StockAgent: Application of RL from LunarLander to stock price prediction

Caitlin Stanton[1] and Beite Zhu[2]

***Abstract*— This work implements a neural network to run the deep Q learning algorithm on the Lunar Lander arcade game (as in figure I), and then adapts this model to instead run on stock data. Our agent learns—on stock data from tech companies such as Google, Apple, and Facebook—when it should buy or sell a stock, given features related to the recent stock price history. Furthermore, our model allows the agent to opt to buy and sell smaller amounts of the stock instead of larger amounts (what we refer to as "soft" buying/selling and "hard" buying/selling, respectively), which increases the nuance and complexity of our model's decision-making.**

I. INTRODUCTION

Reinforcement learning has been at the core of many of the most exciting recent developments in AI. For example, while computers have been relatively successful at playing chess for many years–notably, the computer Deep Blue was able to defeat the reigning world chess champion Garry Kasparov in 1996–the game of Go was considered much harder; it wasn't until reinforcement learning techniques were used in 2015 that the program AlphaGo was finally able to beat a professional human Go player.

Here we use deep Q learning to train an agent to learn the arcade game Lunar Lander, a game where the goal is to steer a landing module and successfully land it on the surface of the moon. After understanding how to apply this model to Lunar Lander, we then use this same technique to a less conventional application of reinforcement learning techniques: investing in the stock market. We rephrase stock market investment as a "game" where states involve data such as recent change and volatility of a stock, and discretize the possible investment amounts in order to create a finite set of actions at each state. In this way, we apply our deep Q learning algorithm from our Lunar Lander model to the problem of stock market prediction.

II. RELATED WORK

Reinforcement learning techniques have already been applied to the Lunar Lander game. Garza implements a Policy Gradient Descent model in [1]. For this method, as developed in [2], the goal is to learn an ideal policy by training a neural network whose input is a state, and whose output is a probability distribution on the possible actions. For our model, we abandoned Policy Gradient Descent in favor of trying deep Q learning.

Predicting stock prices based on past stock data is appealing for obvious reasons. There is even a competition

Fig. I. Lunar Lander environment.

released on Kaggle (starting September 25, 2018 and ending January 8, 2019) by the company Two Sigma related to stock prediction [3]. The goal of this competition is to predict stock prices based on both previous stock data (including market information such as opening price, closing price, trading volume, etc), and news data (including news articles and alerts published about assets, such as article details, sentiment, and other commentary). Unlike our project, the goal for this competition is not to tell an agent when to buy or sell (and how much to buy or sell), but rather to predict whether stock prices will go up or down.

This seems to be a common feature for current applications of machine learning to the stock market: most models wish to predict stock prices, and not directly tell us an investment strategy. For instance, in [4], Jae Won Lee uses reinforcement learning (specifically, TD algorithms) to model stock price behaviour. When referencing the relationship between his research and the problem of how to invest, he writes: "Though the ultimate purpose of reinforcement learning is to get an optimal policy, that is to resolve control problem, the use of TD algorithm in this paper is confined to the prediction problem because the policy in stock market is assumed to be determined by each investor and to be beyond the scope of learning here." Some sources have, however, opted to train an agent to actually develop an investment strategy. In [5], they develop a reinforcement learning algorithm to train an agent to play the stock market. However, unlike our model, their model only allows for three actions: sell, hold, or buy. This eliminates some of the nuance from our model, where our agent has some control over the quantity that gets bought or sold, and not just the fact of buying or selling.

*This work was not supported by any organization

[1]Department of Mathematics, Stanford University, SUNetID: stanton1
[2]Department of Mathematics, Stanford University, SUNetID: jupiterz

	Volatility	Delta1	Delta2	Delta3	Current_price
0	0.01638255165926625	-0.015211898087533624	-0.027310297586317058	-0.03740027951644749	70.8046
1	0.021574289473426142	-0.027310297586317058	-0.03740027951644749	0.0053906275218130765	71.6374
2	0.014799547108176016	-0.03740027951644749	0.0053906275218130765	-0.010495965142995495	72.3087
3	0.029908344460410175	0.0053906275218130765	-0.010495965142995495	-0.09966264861973934	65.6535

Fig. II. Processed stock data for Apple.

III. DATA

A. Lunar Lander Environment

The Lunar Lander environment was provided by OpenAI gym `https://gym.openai.com/` [6] . There was no additional processing needed. The provided states for the simulation consist of length-8 vectors, containing the following information about the lunar landing module: position in space, orientation in space, velocity, angular velocity, whether the right foot is in contact with the ground, and whether the left foot is in contact with the ground.

