ECE 493 Assignment 4 Report
Akshay Patel | Student ID: 20732500

Unless otherwise noted, all figures and algorithm explanations are from the second edition Reinforcement Learning textbook used for this course, written by Richard S. Sutton and Andrew G. Barto.

Domain Description

Maze World is the deterministic environment that was used for this assignment. States, actions, and dynamics are defined as follows. States are defined as the agent being in a specific cell of the maze, and they are represented by four coordinates. Tasks can have walls that the agent cannot pass through, pits that are terminal states where the agent receives a large negative reward, and all tasks have a goal state for which the agent receives a positive reward. The action space in this environment consists of {up, down, right, left} represented by integers {0, 1, 2, 3} respectively. The dynamics of this environment are that given a state *s*, and an action *a*, taking action *a* from a state *s* will result in the agent being in a state *s'*. However, if the action takes the agent into a wall or takes the agent out of the grid, then the state will remain the same. If the agent reaches the goal or runs into a pit, then the game ends as those are terminal states. The reward function is that reaching the goal gives you a reward of +1 and ends the episode, running into a pit yields a penalty of -10 and ends the episode, running into a wall will give you a penalty of -0.3, and apart from these, the agent will receive a -0.1 reward for reaching any other cell in the grid.

Maze World is deterministic, meaning that if the agent takes a specific action from a specific state, it will always go to the same next state as well as receive the same reward value. The objective of this environment given this reward function is to reach the goal state as quickly as possible to maximize the reward.

Algorithm Explanations

Deep Q Network (DQN)

I referenced the following 3 articles to learn more about DQNs:
https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b
https://towardsdatascience.com/dqn-part-1-vanilla-deep-q-networks-6eb4a00febfb
https://medium.com/@jonathan_hui/rl-dqn-deep-q-network-e207751f7ae4

DQN is a reinforcement learning algorithm that was introduced in 2 papers, Playing Atari with Deep Reinforcement Learning in NIPS 2013 and Human-level control through deep reinforcement learning in Nature in 2015. DQN aims to overcome unstable learning through four main techniques, namely experience replay, target network (in Double Q-Learning DQN),

clipping rewards, and skipping frames. Not all of these techniques are required, and experience replay is very important, likely more so than the other techniques. I will explain only experience replay and target network in this report for brevity.

The formulation of this algorithm is the same as Q-learning (see Assignment 2 for an explanation on this). With DQN, we have a Q-function that is parameterized by the weights of the neural network since we are using the NN to approximate the Q-function.

Experience replay was first proposed in Reinforcement Learning for Robots Using Neural Networks in 1993. Deep neural networks easily overfit current episodes, and once the DNN has overfitted, it's difficult to encourage it to explore and produce different experiences. This can be quite troublesome if the DNN overfits to a local minimum/maximum as that would not be optimal. In order to solve this problem, experience replay stores experiences such as state transitions, rewards, and actions, which are necessary to perform Q-learning and make mini-batches to update neural networks. Experience replay aims to reduce correlation between experiences in DNN updates, increases learning speed through the use of mini-batches, and reuses past transitions to avoid catastrophic forgetting.

Target network is another technique used to fix the Q-value targets temporarily so we do not have a moving target to chase. In TD error calculation, the target function is changed frequently with DNNs. An unstable target function can make training quite difficult, so fixing the parameters of the target function and replacing them every so often helps improve stability. This technique is found in the Double Q-Learning variant of DQN.

With both experience replay and the target network, we have a more stable input and output to train the DQN. The full DQN with Experience Replay pseudocode from the paper, Playing Atari with Deep RL, is given below. We can see experience replay and mini-batch usage clearly in the pseudocode, but since this is just normal DQN, we don't see the target network here.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

