## 5 Timeline and division of labour

1. Project Setup by March 3 (Guo Shunhua, Chen Xin)
    - Download dataset and prepare it for the algorithm; Learn Deep Q Network; Setup Github repository,
    - Finish project proposal
    - Submission Item: Proposal March 13
2. Training process by April 3
    - Load data (Guo Shunhua)
    - Write the algorithm (Guo Shunhua, Chen Xin)
    - Hyperparameter tuning (Chen Xin)
    - Submission Item: Interim Prototype April 3
3. UI Design and Game design by April 15
    - Widgets implementation (Chen Xin)
    - Improvement (Guo Shunhua, Chen Xin)
4. Analysis and Visualization by April 15 (Guo Shunhua, Chen Xin)
    - Analysis and Visualization
    - Submission Item: Final work April 24
5. (Possible) Extensions of implementation
    - Add different difficulty levels of the game
    - Improve the game rule: Player's decisions change the future stock prices

# References


[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. nature, 518(7540):529–533, 2015.

[2] Peter J Huber. Robust estimation of a location parameter. In Breakthroughs in statistics, pages 492–518. Springer, 1992.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. arXiv preprint arXiv:1912.01703, 2019.

[4] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[5] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The sacred infrastructure for computational research. In Proceedings of the 16th Python in Science Conference, volume 28, pages 49–56, 2017.

[6] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. arXiv preprint arXiv:1807.05118, 2018.

[7] Tim Head, MechCoder, Gilles Louppe, Iaroslav Shcherbatyi, fcharras, Zé Vinícius, cmmalone, Christopher Schröder, nel215, Nuno Campos, Todd Young, Stefano Cereda, Thomas Fan, rene rex, Kejia (KJ) Shi, Justus Schwabedal, carlosdanielcsantos, Hvass-Labs, Mikhail Pak, SoManyUsernamesTaken, Fred Callaway, Loïc Estève, Lilian Besson, Mehdi Cherti, Karlson Pfannschmidt, Fabian Linzberger, Christophe Cauet, Anna Gut, Andreas Mueller, and Alexander Fabisch. scikit-optimize/scikit-optimize: v0.5.2, March 2018.

[8] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. Computing in Science and Engineering, 9(3):21–29, May 2007.