Reinforcement Learning Assignment 2
Deep Q Learning

Suju Li Ruiqing Sun Peiwen Xing

Abstract

This report presents insights on the implementation of Deep Q-Networks (DQN) for the OpenAI Cartpole environment, a benchmark problem in reinforcement learning. By integrating DQN with experience replay and target networks, and exploring different exploration policies such as ϵ-greedy and softmax, we aim to evaluate the impact of network hyperparameters, exploration policies, learning rates, and architectural choices on the algorithm's performance. Through systematic experimentation, we identify optimal configurations that significantly enhance the agent's learning efficiency and stability, discussing limitations and avenues for future work.

1. Introduction

In Assignment 1, we used the q learning approach to train the agent to reach the goal with the optimal path in a grid environment. This approach is effective in state limited MDP. While tabular Q-learning provides a solid foundation for understanding the basics of Q-learning, DQN and its variants offer scalable, efficient, and effective solutions for complex environments with high-dimensional state spaces. The integration of deep learning with reinforcement learning has significantly advanced the field, enabling applications that were previously thought to be out of reach for machine learning techniques.

In this assignment, we will implement DQN in the environment provided by OpenAI's Cartpole, which is a classic reinforcement learning problem that serves as a benchmark for various algorithms. In this environment, we aim to implement DQN with the experience replay and target network, while using ϵ-greedy or softmax as its exploration policy. After that, we will conduct experiments to compare the influence of network hyper-parameters, exploration policies, learning rates, experience replay and target network. At last, we will interpret the experiment results, discussing the limitations of our solutions and room for future improvement.

2. Methodology

2.1. DQN

Deep Q-Networks (DQN) represent a significant advancement in the field of reinforcement learning (RL), combining traditional Q-learning with deep neural networks to tackle environments with high-dimensional state spaces. Introduced by Mnih et al. in their groundbreaking paper "Playing Atari with Deep Reinforcement Learning" (2013), DQN was the first successful attempt to apply deep learning to solve complex RL problems directly from high-dimensional sensory inputs.

2.1.1. Q Learning

The foundation of DQN lies in the Q-learning algorithm, a model-free reinforcement learning technique. Q-learning aims to learn the value of an action in a particular state, providing a policy that an agent can follow to maximize its total reward. The Q-value function, $Q(s, a)$, represents the expected return of taking action a in state s and following the optimal policy thereafter. The core of Q-learning is expressed by the Bellman equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \underset{a}{max} Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (1)$$

2.1.2. Integration with Deep Learning

The innovation of DQN is the use of a deep neural network to approximate the Q-value function. When the state space becomes too large for a table representation, a neural network can generalize across states, learning to predict Q-values.

A typical training process of DQN without experience replay and target network is shown below:
1. Initialize the network: Input state s_t, output Q-values for all actions available in state s_t.
2. Utilize a policy (e.g., ϵ-greedy) to select an action a_t. Input a_t into the environment, obtain the new state s_{t+1} and reward r.
3. Compute the TD target:

$$y_t = r_t + \gamma \cdot \max_a Q(s_{t+1}, a; w) \quad (2)$$

Where: r_t is the value obtained based on actual observations.

$\max_a Q(s_{t+1}, a; w)$ is the estimated value based on the Q-network at state s_{t+1}.
4. Calculate the loss function:

$$L = \frac{1}{2}[y_t - Q(s, a; w)]^2 \tag{3}$$

5. Update the Q parameters to make $Q(s_t, a_t)$ as close as possible to y_t. This is treated as a regression problem, utilizing gradient descent for updates.
6. From the above steps, we obtain a tuple known as a transition: (s_t, a_t, r_t, s_{t+1}). After it is used, it is discarded.
7. Input the new state and repeat the update process.

2.2. Experience Replay

Before introducing experience replay, let's take a look at the drawbacks of the original DQN algorithm:
1. After using a transition (s_t, a_t, r_t, s_{t+1}), it is discarded, leading to wastage of experiences.
2. Using transitions sequentially leads to high correlation between consecutive transitions, which is detrimental to learning the Q-network.

