DQN PERFORMANCE WITH EPSILON GREEDY POLICIES AND EXPERIENCE REPLAY BUFFERS

DANIEL PERKINS, LUKE GREEN, OSCAR J. ESCOBAR

Abstract. The application of deep learning in reinforcement learning has produced big breakthroughs in this field of ML. We explore how DQN improves Q-learning as well as see how two components of DQN impact its performace. Our results show that DQN can produce significantly better results than Q-learning when the components are adapted properly to the RL problem.

1. Background & Motivation

Reinforcement learning (RL) is found at the intersection between control theory and machine learning. It was first introduced in the 1950s and is a sequential decision-making process in which an agent learns to perform actions that maximize a specified reward ([Pri23]). The RL framework can be summarized by Fig. 1.

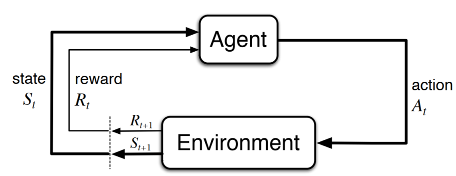


Figure 1. A representation of the RL framework as first introduced by [SB18]. The agent finding itself in state S_t and having received reward R_t takes action A_{t+1} and transitions into new state S_{t+1} and receives a reward R_{t+1}. The cycle then begins again.

We give definitions of other RL verbiage, such as episode, state, time step, etc., in Appendix B. The following definitions are important for this paper:

- The return is the discounted sum of future rewards. We denote it as $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ where $\gamma \in (0, 1]$ is a discount factor that makes rewards closer in time more valuable than those farther away.
- The policy is the rule that determines the agent's action at each state. This function may be either deterministic or stochastic. We denote it as $\pi[a|s]$, which is a probability distribution of actions a

Date: 4 October 2024.

given the state s[1]. Note that it does give an ordering on which actions to take. It only gives a probability for taking an action at a specific state.

- The action-value function is the expected future return for a given state-action pair and policy. We denote it as $q_\pi(s, a) = \mathbb{E}[G_t | s_t, \pi]$. It is a way of representing how good the action that the agent takes in a given state is, after considering expected future rewards.

Unlike conventional supervised and unsupervised methods, vanilla RL models do not learn from a given dataset and, consequently, have no normal loss function. Instead, they are control algorithms that interact with an environment to complete some task and use the reward as a signal for evaluating performance. The optimization problem of RL is to find an optimal policy that will bring the agent the most future return. Reinforcement learning has the potential to surpass human performance by evaluating and computing a far greater number of future state-action pairs than humans are capable of processing.

1.1. **Q-Learning.** Q-Learning ([WD92]) is an algorithm that learns the optimal policy by calculating state-action values from the direct experiences of the agent's interactions with the environment. That is, Q-learning computes the value of a state-action pair as the agent takes live actions and receives rewards, so that it learns by trial-and-error. In order for Q-learning to converge, there is a need for every possible action and state to be visited infinitely many times [SJLS00]. This is of course infeasible to calculate for every possible state-action pair in a problem with high dimensional state and action space.

Specifically, Q-Learning estimates the action-value function by using the reward and the difference in estimated values it has of the current state and an updated estimate. The Q-values are initially set to zero and stored in a matrix. Then, they are iteratively updated with the Bellman equation:

$$Q_{\text{new}}(s_t, a_t) = Q_{\text{old}}(s_t, a_t) + \alpha \Big[r_t + \gamma \max_{a \in A_{s_{t+1}}} Q_{\text{old}}(s_{t+1}, a) - Q_{\text{old}}(s_t, a_t) \Big] \tag{1}$$

where α is the learning rate and $r_t + \gamma \max_{a \in A_{s_{t+1}}} Q_{\text{old}}(s_{t+1}, a)$ is the updated approximation of the current state-action pair.

By learning the value function for each action-value pair, Q-learning is guaranteed to find the optimal policy. It can simply choose the action that maximizes the value at each state the agent finds itself in. When the action space and state space are finite and small, a simple bottom-up dynamic programming implementation will work well ([Wat89]). However, these algorithms are computationally expensive because of their exhaustive search. Therefore, as previously stated, they do not perform well on most real-world tasks which often have large or even infinite action and/or state spaces.

[1] Note that each state s can have a different probability distribution over its own A_s.

