DEEP REINFORCEMENT LEARNING APPLIED TO STOCK TRADING

Jan Komisarczyk[1], **Younghoon Kim**[2], and **Mathias Ruoss**[3]

[1, 2, 3]**Department of Mathematics, ETH, Zurich, Switzerland**
{jkomisarczyk, kimyoun, mruoss}@student.ethz.ch

Abstract

In this work, we apply three different RL algorithms (DDPG, A2C, PPO), with two different architectures (CNN, Transformer), to our own stock trading environment comprising the 30 stocks of the Dow Jones Industrial Average (DJIA). We analyze the experimental results in multiple perspectives. First, we compare the performance of each RL algorithm with the benchmarks — the DJIA and the minimum volatility portfolio — and show that the RL algorithms perform better, even when taking transaction costs into account. Second, we compare the performance between the three RL algorithms. Finally, we demonstrate through the results that the Transformer architecture could improve the performance but could also lead to worse results.

1 Introduction

Reinforcement Learning (RL), a data-driven method to balance between exploration and exploitation, has been proven to be an advantageous approach for automated trading [4]. In this project, we aim to understand the inner workings of Reinforcement Learning by applying three different RL algorithms to stock trading, namely to the 30 stocks within the Dow Jones Industrial Average (DJIA) index. Furthermore, we try to find out the impact of the complexity of the models used in RL by comparing the performance of the same three RL agents implemented with Convolutional Neural Networks (CNN) and the Transformer architecture [13].

Following [15], we choose the following algorithms: Deep Deterministic Policy Gradient (DDPG), Advantage Actor-Critic (A2C), and Proximal Policy Optimization (PPO). The comparison between DDPG and the other two algorithms lets us observe the difference in the learning behavior of deterministic policies and stochastic policies. The difference between A2C and PPO lets us consider the impact of importance sampling and regularization. Due to the page limit, we defer the discussion about the characteristics of these algorithms to Appendix A.

For the RL environment, we again follow the work of the same authors, [14]. Although [15] is a more recent work, the definition in the paper seemed to involve an arbitrary stopping condition named the turbulence index. The threshold for this index can be set in hindsight to optimize the performance of the RL agents. Since our goal is to investigate the difference between the RL algorithms, we instead build our own environment upon the environment defined in [14] that does not involve the turbulence index. We discuss more on our environment in Section 2.

We demonstrate that all three RL algorithms have superior performance compared to our benchmarks through our experimental results in Section 3. DDPG combined with the Transformer architecture shows the best result for most of the performance criteria. However, the impact of the complexity shows a mixed picture for the other agents, which we discuss at the end of Section 3 and Section 4.

2 Dataset, Benchmark and Environment

In this study, we cover 30 equities issued by prominent companies listed on U.S. stock exchanges which make up the Dow Jones Industrial Average (DJIA) Index, as of June 26, 2018. To train and test the performance of the agent, we make use of closing prices and the data span from January 1, 2009, to November 4, 2021.

Here, however, we need to point out two issues. Firstly, the Dow Jones Industrial Average is a price-weighted index that comprises 30 large companies and since a committee made up of *S&P* representatives and Wall Street Journal editors decides and changes which companies are included in the DJIA, we needed to fix the set of 30 stocks and choose a date such that selected equities had a long enough listing history for our algorithms to be deployed. The bucket of stocks on June 26, 2018, was chosen to allow us to validate and test the algorithms on data spanning 2 years each. The full list of all selected stocks can be found in the Appendix D.

Secondly, to execute all RL strategies we have used the daily closing prices, instead of bid-ask prices. In our favor, the stocks comprising the DJIA index are considered to be one the most liquid securities, which means that the bid-ask spread should be narrow, and thus this approximation is no longer a valid problem.

Our experiments consist of three stages for which we split the whole dataset accordingly. As shown in Figure 1, the data from January 1, 2009 to December 31, 2017 are used for training to generate a well-trained trading agent. The data from January 1, 2018 to November 30, 2019 are used for validation, which is carried out for hyperparameter tuning such as learning rate, number of iterations, etc. Finally, we evaluate the profitability of the proposed schemes in the trading stage on the data ranging from December 1, 2019, to November 4, 2021.

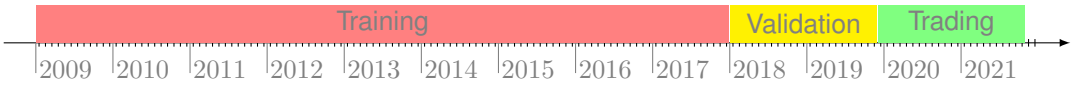

Figure 1: Data Split

When it comes to benchmarks for our selected stocks, the first and natural choice is the Dow Jones Industrial Average. On top of that, we have established an additional portfolio from the Modern Portfolio Theory [5], namely the *minimum volatility* portfolio. The portfolio was obtained using the PyPortfolioOpt library in Python [6], which constructs a portfolio that minimizes the historical volatility of the past 252 trading days using a convex optimization algorithm. Since a prospective trader will not only be interested in profits but also in other performance measures, we report the key distribution statistics of the daily returns as well as various risk-adjusted performance metrics for the strategies in Section 3.

