Deep Q Network (DQN) Algorithm for Reinforcement Learning in a "Function Approximation" Setting

Pallavi Ravada*
University of Illinois Urbana-Champaign, Urbana-Champaign, Illinois, 61820

I. Introduction

This report details the implementation of the DQN algorithm for reinforcement learning in a "function approximation" setting. As compared with Q-learning from the previous assignment, DQN can operate on a continuous state space, however it still requires a finite action space. A Brief algorithm description, implementation notes, and results will be given in the following sections. For detailed algorithm pseudo-code please refer to [1].

II. Algorithm Description and Implementation

A. Environment

The algorithm is tested in the following provided environment:

- A simple pendulum with continuous state space and discretized action space, for which an explicit model is not available ("`http://underactuated.mit.edu/pend.html`")

For this environment the reinforcement learning problem is expressed as a Markov Decision Process with an infinite time horizon and a discount factor of $\gamma = 0.95$.

B. DQN

Q-learning was a suitable approach for models dealing with simple environments in which the Q function could be represented as a tabular format. However when the state space dramatically increases, coupled with an increase in the available actions, the tabular method is no longer sufficient. This is where DQN comes into play, combining the Q-learning algorithm with a deep neural network (DNN) to represent the Q-function.

In reality the implementation of DQN utilizes two DNNs in conjunction for learning stabilization. The main neural network, represented by weight vector θ, is responsible for estimating Q-values for the current state s and action a. The target neural network, with the weight vector θ', is structurally identical to the main network but is tasked to estimate Q-values for the next state s' and action a'. Learning only takes place in the main network, the weights being updated based on the observed data. The target network remains "frozen" during the learning process for a designated number of iterations to mitigate bias in the target network. After the iteration count has been reached, the weights from the main network are copied into the target network. Transferring the learned knowledge from the main network to the target network allows the Q-value estimations from the target network to be more accurate.

The coding implementation of DQN was designed to be modular. In the DQN.py file three classes were created:

1) DeepQNet to initialize the deep neural network for DQN
2) ReplayBuffer for storing the learned experience
3) Agent the DQN agent for interacting with the environment

The class DeepQNet is composed of an initialization fucntion which creates the layers for the DNN and a forward function which maps the state to action values. As per the homework instructions a neural network with two hidden layers was used that each had 64 units. Additionally a tanh activation function was used at both the hidden layers, with a linear activation function at the output layer.

The class ReplayBuffer is composed of three functions in addition to the initialization function which sets the memory capacity. The add function adds a new memory, the sample function samples a random existing memory, and the length function returns the current length of the memory.

The class Agent is where the DQN algorithm interacts with the pendulum environment. In the initialization function the main and target neural nets are initialized as well as the memory replay buffer. There are three more functions, the first being a function that chooses an action for the agent to take. The next function optimizes the DNN through gradient

*Graduate Student, Department of Aerospace Engineering.

descent methods. The gradients are clipped and during backward propagation of the network and the clipped gradients are used to update the weights. The loss function criterion is determined by SmoothL1Loss. The final function trains the DNN on the pendulum environment using the Q-learning algorithm.

To generate all of the required plots the DQN agent can be initialized and then trained on the given environment. To get a better sense of the data results for 5 training runs were completed and their results averaged for the learning curve and ablation study plots.

The hyperparamters chosen are as follows informed by "Extended Data Table 1 List of hyperparameters and their values" from [1]: learning rate $\alpha = 1 \cdot 10^{-4}$, mini batch size = 128, number of episodes = 1000, and reset number = 100. These numbers were chosen as more training was required to determine a good policy for the pendulum.

III. Results

The results of the algorithms for the gridworld and pendulum environments are given below. A gif of the pendulum can be found in the github repository. The pendulum begins its upswing until it reaches the "maximum" anglular position. Once there agent atempts to maintain that position for the 10 second trajectory duration. As can be seen from the gif the agent is fairly successful at maintaining this position with the exception of some slight deviations from the desired position.

