Deep Q-Network for Stochastic Process Environments

Kuangheng "Gerald" He
Project Repository: `https://github.com/skylinehk/flappy-bird-stochastic-process-stock-predict`

Abstract. Reinforcement learning is a powerful approach for training an optimal policy to solve complex problems in a given system. This project aims to demonstrate the application of reinforcement learning in stochastic process environments with missing information, using Flappy Bird and a newly developed stock trading environment as case studies. We evaluate various structures of Deep Q-learning networks and identify the most suitable variant for the stochastic process environment. Additionally, we discuss the current challenges and propose potential improvements for further work in environment-building and reinforcement learning techniques.

Keywords:

Reinforcement Learning, Deep Q-Learning, Stochastic Process, Flappy Bird, Portfolio Management Strategy.

1 INTRODUCTION

The primary objective of our research is to develop optimal policies for predicting the behavior of a stochastic process environment. To achieve this, we will be using the game Flappy Bird as our starting point, as it has been previously demonstrated as a suitable environment for reinforcement learning. In the first part of our project, our goal is to train a Deep Q-Network(DQN) agent to successfully play the game of Flappy Bird. See figure 1.

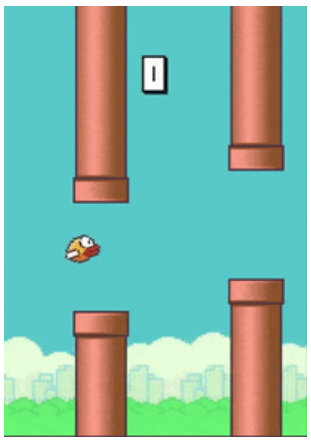

Figure 1. The game image of Flappy Bird env

The game involves navigating a bird through pipes while avoiding obstacles to earn points. The goal is to pass the bird through pipes. The observation space consists of only one variable, which is the position of the bird denoted by $s \in \mathbb{R}$. The action space includes two possible actions: moving the bird up and down denoted by $a \in 0, 1$, where 0 corresponds to moving the bird up and 1 corresponds to moving the bird down. The reward function $r(s, a)$ is defined as follows: when the bird passes through the pipe, the reward is 1, otherwise, the reward is 0. The episode terminates when the bird crashes into the pipe. The agent will be provided with position information and the current score, and it must learn to recognize the bird and pipes and locate them on its own. The game's state space is challenging, requiring the agent to generalize its learning to successfully play the game better than even human players.

In the second part of our project, we modified the game to a stochastic process environment and developed a stock trading simulation with essential metrics. The DQN agent will be trained using a network design developed in the first part of our project and will trade using the trained policy to earn profit in a backtest class.

Overall, our research aims to develop a robust and generalized DQN agent that can predict and adapt to the behavior of stochastic process environments such as stock trading simulations. This will contribute to the ongoing effort to develop more efficient and effective AI algorithms that can adapt to complex and dynamic environments.

2 RELATED WORK

Previous work in this area has primarily been conducted by Google DeepMind. Mnih et al. successfully trained agents to play Atari 2600 games using deep reinforcement learning, achieving performance exceeding that of human experts on multiple games. In the domain of game environments, many successful attempts have been made to train agents using gym. For instance, Kevin Chen (2017) employed Deep Q-Networks (DQN) to train an agent to play Flappy Bird, achieving the highest score of 215. 1 In

the realm of stock trading, Lin Willam Cong et al. demonstrated the flexibility of using deep reinforcement learning to manage portfolios without tagging information.

Table 1. Highest score in flappy bird by Chen, 2017

training difficulty	flap every n	human	DQN
easy	Inf	Inf	Inf
medium	11	Inf	Inf
hard	1	65	215

In the realm of stock trading, Lin Willam Cong et al. demonstrated the flexibility of using deep reinforcement learning to manage portfolios without tagging information.