The essence of Markov Decision Processes in Reinforcement Learning (RL) is that an agent interacts with an environment, which changes its states in response to action choices made by the decision-maker. The state of the environment affects not only the immediate reward obtained by the agent but also the probabilities of future state transitions. Hence, the **state** s can be viewed as a position of the agent at a specific time-step in the environment. So, whenever an agent performs an action, the environment gives the agent a reward and a new state, which the agent reached by performing that action. For further definitions and discussions on the RL setting, we refer the readers to Appendix A.

In our case, the training process will involve observing stock price returns, taking an action, and receiving a reward so that the agent can adjust its strategy accordingly. By interacting with the environment, the trading agent should derive a trading strategy with maximized rewards as time proceeds.

2.1 Xiong et al. [14]'s Environment

Before we discuss the definition of our reinforcement learning environment, we first introduce the environment in the work of Xiong et al. [14], upon which we have built our own environment. In Xiong et al. [14], the components of the RL environment are:[1]

- **Action:** $[-1, 1]^{30}$, where the action for each stock is a value within $[-1, 1]$, which can be interpreted as a trading signal. 1 means a strong buy signal, -1 means a strong sell signal. This real number is translated into the number of shares k to buy/sell with $k = \text{actions} \cdot h_{\text{max}}$, where h_{max} is a predefined parameter that determines the maximum amount of shares to trade, set by the authors to $h_{\text{max}} = 100$.
- **State:** Let us consider time t. At the end of day at time t, we will know the open-high-low-close price of the DJIA constituents stocks. We can use this information to calculate technical indicators such as MACD, RSI, CCI, ADX, based on which the agent will take an action.

[1]The parameters not mentioned in the paper were inferred from the authors' implementations in `https://github.com/AI4Finance-Foundation/FinRL`

Here, however, the authors limited themselves to the minimum in order to compute a reward, namely $\{$*balance, closing price for every stock, no. of shares for every stock*$\}$ at time t, which is a 61-dimensional vector $(1 + 30 \cdot 2)$.

- **Reward function:** The reward is calculated as $PV(s') - PV(s)$, where $PV(s)$ represents the portfolio value at state s, and s' stands for the next state. If at time t the agent is in state s and takes an action (buying/selling some number of shares for each stock), the agent moves to time $t + 1$. The environment changes to state s', which is the end of the day at time $t + 1$. Here, the agent knows closing prices of stocks and thus also knows the new value of its portfolio.

Remark (Scaling Factors)**.** *The authors rescale the number of shares, current balance, and closing prices when combining them to the state observation vector. Moreover, they keep the state variable for the balance above* $1*, which to the best of our knowledge is an arbitrary cut-off value provided for the stability of training. The rewards are also scaled according to the scaling factor for the balance.*

2.2 Our Environment

However, our RL algorithms were not able to show consistent behavior in the environment, thus we alter the aforementioned definitions and build a new environment to help converge the RL agents.

We change the following components:

- **Action:** We no longer consider that the agent outputs the number of shares we should buy/sell for each stock. Instead, we look at the percentage structure of our portfolio — what *weight* (percent) a given stock and the balance (cash account) should have in our portfolio. Hence, the agent outputs a vector of values between -1 and 1, which are then rescaled by a factor of 10, and after a transformation using the softmax function, we have a vector of new portfolio weights that restructure our portfolio.
- **State:** We view it as $\{$*daily returns for every stock, weight of balance, weight of every stock*$\}$ at time $t, t-1, \ldots, t-24$, which is a 61×25-dimensional matrix. The main difference is that also here we switch to relative sizes and add a history to the state. Now, the agent is able to observe the last 25 daily returns as well as the past 25 portfolio weightings and make a new decision based on that.

With the new definitions, we can remove a number of predefined parameters in the environment of [14], such as h_{max} and most scaling factors. We only scale the rewards by a factor of 100. Moreover, we compute the reward as a relative return (relative change in the value of total assets, i.e. *balance* + *portfolio*) instead of the absolute change measured in dollars.

3 Experiments

We now present the setup of our experiments and their results. Our goal is to answer two hypotheses: First, whether the RL agents are able to beat the benchmarks — the DJIA and the minimum volatility portfolio — in the test period (trading stage) and second, whether replacing the CNN architecture with a Transformer block improves the performance.

3.1 Setup

We train all three RL agents (DDPG, A2C, PPO) with both a CNN and a Transformer architecture from January 1, 2009, to December 31, 2017, and validate them on the period from January 1, 2018, to November 30, 2019. All six agent-architecture combinations are trained with three different seeds to make the eventual test results more robust. Further, we impose a transaction cost of 0.1% for each trade. The hyperparameters used for the final models can be seen in Appendix B. For each agent, we use the same configuration for both the CNN and Transformer architecture. The CNN and Transformer architecture details and more information on the training procedure can be found in Appendix C.