In my implementation, I train the default DQN algorithm given in Stable Baselines (https://github.com/hill-a/stable-baselines/tree/master/stable_baselines). Note that this is actually the double Q-learning variant of DQN which uses the target network. Since this variant is still DQN, I sincerely hope I am not penalized for training a more stable variant of DQN, especially since the assignment didn't specify a specific DQN variant to follow. If I was told to specifically do the simple DQN, I could have easily done that by setting one default boolean to False in Stable Baselines. I train the algorithm for ~2000 episodes with each episode being capped at a maximum of 300 steps in Maze World in order to prevent extremely long episodes which were quite common, especially in Task 2 for DQN. I used the default hyperparameters for DQN from Stable Baselines, with learning rate = 0.0005, reward decay = 0.99 so as to account for future reward, and epsilon greedy that decays from 1 in the beginning to 0.02 in the final episode to encourage exploration in the beginning but encourage exploitation as the agent becomes more experienced. I did not have much time for this assignment to experiment with tuning the hyperparameters to increase performance, but I have no doubt that tuning the parameters may have resulted in increased performance in the tasks. Also, I used the Multi-Layer Perceptron policy for training the DQN.

Advantage Actor Critic (A2C)

I referenced the following article(s) to learn more about A2C:
https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f
https://medium.com/deeplearningmadeeasy/advantage-actor-critic-a2c-implementation-944e98616b

A2C is a policy gradient algorithm and it is on-policy. This means that we are learning the value function for one policy while following it, and so in other words, we can't learn the value function by following another policy. DQN on the other hand, uses another policy through experience replay because by learning from older data, we use information generated by a policy different from the current state. Since A2C is on policy, we can collect experiences but we must process them immediately.

A2C is an actor critic method that leverages an advantage function. On each learning step, A2C updates both the actor parameter with policy gradients and advantage value, and the critic parameter with minimizing the mean squared error with the Bellman update equation.

The advantage function basically means how much better it is to take a specific action compared to the average, general action at the given state. This value is called the advantage value, given below.

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$

This does not mean we need to use two neural networks for both the Q value and the V value in addition to the policy network; this would be quite inefficient. Instead, we use the relationship

between the Q and the v from the bellman optimality equation, leading us to be able to rewrite the advantage as:

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

And so, we only need to use one neural network for the V function that is parameterized by v above. Now, we can rewrite the update equation as the following, which is the Advantage Actor Critic.

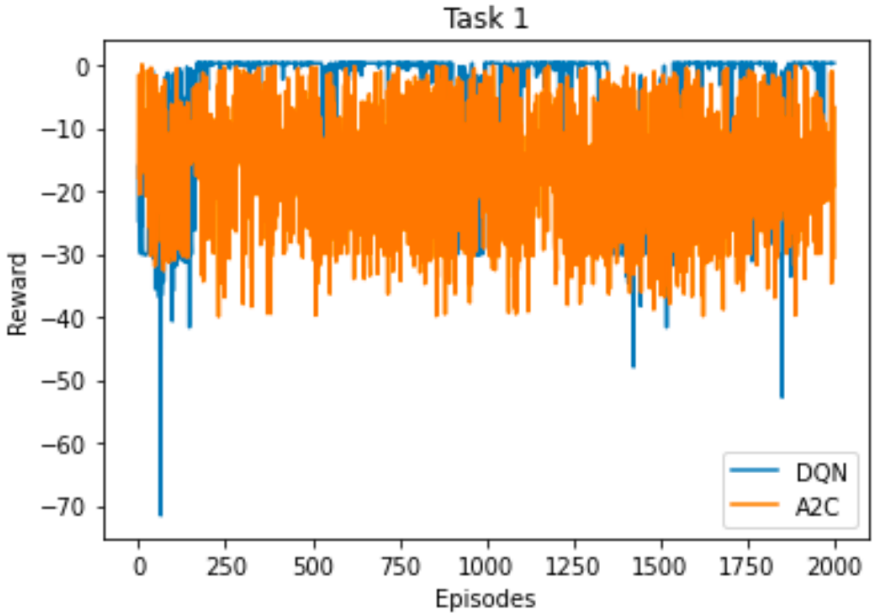
$$\begin{aligned}\nabla_\theta J(\theta) &\sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)) \\ &= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)\end{aligned}$$