3 METHOD

3.1 Deep Q-network

The Deep Q-Network (DQN) algorithm used for the Flappy Bird environment is based on an implementation of a neural network with three linear layers and two dropout layers which takes in the number of input features, the number of hidden units, and the number of output actions as inputs. The forward function of the network applies ReLU activation to the output of each linear layer and applies dropout with a probability of 0.1 to the output of the first two linear layers. The output of the final linear layer represents the Q-values for each possible action. The DQN algorithm learns the optimal Q-values by minimizing the mean squared error loss between the predicted Q-values and the target Q-values, calculated using the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \quad (1)$$

where $Q(s,a)$ is the Q-value for a state-action pair, s is the current state, a is the current action, r is the reward for taking action a in state s, γ is the discount factor, and α is the learning rate. The agent selects actions according to an ϵ-greedy policy, where with probability ϵ it selects a random action and with probability $1 - \epsilon$ it selects the action with the highest Q-value.

3.2 Kaiming Initialization

We utilized the Kaiming initialization method for the weights of our DQN network. The Kaiming initialization is a method of weight initialization to improve the convergence speed and performance of deep neural networks. This method initializes the weights using a Gaussian distribution with mean 0 and variance $\frac{2}{n}$, where n is the number of input features. The Kaiming initialization is applied to the weights of the linear layers using the PyTorch with a nonlinearity of ReLU. These weight and bias initialization methods help to improve the performance of our DQN agent in learning the optimal Q-values and navigating the Flappy Bird environment.

3.3 Replay Memory

Our implementation of the DQN algorithm in the Flappy Bird game involved the replay memory method. The ReplayMemory class stored recent transitions consisting of the current state, action taken, next state observed, and corresponding reward, in a deque data structure with a maximum capacity. The push method added a new transition to the memory buffer, and the sample method randomly selected a batch of transitions for training. This method helped decorrelate transitions, improve sample efficiency, and provide a more stable training process.

3.4 Model Optimization

The optimize model function is realized by computing the expected state-action values using the Bellman equation defining the Huber loss function:

$$\mathcal{L}(y,\hat{y}) = \begin{cases} \frac{1}{2}(y-\hat{y})^2, & \text{if } |y-\hat{y}| \leq \delta \\ \delta|y-\hat{y}| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases} \quad (2)$$

The function calculates the loss between expected and predicted state-action values, backpropagating the loss, clipping gradients, and performing an optimization step. It utilizes the policy and target networks to update the Q-values of the agent's policy.

3.5 Flappy Bird Agent Training Pipeline

With the previously stated methods above, we build the full training pipeline to train the agent for Flappy bird environment in algorithm 1. The technique used in our approach is Q-learning with experience replay, where we store every experience at every frame in a replay memory.

Algorithm 1 Deep Q-learning algorithm for Flappy Bird

Initialize DQN weights
Initialize replay memory with capacity MEMORY
repeat
    Reset the environment and observe the state
    repeat
        Compute Q-values for the current state using the policy network
        if random number > ϵ **then**
            Take the action with the highest Q-value
        else
            Take a random action
        end if
        Observe the transition and compute the reward
        Define the next state of the environment
        Push the transition into the replay memory
        if the replay memory is full **then**
            Sample a batch from the replay memory
            Compute the expected state-action values for the batch
            Compute the Huber loss between the state-action values and the expected values
            Zero the gradients of the optimizer
            Backpropagate the loss by the network
            Clip the gradients to prevent exploding gradients
            Take an optimizer step
        end if
        Compute the soft update of the target network weights
    until flappy bird crashes
until reach max episode

3.6 Stock Trading Environment

In this part, we aim to develop a reinforcement learning algorithm to optimize stock trading decisions. To achieve this goal, we define a custom TradingEnv class to simulate stock trading in the real world. The initial state of the environment is set to a cash balance of $ 1 million, and the commission fee is set to 0.1%. The TradingEnv class has two methods: `reset()` and `step()`. The `reset()` method initializes the environment variables to their initial values and creates an observation consisting of four elements: the *currentprice* of the stock, the *currentvalue* of the account, the *currentposition* of the account (i.e., how many stocks the account holds), and the *currentcashbalance* of the account. The `step()` method takes an action as input and updates the environment variables accordingly. The action can be to buy or sell any action value of the stock. The current market value and balance of the account are updated after each action. The $dailyprofit$ and $totalprofit$ from the initial value of the account are calculated. A *reward* is determined based on the total profit made by trading. The method also returns the observation, reward, and done flag. The episode is considered done when the time comes to an end, which is the last timestamp of the training data. The stock data consists of four symbolic stocks with the highest trading volume from 2018 to 2020. We will use this environment to evaluate the performance of the reinforcement learning algorithm on simulated stock trading scenarios.

