Reinforcement Learning 2024
Assignment 2: Deep Q Learning

Ritesh Sharma (4158059)

Abstract

This document provides the implementations and results of Reinforcement Learning Assignment 2. We implemented a Deep Q-Network (DQN) for OpenAI's Cartpole environment. This implementation included experience replay along with a target network. Hyperparameter tuning is done to achieve the best possible performance. After which an Ablation Study is performed. The Bonus selection of the assignment is also attempted.

1. Introduction

In the previous research we experimented with different tabular reinforcement algorithms, which worked well due to a relatively low state space for the agent. When expanding the state space, these methods will not work. Therefore, we introduce deep reinforcement learning. The algorithm will now make use of neural networks to improve the algorithms. In this case we will implement a deep Q-learning algorithm to solve a relatively simple problem. We will also experiment with regards to all the hyperparameters and see which ones will work better with regards to this problem.

2. Environment

The main setup for the following experiments is the following: We used the Torch (Paszke, 2019) library in Python to create the neural network.

2.1. Cartpole

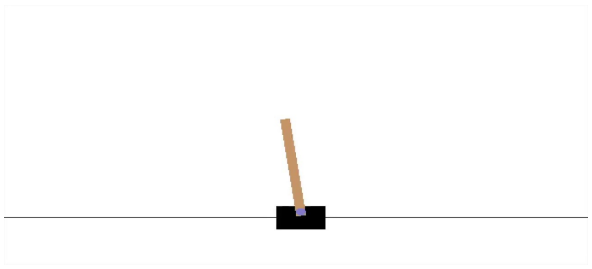

Figure 1. The cartpole environment (OpenAI, 2016–b)

The environment in which we will deploy the deep q-learning agent will be the CartPole-v1 environment (OpenAI, 2016–b) by OpenAI. The goal is to keep a pole in the air for as long as possible. This pole is attached to a cart which can move frictionless on a line from left to right.
The action space for an agent in this environment is relatively small. Since the cart is moving on a 2-d line, the agent can only push it left or push the cart right. The environment makes it so that the push is a fixed force.
The statespace on the other hand, is basically infinite and is denoted by four float values.

Table 1. Statespace CartPole (OpenAI, 2016–b)

NUM	OBSERVATION	MIN	MAX
0	CART POSITION	−4.8	4.8
1	CART VELOCITY	-INF	INF
2	POLE ANGLE	-24°	24°
3	POLE ANGULAR VELOCITY	-INF	INF

3. Deep Q-learning

Deep Q Learning (DQN) extends and refines the concept of Q-learning. This technique combines deep learning with Q-learning to overcome the limitation of Q-learning in which the Q-function is represented as a table and the size of the Q-table grows exponentially when there are large or continuous state spaces. To get around this problem, DQN uses a deep neural network to get close to the Q-function. This makes it easier for DQN to deal with large or continuous state spaces. DQN includes a target network that makes the training more stabilized by providing target Q-values during the updates. It also uses experience replay to store and sample past experiences to reduce the data correlation.

3.1. Methodology

1. **Imports and Argument Parsing:** The code starts by importing necessary libraries such as PyTorch, Gym, and argparse for argument parsing. It defines command line arguments for toggling options like experience replay and target network. `parser.setdefaults()` function sets default

values for these arguments.

2. **Neural Network Architecture (DQN):** Then DQN class is defined as a subclass of `nn.Module`, that represents the neural network architecture for the Q-network. After that the constructor `__init__` initializes the network architecture based on the provided parameters (`input_dim`, `output_dim`, `neurons`, `architecture`). It sets up different network architectures based on the value of the `architecture` parameter (1 or 2). The `forward` method defines the forward pass of the network based on the selected architecture value.

3. **Experience Replay:** After that the `experience_replay` class is defined that manages the experience replay buffer. It initializes a deque buffer with a maximum capacity specified by the `capacity` parameter. It also provides methods for adding experiences (`push`) and sampling mini-batches (`sample`) from the buffer.