There are four actions allowed at any given point in the game: firing the main engine, firing the right engine, firing the left engine, or doing nothing. There are rewards for landing with feet down, and penalties for wasting time, landing far away from the pad, and wasting fuel.

B. Stock Dataset

We used IEXs API to download 5 years worth of stock data from Apple and Google. This included the daily price (opening, closing, high and low) of the stock. To simplify our model, we just considered the closing price of the stock on any given day. We also wanted to predict stock prices on a slightly longer time scale, so we restricted to looking at stock prices each week (choosing the closing price on the previous Friday as the stock's price at the beginning of the following week).

We did, however, save some data about how the stock changed in the lead-up to a given week. We added a feature corresponding to the volatility of the stock price in the previous three weeks. More precisely, for week n, the volatility feature for that week is equal to the standard deviation of the daily stock prices from weeks $n-3$ through $n-1$, divided by the mean of the stock price during this range. Normalizing by the mean price ensures that this feature is independent of scaling the stock price (as this shouldn't impact how much we choose to invest).

We also added three features, which we call delta1, delta2, and delta3, that correspond to the weekly net change in stock price from the three preceding weeks. So in total, our data (each row corresponding to one week in our 5 year span) contains 7 features: the volatility of the stock, delta1, delta2, delta3, the price of that stock at the beginning of the week (i.e. the closing cost from the previous Friday), our current cash on hand, and the value of the stock we own. Our initial state consists of our first line of processed stock data, and the fact that we have \$0 in cash and \$0 invested into the stock.

To make our situation similar to the finite-action state from Lunar Lander, we imposed only a finite number of actions that our agent can take when playing the stock market. Specifically, at any given moment in time, we are allowed to do a "hard" buy or sell, a "soft" buy or sell, or do nothing. Our default values of hard and soft were \$100 and \$10 respectively, though we did test different "soft" values later while keeping "hard" fixed (see section V-C for details). By letting "hard" and "soft" correspond to the amount we invest in the stock market on a given week and not to the number of shares, we ensure that our model is independent of the average price of the stock; our model just cares about how the stock price fluctuates from week to week, and not the absolute price of the stock. Notice also that for simplicity of our model, we allow negative cash and negative stock value in our portfolio. Some of this even makes sense (for instance, "negative cash" could correspond to taking out a loan so as to invest more in the stock market).

Our reward function is just the change in portfolio value each week.

IV. METHODS

A. Deep Q Learning

For both Lunar Lander and the stock market, we used deep Q learning to train our agent. The goal here is to learn the Q function, which gives our total maximum expected reward if we start at state s and take action a. Thus Q should satisfy the optimal Bellman equation:

$$Q(s,a) = R(s,a) + \gamma \cdot \max_{a' \in A} Q(s',a'),$$

where s' is the state after taking action a from s, and γ is the discount factor (intuitively, it parametrizes how much we value future versus current reward).

For deep Q learning, we use a neural network to learn the Q function. Specifically, our network takes in a state s, and outputs a vector of length equal to the number of actions, where each entry corresponds to $Q(s,a)$ for that particular action. Since Q corresponds to net reward, in order to implement a trained network, at state s we would take action a which corresponds to the largest entry in our output from the neural network.

B. Loss Function

In order to train our model, we need a loss function. Given a state s and an action a, our loss is just the difference between what our model predicts, and what the optimal

Bellman equation should give. In other words, if we let $\hat{Q}$ be our target function generated via

$$\hat{Q}(s,a) = R(s,a) + \gamma \max_{a' \in A} Q(s,a),$$

our loss is then given by:

$$Loss = \|Q - \hat{Q}\|_2.$$

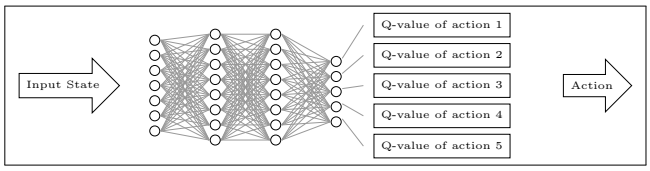

Fig. III. Graph of our model, *StockAgentDeepQNetwork*.

V. Discussion

A. Training

For the Lunar Lander model, we used a fully-connected neural network of shape $(\text{input}, 8, 8, 4)$. For stock prediction, we used a fully-connected neural network of shape $(\text{input}, 10, 10, 5)$. 500 epochs could be finished on an Apple laptop within a few minutes.

B. Result

Our base line model as in figure III, *StockAgentDeepQNetwork*, or SADQN, involves setting "hard" and "soft" to 100 and 10 dollars per transaction, respectively. We set the exploration rate set equal to 0.05. This converges after around 200 epochs. One can see the plot of cost and reward in figure IV.