3.7 Stock Trading Agent

The agent class in this project implements the deep Q-learning algorithm with experience replay to train a stock trading agent. The constructor initializes hyperparameters and memory parameters such as γ, ϵ, learning rate, input dimensions, batch size, memory size, ϵ end, and ϵ decay. The action space is continuous and ranges from -1 to 1. The Q-network is initialized, and the memory buffers are allocated using the input dimensions and memory size. The `store_transition()` method stores a transition tuple of state, action, reward, next state, and done in memory. The `choose_action()` method chooses the action with the highest value for greedy exploration or a random action for exploration based on the ϵ value. The `learn()` method computes gradients and updates the weights using a randomly generated batch index, and the Bellman equation is used to compute the Q-targets. The ϵ value is decayed over time to ensure the agent performs less exploration as it gains more experience.

3.8 Stock Trading Agent Training Pipeline

With the previously stated methods above, we build the full training pipeline to train the agent for the stock trading environment in algorithm 2.

Algorithm 2 Training loop for stock trading environment

Read training data and create a trading environment
Initialize the agent with hyperparameters
Set the number of episodes to simulate
Initialize empty lists for episode rewards, episode durations, and epsilon history
for each episode **do**
    Initialize cumulative reward and done variables, and reset the environment
    while not done **do**
        Choose an action using the agent's policy
        Take an action into the environment
        Update the cumulative reward
        Store the transition in the agent's memory
        Learn from the stored transitions
        Update the observation for the next timestep
    end while
    Append the final cumulative reward and epsilon value to the lists
    Calculate the episode number, cumulative reward, average reward, and current epsilon
    Append the episode duration to the list of episode durations
    Calculate the rewards
end for reach max episode

4 RESULTS AND DISCUSSION

4.1 Flappy Bird Environment

We implemented an enhanced DQN for the Flappy Bird game, which incorporated dropout layers and utilized the Kaiming initialization method to improve performance. Compared to the Lunar Landing environment, where the agent receives continuous rewards, Flappy Bird only provides rewards when the agent passes through the pipes correctly. To address this, we trained a more robust network with a larger memory replay size of $MEMORY = 50000$, and a batch size of $BATCH\_SIZE = 256$, which resulted in stable activation. The enhanced DQN also employed the following hyperparameters: discount factor $\gamma = 0.99$, exploration rate starting at $\epsilon = EPS\_START = 0.99$ and decaying exponentially to $\epsilon = EPS\_END = 0.01$ over $EPS\_DECAY = 1000$ steps, target network update rate $\tau = 0.005$, learning rate $LR = 1e-4$, and hidden layer size $HIDDEN = 256$, which resulted in more stable and faster activation. The enhanced DQN triggered a faster training process and led to a significant improvement in test results. Specifically, the Average Reward over 20 Episodes increased from 158.70 to 801.00, while the average trigger episode decreased from 2212.73 to 1623.58.2.

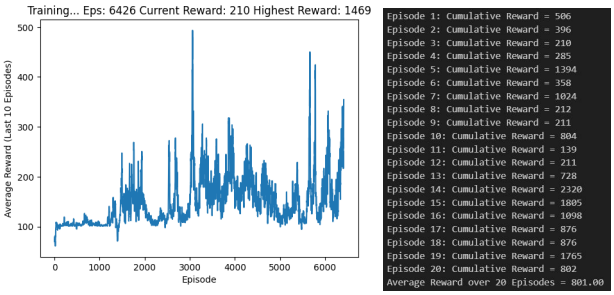

Figure 2. The image of Flappy Bird training result

Dropout layers randomly drop out a certain percentage of neurons during each training iteration, which prevents overfitting and encourages the network to learn more robust features. The Kaiming initialization method takes into account the non-linearity of activation functions, which helps to mitigate the vanishing/exploding gradient problem during training. Overall, the enhanced DQN achieved significant improvements in performance for the Flappy Bird game.