4. **DQNAgent Class:** The `DQNAgent` class initializes the DQN agent and manages its training process.It sets up the policy network (`policy_net`) and optionally a target network(`target_net`). It also provides (`select_action`) method which is responsible for choosing an action for the agent based on the current state and the specified policy. (`evaluate`) method evaluates the DQN agent by running it in the environment for a specified number of episodes and computing the average return. It also ensures proper handling of the network mode (training vs. evaluation) during the evaluation process. (`direct_update` method performs a direct update of the Q-network with Experience Replay based on the observed transition using the Bellman equation. The update can be performed with or without using the target network, depending on the value of the `experiment_target_network` flag. (`optimize_model`) method computes the Q-values for the current and next states to optimize the Q-network and computes the expected Q-values using the Bellman equation. It calculates the loss and updates the model parameters through backpropagation.

5. **Main Function (DQN_Main):** The `DQN_Main` function coordinates the training process of the DQN agent. It sets up the environment, initializes the DQN agent and iterates over a specified number of episodes. It interacts with the environment within each episode loop. During which it collects experiences, updates the Q-network and periodically evaluates the agent's performance.

6. **Evaluation:** During evaluation, the agent's policy is evaluated by running the policy in the environment for a certain number of episodes. The average return over these evaluation episodes is calculated and returned.

3.2. Exploration Strategies

To balance exploration and exploitation we have used different exploration strategies such as:

3.2.1. ϵ-GREEDY

The ϵ-Greedy policy randomly selects an action with probability ϵ (epsilon) and exploit the best-known action that has the highest expected reward with probability 1-ϵ as shown in the equation (1).

$$\pi(a|s) = \begin{cases} 1.0 - \epsilon \cdot \frac{|A|-1}{|A|}, & \text{if } a = \text{argmax}_{b \in A} \hat{Q}(s,b) \\ \epsilon/|A|, & \text{otherwise} \end{cases} \tag{1}$$

- $\pi(a|s)$ represents the probability of choosing action a while in state s.
- ε is the exploration rate, where $\varepsilon = 1$ indicates a uniformly random policy, while $\varepsilon = 0$ indicates a greedy policy.
- $|A|$ is the number of actions that can occur in state s.
- $\hat{Q}(s,a)$ represents the estimated value of performing action a in state s.

3.2.2. BOLTZMANN POLICY

The Boltzmann policy (softmax policy) assigns probabilities to each action based on their Q-values. It uses a temperature parameter τ (tau) to control the exploration rate. Higher temperatures lead to more exploration as shown in equation (2).

$$\pi(a|s) = \frac{e^{\hat{Q}(s,b)/\tau}}{\sum_{b \in A} e^{\hat{Q}(s,b)/\tau}} \tag{2}$$

- τ is the temperature parameter controlling the degree of exploration.
- A is the set of all actions that can occur in state s.

3.3. Experience Replay

Experience replay is an important technique which is used in Deep Q-Learning (DQN) that enhances the training stability and efficiency from past interactions. It stores the observed experiences which are represented as state-action-reward-next state transitions into a memory buffer during interaction

with the environment. These transitions are then randomly sampled during the training phase. This random sampling helps in breaking the temporal correlations between consecutive experiences thus promoting a more stable learning process. Experience replay makes sure that the agent learns from a wide range of past events and keeps it from getting stuck in states that aren't ideal.

3.4. Target Network

The target network is used to stabilize the training process and improve convergence. In Q-learning algorithms, the same set of parameters is used to both estimate the target Q-values and update the Q-network. This leads to instability during training because the target Q-values keeps changing as the Q-network parameters are updated. To deal with this issue DQN introduces a separate target in which a separate copy of the Q-network that is periodically updated to match the parameters of the primary Q-network. The target network provides more consistent and smoother training process and facilitates convergence towards optimal Q-values by delaying updates and reducing volatility in target Q-values.

4. Results

4.1. Tuning Hyper-parameters

4.1.1. Architecture Analysis

In this section of hyper-parameter tuning, we aim to find the best network architecture possible. We are going to look into several hidden layers and several neurons to perform this experiment. We selected two values for the hidden layer [1, 2] and three values for the number of neurons [32, 64, 128].

