AI Learns and Plays SuperMario

YaoHui Chen Student ID: 2020533110
PengChen Ding Student ID: 2020533040
Ken Chen Student ID: 2020533036

Abstract

This project explores the application of Q-Learning, a reinforcement learning algorithm, to train an agent to play the popular video game Super Mario. The goal is to teach the agent how to navigate complex levels, overcome obstacles, defeat enemies, and ultimately complete the game successfully. Q-Learning involves iteratively updating a Q-table to learn the optimal policy by maximizing long-term cumulative rewards. Our goal is to use Q-Learning complete the game successfully and get as much score as possible.

1 Introduction

Super Mario is a popular game that requires precise timing, strategic decision-making, and quick reflexes to navigate through various levels, defeat enemies, and rescue Princess Peach. Q-Learning is a reinforcement learning algorithm that has been widely used to teach agents how to play complex video games, including the Super Mario. Q-Learning provides a framework for the agent to learn an optimal policy by iteratively updating a Q-table, which stores the expected rewards for different state-action pairs.

The agent starts with no prior knowledge and explores the game environment, gradually learning which actions lead to positive outcomes. Techniques such as discretization of the state space and reward shaping are employed to tackle the large state and action spaces of Super Mario. Through iterative learning and exploration, the agent becomes proficient in gameplay, showcasing the potential of reinforcement learning in tackling challenging gaming environments. The application of Q-Learning to Super Mario not only has implications for gaming but also provides insights into the development of intelligent agents capable of handling real-world problems.

2 Methodology

2.1 Markov Decision Process

A Markov Decision Process is defined by a set of states, a set of actions, a transition function, a reward function, a start state and a terminal state. Thus, SuperMario Game is obviously a typical MDP problem.

2.1.1 States

Intuitively, the states include the location of Mario, the location of the monsters, the location of the coin and the flag, the terrain in the game. To be precise, in our project, the state is a grey-scale map which provides us information of that moment and the agent will decide what to do depending the grey-scale map.

2.1.2 Actions

In the game, Mario can move left, move right, stay still or jump, which altogether form the set of action.

Submitted to 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Do not distribute.

2.1.3 Transition Function

Transition function can be defined by the probability of transforming from one state to another.

2.1.4 Reward Function

The reward function can be determined by many factors, such as whether Mario get to the terminal, whether Mario die, how many monsters Mario beats, how many coins Mario gets and the duration of the game. We can set different weights to these factors to train the agent.

2.2 Deep Q Network

Deep Q Network is a kind of Deep Reinforcement Learning, which is a combination of Deep Learning and Q-learning. Due to the limitations of Q-learning that it is impossible to choose the best action when the number of combinations of states and actions is infinite, using a deep neural network to help determine the action is reasonable.

2.2.1 Q-learning algorithm

Q-learning is a model-free reinforcement learning technique proposed by Watkins in 1989. It is able to compare the expected utility of available actions (for a given state) without requiring a model of the environment. It can handle random transitions and rewards without tuning. It has been shown so far that for any finite MDP, Q-learning eventually finds an optimal policy such that the expected value of the total payoff of all successive steps, starting from the current state, is maximally achievable. Before learning begins, Q is initialized to any fixed value (chosen by the programmer) that is possible. Then at each time t, the Agent chooses an action a_t,enter a new state s_t and update Q-value. The key of Q-learning is value iteration, such that:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [r_t + \gamma \max_{\pi} Q(s_{t+1}, a_t) - Q(s_t, a_t)]$$