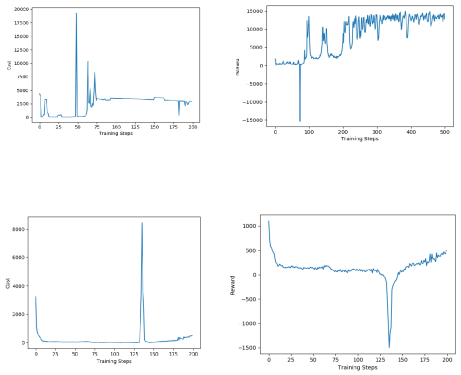

Fig. IV. Plots of reward and cost against training steps.The first row are the cost and reward of AAPL, and second row are the cost and reward of GOOG. The plummets and spikes are due to the random explorations.

Our model can return 500-2000 dollars of profit in the course of 5 years on a strong stock. The majority of its actions consist of soft buys, and the agent will go into negative cash budget, but with a sizable stock value thus resulting in a positive portfolio.

C. Hyper-parameter Choices/ Experiments

All other experiments are conducted around the above base model. Here is what we have tried:

i. **Soft buy/sell tweaking**: As we listed out the actions from our base model after training, we realized that our model has a very strong preference for "soft" actions. In other words, the agent is risk-averse. To experiment with this, we have tried various scalings of soft action values. We fix the hard buy/sell value at 100 and alter the soft buy/sell values, as seen in in the following table in figure V. One can see that the final portfolio value is very much positively proportional to the soft action scale.

Network	GOOG	AAPL
Soft=1	66.61	235.42
Soft=5	448.15	342.22
Soft=10	50.77	943.07
Soft=20	120.14	19.247
Soft=50	5716.30	8566.56

Fig. V. Final portfolio value under different soft buy/sell value.

ii. **Less competitive stock testing**: When training on Apple and Google stock, we noticed that the agent chose "soft" buy the vast majority of the time. Part of the issue here might be that Google and Apple are relatively stable stocks, and have an upward long-term trend in price. In order to really exhibit the predictive capabilities of our model, it made sense to apply it to stocks that are more volatile, and which maybe don't have such an obvious long-term trend.
Staying in the tech realm, three additional stocks we looked at were Facebook, Twitter, and NVIDIA. Using just our baseline model (with 500 epochs, since more volatile stocks take longer to converge), this is how we performed on each of these stocks, as compared to always performing "soft" buy as our action:

MODEL	FB	TWTR	NVDA
Baseline	1435.60	205.02	1761.86
"Soft" buy	1142.12	235.96	6046.56

Fig. VI. Final portfolio value of Facebook, Twitter, and NVIDIA.

We can see that our model performs better than naively performing "soft" buy for Facebook, performs similarly for Twitter, and performs worse for NVIDIA. We don't currently know what is causing these discrepancies, but we think it's likely that our model is too risk-averse. For example for NVIDIA, a highly volatile stock, our model is usually doing nothing, or performing a "soft" buy.

iii. **Exploration rate tweaking**: Having random exploration is important in learning a game like Lunar Lander so that the agent can experience different states of the game and handle various situations. In the Lunar Lander game, our agent in fact grows more 'curious' and

explores more as time goes on. However, we felt this is unnecessary in a stock prediction situation, and thus we have experimented with various fixed exploration rates. The result can be seen in figure VII. One can see that unlike the Box2D games, having a reasonable exploration rate like 5 percent actually negatively influences the training. This is probably due to the fact that the process of stock prediction is much more formulated and deterministic. Unlike other games where various actions could lead to different states that could still achieve optimal outcome, if the price is going up, a hard buy is just the absolute best action. Thus, for the best training outcome, one should stick to a minimal exploration rate.

Exploration rate	GOOG	AAPL
$\varepsilon = 0$	249.63	1905.90
$\varepsilon = 0.01$	458.92	1713.68
$\varepsilon = 0.05$	592.98	476.66
$\varepsilon = 0.1$	116.36	264.31
$\varepsilon = 0.2$	-30.76	129.55

Fig. VII. Final portfolio value under different exploration rates.

iv. **SADQNbold, the risk rewarding experiment, and γ tuning:** As we observed, our agent is highly risk-averse. Even though it can forecast stock behaviour in the long run, it will still choose the soft action instead of the hard one. To encourage our agent to take risks, we have implemented a new model called SADQNbold. SADQNbold has the extra two parameters:

```
volatility_weight
exploration_hard_chance.
```