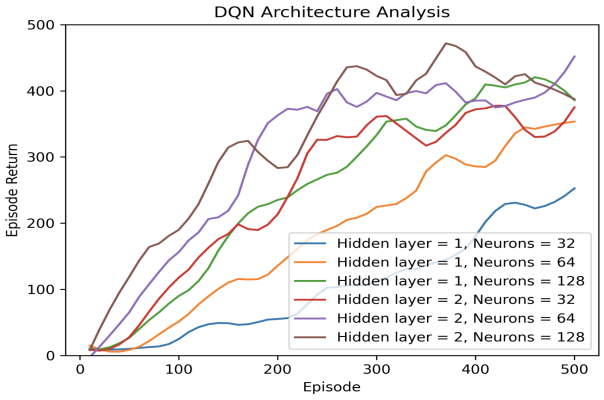

Figure 2. DQN performance at different number of hidden layers and neurons.The plot is generated with 10 repetitions. The total number of episodes is 500 where the policy used is ϵ-greedy at ϵ = 0.01 and the learning rate (α) is 0.001. Smoothing window is at 9.

Based on Figure 2, we can see that DQN works best in this environment when there are two hidden layers, each with 128 neurons. It's quick to learn, and the results are better than other values of neurons and hidden layer. Therefore, The architecture of DQN selected for ablation study is two layered having 128 neurons each. We also have a input layer of four neurons (cart position, cart velocity, pole angle, pole velocity) and a output layer of two neurons (push left or right). We do not need Convolutional layers because Cartpole is a simple environment that does not provide pictorial information as returns.

4.1.2. Exploration Policy

In this section of hyper-parameter tuning, our aim is to find the best value of ϵ for ϵ-greedy policy and best value of temperature τ of Softmax policy. The value at which we will achieve best performance will be selected for the Ablation Study. We checked the DQN performance for three different values of Epsilon (ϵ) = [0.001, 0.1, 0.3] and Temperature (τ) = [0.01, 0.1, 1.0].

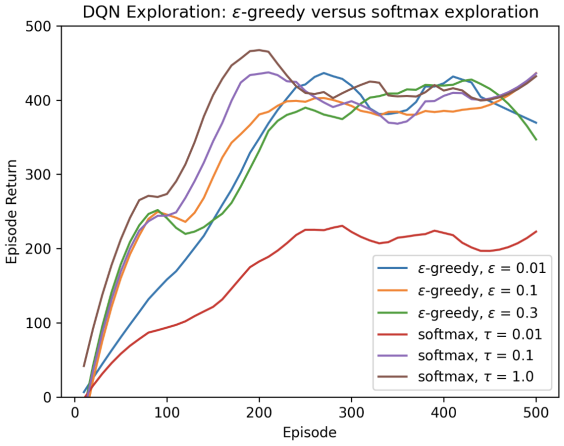

Figure 3. DQN performance with different exploration policies. The plot is generated with 10 repetitions. The total number of episodes is 500 where the discount factor (γ) is 0.99 and the learning rate (α) is 0.001. Smoothing window is at 9.

From Figure 3, we can observe that the Softmax policy with temperature $(\tau) = 1.0$ performs best. DQN can learn quickly with good episode returns. In terms of ϵ-greedy policy, $\epsilon = 0.01$ performs best. It takes some time to learn but learning is stable compared to other values of ϵ. Even the episodic returns are also a bit better.

We also decided to see if linear annealing would yield better results. We let both the ϵ and τ go from 1.0 to 0.1 with steps of $5 * 10^{-4}$

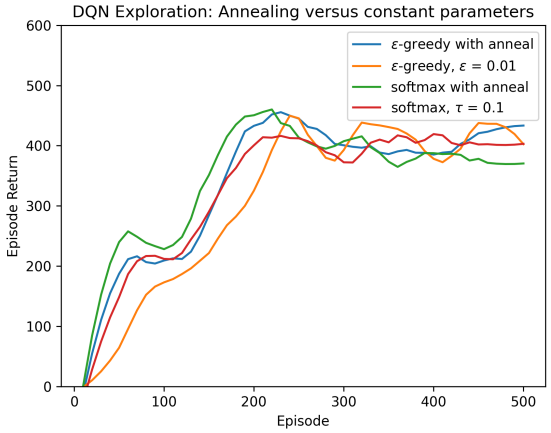

Figure 4. DQN with linear annealing versus constant parameters

In Figure 4 we can see that when using linear annealing, the agents found a better return a little sooner than when using the constant parameters, this is because the agent has more room to explore with a high ϵ and τ. However after around 250 episodes they all performed similarly