1.2. **RL Challenges.** While RL can be quite a good tool for handling sequential decision making, there are a few hurdles that it must overcome.

1.2.1. *Exploration-Exploitation-Trade off.* Since Q-learning must learn an optimal policy strictly from experience, the agent has to explore action space to find what actions are good and which are bad. However, we also need the agent to exploit what it has already learned so that the algorithm converges to an optimal policy. Thus, there is a need to properly balance exploration and exploitation if we want the agent to achieve a goal.

1.2.2. *Credit Assignment Problem.* While a reward is a good signal that the agent can use in the immediate sense for judging how good a taken action was in helping it achieve the long term goal, the agent has difficulty determining which actions or sequence of actions actually led to the reward. This is known as the *credit assignment problem* (CAP).

In order for the agent to accomplish the goal (i.e. solving the environment), it must be able to determine which actions actually contribute to solving the environment and which don't. Moreover, due to the sequential nature of RL and indirectly from CAP, how can the agent learn to single out good actions needed to accomplish a task seeing that those come from the sequential decision making? Lin also mentioned that "if an input pattern has not been presented for quite a while, the [agent] typically will forget what it has learned for that pattern and thus need to re-learn it when that pattern is seen again later" [LJ94]. Thus, for RL problems with bigger state and action spaces, there appears to be a necessity to use previous actions to "remember" what has been accomplished as well as to disassociate actions.

1.3. **Deep Reinforcement Learning.** Until recently, Temporal Difference Learning methods like Q-Learning ([WD92]) and SARSA ([RN94]) were the state of the art in reinforcement learning. Problems with large state spaces were thought to be unsolvable. However, with the advent of deep learning, RL has hit a new frontier. In 2013, fitted Q-learning was introduced ([MKS+13]). Instead of storing all possible values in a large matrix, a neural network is used to estimate the action-value function; the Q-values $Q(s_t, a_t)$ are replaced with a neural network $q[s_t, a_t, \phi]$. Since $q[s_t, a_t, \phi]$ should be close to $r_t + \gamma \text{max}_{a\in A}\{q[s_{t+1}, a, \phi]\}$, we get the loss function

$$L(\phi) = (r_t + \gamma \text{max}_{a\in A}\{q[s_{t+1}, a, \phi]\} - q[s_t, a_t, \phi])^2 \tag{2}$$

In theory, performing gradient descent on this function will yield a deep neural network that approximates the true state-value function. However, unlike typical supervised learning algorithms, at each step, the target, $r_t + \gamma \text{max}_{a\in A}\{q[s_{t+1}, a, \phi]\}$ is changing at each step. So, to reduce variance, it is common to only alter the target every hundred iterations or so. So, letting $\hat{q}[s_{t+1}, a, \phi]$ denote the most recently saved target, this loss function can be written as

$$L(\phi) = (r_t + \gamma \text{max}_{a\in A}\{\hat{q}[s_{t+1}, a, \phi]\} - q[s_t, a_t, \phi])^2 \tag{3}$$

This new framework has paved the way for solving much more complicated problems. For example because the game Go has a state space with over 10^{170} elements. It was once considered to be impossible to develop a RL learning algorithm that can beat the best humans. But, in 2016, a deep RL model beat the world champion ([SHM+16]).

1.4. **Assumptions & Objective.** Seeing that there are many stochastic pieces to work with, as given in the definitions, as well as big hurdles that plague RL, we have decided to work with a very simplistic model where our policies will be deterministic as well as our reward. Moreover, the environment will have a finite and deterministic nature. Furthermore, to help our analysis, we decide to focus on a problem that has a small and finite action and state space as well as having the property that the agent gets a full observation of the state. That is, we work under a Markov Decision Process not a Partially Observable Markov Decision Process. This will lessen the number of variables we must account for as well as reduce the stochastic nature of RL problems.

In particular, it is our interest to further investigate how DQN can help us achieve a higher reward than what the Q-learning algorithm can give. Furthermore, we seek to also learn how various *epsilon-greedy* policies affect the performance of DQN and if it makes it worse or better than Q-learning. Lastly, we explore the affect of *experience replay*, with decaying *epsilon-greedy* policies, on DQN and its performance compared to Q-learning.