Experience replay can overcome these drawbacks in the following ways:
1. Experience replay shuffles the sequences, eliminating correlation and making the data independent and identically distributed. This reduces the variance of parameter updates and speeds up convergence.
2. Experience replay allows for the reuse of experiences, resulting in higher data utilization rates, particularly beneficial in scenarios where acquiring new data is challenging.

During reinforcement learning, the most time-consuming step is often interacting with the environment, while training the network tends to be faster, especially with GPU acceleration. Using a replay buffer can reduce the number of interactions with the environment. Experiences do not need to be exclusively from a single policy; instead, experiences obtained from past policies can be reused multiple times in the replay buffer.

Experience replay constructs a replay buffer to store n transitions, referred to as experiences. A policy π interacts with the environment, collecting many transitions, which are placed into the replay buffer. The experiences in the replay buffer may come from different policies. The replay buffer only discards old data when it becomes full. During training, a batch of transitions is randomly sampled from the buffer, and multiple random gradients are computed. The network parameters w of the Q-network are then updated by averaging these gradients.

2.3. Target Network

The key concept of target network is bootstrapping. In reinforcement learning, bootstrapping refers to using estimated values of subsequent states to update the estimated value of the current state.

As shown in Equation 2, TD target y_t partly comes from the estimation of the Q-network, and we use y_t to update the Q-network itself, so this is a form of bootstrapping.

When calculating the TD target, we maximize the Q-value: $\max_a Q(s_{t+1}, a; w)$. This maximization, along with the bootstrapping process mentioned above, can lead to overestimation issues. Using target networks can mitigate bootstrapping to some extent, thus alleviating overestimation issues. We use the second network, referred to as the target network, denoted as $Q(s, a; \mathbf{w}^-)$. The network architecture is the same as the original network $Q(s, a; \mathbf{w})$, except for different parameters ($\mathbf{w}^- \neq \mathbf{w}$). The original network is called the evaluation network.

The two networks serve different purposes: the evaluation network $Q(s, a; \mathbf{w})$ is responsible for controlling the agent and collecting experience, while the target network $Q(s, a; \mathbf{w}^-)$ is used to compute the TD target:

$$y_t = r_t + \gamma \cdot \max_a Q(s_{t+1}, a; \mathbf{w}^-) \tag{4}$$

During the update process, only the weights $\mathbf{w}$ of the evaluation network $Q(s, a; \mathbf{w})$ are updated, while the weights $\mathbf{w}^-$ of the target network $Q(s_{t+1}, a; \mathbf{w}^-)$ remain unchanged. After a certain number of updates, the updated weights of the evaluation network are then copied to the target network for the next batch of updates. This allows the target network to also receive updates. Since the target network remains unchanged for a period of time, the target values of returns are relatively fixed, which increases the stability of learning.

2.4. Learning Rate Scheduling

Learning rate scheduling is a strategy used in training neural networks that involves adjusting the learning rate over time. The goal of learning rate scheduling is to dynamically find a balance that allows for both efficient and effective model training. There are several strategies for learning rate scheduling, including: time-based decay, step decay, exponential decay and Cosine annealing.

We would love to use CosineAnnealingWarmRestarts as the learning rate scheduling for our model training. The fundamental principle behind is to start with a relatively high learning rate and decrease it following a half-cosine curve to a minimum value, before "restarting" the learning rate back to the high value. This process is repeated several times, with each cycle called a "restart". The length of these cycles can either be kept constant or increased after each restart.

Mathematically, η_t at a given epoch *t* can be described as:

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + cos(\frac{T_{cur}}{T_i}\pi)) \quad (5)$$

Where: η_{min} is the minimum learning rate, η_{max} is the maximum (initial) learning rate, T_{cur} is the number of epochs since the last restart, T_i is the total number of epochs in the current cycle.