3.2 Results

Table 1 presents the results of testing our trained agents on the trading (test) period from December 1, 2019, to November 4, 2021, compared to the performance of the DJIA and the minimum volatility portfolio in the same period. Due to the state space requiring the past 25 days, the agents have to wait and so we do not start actual trading until January 9, 2020. Daily returns are averaged over three different seeds for each agent-architecture combination before evaluating the performance statistics and can also be seen as excess

Table 1: Performance Statistics

	A2C		DDPG		PPO			
01/2020 - 11/2021	CNN	Transf.	CNN	Transf.	CNN	Transf.	DJIA	Min Vol.
Cumulative Return	34.71%	29.00%	34.80%	43.37%	31.47%	25.63%	24.75%	3.75%
CAGR	17.80%	15.03%	17.84%	21.90%	16.23%	13.36%	12.93%	2.05%
Volatility (ann.)	27.03%	27.06%	29.51%	28.44%	27.28%	28.59%	28.39%	21.02%
Max Drawdown	−34.04%	−34.68%	−37.52%	−36.27%	−34.06%	−35.82%	−37.09%	−26.26%
Max DD days	270	277	266	204	270	285	277	279
Sharpe Ratio	0.74	0.65	0.70	0.84	0.69	0.58	0.57	0.20
Calmar Ratio	0.52	0.43	0.48	0.60	0.48	0.37	0.35	0.08
Skewness	−0.38	−0.37	0.08	−0.04	−0.29	−0.11	−0.58	0.08
Kurtosis	14.26	14.92	13.03	11.27	13.21	11.64	14.91	10.26
Alpha	0.05	0.02	0.04	0.08	0.03	0.01	0.00	−0.06
Beta	0.95	0.95	1.01	0.97	0.95	0.99	1.00	0.61

returns as we assume a risk-free rate of 0%. The Calmar ratio is calculated as $\frac{CAGR\%}{MaxDrawdown\%}$. Alpha and beta are calculated by taking the DJIA as the market portfolio. Max drawdown days are the days it took to escape the maximum drawdown. Volatility, Sharpe ratio, and alpha are annualized. A plot of the cumulative returns over the test period can be viewed in Figure 2.

3.2.1 RL agents vs. DJIA

We see that all our agents outperform the DJIA and the minimum volatility portfolio in terms of cumulative return, compound annual growth rate (CAGR), Sharpe ratio, and Calmar ratio. Moreover, all but one agent have a smaller max drawdown than the DJIA and most of the agents spend a similar amount of time escaping the max drawdown (max DD days). The annualized volatility is similar to the DJIA for all agents, while some are above and some below. Further, all but one agent have a beta closely below 1, showing that the returns move similarly to the DJIA returns.

3.2.2 A2C vs. DDPG. vs. PPO

Comparing the three different agents, we observe that DDPG performed better than A2C and A2C performed better than PPO in terms of cumulative return, CAGR, Sharpe ratio, and Calmar ratio. This also shows in the size of the alphas, as the alphas for the DDPG agent are the highest on average with 4% and 8%, compared to A2C and PPO with alphas of 5% and 2%, and 3% and 1%, respectively. All agents exhibit similar volatility and max drawdown, but A2C does a little better in that respect.

3.2.3 CNN vs. Transformer

Looking at the architectures for each agent, we have mixed results. For DDPG, the Transformer architecture performed much better and is the best performing agent overall with a cumulative return of 43.37%, a CAGR of 21.90%, a Sharpe ratio of 0.84, and a Calmar ratio of 0.60, compared to the CNN architecture with 34.80%, 17.84%, 0.70, and 0.48, respectively. Also, the Transformer architecture generated double the alpha of the CNN architecture and escaped a similar max drawdown 62 days earlier.

However, for A2C, the CNN architecture performed better than the Transformer architecture. The CNN architecture generated a cumulative return of 34.71%, a CAGR of 17.80%, a Sharpe ratio of 0.74, and a Calmar ratio of 0.52, compared to the Transformer architecture with values of 29.00%, 15.03%, 0.65, and 0.43, respectively. Here, the CNN architecture generated more than double the alpha (5%) of the Transformer architecture (2%). For PPO, it is also the CNN architecture that performed better. It generated a cumulative return of 31.47%, a CAGR of 16.23%, a Sharpe ratio of 0.69, and a Calmar ratio of 0.48, while the Transformer architecture exhibits 25.63%, 13.36%, 0.58, and 0.37, respectively. The alpha is 3% for the CNN architecture, while it is only 1% for the Transformer architecture. We see that PPO with a Transformer architecture performed only slightly better than the DJIA.