2. Simulation Environment

In this study, we utilize the Cart Pole environment from Gymnasium ([JA24]) to test our deep reinforcement learning (DRL) algorithm. This environment is well-suited for investigating the mathematical underpinnings of Deep RL, as it features a small action space and relatively straightforward dynamics. The simplicity enables rapid training, which is essential for efficiently testing a wide range of hyperparameter configurations.

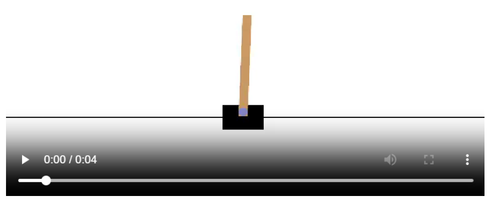


Figure 2. The Cart Pole Environment

The Cart Pole Environment presents a setting in which a pole is balanced on a moving cart (Figure 2). The goal is to prevent the pole from falling over by adjusting the cart's position. The environment consists of the following:

- The state is observed as a four-dimensional vector $s_t = \begin{bmatrix} x_t & v_t & \theta_t & \omega_t \end{bmatrix}$ where $x_t \in [-4.8, 4.8]$ and $v_t \in (-\infty, \infty)$ denote the cart's position and velocity, respectively, and $\theta_t \in (-0.418, 0.418)$ and $\omega_t \in (-\infty, \infty)$ represent the pole's angle and angular velocity.
- The initial state s_0 is an observation where each element of s_0 is sampled uniformly from $(-0.05, 0.05)$.
- The action space $A = \{0, 1\}$ is the same at each state. 0 denotes the action of moving to the left and 1 is the action of moving to the right.
- At each time step, the reward $r(s_t, a_t)$ is 0 if the pole falls and 1 otherwise.
- The simulation ends if the pole angle θ_t is greater than ± 0.2095, if the cart position x_t is greater than ± 2.4, or if the episode length t is greater than 500.

To solve this environment using normal Q-learning in (1), we had to discretize the state space by taking a certain number of values for each of the components of s_t. We employed a linear decay for the epsilon-greedy algorithm (explained in 3). Our results are shown Fig. 3.

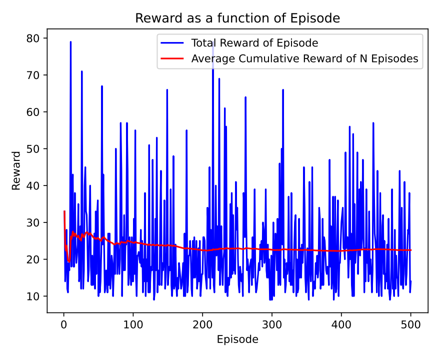


FIGURE 3. The rewards obtained using an exponential decay 0.9999^t for ε-greedy in normal Q-learning. Compare to Fig. 4.

As shown, the results are fairly good with normal Q-learning. The agent is able to accomplish the task of balancing the pole, and with more episodes, it performs better on average. Note that due to the stochastic nature of our policy the rewards are noisy as shown in the graph. Notice also how the average reward (red) somewhat increases at times but stays overall stagnant and even decreases at times.

3. Deep Q-Network: Epsilon Greedy Algorithm/Policy

To balance exploration and exploitation (Section 1.2.1), DRL commonly uses the *epsilon-greedy algorithm*, which specifies a value $\varepsilon \in [0, 1]$ that signifies the rate at which the agent must explore. By drawing some random value from $c \sim \text{Uniform}[0, 1]$, we can *explore* the action space when $c < \varepsilon$

or *exploit* by taking the best action when $c \geq \varepsilon$. This strategy enables the agent to explore alternative actions that could potentially yield higher rewards than the current policy.

Setting ε too high greatly increases the temporal complexity of the algorithm. But, setting it too low reduces the probability that it finds the optimal policy. It is common to let ε decay overtime so that the model learns more in the beginning and exploits what it learns later on. In our paper, we investigate the effects of various schedules for ε.

3.1. **Theory.** In the work of Signh et al, they gave proofs for guaranteed convergence of an algorithm similar to Q-learning, called *SARSA(0)*[2], employing different *behavioral/learning* policies (see B). They then discussed how these concepts apply to Q-learning. In particular, they stated that when using decaying behavioral policies, actions may converge to the optimal actions but at the cost of losing ability to adapt as the value decays. Moreover, they identified two crucial properties that decaying policies should have: 1) "each action is visited infinitely often in every state that is visited infinitely often", and 2) "in the limit, the learning policy is greedy with respect to the Q-value function with probability 1"[SJLS00].