2.2.2 DQN algorithm

In the Super Mario the environment is deterministic, so all the equations listed below are formulated deterministically for simplicity. Our aim will be to train a policy that can maximize the reward $R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$,here the γ is the discount factor,which should be a constant between 0 and 1 to make sure the sum can converge.In the Q-learning algorithm, we get a table of the Q values of the combinations of states and actions, then we construct a policy that maximizes the rewards:

$$\pi^*(s) = \operatorname*{argmax}_a \ Q^*(s, a)$$

However, since the number of the combinations of states and actions is infinite in this scene, so we use a neural network to resemble Q^*. And by Bellman equation,we get:

$$Q^{\pi}(s, a) = r + \gamma Q^{\pi}(s', \pi(s'))$$

The difference between the two sides of the equation is known as the difference discussed in the lecture:

$$\delta = Q(s, a) - (r + \gamma \max_{a'} Q(s', a'))$$

2.2.3 input and feature extraction

In our project, we use the graph of the game as input, then do feature extraction to the graph.

The feature extraction process can be described as follows:

Firstly, the initial convolutional layer, **nn.Conv2d(inputShape[0], 32, kernelsize=8, stride=4)**, takes the input with inputShape channels and produces 32 feature maps. By employing an 8x8 kernel size and a stride of 4, this layer captures essential spatial information. Subsequently, the resulting feature maps are passed through the activation function, **nn.ReLU()**, which introduces non-linearity and enhances the network's ability to capture complex patterns.

Next, the second convolutional layer, **nn.Conv2d(32, 64, kernelsize=4, stride=2)**, further refines the features extracted from the previous layer. This layer takes the 32 input channels and generates 64 output channels. With a 4x4 kernel size and a stride of 2, it effectively captures spatial dependencies and patterns in the input data. Again, an activation function, **nn.ReLU()**, is applied to introduce non-linearity and enable the network to capture complex relationships within the features.

Lastly, the third convolutional layer, **nn.Conv2d(64, 64, kernelsize=3, stride=1)**, refines the features even further. This layer operates on the 64 input channels and produces 64 output channels. Utilizing a 3x3 kernel size and a stride of 1, it focuses on capturing intricate details and fine-grained patterns within the input. An activation function, **nn.ReLU()**, is employed to introduce non-linearity and enhance the network's ability to model complex feature interactions.

By stacking these convolutional layers and applying activation functions, the input data undergoes multiple transformations, allowing the network to extract and represent important spatial features. In the given code, this feature extraction layer effectively transforms the input image data into a high-dimensional feature vector.

It is important to note that this implementation serves as an example, and the specific configuration of convolutional layers and activation functions can be customized based on the problem domain and desired feature representation.

2.3 Priority DQN

Priority DQN is an extension of the DQN algorithm that introduces a prioritized experience replay mechanism. In Priority DQN, the replay buffer assigns priorities to experiences based on their TD (Temporal Difference) error. TD error represents the difference between the predicted Q-value and the target Q-value for a given state-action pair. Experiences with higher TD errors are considered more important or informative.

Concretely, instead of uniformly sampling experiences from the replay buffer, Priority DQN samples experiences with probabilities based on their priorities. Experiences with higher priorities have a higher chance of being selected for training. After each update step, the priorities of the sampled experiences are updated based on their TD errors. Experiences with larger TD errors are given higher priorities, which means they are more likely to be sampled in future training iterations.

2.4 Some Theory on DQN

2.4.1 DQN are not promised to converge.

In supervised learning problems, we minimize the loss to get the parameters in optimal solution. The algorithm always converge to the optimal solution. While in off-policy unsupervised learning, the loss are not promised to converge. In paper Full Gradient DQN Reinforcement Learning: A Provably Convergent Scheme[1], the writer has given full explanation that DQN are not promised to converge. So, it is not necessary for us to focus on the loss curve.

2.4.2 Average Final Reward (short for AFR) is a better indicator than Loss.

In the context of the game Super Mario, the question that truly matters to us is: what is the highest score that Mario can achieve in a level through the strategy we have learned? As pointed out in OpenAI's documentation on reinforcement learning [2], we can consider the loss as a way of calculating the loss of a Convolutional Neural Network (CNN). However, what we should really be concerned about is the average reward, as this represents the performance of the strategy. The non-convergence of the loss can sometimes be related to the policy we are training. Certain samples may trigger hidden rewards, contributing to the observed behavior. This is our reason for choosing AFR as indicator for learning curve.