4.1.3. Discount Factor (γ)

Our goal in this part of hyper-parameter tuning is to find the best value for the discount factor (γ). For the Ablation Study, the value at which DQN performs the best will be selected. We tested how well the DQN worked with four different values of Gamma (γ): [1.0, 0.99, 0.95, 0.9]

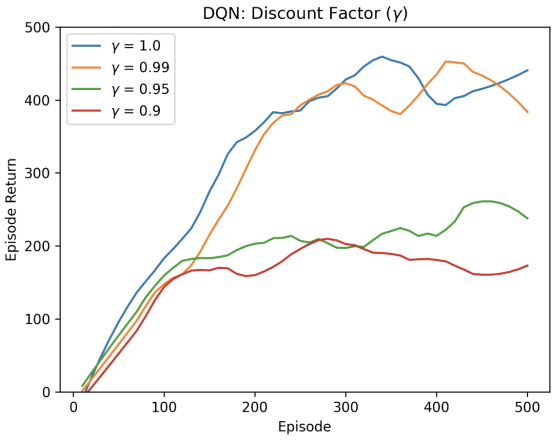

Figure 5. DQN performance at different discount factor values.The plot is generated with 10 repetitions. The total number of episodes is 500 where the policy used is ϵ-greedy at $\epsilon = 0.01$ and the learning rate (α) is 0.001. Smoothing window is at 9.

From Figure 5, we can observe that the (γ) value at 1.0 and 0.99 performs best and provides better episodic returns for DQN implementation. DQN at (γ) = 0.99 takes time to learn but its performance is more stable. Therefore, the discount factor (γ) selected is 0.99.

4.1.4. Learning Rate (α)

To find the best value for the Learning Rate (α), we need to tune the hyperparameters one more time. The value at which DQN works best will be chosen for the Ablation Study. We tried the DQN with four different Learning Rate (α) [0.1, 0.01, 0.001, 0.0001] to see how well it performed.

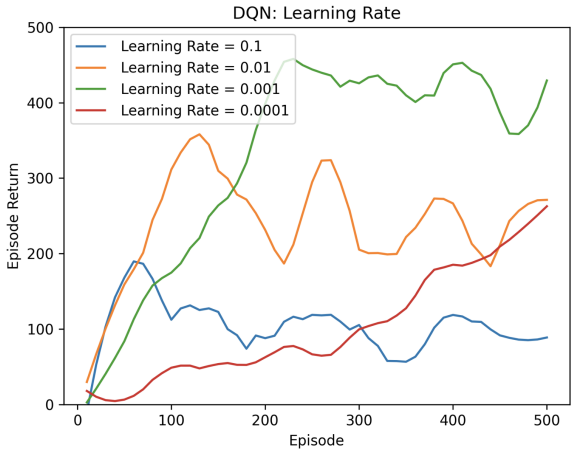

Figure 6. DQN performance at different learning rates.The plot is generated with 10 repetitions. The total number of episodes is 500 where the policy used is ϵ-greedy at $\epsilon = 0.01$ and (γ) selected is 0.99. Smoothing window is at 9.

We can see from Figure 6 that the (α) value at 0.001 works best and gives better episodic returns for DQN implementation. The performance is quick and stable. There is a second-best learning rate of 0.01, but it can't reach the episodic returns of 0.001.

4.1.5. Batch Size

Our goal in this section is to find the best batch size which will provide the best performance of DQN in this Cartpole environment. We checked the performance of DQN on for values of batch size [16, 32, 64, 128].