A. Learning Curve

Figure 1 shows the learning curve for the agent. As the episodes progress the return increases then remains relatively constant for the remainder of the episodes as the agent learns the proper steps to maintain the desired pendulum position. The blue curve shows the undiscounted return while the red curve shows the discounted return. The results over 5 runs with their standard deviations were averaged and plotted to get a fuller picture of the data trends.

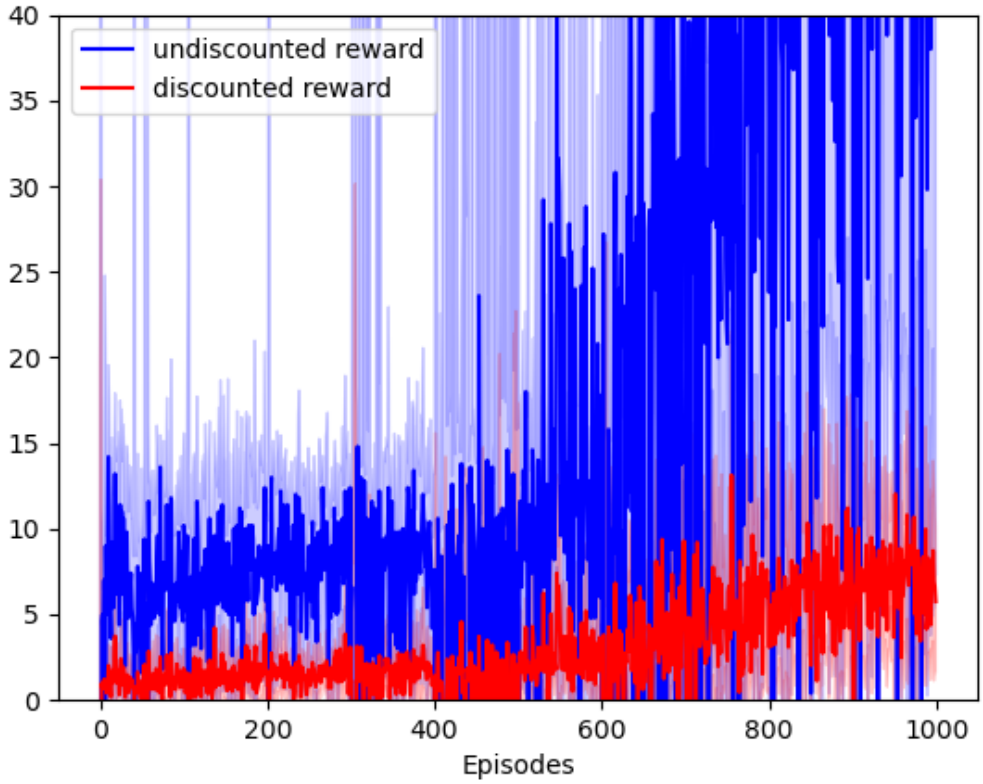

Fig. 1 Learning Curve

B. Example Trajectory of a Trained Agent

Figure 2 shows the trajectory of the trained agent, which shows a periodic trajectory. The tau value is within the max allowable limits for the duration of the trajectory ($-5 \leq \tau \leq 5$). The reward also jumps to 1 at two points along the trajectory.