3 Result

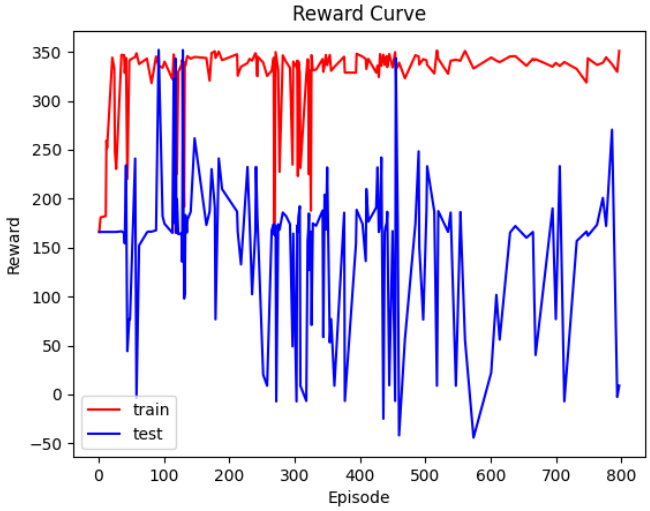
In order to make the DQN algorithm learning model more intuitive, we decided to use our defined Reward to investigate the accuracy and generalization of the model.

As has been proven, for probabilistic random enemies and game scenarios, Super Mario can obtain a convergent policy. Even though Reward is not directly equivalent to policy, nor can it directly show the characteristics of policy, we can roughly summarize the effect of policy by observing the relationship between Reward and Episode.

In the training process of our model, we chose the 1-1 level. In order to avoid computing too much data at the same time and causing too much GPU memory load or low computational efficiency, as well as adjusting our trained network, we decided to train the model in two steps, that is, pre-training and the training. Between every two training batch, we recorded the previous Episode and the corresponding ϵ value.

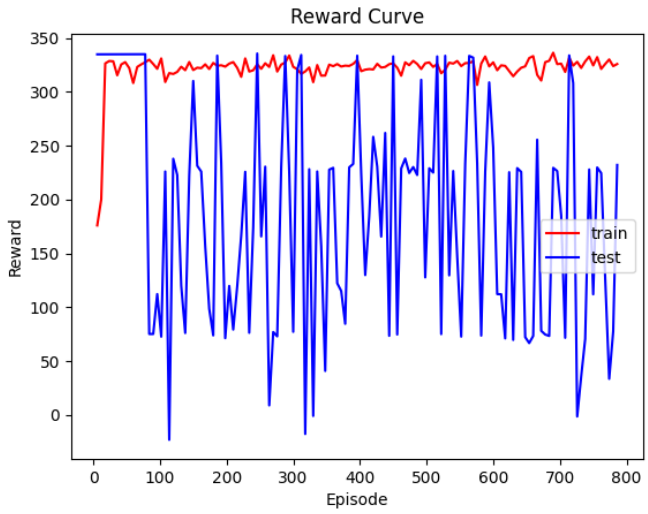
3.1 Pre-training Step

We pre-trained the model at the beginning, here is the result of the second batch of this pre-training step.



It can be seen from the figure that in the second batch of nearly 1000 sampling points, train reward has reached the upper bound and begins to show a stationary law. Combined with the above proof, we can infer that the upper bound reached by train reward is the local optimal strategy of this level.