3. Experiment

We designed a total of four experiments. The first experiment is to observe the effect of model parameters on DQN performance. We picked the number of hidden layers and the dimension of the hidden layers as the two hyperparameters to be tuned. The second experiment is to compare the effects of ϵ-greedy and softmax strategies on the training process of DQN. In addtion to this experiment, we will discuss the tuning of ϵ in detail. The third experiment is to discuss the effect of learning rates on the learning of the model. The fourth one is to compare the performance difference between DQN with DQN-ER[1], DQN-TN, DQN-ER-TN.

Our experiment codes and detailed instructions could be found **here**. For whoever wants to try it out, please download it and type these commands in the terminal:

- run DQN:
  ```
  python dqn.py
  ```
- run DQN-ER:
  ```
  python dqn.py --experience_replay
  ```
- run DQN-TN:
  ```
  python dqn.py --target_network
  ```
- run DQN-ER-TN:
  ```
  python dqn.py --experience_replay
  --target_network
  ```

3.1. Network Parameters

Adjusting the neural network's depth (*LAYER_COUNT*) and width (*HIDDEN_DIM*) critically influences the agent's learning efficacy and performance.

Baseline Setting: Initially, a baseline was established with one hidden layer (*LAYER_COUNT*=1) and 128 nodes (*HIDDEN_DIM*=128), based on prior experience.

Iterative Testing Process: The depth was tested from 1 to 3 layers, incrementing by one, and width from 64 to 256 nodes, doubling at each step. This process systematically varied one parameter at a time to determine its impact independently. The DQN model was trained for 600 episodes across each configuration, assessing metrics such as average reward, learning speed, and stability.

Fine-tuning: Analysis of the results identified performance patterns, guiding the fine-tuning towards configurations showing optimal initial performance which involves smaller increments and exploring in between previously tested values to find a more optimal setting.

3.2. Exploration Policy[2]

In the ϵ-Greedy strategy, the algorithm randomly selects an action with probability ϵ and chooses the current believed optimal action with probability $(1 - \epsilon)$. Over time, the value of ϵ gradually decreases, meaning that the algorithm gradually reduces exploration and increases exploitation of known optimal actions.

The Softmax strategy selects actions based on the probability distribution of action-value function (Q-values). A key parameter of this strategy is the temperature parameter (*temp*), which controls the level of randomness in action selection. A lower *temp* value means that the algorithm tends to select actions with higher Q-values, while a higher *temp* value increases the randomness of exploration.

3.2.1. Decay Methods

ϵ-greedy: We designed an ϵ-greedy strategy that decays as training progresses, as seen in the following equation.

$$eps \leftarrow eps\_min + (eps\_start - eps\_min) * e^{-\frac{episode}{decay\_factor}} \quad (6)$$

Where: *eps* is the value of ϵ of every episode, *eps_start* is the starting value, *eps_min* is the minimum value of ϵ, and *decay_factor* is the decay rate.

softmax: We designed a softmax policy whose *temperature* decays during training, as seen in the following equation.

$$temp \leftarrow max(temp\_start * 0.99^{num\_episode}, temp\_min) \quad (7)$$

Where: *temp* is the value of *temperature* of every episode, *temp_start* is the starting value, *temp_min* is the minimum value of *temperature*, and *num_episode* is the n^{th} episode of the training.

3.2.2. Value of ϵ

Besides the decay ϵ-greedy, three fixed ϵ were also tested out, where $\epsilon = [0.9, 0.5, 0.3]$. We would like to observe the training speed, stability, and final performance of the model under different ϵ.

[1]Where the notation - means reduction. It means DQN without experience replay with the notation of DQN-ER.

[2]We designed and experimented a step based ϵ-greedy decay strategy, which could be our effort for bonus.

3.3. Learning Rate[3]

Like the section 3.2.2 mentioned above, we compared training with a decaying learning rate as well as three fixed learning rates, where $learning_rate = [0.00005, 0.0001, 0.0005]$. As for learning rate scheduling strategy, we used CosineAnnealingWarmRestarts from PyTorch and $T\_0 = 10, T\_mult = 2$ as its parameters.