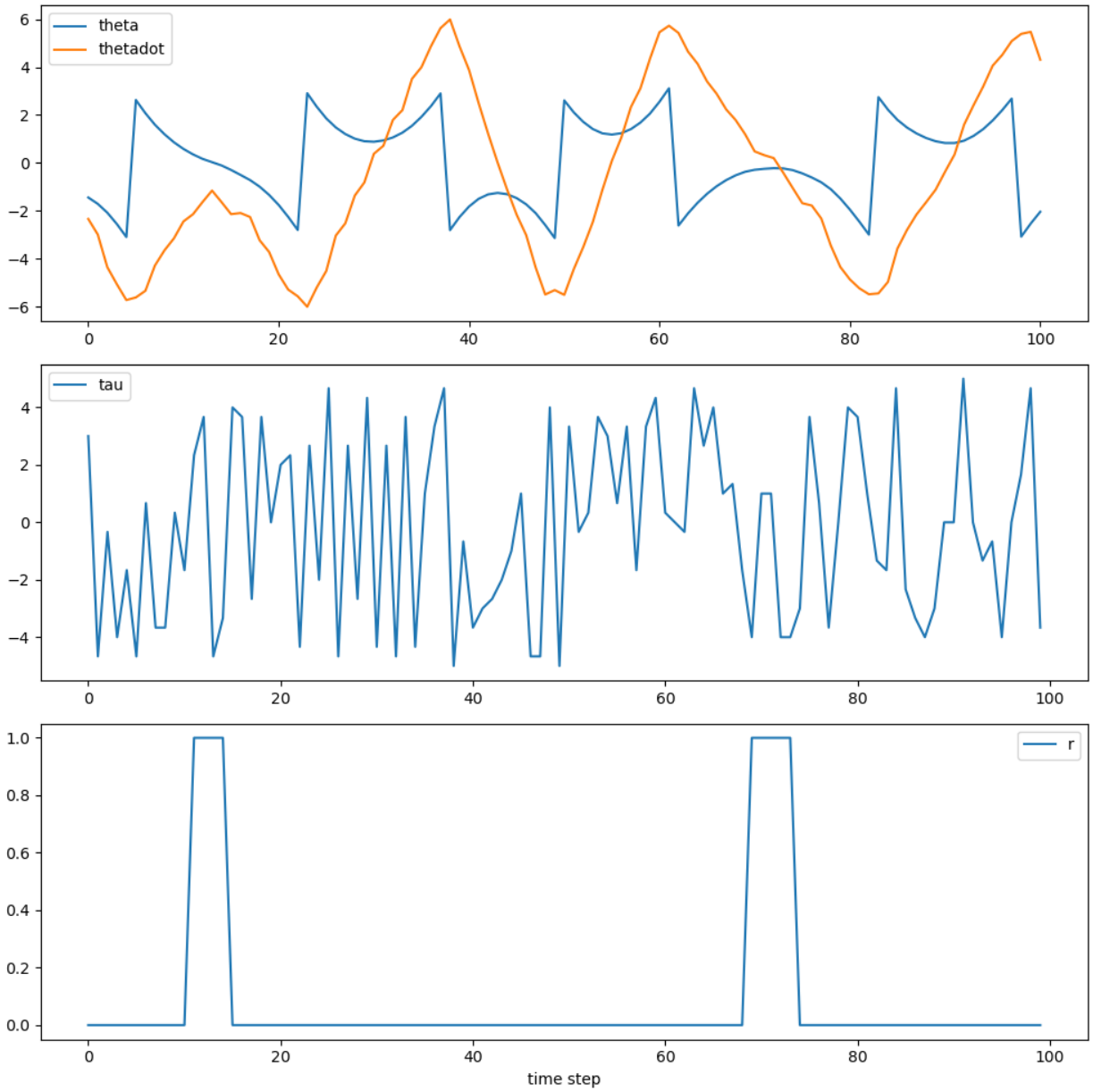

Fig. 2 Example Trajectory of a Trained Agent

C. Policy & State-Value Function

Figure 3 shows the policy for the pendulum model. The axes on the plot represent the theta and thetadot values respectively, and the corresponding plot color indicates the tau value to be employed to maintain the pendulum position. Figure 4 shows the state value function for the pendulum model. The axes are the same as the policy graph but the colorbar now represents the state-value function. The oscillatory behavior indicative of a pendulum can be seen in the color gradient of the plot.