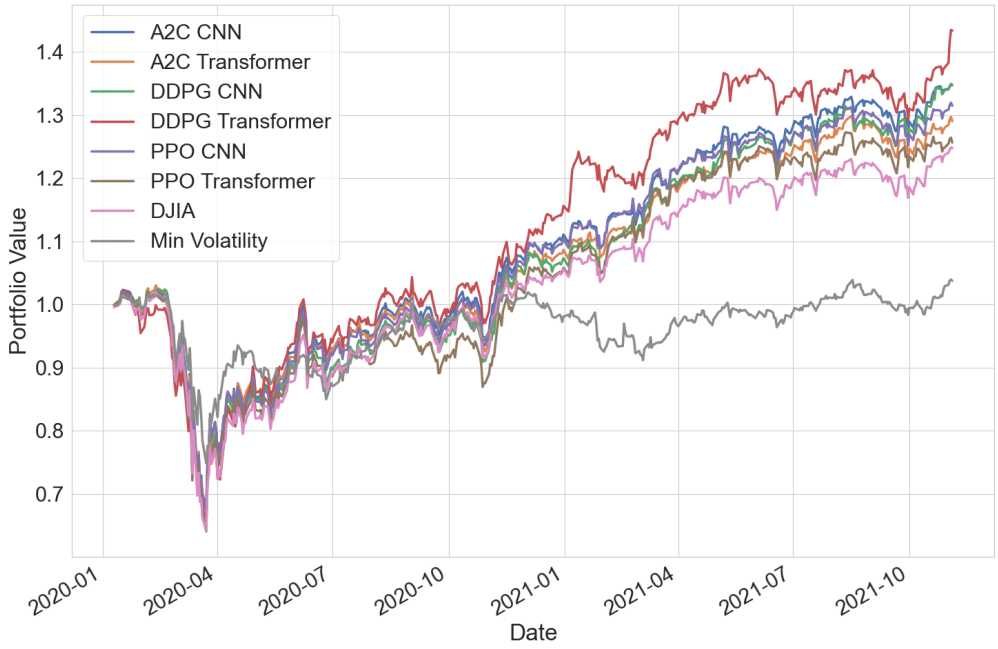

Figure 2: Cumulative Returns of different Agents and Architectures from 01/2020 to 11/2021

4 Discussion

Averaged over three seeds, all agents outperform the market even when taking risk and transaction costs into account, which can promote the use-case of these technologies in finance even more. Furthermore, the DDPG agent using a Transformer architecture performs best out of all agent-architecture combinations.

However, while still outperforming the market, the Transformer architecture makes performance worse for A2C and PPO. Furthermore, for both architectures, learning proved difficult for all agents. While training losses decreased in general, the behavior and magnitudes of them were hard to interpret. Also, the validation score (profits in the validation period) was quite sensitive to the validation period chosen. The same agent which performed well for a specific validation period, showed different validation performance when increasing the validation period by one year. A reason could be that our validation period of close to two years is too short and does not reflect representative market dynamics, since the market grew only about 11% in that time period. Moreover, the test period is special too, as it contains the stock market crash in February/ March 2020 due to the emergence of COVID-19.

Future research should explore more RL agents, train them on larger datasets, improve the environments, and tune hyperparameters even more. Finally, compared to the original Transformer introduced by [13], we only use one instead of six Transformer blocks, so with enough computing power and/or a more efficient design, there might be room for improvement.

5 Conclusion

In this paper, we investigated the performance of the DDPG, A2C, and PPO algorithms applied to stock trading on a self-designed environment. We found that they all outperform the DJIA in several risk-adjusted metrics and taking transaction costs into account. Moreover, we found that replacing the convolutional neural networks with just one Transformer block improves the DDPG agent but worsens the performance of A2C and PPO. More experiments and hyperparameter tuning should be done in this direction to make the results more robust. Furthermore, the results should be compared to other traditional portfolio investment strategies.

References

[1] Thomas Degris, Patrick M Pilarski, and Richard S Sutton. "Model-free reinforcement learning with continuous action in practice". In: *2012 American Control Conference (ACC)*. IEEE. 2012, pp. 2177–2182.

[2] Prafulla Dhariwal et al. *OpenAI Baselines*. `https://github.com/openai/baselines`. 2017.

[3] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[4] Xiao-Yang Liu et al. "FinRL: Deep reinforcement learning framework to automate trading in quantitative finance". In: *arXiv preprint arXiv:2111.09395* (2021).

[5] Harry Markowitz. "Portfolio Selection". In: *The Journal of Finance* 7.1 (1952), pp. 77–91. ISSN: 00221082, 15406261. URL: `http://www.jstor.org/stable/2975974`.

[6] Robert Andrew Martin. "PyPortfolioOpt: portfolio optimization in Python". In: *Journal of Open Source Software* 6.61 (2021), p. 3066. DOI: `10.21105/joss.03066`. URL: `https://doi.org/10.21105/joss.03066`.

[7] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937.

[8] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[9] John Schulman et al. "High-dimensional continuous control using generalized advantage estimation". In: *arXiv preprint arXiv:1506.02438* (2015).

[10] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[11] John Schulman et al. "Trust region policy optimization". In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

[12] David Silver et al. "Deterministic policy gradient algorithms". In: *International conference on machine learning*. PMLR. 2014, pp. 387–395.

[13] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: `1706.03762 [cs.CL]`.

[14] Zhuoran Xiong et al. *Practical Deep Reinforcement Learning Approach for Stock Trading*. 2018. arXiv: `1811.07522 [cs.LG]`.

[15] Hongyang Yang et al. "Deep reinforcement learning for automated stock trading: An ensemble strategy". In: *Available at SSRN* (2020).

A Reinforcement Learning Algorithms

We consider a standard reinforcement learning setup [3], which consists of an agent interacting with an environment in discrete timesteps. At each timestep t the agent receives an observation x_t, takes an action a_t and receives a scalar reward r_t. In the following descriptions, the actions are all assumed to be real-valued $a_t \in \mathbb{R}^n$. In general, the environments may be partially observed so that the state for the Markov Decision Process (MDP) may need to include the entire trajectory, i.e., $s_t = (x_1, a_1, \ldots, a_{t-1}, x_t)$. However, in this paper, we assume that the features for the observations were well-chosen so that the environment is fully-observed, $s_t = x_t$.