3.4. DQN-ER, DQR-TN, DQR-ER-TN

3.4.1. DQN-ER

The streamlined DQN variant removes experience replay, opting for immediate, sequential learning in dynamic environments. It utilizes a policy network and a periodically updated target network, both adjustable in layer count and dimensions, to approximate Q-values. Algorithm 1 is a brief pseudo codes of its design:

Algorithm 1 Training DQN without Experience Replay

```
Initialize episode_durations list, policy_net, target_net
Set optimizer for policyNet with learning rate LR
for episode from 1 to num_episodes do
    s ← Initial state from the environment
    while episode not done do
        Select an action $a_t$:
            With probability $\epsilon$, choose a random action
            Otherwise, $a_t = \max_a Q^*(\varphi(s_t), a; \theta)$
        Execute $a_t$ in the environment and observe next_state, reward, done
        Use the observation to update policy_net
        Periodically update targetNet weights with $\tau$
        s ← next_state
        Gradually decrease $\epsilon$ towards $\epsilon_{\text{end}}$
    end while
    Append t to episode_durations
end for
return policy_net, episode_durations
```

3.4.2. DQN-TN

DQN-TN refers to a variant of DQN without a target network. In this variant, there is only one neural network, the policy network, which is used for both action selection and Q-value estimation. This approach simplifies the structure of the DQN algorithm, making the entire training process more concise and straightforward. However, due to the absence of a target network, it may lead to instability and convergence issues during training. Algorithm 2 is a brief pseudo codes of its design:

[3]We conducted an experiment comparing decaying learning rate with fixed ones, which could be our effort for bonus.

Algorithm 2 Deep Q-learning without Target Network

```
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for episode = 1, M do
    Initialize sequence $s_1 = \{x_1\}$ and preprocess sequence $\varphi_1 = \varphi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\varphi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\varphi_{t+1} = \varphi(s_{t+1})$
        Store transition $(\varphi_t, a_t, r_t, \varphi_{t+1})$ in D
        Sample random minibatch of transitions $(\varphi_j, a_j, r_j, \varphi_{j+1})$ from D
        Set $y_j = r_j$ for terminal $\varphi_{j+1}$, otherwise set $y_j = r_j + \gamma \max_{a'} Q(\varphi_{j+1}, a'; \theta)$
        Perform a gradient descent step on $(y_j - Q(\varphi_j, a_j; \theta))^2$
    end for
end for
```

3.4.3. DQN-ER-TN

DQN-ER-TN refers to a variant of DQN without experience replay and target network. The policy network is updated from immediate, sequential game data given by the environment. This network is used for both action selection and Q-value estimation. Algorithm 3 is a brief pseudo codes of its design:

Algorithm 3 Training DQN without Experience Replay and Target Network

```
Initialize episode_durations list, policy_net
Set optimizer for policyNet with learning rate LR
for episode from 1 to num_episodes do
    s ← Initial state from the environment
    while episode not done do
        Select an action $a_t$:
            With probability $\epsilon$, choose a random action
            Otherwise, $a_t = \max_a Q^*(\varphi(s_t), a; \theta)$
        Execute $a_t$ in the environment and observe next_state, reward, done
        Use the observation to update policy_net
        s ← next_state
        Gradually decrease $\epsilon$ towards $\epsilon_{\text{end}}$
    end while
    Append t to episode_durations
end for
return policy_net, episode_durations
```

3.5. DQN with CNN[4]

In all the above experiments, the models are using a simple fully connected model, we followed these experiments by doing a DQN based on a convolutional neural network (CNN) which contains 3 convolutional layers with a kernel size of 5 and a stride of 2. We run the model three times to get its average performance.

4. Results[5][6][7]

4.1. Network Parameters

Figure 1 and 2 can be intuitively analysed to identify patterns or trends. The following phenomena can be observed when adjusting the network's depth (*LAYER_COUNT*) and width (*HIDDEN_DIM*):

Impact of Layer Count on Performance: The transition from 1 to 2 layers resulted in a marked enhancement in performance, with less volatility and higher average rewards. However, expanding the network to three layers did not continue this trend. This suggests that for this particular task, there exists an optimal depth beyond which additional layers may detract from performance, likely due to overfitting or difficulty in training optimization.