4.2 Stock Trading Environment

In this part, we developed a reinforcement learning algorithm to optimize stock trading decisions in both single-stock and multiple-stock environments. In the single-stock environment, the results were impressive. After training the policy for 2000 episodes, the average profit for 1000 steps was 121 % of the starting fund. The policy tends to sell when the stock surges, mimicking the behavior of a professional investor. See fig 3.

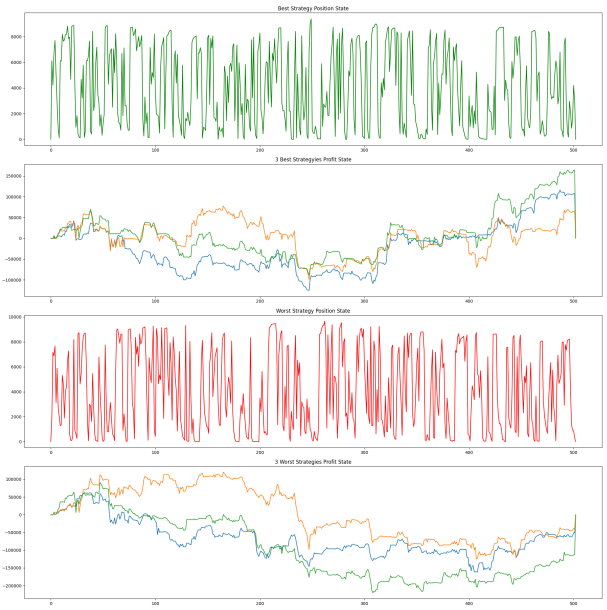

Figure 3. The result plots for profit and position stats in test

However, this is just a simulation using historical data, and the model may overfit the data. To evaluate the performance of the model, we used the backtest class to analyze the pattern of the position and profit. We found that the model's risk preference could be tuned to achieve even better results.

The backtest class allows us to analyze the performance of the policy over multiple episodes and identify patterns in the position and profit. By tuning the model's risk preference, we can further improve the performance of the policy in future simulations.

5 CONCLUSIONS

The success of our enhanced DQN in the Flappy Bird game and our previous work in stock trading demonstrate the potential of deep reinforcement learning in stochastic processes where some information is missing or incomplete. By learning directly from the pixels and score in Flappy Bird, we achieved super-human results, surpassing human-level performance. Similarly, in the stock trading environment, we were able to make successful trades despite missing or noisy data. Although training was not always consistent and overfitting or forgetting may have occurred, the potential of the DQN to successfully perform in these environments highlights the power of reinforcement learning in addressing problems where incomplete or noisy data is present. Additionally, our experience with uniform sampling from the replay memory suggests that prioritizing important experiences could lead to even better performance and more efficient training. Overall, our work underscores the promise of reinforcement learning in

addressing a variety of stochastic problems and highlights several important avenues for future research in this field.

References

[1] Chen, Kevin. *Deep Reinforcement Learning for Flappy Bird*. Stanford University, 2017. https://cs229.stanford.edu/proj2015/362_report.pdf.

[2] Chuchro, Robert, and Deepak Gupta. *Game Playing with Deep Q-Learning Using Openai Gym*. Stanford University, 2017. http://cs231n.stanford.edu/reports/2017/pdfs/616.pdf.

[3] Cong, Lin, Tang, Ke, Wang, Jingyuan, and Zhang, Yang. *AlphaPortfolio: Direct Construction Through Deep Reinforcement Learning and Interpretable AI*. August 1, 2021.

[4] He, Kaiming, et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. IEEE International Conference on Computer Vision (ICCV), 2015. https://doi.org/10.1109/iccv.2015.123