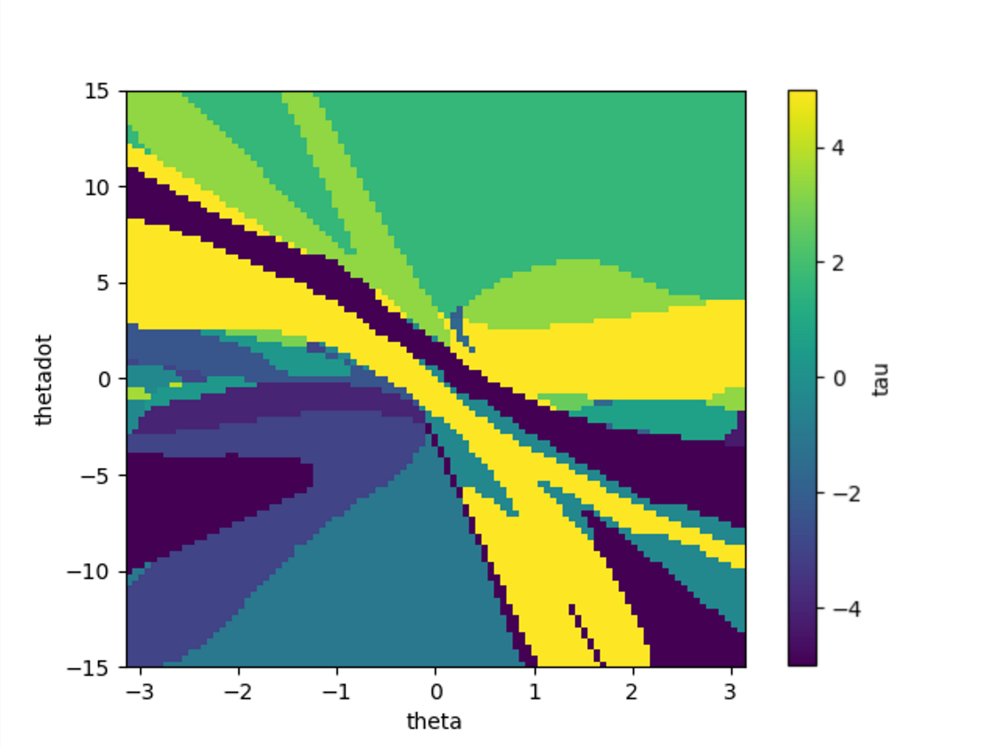

Fig. 3 Policy of a Trained Agent

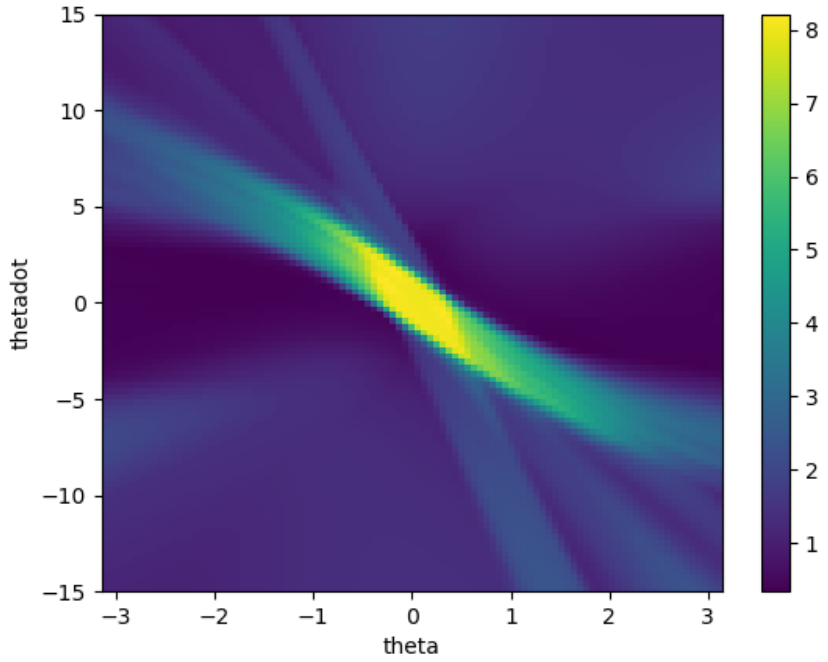

Fig. 4 State-Value Function of a Trained Agent

D. Ablation Study

Figure 5 is an ablation study to document the impact of using replay and target network on the performance of the algorithm. The study was done for the following four cases:

1) With Replay & With Target Q
2) With Replay & Without Target Q
3) Without Replay & With Target Q
4) Without Replay & Without Target Q

In terms of case (1), the base DQN algorithm, the learning curve shows growth and that the agent is able to learn the policy to maintain the pendulum position. A similar pattern can be seen in case (3) where the return increases then levels out, indicating the agent was able to learn how to keep the pendulum position upright. This shows that even without the ability to randomly sample all of the agents previous observations, the agent, in this case for this environment, is still able to form reasonably well. This is not the case for the other two ablation study cases.