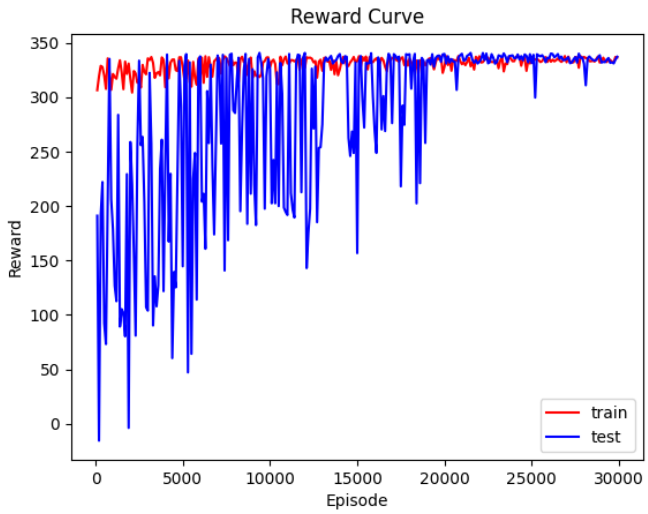
After that, here is the result of the third batch.



From the figure, it is obvious that the sampling mean of train reward has begun to converge, while test reward also begins to show a certain pattern. Specifically, in every 100 samples, the test reward of 1 or 2 samples will reach the upper bound, at the same time, the mean value of the test reward of every 100 samples is also rising.

3.2 Training Step

After the pre-training step, we begun to train our model formally. Here is the training result.



From the plot of the running results, it is clear that train reward and test reward begin to converge together at about 20,000 episodes, and the mean value of test reward is also constantly approaching the upper bound.

3.3 Validation & Reflection

After the training step, we begun to verify the generalization of the model we trained. Here is the result of our validation.

Level	Win (T or F)	Reward
1-1	T	336.75
1-2	F	7.50
1-3	F	20.50
2-1	F	18.00
3-1	F	39.80
4-1	F	35.60
5-1	F	7.60
6-1	F	-44.40
7-1	F	-40.70

Unfortunately, it turns out that our trained model generalizes poorly. For all levels except 1-1, our model failed, even though we have trained more than 30,000 times and the model has been trained more than 10,000 more times while already converging.

We reflected on the problems that occurred during training and concluded that the reasons for poor generalization performance were as follows:

- Level 1-1 was so simple that there were too few features that could be extracted by the DQN algorithm to include features from the more complex levels that followed. Thus, our model was effectively underfitting when testing other levels.
- Different levels have different art style backgrounds, which causes our model to confuse and misjudge when dealing with different art style backgrounds in other levels (for example, in the initial stages of 6-1 and 7-1, Mario stops moving because he mistoo a door in the background for a water pipe). As a result, our model cannot effectively deal with such an environment.

To address these issues, we propose the following solutions:

- In the preprocessing, the training for different levels is added to ensure that the model can extract enough features.
- When building the environment, use the opencv library to pre-process different level scenes to preliminarily identify different styles of art design background, so as to ensure that the background will not be wrongly judged as obstacles by the model.

4 Conclusion

In this project, we utilized Q-Learning and Deep Q-Networks (DQN) to train an agent for playing Super Mario. By implementing an MDP framework and using a deep neural network as a function approximator, we trained the agent to optimize its long-term rewards. We introduced a prioritized experience replay mechanism to enhance the learning process.

During training, we observed the convergence of train rewards and improvements in test rewards, indicating the agent's learning progress. However, the model exhibited poor generalization when tested on different levels, primarily due to the simplicity of the initial level and variations in art style backgrounds.

To tackle these challenges, we proposed solutions such as level-specific training and image pre-processing to handle diverse art styles. These enhancements aim to improve the model's ability to generalize.

Overall, our project showcases the potential of Q-Learning and DQN for training agents in complex video games like Super Mario. While further improvements are needed for better generalization, our work offers valuable insights for the development of intelligent agents in real-world applications.

References

[1] Konstantin E. Avrachenkov, Vivek S. Borkar, Hars P. Dolhare Kishor Patil (2021) Full Gradient DQN Reinforcement Learning: A Provably Convergent Scheme

[2] https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html
[3] https://zhuanlan.zhihu.com/p/503546476