The agent's behavior is defined by a policy π, which is a function that maps states to a probability distribution over the action space $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$. The environment is also allowed to be stochastic, thus we model the MDP by the state space $\mathcal{S}$, action space $\mathcal{A} = \mathbb{R}^n$, initial state distribution $p(s_1)$, the transition dynamics $p(s_{t+1}|s_t, a_t)$, and reward function $r(s_t, a_t)$. The return from a state is defined as the sum of discounted rewards $R_t = \sum_{i=t}^{T} \gamma^{i-t} r(s_i, a_i)$, where T is the maximum cutoff length for the calculation and $\gamma \in [0, 1]$ is a discounting factor. Now the goal of reinforcement learning can be defined as learning a policy π which maximizes the expected return from the initial state distribution $J = \mathbb{E}^{\pi}[R_1]$.

In this project, we tried to implement and compare 3 different algorithms for reinforcement learning: Deep Deterministic Policy Gradient (DDPG), Advantage Actor-Critic (A2C), and Proximal Policy Optimization (PPO). We introduce the basic concepts and the pseudocodes for each of these 3 algorithms in the following subsections.

A.1 Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradient (DDPG) [3] extends the Deterministic Policy Gradient (DPG) [12] by introducing non-linear function approximators such as the neural networks for the actor function and critic function of DPG. The actor function μ of DPG is a parameterized version of the policy π which *deterministically* maps the states to specific actions, i.e., $\mu(\cdot|\theta^{\mu}) : \mathcal{S} \to \mathcal{A}$. The critic function Q is a parameterized version of the action-value function Q^{π} that describes the expected return after taking action a_t in state s_t when following the policy π:

$$\begin{aligned} Q^{\pi}(s_t, a_t) &= \mathbb{E}^{\pi}[R_t|s_t, a_t] \\ &= \mathbb{E}\left[r(s_t, a_t) + \gamma \mathbb{E}^{\pi}[Q^{\pi}(s_{t+1}, a_{t+1})]\right] \end{aligned} \tag{1}$$

where the second equality comes from the Bellman equation. In the case of DPG where the policy is deterministic, the second equality can be simplified as the following:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}\left[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, a_{t+1})\right]. \tag{2}$$

Note that the expectation now does not have the dependence on π, which makes it possible to learn Q^{μ} off-policy using the random replay buffers as in [8].

The random replay buffer is known to help simulate samples from an independently and identically distributed dataset, to mitigate the autocorrelation in the transition tuples (s_t, a_t, r_t, s_{t+1}). The replay buffer is a finite-size cache R that stores a fixed number of transition tuples. Transitions are sampled from the environment using some policy π' that does not necessarily have to be the policy π that we are learning and stored into R. At each timestep a new transition tuple (s_t, a_t, r_t, s_{t+1}) is inserted into the replay buffer, and the actor and critic are updated by sampling a minibatch randomly from R. When the replay buffer is full the oldest samples are discarded. The size of R should be large so that it allows the algorithm to learn across a set of uncorrelated transitions [3].

In DDPG, the critic function Q is updated by minimizing the loss between the left-hand side and the right-hand side in (2), using the samples from the replay buffer:

$$\begin{aligned} L &= \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \\ \text{where } y_i &= r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \end{aligned} \tag{3}$$

where N is the size of the minibatch sampled from R.

The Q' and μ' in (3) are copies of the critic function Q and the actor function μ, the parameters of which are updated *slowly* compared to the original parameters. Creating the Bellman update target using these copies instead of the original is shown to improve stability in training [8]. The parameters of these copied networks, also called as *target* networks in [3] are updated by an exponential moving average of the original parameters:

$$\theta^{Q'} \leftarrow \tau\theta^{Q'} + (1-\tau)\theta^{Q}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu'} + (1-\tau)\theta^{\mu}$$

where $\tau \ll 1$.

The actor function μ is updated by applying the chain rule to the expected return from initial state distribution J, using the critic function Q as an approximation:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)\big|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_i} \tag{4}$$

where again the minibatch of size N is sampled from R to approximate the gradient.

One thing to note about the actor function in DDPG is the noise process $\mathcal{N}$ that is injected into the actions. It may be difficult to explore the action space when using a deterministic policy [3], for instance, the actor may always produce the same action for a given state when stuck in a local minimum. To this end, in DDPG the actions are sampled from an exploration policy $\tilde{\mu}$ instead of the deterministic policy μ, which is constructed by adding noise sampled from a noise process $\mathcal{N}$ to μ:

$$\tilde{\mu}(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N} \tag{5}$$

in this project, we followed the approach in [3] and used the Ornstein-Uhlenbeck process for $\mathcal{N}$. For more details on the noise process and other hyperparameters for DDPG, we refer to Appendix B.

We share the pseudocode for the overall training process of DDPG in Algorithm 1.

A.2 Advantage Actor-Critic

The Advantage Actor-Critic (A2C) is a synchronous implementation of the Asynchronous Advantage Actor-Critic (A3C) [7]. For this project, we mostly have followed the implementation of OpenAI Baselines [2] which is based on the Stochastic Actor-Critic [1], but we made some deviations from the repository to be consistent with our implementation of the Proximal Policy Optimization algorithm which we shall explain in A.3. In this section, we explain the details that are relevant to our version of A2C.