Effect of Hidden Layer Dimension on Learning: Layer dimension had a notable impact on learning speed and performance stability. With 64 nodes, the network's performance was quite erratic, indicating inadequate capacity for learning. Upon increasing the node count to 128, a substantial gain in reward stability appeared, signifying a more competent learning process. However, increasing the node count to 256 further increased volatility, which might reflect the model's overfitting to the training data or its capture of irrelevant patterns (noise).

Trade-off Between Model Complexity and Efficiency: The experiments showed an evident trade-off between the complexity of the DQN model and its performance.While some level of complexity is required for effective task learning, overly intricate models, characterized by an excessive number of layers or nodes, did not result in improved performance and showed indications of overfitting. Additionally, these complex models entail higher computational costs, potentially affecting training time and resource efficiency. The network with two layers and 128 nodes seemed to strike the optimal balance, offering robust performance without excessive computational demands.

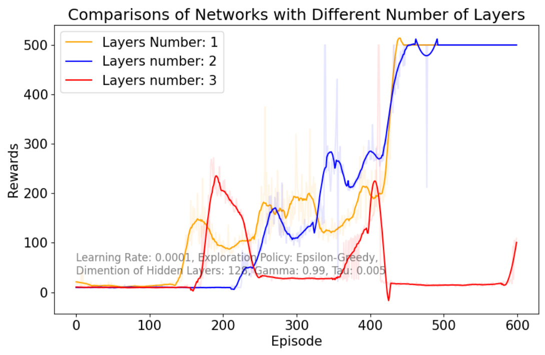

Figure 1. Network Performance with Different Layer Counts

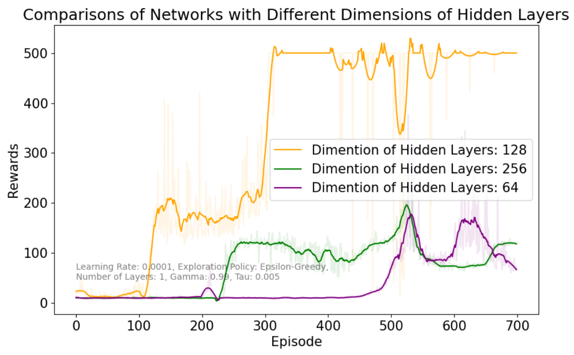

Figure 2. Network Performance with Different Numbers of Nodes

4.2. Exploration Policy

4.2.1. ϵ-GREEDY VS SOFTMAX

Figure 3 illustrates a comparison of two different exploration strategies—ϵ-Greedy and Softmax.

In the figure, the score of the ϵ-Greedy strategy starts relatively low, but as ϵ decreases, the algorithm begins to exploit known optimal strategies more, leading to a gradual increase in score.

The score of the Softmax strategy is initially higher than that of the ϵ-Greedy strategy, indicating that the Softmax strategy may be able to find effective actions more quickly during the exploration phase.

From the figure, it can be observed that the performance of

[4]We implemented a CNN based DQN and conducted related experiment, which could be our effort for bonus.

[5]The results of the four experiments will be shown in this section. When looking at these results, it is important to note that the yellow line in all the graphs represents the same model, which is the one that we believe performs the best.

[6]Among all the graphs, the lines with some transparency represent the original reward data and the solid lines represent the smoothed data.

[7]The gray texts in every graph indicate the parameters which are shared among models in the same graph.

the ϵ-Greedy strategy remains relatively stable after reaching the optimal strategy, while the Softmax strategy shows significant fluctuations in performance indicators after reaching the optimal strategy. One possible reason is that the adjustment of the temperature parameter is not smooth enough. In the Softmax strategy, the temperature parameter has a significant impact on the probability distribution of action selection. If the temperature parameter decreases too quickly or the adjustment is not smooth during training, it can cause the probability distribution to transition rapidly from being relatively uniform to sharp, leading to drastic changes in the behavior of the agent.