Signh et al. represented this mathematically by letting $t_s(i)$ denote the time step at which the i^{th} visit to state s occurs and considering some action a. To visit ever state infinitely often, they showed that probability of executing action a must satisfy $\sum_{i=1}^{\infty} \mathbb{P}(a = a_t | s_t = s, t_s(i) = t) = \infty$[SJLS00]. This of course is affected by ε-greedy since it takes actions by comparing a randomly sampled value $c \sim \text{Unif}[0, 1]$ to the current value ε. Thus, careful consideration must be given to how we decay ϵ to obtain optimal results.

3.2. **Exponential Decay.** One of the most common schedules for ϵ is exponential decay. At each iteration t, we set $\epsilon = \beta^t$ for some $\beta \in [0, 1]$. This simple formula gradually reduces the exploration rate as the agent gains more experience, allowing it to increasingly exploit the learned policy while still exploring in the early stages. By adjusting β, we can control the rate at which exploration diminishes, balancing the trade-off between discovering new strategies and refining the current policy.

[2]SARSA(0) is given by $Q_{\text{new}(s_t, a_t)} = Q_{\text{old}}(s_t, a_t) + \alpha\Big[r_t + \gamma Q_{\text{old}}(s_{t+1}, a_{t+1}) - Q_{\text{old}}(s_t, a_t)\Big]$. It uses the tuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, hence the name. Compare to (1).

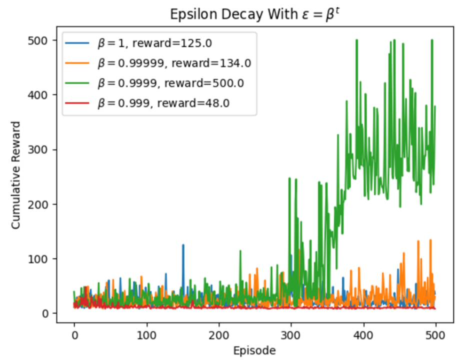


FIGURE 4. Cumulative reward with exponential epsilon decay for various values of β

Figure 4 shows the empirical results for the deep Q-learning model with an exponential decay schedule for the epsilon-greedy algorithm. When β was too large, the agent selected random actions too frequently, resulting in lower rewards. Conversely, when β was too small, the model was able to learn effectively only in the early iterations, ultimately getting stuck in a suboptimal solution. We found that for the Cart Pole problem, the optimal value of β was 0.9999, as it produced the highest cumulative reward.

3.3. **Other Decaying Epsilon Schedules.** We show results of our investigations for various other epsilon decaying schedules in Figure 7 in Appendix A. The linear (7a) and logarithmic (7b) schedules did not obtain rewards as high as the inverse (7c) and sinusoidal decay (7d) schedules. This suggests that the DRL is more likely to get better results if the decay of ϵ is rapid (super-linear). The inverse decay schedule has the highest average reward empirically. However, this does not necessarily mean that it is the optimal schedule. All of the schedules we tested were noisy, differing substantially in each run.

Furthermore, each schedule demands considerable hyperparameter tuning to determine the ideal rate of decay. Consequently, we conclude that the optimal selection of the ϵ-decay algorithm and its associated hyperparameters is highly task-dependent.

4. Deep Q-Network: Experience Replay

As mentioned in Section 1.2.2, two big hurdles in RL are disassociating actions from their sequential nature and then remembering what is already known. In his work, Lin created what he called *experience replay*. This entailed the use of a replay buffer, $\mathcal{D}$ of some fixed size D, that stores the tuple $(s_t, a_t, r_{t+1}, s_{t+1})$, labeled as experience e_t, as the agent steps through various time-steps t. The aim of this was to give a concise summary of the experience or lesson the agent went through at any time t.

In the simple implementation, the DQN algorithm uniformly at random samples some batch $E = \{e_i\}_{i=1}^N$ comprised experiences and then uses the behavioral/policy network with $(s_i, a_i) \in e_i$ in order to compute the current q-value $q(s_i, a_i)$ for each $e_i \in E$. We then compute the optimal/target q-value for (s_i, a_i) with $q^*(s_i, a_i) = r_{i+1} + \gamma \max_{a_{i+1} \in A_{s_{i+1}}} q(s_{i+1}, a_{i+1})$ by using the *target* network (see B). We then subtract to get the loss and apply gradient descent to minimize the loss in order to converge to the optimal q-value.