The actor function π of A2C is a parameterized version of the policy π which maps the states to a probability distribution voer the action space, i.e., $\pi(\cdot|\theta^\pi) : \mathcal{S} \to \mathcal{P}(\mathcal{A})$. The critic function V is a parameterized version of the state-value function V^π that denotes the expected return in state s_t when following the policy π:

$$\begin{aligned} V^\pi(s_t) &= Q^\pi(s_t,a_t) - A^\pi(s_t,a_t) \\ &= \mathbb{E}^\pi[R_t|s_t] \\ &= \mathbb{E}[r(s_t,a_t) + \gamma V^\pi(s_{t+1})] \end{aligned} \tag{6}$$

where A^π in the first equality is the advantage function and the third equality again comes from the Bellman equation. In comparison to DDPG, although the expectation in (6) does not have the dependence on π, the state-value function itself has the dependence on π hence the learning should be mainly done on-policy.

Nonetheless, A2C also uses a small episodic replay buffer to store the episode trajectory $(s_t, a_t, r_t, \ldots, s_T, a_T, r_T)$. Using the stored episodes A2C computes an estimate $\hat{A}$ for the advantage function A^π using a collection of Temporal Difference (TD) errors, also known as the Generalized Advantage Estimation (GAE) [9]:

$$\begin{aligned} \hat{A}_t &= (1-\lambda)(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \cdots + \lambda^{T-t-1}\hat{A}_t^{(T-t)}) \\ \text{where } \hat{A}_t^{(n)} &= r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}) - V(s_t) \end{aligned} \tag{7}$$

where in practice we usually truncate the trajectory to some fixed length of $n < T - t$ to reduce the computational burden [2].

Algorithm 1 Pseudocode of DDPG

Randomly initialize critic network $Q(s, a|\theta^Q)$ and the actor network $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialize target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1 **to** E **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for $t = 1$ **to** T **do**
    Selection action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current actor and exploration noise
    Execute action a_t and observe reward and new state r_t, s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Create Bellman update targets $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the mean squared error from target:

$$L = \frac{1}{N} \sum_i \left(y_i - Q(s_i, a_i|\theta^Q)\right)^2$$

    Update actor using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)\big|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)\big|_{s=s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Combining the errors from multiple steps by using the weighting parameter λ allows the algorithm to exploit the bias-variance trade-off [9]. If one uses a higher value for λ, the bias of the estimate $\hat{A}$ may increase but the variance decreases, so that the Bellman update target and the policy gradient could become more stable. In A2C, the critic function V is updated by minimizing the squared distance from Q, which is estimated by the advantage function $\hat{A}$:

$$L = \frac{1}{N} \sum_i (y_i - V(s_i|\theta^V))^2$$
$$\text{where } y_i = V'(s_i|\theta^{V'}) + \hat{A}_i \tag{8}$$

V' in the above equation is the equivalent of Q' in DDPG, the target functions for creating the Bellman targets. In A2C, however, the target functions are not updated using weighted averages but are copied from the original functions right before the update.

The actor function π is updated by the advantage policy gradient [7]:

$$\nabla_{\theta^\pi} J \approx \frac{1}{N} \sum_i \hat{A}_i \nabla_{\theta^\pi} \log p(a_i|s_i, \theta^\pi). \tag{9}$$

In comparison to DDPG, the actor function π can be stochastic. In our implementation, π was set to follow a multivariate normal distribution with a diagonal covariance matrix, and thus no external noise process was required to help the exploration of the agent. Instead, we added an intrinsic bonus reward for the entropy of the multivariate normal distribution. See Appendix B for more details on our implementation of A2C.

We share the pseudocode for the overall training process of A2C in Algorithm 2.

Algorithm 2 Pseudocode of A2C

Randomly initialize critic network $V(s|\theta^V)$ and the actor network $\pi(s|\theta^\pi)$ with weights θ^V and θ^π
for episode = 1 **to** E **do**
  Receive initial observation state s_1
  Initialize episodic buffer R
  for $t = 1$ **to** T **do**
    Select action $a_t \sim \pi(s_t|\theta^\pi)$ from the current actor
    Execute action a_t and observe reward and new state r_t, s_{t+1}
  end for
  Store the episode $(s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$ in R
  Compute advantage estimates $\hat{A}_t$ using Generalized Advantage Estimate (GAE):

$$\hat{A}_t = \begin{cases} r_t - V(s_t|\theta^V) & \text{if } t = T \\ r_t + \gamma V(s_{t+1}|\theta^V) - V(s_t|\theta^V) + \gamma\lambda\hat{A}_{t+1} & \text{if } t < T \end{cases}$$

  Create Bellman update targets using the advantage estimates $y_i = \hat{A}_i + V(s_i|\theta^V)$
  for epoch = 1 **to** e **do**
    Update critic by minimizing the mean squared error from target:

$$L = \frac{1}{N}\sum_i (y_i - V(s_i|\theta^V))^2$$

    Update actor using the advantage estimates:

$$\nabla_{\theta^\pi} J \approx \frac{1}{N}\sum_i \hat{A}_i \nabla_{\theta^\pi} \log p(a_i|s_i, \theta^\pi)$$

  end for
end for

A.3 Proximal Policy Optimization

The Proximal Policy Optimization (PPO) [10] is a practical implementation of the Trust Region Policy Optimization (TRPO) [11]. TRPO aims to improve the training stability of advantage policy gradient methods by enforcing a constraint on the size of the policy update.