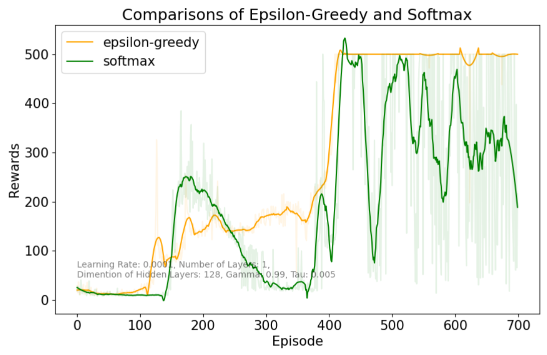

Figure 3. Comparisons of ϵ-Greedy and Softmax

4.2.2. Value of ϵ

In figure 4, we can observe the impact of different epsilon values on the training speed. Specifically, the performance of the decayed epsilon strategy is significantly better than that of the fixed epsilon strategy. In the decay strategy, the epsilon parameter gradually decreases over time or with an increase in training episodes. The purpose of this strategy is to allow the agent to have high exploration in the early stages of training, enabling it to explore the environment extensively and learn possible actions and outcomes. As training progresses, the epsilon value gradually decreases, reducing the agent's exploration behavior and relying more on learned knowledge to make decisions. In the fixed epsilon strategy, the agent uses the same epsilon value throughout the entire training process. While this strategy is simple and easy to implement, a fixed epsilon value may not be suitable for different stages of training, potentially leading to poor balance between exploration and exploitation and fluctuations in performance during training.

From Figure 4, it can also be observed that lower epsilon values result in better performance. Generally, higher epsilon values imply more exploration behavior, which may slow down the training process as the agent requires more time to explore and learn the environment. Conversely, lower epsilon values may lead to faster training speeds as the agent tends to exploit known strategies more.

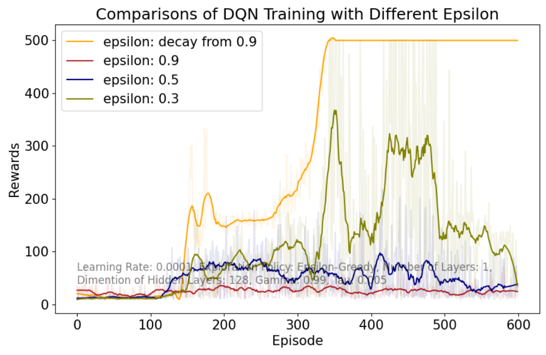

Figure 4. Comparisons of DQN Training with Different ϵ

4.3. Learning Rate

Figure 5 shows the training process under different learning rates. It's evident that with a higher learning rate, the network learns faster in the beginning and towards convergence. However, it can also cause the training process to become unstable. The optimizer may "overshoot" the minimum of the loss function, leading to divergent behavior where the loss oscillates or even increases. The learning speed could be illustrated by the rate at which the curve rises and the order in which it reaches convergence. And the stability could be indicated by the oscillations of curves with some degree of transparency.

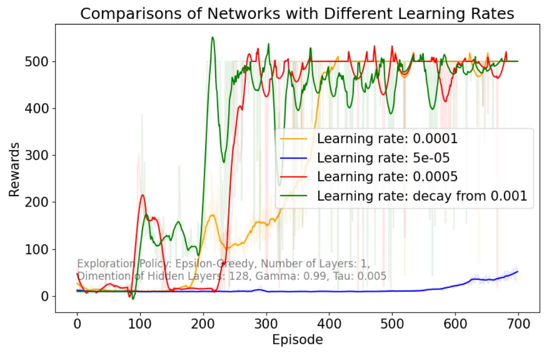

Figure 5. Network Performance with Different Learning Rates

Fixed Learning Rates: The network with $learning\_rate = 0.0005$ learns the fastest, but performs the least consistently. But the network with

$learning\_rate = 0.00005$ shows no sign of learning within 700 episodes. Thus we believe $learning\_rate = 0.0001$ could be a good option when considering the trade-off between learning speed and stability.