Lin's use of experience replay in this fashion allowed the agent to "remember" previous actions and, most importantly, learn how to break the correlation among actions due to the sequential nature from which it experiences them. Moreover, as given by Fedus et. al, the stability of the algorithm improved as well as converging faster than normal Q-learning [FRA$^+$20].

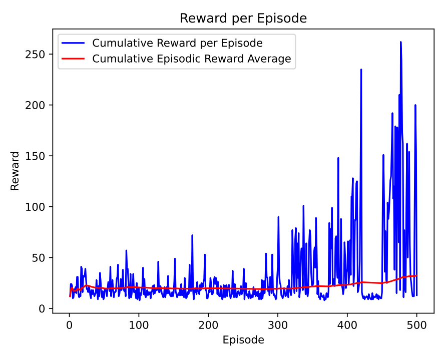


Figure 5. The cumulative reward obtained at each episode using a DQN without any experience replay. Compare to Figures 3 and 6 . Notice how the results for the latter episodes are significantly higher than those obtained by Q-learning but only about half those given by prioritized experience replay.

4.1. **No Experience Replay.** We plot the results of DQN without any experience replay in Fig. 5. Despite not being able to perform the disassociation or remembrance, the use of a 5-layer linear network comprised of ReLu activation functions and 2 hidden layers with a width of 8 neurons, allowed us to significantly improve, at times, the reward obtained by normal Q-learning. However, due to the stochastic nature of our policies, there were times where the network only did slightly better than Q-learning.

This can be explained by the fact that the hidden layers can compute features that the normal Q-learning function is not able to experience or compute. Moreover, the layers can adapt to a continuous state-space and do not need discretization, which is a major weakness of Q-learning.

4.2. **Prioritized Experience Replay.** Prioritized Experience Replay (PER) was first introduced by ([SQAS16]) in 2016 as a way to further improve (Uniform) Experience Replay by replaying experiences from which we can learn the most more frequently. This is measured by the Temporal Difference Error:

$$|r_t + \gamma \max_{a \in A}\{q[s_{t+1}, a, \phi]\} - q[s_t, a_t, \phi]|. \tag{4}$$

The priority p_i of the i^{th} experience is equivalent to its TD-error. We then assign a probability to each experience in the replay buffer based on its priority:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}. \tag{5}$$

The hyperparameter α controls how much weight we are giving to the probabilities. We only recalculate the TD-errors of sampled experiences in order to reduce the computation time. By replaying the experiences with high TD-error more frequently, we introduce bias into our model. In order to counteract this, every time an experience is replayed, we must decrease its effect on the training updates. We do this by weighting the errors δ_i as $w_i\delta_i$, where w_i is defined as

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta \cdot \frac{1}{\max\limits_i w_i}, \tag{6}$$

N being the size of the replay buffer and β emphasizing how much affect the weight should have. When implemented, β should start at β_0 and increase to 1. The best results came by initializing $\beta_0 = 0.4$.

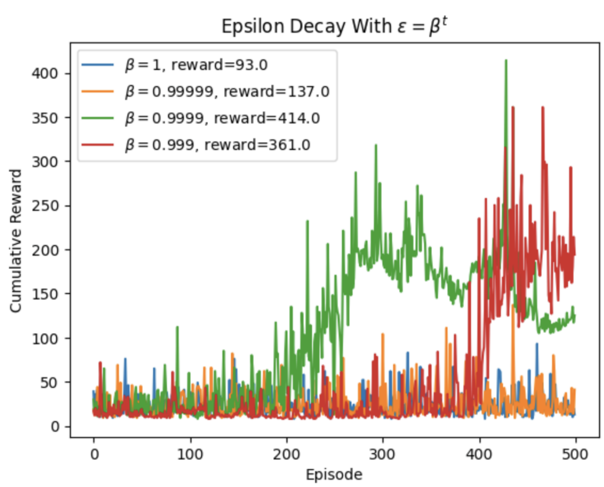


FIGURE 6. Cumulative reward with Prioritized Experience Replay Buffer and exponential epsilon decay for various values of β

Figure 6 shows the performance of exponential epsilon decay with Prioritized Experience Replay. Overall, the results were very similar to those found in Figure 4, which used normal/uniform experience replay. For $\beta = 0.9999$, the cumulative reward exceeds 200 before reaching its 300th episode, demonstrating how PER can learn more efficiently. Likewise, we see that for $\beta = 0.999$, performance exceeded a cumulative reward of 200 after 400 episodes. Overall, PER was slightly more efficient, though, no more accurate than uniform experience replay. In fact, for $\beta = 0.9999$, the cumulative reward was about 100 points short of that reached by the uniform replay experience. As stated before, performance varied widely on each run. Therefore, it is difficult to really see how these two implementations compare.