That is, if we take a look at the advantage policy gradient in A2C in (9), the samples (s_i, a_i) in the replay buffer R have been collected from the environment using the policy before update π'. If the size of the buffer R is large enough the samples from π' would correctly represent the distribution of the state-action pairs given π'. However, in practice the size is set to be small (e.g. 128 steps [2]) since the probability term $p(a_i|s_i, \theta^\pi)$ in (9) quickly diminishes to 0 as the policy π deviates from π'.

To this end, TRPO directly discounts the probability of observing the state-action pair (s_i, a_i) under the old policy π':

$$J \approx \frac{1}{N}\sum_i \rho(s_i, a_i)\hat{A}_i$$
$$\text{where } \rho(s_i, a_i) = \frac{p(a_i|s_i, \theta^\pi)}{p(a_i|s_i, \theta^{\pi'})} \tag{10}$$

Furthermore, TRPO enforces the Kullback-Leibler (KL) divergence between the two policies to be small, i.e., $\mathbb{E}^{\pi'}[D_{KL}(\pi'(\cdot|s)\|\pi(\cdot|s))] \leq \delta$ for some small $\delta > 0$, to stabilize the ratio $\rho(s, a)$. Without this constraint, TRPO could have extremely large parameter updates and big policy ratios [11]. However, computing the KL divergence between two distributions may require substantial computational resources. PPO aims to simplify this constraint by forcing the ratio $\rho(s, a)$ to stay within a small interval around 1, i.e., $[1 - \varepsilon, 1 + \varepsilon]$ for some

small $\varepsilon > 0$. This is done by simply clipping the ratio to be within the interval,

$$\text{clip}_\varepsilon(x) = \begin{cases} \varepsilon & \text{if } x > \varepsilon \\ -\varepsilon & \text{if } x < -\varepsilon \\ x & \text{if else} \end{cases} \tag{11}$$

$$J \approx \frac{1}{N} \sum_i \min \left[\rho(s_i, a_i)\hat{A}_i, \text{clip}_\varepsilon(\rho(s_i, a_i))\hat{A}_i \right] \tag{12}$$

and take the minimum between the clipped version of the advantage policy gradient and the non-clipped version as in (12).

The rest of the algorithm is mostly the same as A2C, but there are some details in the implementation of OpenAI Baselines [2] that are different from the original description of PPO [10]. Firstly, [2] creates a clipped version of the critic and takes the maximum between the mean squared error using the clipped version and the non-clipped for updating the critic function. The second point to note is that the advantage estimate $\hat{A}$ is normalized inside the advantage policy gradient (12):

$$\bar{A}_i = \frac{\hat{A} - \text{mean}(\hat{A})}{\text{std}(\hat{A})} \tag{13}$$

where $\text{mean}(\hat{A})$ is the mean of $\hat{A}$ and $\text{std}(\hat{A})$ is the standard deviation of $\hat{A}$. See Appendix B for more details on our implementation of PPO.

We share the pseudocode for the overall training process of PPO in Algorithm 3.

Algorithm 3 Pseudocode of PPO

Randomly initialize critic network $V(s|\theta^V)$ and the actor network $\pi(s|\theta^\pi)$ with weights θ^V and θ^π
for episode = 1 **to** E **do**
  Receive initial observation state s_1
  Initialize episodic buffer R
  for $t = 1$ **to** T **do**
    Select action $a_t \sim \pi(s_t|\theta^\pi)$ from the current actor
    Execute action a_t and observe reward and new state r_t, s_{t+1}
  end for
  Store the episode $(s_1, a_1, r_1, \ldots, s_T, a_T, r_T)$ in R
  Compute advantage estimates $\hat{A}_t$ using Generalized Advantage Estimate (GAE):

$$\hat{A}_t = \begin{cases} r_t - V(s_t|\theta^V) & \text{if } t = T \\ r_t + \gamma V(s_{t+1}|\theta^V) - V(s_t|\theta^V) + \gamma\lambda\hat{A}_{t+1} & \text{if } t < T \end{cases}$$

  Create Bellman update targets using the advantage estimates $y_i = \hat{A}_i + V(s_i|\theta^V)$
  Create a normalized version of the advantage estimates using mean μ and standard deviation σ of the estimates, $\bar{A}_i = (\hat{A}_i - \mu)/\sigma$
  Store the parameters before the update $\theta^{V'} \leftarrow \theta^V, \theta^{\pi'} \leftarrow \theta^\pi$
  for epoch = 1 **to** e **do**
    Create a clipped version of the critic:

$$V_{\text{clip}}(s_i) = V(s_i|\theta^{V'}) + \text{clip}_\varepsilon(V(s_i|\theta^V) - V(s_i|\theta^{V'}))$$
$$\text{where clip}_\varepsilon(x) = \begin{cases} \varepsilon & \text{if } x > \varepsilon \\ -\varepsilon & \text{if } x < -\varepsilon \\ x & \text{if else} \end{cases}$$

    Update critic by minimizing the mean of the element-wise maximum among the squared error and the clipped version of the error:

$$L = \frac{1}{N}\sum_i \max\left[(y_i - V(s_i|\theta^V))^2, (y_i - V_{\text{clip}}(s_i))^2\right]$$