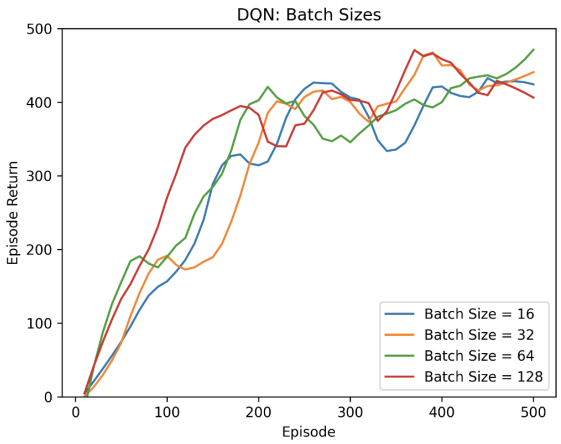

Figure 7. DQN performance at different batch sizes.The plot is generated with 10 repetitions. The total number of episodes is 500 where the policy used is ϵ-greedy at $\epsilon = 0.01$ and (γ) selected is 0.99. Smoothing window is at 9.

Figure 7 shows that the results were the same for all batch sizes that were tested. Performance of DQN at batch size = 64 appears to be stable and quick to learn. Therefore we will select this value for the Ablation Study.

4.1.6. Target Network update

In this section, we experimented with updating the target network at different episodes. We aim to find the best way to update a Target Network which will provide the best results for our DQN.

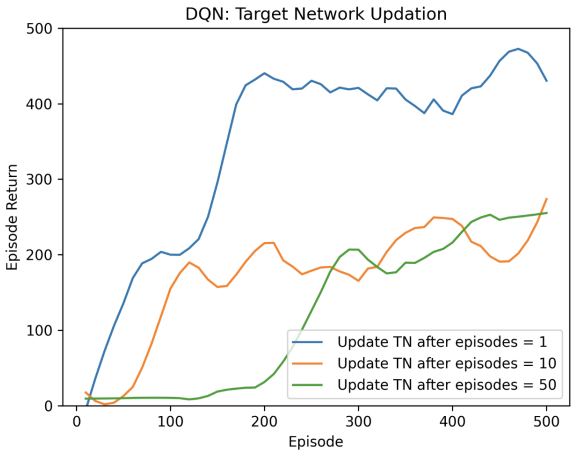

Figure 8. DQN performance when we update Target network at different episodes. The plot is generated with 10 repetitions. The total number of episodes is 500 where the policy used is ϵ-greedy at $\epsilon = 0.01$ and (γ) selected is 0.99. Smoothing window is at 9.

From Figure 8, we can observe updating the target network at every episode yields the best results in our implementation of DQN. Therefore, we will select TN update to 1 for our Ablation study.

5. Observation

The best-performing hyperparameters for the Deep Q-Network Ablation Study after conducting experiments are shown in Table 2.

Parameter	Value
Learning Rate	0.001
Discount Factor	0.99
N-step	64
Evaluation	At every 5000 timestep

Table 2. Optimal hyperparameters for DQN Ablation Study.

6. Ablation Study

With using all the best hyper-parameters, we performed an ablation study of DQN. It's important to note that (-) means that we are not using that component in our DQN for the CartPole environment. Here, we performed the study on four variations of DQN. They are DQN with both experience replay(ER) and a target network(TN), DQN without experience replay(ER), DQN without target network(TN) and DQN without both experience replay(ER) and a target network(TN).

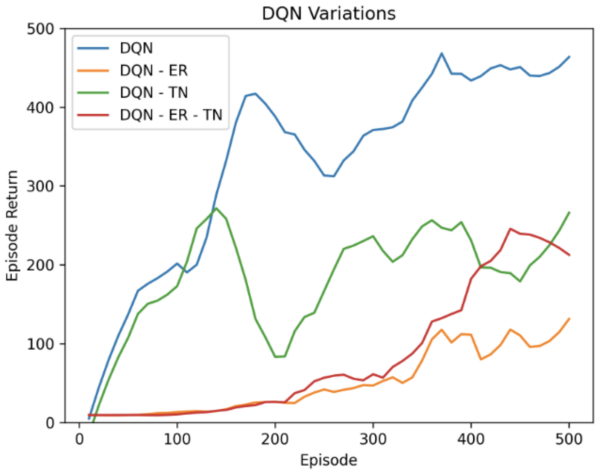

Figure 9. Ablation Study: All the hyper parameters mentioned in Observation is used.