The results were similar for the other epsilon decay schedules reinforced with PER. 8a, 8b, and 8d show that each model reached a cumulative reward about 100 points higher than their respective models seen in 7a, 7b, and 7d. However, inverse epsilon decay, which had the best performance with a uniform experience replay (reaching nearly 300 points), performed dreadfully with PER as seen in 8c. Overall, the Deep Q Network was able to learn in fewer episodes, but the trained model didn't always perform as well as it did with traditional uniform experience replay. In addition, the runtime for PER averaged about 2 minutes, while uniform experience replay averaged 20 seconds. So, even though PER learned in fewer episodes, the runtime was still longer overall.

5. Conclusion

Overall, our results show that the use of neural networks does improve upon what Q-learning can produce. To very weakly-ish rival DQN, Q-learning must use well over 1000 episodes to train in order to achieve a cumulative reward over 100 (see Fig. 10). DQN further extends the ability of solving RL problems in general. Specifically, the use of prioritized experience replay suggest that DQN can converge faster than normal experience replay. Further testing would have to be done since replays outside of normal experience replay are typically used for harder RL problems like playing Atari.

As per the use of epsilon-greedy policies, our results demonstrate the necessity of exploring in the early stages of learning and then exploiting in the later stages to achieve maximal performance. The policies which worked best in our experience were those which were super-linear. Each model requires tuning of the hyperparameters to start with an epsilon large enough to explore many states at the beginning of training, while reaching an epsilon small enough to adequately exploit at the end of training. Moreover, our results also show the need for carefully using policies that meet the criteria stated in 3.1 which is something we were unsuccessful in doing. Overall, however, our results show the great usefulness in implementing neural networks to solve RL problems.

References

[DMM+22] Christoph Dann, Yishay Mansour, Mehryar Mohri, Ayush Sekhari, and Karthik Sridharan. Guarantees for epsilon-greedy reinforcement learning with function approximation, 2022.

[FRA+20] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay, 2020.

[JA24] X. James and Y. Al. Gymnasium: A toolkit for reinforcement learning, 2024.

[LJ94] Lin Long-Ji. Reinforcement learning for robots using neural networks. *CMU-CS*, Jan 06, 1994.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[Pri23] Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023.

[RN94] G. Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction (2nd edition)*. MIT Press, 2018.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SJLS00] S. P. Singh, T. Jaakkola, M. L. Littman, and C Szepesv ´ari. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38:287–308, 2000.

[SLM+17] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[SQAS16] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.

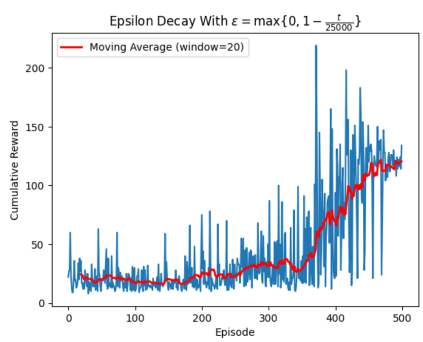
[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[Wat89] Christopher Watkins. Learning from delayed rewards. 01 1989.

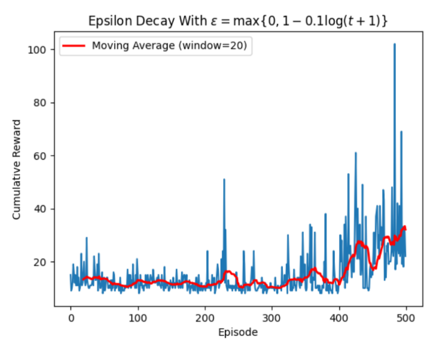
[WD92] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

Appendix A. Additional Graphs

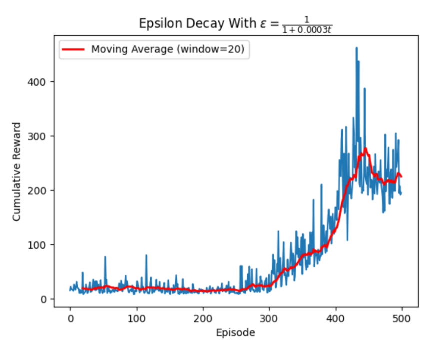
In this section we give additional graphs that help visualize our results.