Decayed Learning Rates: The key motivation of using a decay method for learning rate scheduling is we can use a high value of *learning_rate* in the beginning and gradually decay it while training, by which the network can have faster convergence and better stability. As shown in Figure 5, the model (green line) with a *learning_rate* decaying from 0.001 shows the benefits of properly scheduling the textitlearning_rate. It is the first one that reaches a reward of 500 and has similar stability with the model whose $learning\_rate = 0.0005$ after 600 episodes.

4.4. Comparisons of DQN Reduction Variants

As shown in Figure 6, we compared the performance differences between DQN-ER, DQN-TN, and DQN-ER-TN, and found that they all performed very poorly, showing no signs of learning within 1000 episodes, and their rewards stabilized around 10. In Figure 7, we can see the significant difference in performance between DQN and these variants.

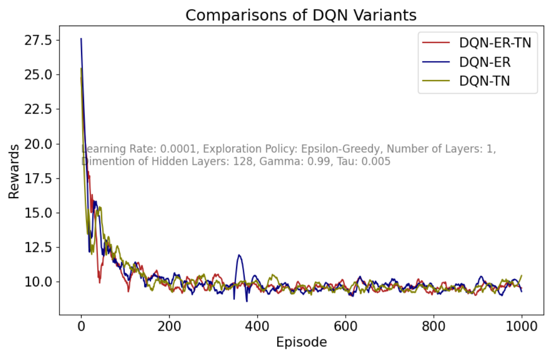

Figure 6. DQN Reduction Variants Performance

4.5. DQN with CNN

As shown in Figure 8, the DQN integrated with CNN reached a perfomance plateau after 300 episodes, which is worse than the previous fully connected network. Due to time limited, we didn't tune the hyper-parameters for this model and training process. It's possible that DQN with CNN could have more potentials to explore. Besides, Figure 8 suggests a more gradual and consistent learning process of this DQN. We can say that it has better generalization from the training data.

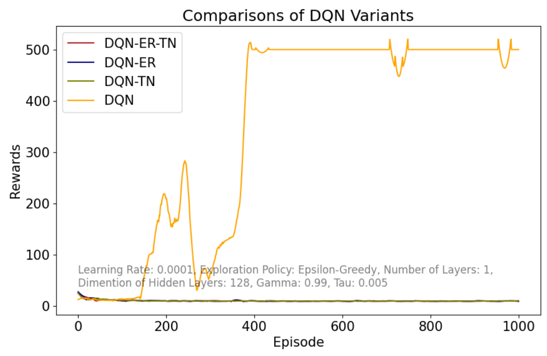

Figure 7. Comparisons of DQN and DQN Reduction Variants

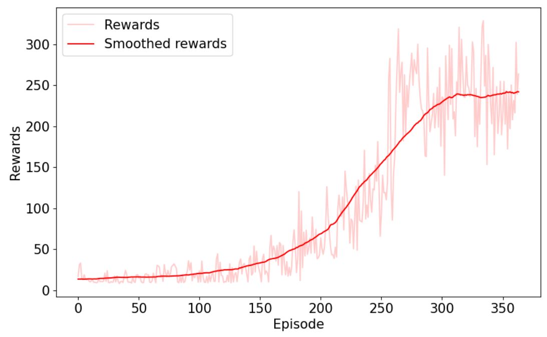

Figure 8. Perfomance of DQN integrated with CNN

5. Analysis

The experiment conducted aimed to assess the efficacy of DQN in the context of the OpenAI Cartpole environment, focusing on the influence of various factors such as network hyperparameters, exploration policies, learning rates, experience replay, and target networks. The DQN architecture was implemented with experience replay and target networks as foundational elements to stabilize learning and mitigate issues related to correlated experiences and non-stationary targets.

Initially, network hyperparameters, including the size and number of layers in the neural network, were systematically varied. It was observed that deeper networks with more layers did not necessarily translate to better performance, highlighting the importance of an appropriate complexity level for the task at hand. Overly complex models tended to overfit, while simpler models were unable to capture the necessary features for optimal decision-making.