In terms of case (2), with replay but without the target Q network the return is about constant through the whole trajectory. This indicates the agent is unable to learn policy to maintain pendulum position and pendulum continues to swing around in circles. The same pattern can be seen for case (4), where neither the replay buffer nor the target Q network are used. As expected this situation has the poorest performance, yielding an policy that causes the pendulum to circularly swing for the duration of the trajectory.

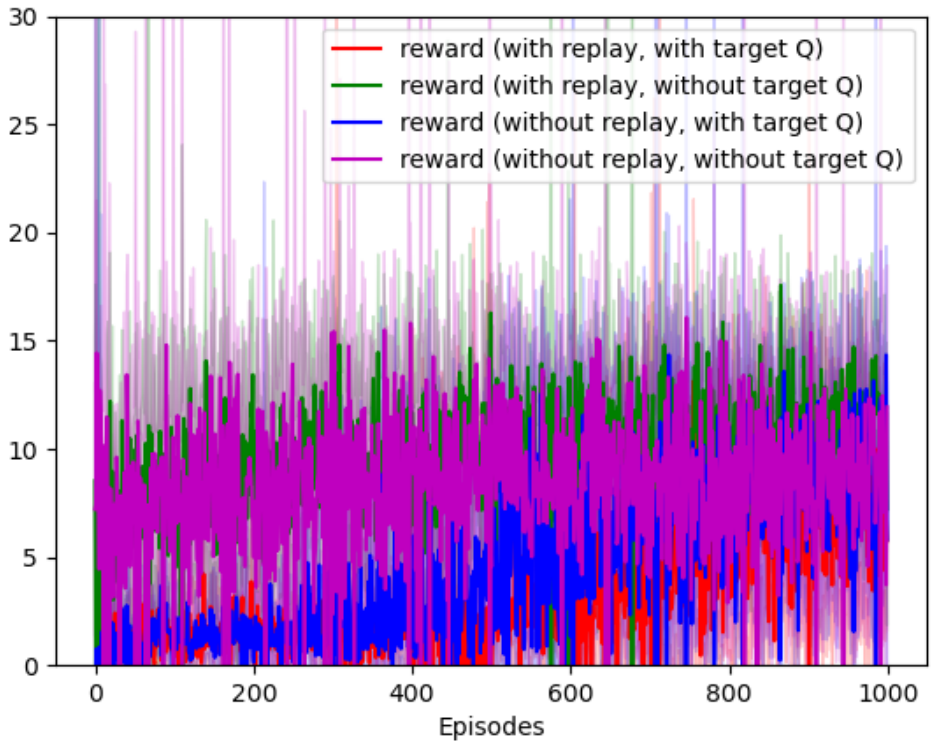

Fig. 5 Ablation Study

IV. Conclusion

Overall DQNs are more versatile than Q-learning, able to handle a continuous action space with an increased number of corresponding actions. They are very efficient at determining a good policy with intuitive hyperparameter tuning. However, there is always room for improvement. With prioritized experience replay or double Q networks the performance of DQN can be augmented.

References

[1] V. Mnih, D. S. A. A. R. J. V. M. G. B. A. G. M. R. A. K. F. G. O. S. P. C. B. A. S. I. A. H. K. D. K. D. W. S. L., K. Kavukcuoglu, and Hassabis, D., "Human-level control through deep reinforcement learning," *Nature*, 2015, p. 529–533. https://doi.org/10.1038/nature14236.