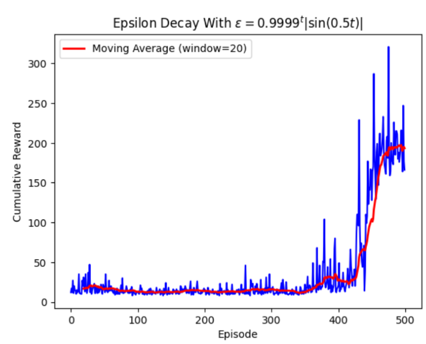
(A) Linear epsilon decay with the rule $\epsilon = \max(0, 1 - \frac{t}{25000})$



(B) Logarithmic epsilon decay with the rule $\epsilon = \max(0, 1 - \frac{1}{10}\log(t+1))$

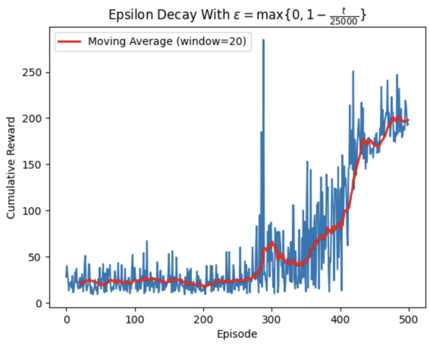


(C) Inverse epsilon decay with the rule $\epsilon = \frac{1}{1+\frac{3}{1000}t}$

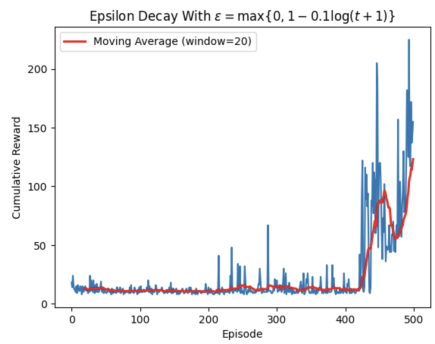


(D) Sinusoidal epsilon decay with the rule $\epsilon = 0.9999^t|\sin(\frac{1}{2}t)|$

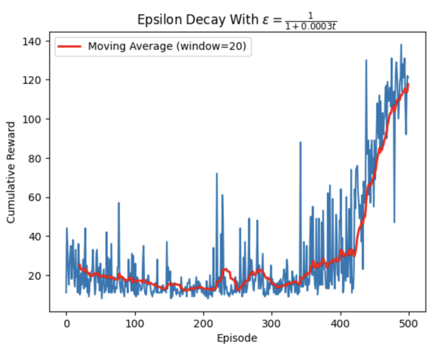
Figure 7. Cumulative reward with various epsilon decay schedules



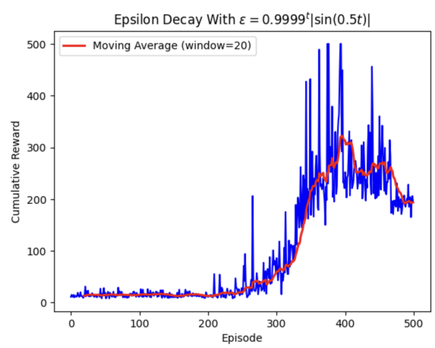
(A) PER and Linear epsilon decay with the rule $\epsilon = \max(0, 1 - \frac{t}{25000})$



(B) PER and Log epsilon decay with the rule $\epsilon = \max(0, 1 - \frac{1}{10}\log(t+1))$



(C) PER and Inverse epsilon decay with the rule $\epsilon = \frac{1}{1+\frac{3}{1000}t}$



(D) PER and Sinusoidal epsilon decay with the rule $\epsilon = 0.9999^t|\sin(\frac{1}{2}t)|$