    Calculate the probability ratios between the current policy and the old policy:

$$\rho(s_i, a_i) = \frac{p(a_i|s_i, \theta^\pi)}{p(a_i|s_i, \theta^{\pi'})}$$

    Update actor by maximizing the mean of the element-wise minimum of the original advantage policy gradient and the clipped version:

$$\nabla_{\theta^\pi} J \approx \frac{1}{N}\sum_i \nabla_{\theta^\pi} \min\left[\rho(s_i, a_i)\bar{A}_i, \text{clip}_\varepsilon(\rho(s_i, a_i))\bar{A}_i\right]$$

  end for
end for

B Hyperparameters

Table 2: Hyperparameter configuration

Parameter	A2C	DDPG	PPO
Optimizer	RMSProp	RAdam	RAdam
Batch Size	32	128	32
Learning Rate Critic	-	1e-4	-
Learning Rate Actor	1e-4	1e-5	1e-4
Gradient Clipping	0.5	0.5	0.5
σ	0.1	0.1	0.1
γ	0.99	0.99	0.99
λ	0.95	-	0.95
1 - τ	-	0.99	-
Correlation Coeff.	0.5	-	0.5
Entropy Coeff.	1e-4	-	1e-1
Warmup	-	1000	-
Buffer Size	-	50000	-

C Experiment Details

C.1 Training

For DDPG, we stopped training after 4 and 3 Million steps for the CNN and Transformer architecture, respectively. Around that time, the actor and critic loss seemed to have converged and the validation score, which we defined to be the profits in the validation period, stayed constant for a while. We observed that the actor and critic loss both overshot dramatically at the beginning and much stronger with the Transformer architecture, before reverting exactly as fast to a more expected trajectory. For A2C, we stopped training after 30 million and 10 Million steps for the CNN and Transformer architecture, respectively. For PPO, we stopped after 10 million steps for both architectures. For these two agents, it proved difficult to interpret a good stopping signal. The training losses decreased overall but in different movements and the entropy did not always move in expected ways. For instance, the entropy was increasing for our final PPO hyperparameter configuration, but the validation score seemed to perform better than for other configurations where the entropy moved downwards. Therefore, for these two agents, we mainly concentrated on the validation score.

C.2 CNN architecture

For DDPG, the actor network starts with a one-dimensional convolutional layer, which takes the size of the first dimension of the state space matrix (1 cash balance + 30 stock portfolio weight + 30 stock returns = 61) as the input and maps this vector to 128 feature vectors. The kernel is of size 5 and moves along the columns, which means along the past 25 cash balances, portfolio weights and returns, respectively. The convolutional layer is followed by an element-wise ReLU activation function. This is then repeated twice before flattening all vectors and using a final fully-connected layer to map to the action space of 31 (weights for each stock and the cash account). The critic network, which has to process a state and an action, uses the same structure as the actor network for the state (matrix) and a linear feedforward neural network architecture for the action (vector). The feedforward neural network consists of two linear layers, each followed by a ReLU activation. The outputs of both of these networks-within-the-network are then concatenated and mapped to a single number via a final fully-connected layer, to represent the estimate for the Q value. For both the actor and the critic network, the final linear layers are initialized uniformly between (-3-e3, 3e3).

For A2C and PPO, we again have three convolutional layers with the same structure as in the DDPG networks followed by ReLU activations. They take in a state and transform it into a representation which feedforward layers map to the final output values needed in the algorithm.

C.3 Transformer architecture

Following [13], we replace all convolutional layers in the CNN architecture described in subsection C.2 by one Transformer encoder block. The original Transformer of the paper consists of six such Transformer

blocks, but this proved to be too expensive in our training. A Transformer block consists of a self-attention layer with a residual connection followed by a layer normalization. A feedforward network is then applied independently to each embedding dimension and another residual connection is added before a final layer normalization. A self-attention layer applies "multi-headed" attention by using 8 self-attention heads. The self-attention head first linearly maps the input sequence (length 61) with embedding dimension 25 to query, key, and value representations of the same size. Then, for each element in the sequence, its query matrix is multiplied with the key matrix of all other sequence elements to produce a score. This is followed by a softmax to produce weightings that are multiplied with the value representation of each element to generate a final representation. With 8 heads, we get 8 final representations that we concatenate and linearly map to the dimension that the following feedforward network requires, which in our case is 25.

D Stock Portfolio Dow Jones 30

Table 3: Equities constituting the DJIA 30 as of June 26, 2018

Company	Ticker
Apple Inc.	AAPL
American Express Co.	AXP
Boeing Co.	BA
Caterpillar Inc.	CAT
Cisco Systems Inc.	CSCO
Chevron Corp.	CVX
DuPont de Nemours, Inc.	DD
Walt Disney Co.	DIS
Goldman Sachs Group Inc.	GS
Home Depot Inc.	HD
International Business Machines Corp.	IBM
Intel Corp.	INTC
Johnson & Johnson	JNJ
JPMorgan Chase & Co.	JPM
Coca-Cola Co.	KO
McDonald's Corp.	MCD
3M Co.	MMM
Merck & Co Inc.	MRK
Microsoft Corp.	MSFT
Nike Inc.	NKE
Pfizer Inc.	PFE
Procter & Gamble Co.	PG
Raytheon Technologies Corp.	RTX
Travelers Companies Inc.	TRV
UnitedHealth Group Inc.	UNH
Visa Inc.	V
Verizon Communications Inc.	VZ
Walgreens Boots Alliance Inc.	WBA
Walmart Inc.	WMT
Exxon Mobil Corp.	XOM