In Figure 9, we can see the episode returns for the agent while it uses experience replay(ER) and a target network(TN). As was expected, using both got the most returns. DQN with ER and TN (blue line) was able to perform best. It is quick to learn, stable and performance is also great. When we did not use Target Network (Green line), the result

became unsteady and the learning was unstable. We believe that DQN needs a target network to maximise its episodic returns. DQN is still able to learn but naturally, it's not as good as DQN with both TN and ER. The stability, performance and speed are learning dropped.
DQN without TN and ER (red line), learns better than DQN without ER (orange line). This suggests that removing Experience Replay has a greater negative impact on DQN than not using TN and ER. We believe that without ER, the agent is more susceptible to catastrophic forgetting or overfitting to recent experiences. Both DQN without TN and ER and DQN without ER perform poorly but we believe that learning will happen with more episodes.

7. Bonus

7.1. Acrobot environment

For extra experimentation we looked at how this setup would work when it is used for a different problem: the Acrobot environment (OpenAI, 2016–a). Which is also by OpenAI.

Num	Observation	Min	Max
0	$\cos(\theta_1)$	-1	1
1	$\sin(\theta_1)$	-1	1
2	$\cos(\theta_2)$	-1	1
3	$\sin(\theta_2)$	-1	1
4	Angular velocity θ_1	-4π	4π
5	Angular velocity θ_2	-9π	9π

Table 3. State space for the Acrobot environment showing the observation space along with the minimum and maximum values for each variable.

7.2. Results

We ran some experiments on this environment to see how our DQN would work. As can be seen in Figure 10 having a learning rate that is too high will result in a very bad episode return. It looks like having a learning rate of 0.001 or 0.0001 works best for this agent.

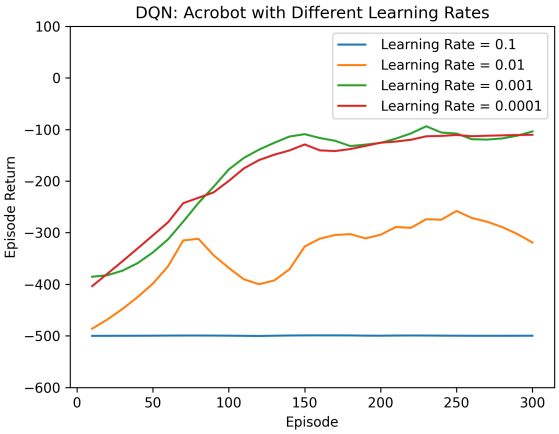

Figure 10. Different learning rates for the Acrobot environment

We also looked at how the performance was compared to the cartpole agent to see which would work better. Both make use of experience replay and a target network as discussed in sections 3.3 and 3.4 respectively. Since the rewards are negative, and the minimum is -500 we decided to add 500 to the rewards per episode in order to adequately compare the agents in the different environments. As one can see in Figure 11 the agent in the acrobot environment was faster to get constant good results.

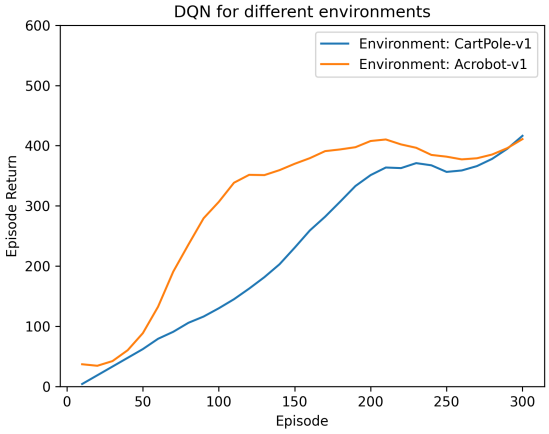

Figure 11. Acrobot environment versus cartpole environment

8. Conclusion

To conclude, we have looked and experimented with many different variations of the deep Q network to get the optimal values for its parameters. All the best values are discussed

in section 4. Even though the speed of the algorithm was low we have gotten positive results and have gotten a better understanding of the impact of certain parameters. In future work, more exploration strategies can be experimented with or the problem could be more difficult in order to mimic real world problems.

References

OpenAI. OpenAI Gym. https://www.gymlibrary.dev/environments/classic_control/acrobot/, 2016–a.

OpenAI. OpenAI Gym. https://www.gymlibrary.dev/environments/classic_control/cart_pole/, 2016–b.

Paszke, A. e. a. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.