FIGURE 8. Cumulative reward with PER and various epsilon decay schedules

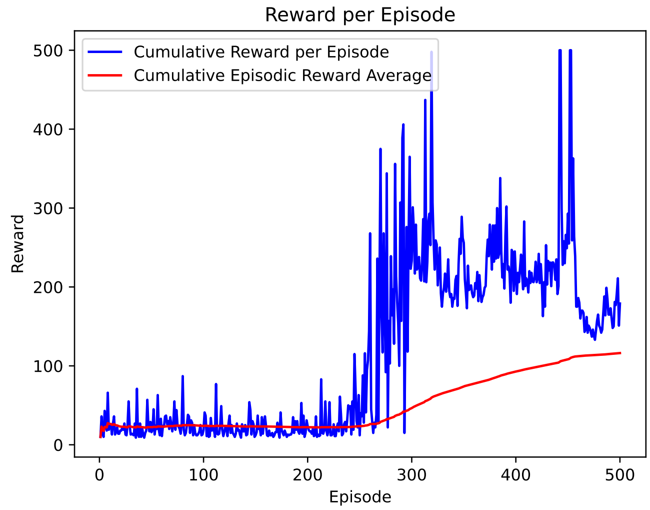


FIGURE 9. A graph showing the average reward as the agent goes through episodes and the overall cumulative reward per episode when employing uniform/normal experience replay. Compare to 6.

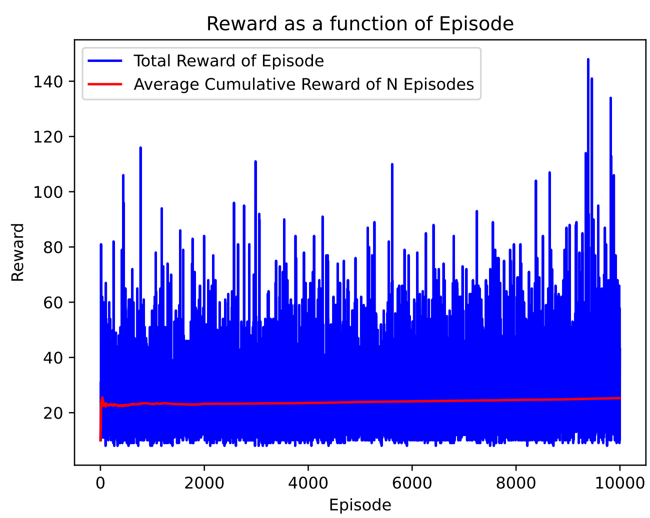


FIGURE 10. A graph of the reward obtained by the agent. We employed a decaying epsilon over 10,000 episodes. Notice that the average (red) slowly increases as we use more episodes.

Appendix B. Reinforcement Learning Definitions

- The agent is the decision maker that learns a sequence of actions to perform for a given task.
- The action space, is the set of all possible actions the agent can perform. We denote this as A, where $a_t \in A$ is a specific action at time t. A_s, called the *set of allowable actions* or *action set*, denotes set of actions from A that an agent can take during a specific state. The action set can be different for each state.
- The behavioral or learning policy is the policy in charge of selecting actions based on the current state and obtained reward (e.g. ε-greedy in our study). In DQN, we call the behavioral policy the *behavioral network* or *policy network* since we employ a neural network to compute it.
- The environment is the world in which the agent is located. It is usually stochastic in nature.
- An observation is the agent's measurement or perception of the state of the environment. Thus, the observation need not be the full state (i.e. MDP vs POMDP).
- The reward is a stochastic or deterministic function that assigns real values to states and actions in order to signal an immediate outcome. The reward helps the agent measure in the immediate sense the value of a specific state-action pair.
- The state space, is the set of all possible representations of the environment. We denote this as S, where $s_t \in S$ is a specific state at time t.
- The target policy is the policy in charge of updating the actual q-values. In DQN and Q-learning, it always exploits (i.e. the max over the actions). The network, in DQN, in charge of computing this policy is termed the *target network*.
- A time step t is the smallest discrete unit of time where the agent interacts with the environment once. This interaction typically comprises a cycle of one state, one action, and one reward. An *episode* is a finite sequence of time steps that starts at some *initial state*, s_0 (i.e. the state at $t = 0$), and ends at some *terminal state*, s_T . The *terminal state* is the final state representing the end of an episode and can be a maximum number of time steps or some other desired state. Note that in this latter case, the *time of termination*, T, can be different for each episode as well as the fact that each episode can have a different terminal state.

Appendix C. The Code