Exploration policies, including ϵ-greedy and softmax were evaluated. The ϵ-greedy policy, with its balance between exploration and exploitation guided by a gradually decreasing epsilon value, demonstrated consistent learning improvement. Compared to the ϵ-greedy policy, the softmax policy, which selects actions based on a probability distribution from the Q-values, showed more variability in early training phases and could not stabilized within 700 episodes, suggesting its potential in environments requiring a more nuanced exploration strategy.

The impact of different learning rates was also scrutinized. A lower learning rate resulted in slower convergence but more stable learning, whereas higher rates led to faster initial improvement at the cost of potential instability in the learning curve. Finding the right balance was crucial for efficient training. Here we could tune a learning rate scheduler to help balancing between learning speed and stability.

Experience replay and target networks were pivotal in stabilizing the DQN training process and improving its performance. Experience replay allowed the model to learn from a diverse set of past experiences, reducing the correlation between consecutive learning updates. The use of a target network helped mitigate the moving target problem by providing a stable target for Q-value updates. These mechanisms collectively contributed to the robustness and effectiveness of the DQN implementation.

The experiments underscored the critical role of finely tuned exploration policies, network architecture, and learning rates in the success of DQN. While the DQN model achieved notable performance in mastering the Cartpole environment, challenges such as parameter sensitivity and the need for extensive experimentation to identify optimal configurations were evident.

6. Limitations

In this experiment, the adjustment of the temperature parameter in the Softmax strategy is not optimal enough, leading to significant fluctuations in performance indicators after reaching the optimal strategy.

Additionally, it can be observed that the performance of DQN largely depends on the selection of hyperparameters, such as network architecture, learning rate, and exploration factors. Manually tuning these parameters can be time-consuming, and improper choices of hyperparameters can severely affect the performance of the network.

Besides, we used the rewards directly given by the environment. The training efficiency could be better if we design a method to reshape the rewards.

At last, limited by time and computational resources, we haven't explored the CNN based DQN into depth. So far, our implementation of it converged on a sub-optimal performance. It's worthy to testify it stability for a longer training process based on the results we have in section 4.5.

7. Conclusion and Future Work

Our comprehensive study on the optimization of a Deep Q-Network for the CartPole-v1 challenge within the OpenAI Gym framework has produced a definitive set of optimal parameters that enable peak performance.

In conclusion, the confluence of a one-layer network with 128 nodes per layer, an adaptively decaying -greedy policy, and an optimally selected learning rate constitute the most effective set of parameters for the DQN agent applied to the CartPole-v1 task. This precise amalgamation fosters a learning environment where the agent not only excels rapidly but also sustains a high-quality performance throughout its learning phase.

For future research, several avenues can be pursued:

1. Implementing regularization methods like dropout or L2 regularization could further investigate the prevention of overfitting in more complex networks.

2. Employing automated hyperparameter optimization techniques, like grid search, random search, or Bayesian optimization, could identify better performing models without extensive manual tuning.

3. Investigating the transferability of learned policies to different but related tasks could enhance the efficiency of the learning process and the versatility of the agent.

4. Designing different rewards reshaping methods and comparing their effects on the model training.

5. Tuning the hyper-parameters for CNN based DQN and training it for more episodes on a high performance computer.

8. Division of Labor

Three members of our group made equal contribution to this assignment.
Suju Li: Implemented DQN and DQN-ER-TN architecture. Conducted experiments described in 3.3, 3.4, 3.5.
Ruiqing Sun: Implemented DQN and DQN-ER architecture. Conducted experiments described in 3.1.
Peiwen Xing: Implemented DQN and DQN-TN architecture. Conducted experiments described in 3.2.

References

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double Q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1).

Bellemare, M. G., Dabney, W., & Munos, R. (2017). Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*.

Zhang, C., Li, C., Zhang, S., & Huang, K. (2020). Dueling Double Deep Q Network with Expert Experience Replay Mechanism for UAV Maneuver Decision-Making. *IEEE Access*, 8, 215707-215716.

Kim, K. (2022). Multi-Agent Deep Q Network to Enhance the Reinforcement Learning for Delayed Reward System. *Applied Sciences*, 12(7), 3520.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized Experience Replay. *arXiv preprint arXiv:1511.05952*.