The volatility weight is a variable ν that currently associates reward with volatility using the formula

$$\text{Reward} = \text{Profit} * (1 + \nu) * \text{volatility},$$

thus making a riskier profit more rewarding. The exploration hard chance, or *EHC*, puts a different weight on hard actions when sampling for an action in the exploration part of training. With a bigger probability of taking hard actions, the agent will see more of the benefits of hard actions. (Note that when $\nu = 0$ and $EHC = 0.2$ we have the baseline model.) Another related parameter is γ, the discount factor in the Bellman equation for generating our $\hat{Q}$. Making this number closer to 0 will make the agent more short-sighted thus taking bolder actions. In experiments, we found that lowering γ or increasing ν and *EHC* will indeed encourage more hard actions. On the other hand, it makes the agent highly unstable with respect to different stocks. More volatile stocks could lead to an unprofitable agent. In the experiment as in figure VIII on the stock of AAPL, the actions consist primarily of hard buy and sell. The final portfolio value is around 12,000 dollars.

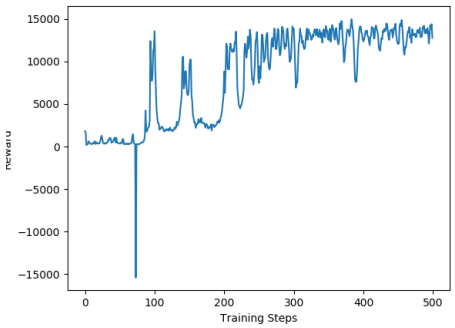

Fig. VIII. Plots of reward against training steps using SADQNbold with $\nu = 1$, $\gamma = 0.8$ and $EHC = 0.25$.

VI. Conclusion and Future Directions

Conclusion By drawing connections between the game of Lunar Lander and stock investment, we have established a baseline structure of a stock predicting agent using the model of deep Q learning. The model is demonstrated to be rather risk averse but can master long term investment strategy with reasonably volatile stocks.

Future directions, shot term vs. long term: In our data split there is a mismatch, as we are training on approximately 5 years of data and trying to test the agent on 10 weeks of data. However, what the agent picks up from the base model is a long term strategy and is not optimal on a 10 week basis. Thus we tried to implement *StockAgentDQNShort*. This agent, instead of training the whole 5 year period, trains on several episodes of 10 consecutive weeks (or whatever the test data length) randomly selected from the training data. But as one can observe from figure IX, the randomness we

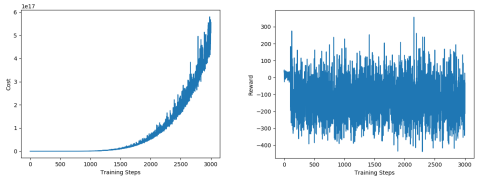

Fig. IX. Plots of reward and cost against training steps. The stock is based on AAPL

introduced is giving the model a hard time converging, and thus the reward graph is highly fluctuating. One explanation why this approach failed is that, though we match the data with train and test, this is not the traditional way humans predict stock. The historical data is always there for reference whenever one makes a prediction in real life, so somehow restricting our model to training on 10 weeks of data is not a realistic solution to this data mismatch problem.

One possible solution is to strengthen the model so that it includes more than 3 weeks of data in the past, thus resulting a network that has more input features and thus more complexity in general. Another solution, which is closer to human prediction, is to include real world events to

help short term prediction [3][7]. Indeed, news events like the release of new products or the new employment of a CEO can hugely influence stock price in a short span of time. The significance of such events usually outweighs past statistical information of the stock market, and thus should be considered in the model.

CODE

The code for this project is available at `https://github.com/zhubeite/CS-229-RL-project`.

ACKNOWLEDGMENT

We would like to thank our mentor, Mario Srouji, for his guidance throughout this project.

REFERENCES

[1] Gabriel Garza. Deep reinforcement learning - policy gradients - lunar lander!, 2018. [Online; posted 17-January-2018].

[2] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[3] Two Sigma. Two sigma: Using news to predict stock movements, 2018. [Online; posted 25-September-2018].

[4] Jae Won Lee. Stock price prediction using reinforcement learning. In *ISIE 2001. 2001 IEEE International Symposium on Industrial Electronics Proceedings (Cat. No.01TH8570)*, volume 1, pages 690–695 vol.1, June 2001.

[5] Zhuoran Xiong, Xiao-Yang Liu, Shan Zhong, Hongyang Yang, and Anwar Walid. Practical deep reinforcement learning approach for stock trading. *CoRR*, abs/1811.07522, 2018.

[6] Openai gym.

[7] Stefan Feuerriegel and Helmut Prendinger. News-based trading strategies. 2018.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529 EP –, 02 2015.

[9] Fuli Feng, Huimin Chen, Xiangnan He, Ji Ding, Maosong Sun, and Tat-Seng Chua. Improving stock movement prediction with adversarial training. *arXiv preprint arXiv:1810.09936*, 2018.