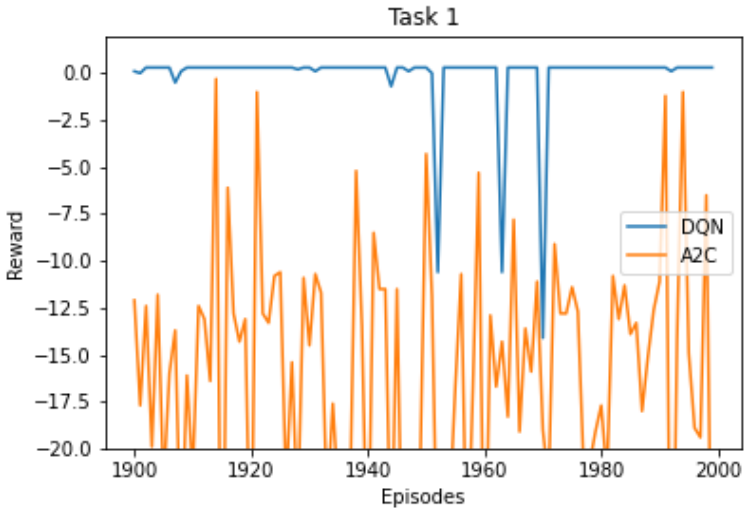
In my implementation, I train the default A2C algorithm given in Stable Baselines (https://github.com/hill-a/stable-baselines/tree/master/stable_baselines). I train the algorithm for ~2000 episodes with each episode being capped at a maximum of 300 steps in Maze World in order to prevent extremely long episodes which were somewhat common. I used the default hyperparameters for A2C from Stable Baselines, with learning rate = 0.0007, and reward decay = 0.99 so as to account for future reward. I did not have much time for this assignment to experiment with tuning the hyperparameters to increase performance, but I have no doubt that tuning the parameters may have resulted in increased performance in the tasks. Also, I used the Multi-Layer Perceptron policy for training the A2C algorithm.

Resources required for both algorithms

In terms of computational resources, both of these algorithms require significantly more compute in terms of memory usage and floating point operations than the algorithms previously studied in the class due to the deep neural network architecture used. However, the double Q-Learning DQN requires more computational resources compared to A2C due to the experience replay storage, batch updates, and the target network. In terms of resources required to implement these algorithms, there is not a lot of effort required if the Stable Baselines library is used. If it's not used, then these algorithms require much more effort, time, and cognitive ability to implement correctly compared to any of the previous algorithms we have studied.

Results and Analysis





As we can see from the plots above DQN performs reasonably well in Task 1 while A2C struggles quite a bit. Over the last 100 episodes, DQN is able to converge and achieve what seems to be the maximum reward of +0.3 from this environment in this task, but A2C is not able to converge. The maximum reward obtained by DQN is +0.3 which is the max reward for this task, whereas A2C achieves a max of +0.1, which surprisingly is not in the last 100 episodes (it's somewhere before). The variance over the final 100 episodes for DQN is approximately 4.3 whereas the variance for A2C is 53.1. From the variance, we can clearly see that A2C is not very stable in this task and does not converge at all.

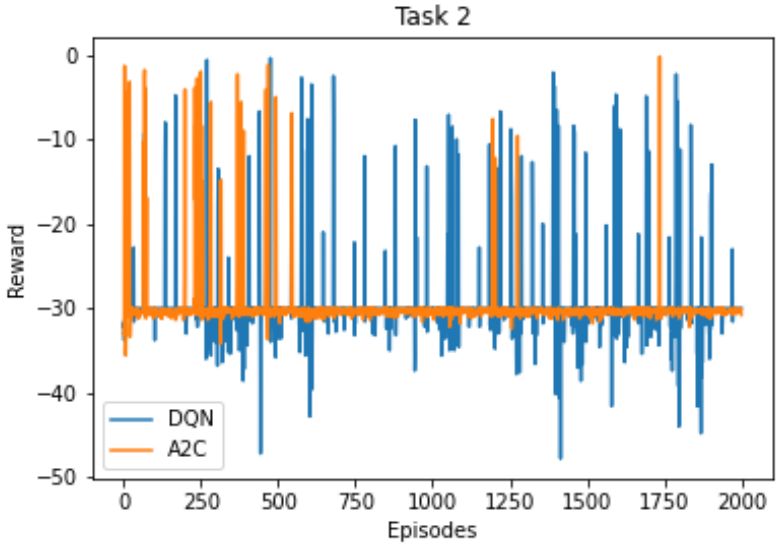
A2C, being a policy gradient algorithm, also likely suffers from high variance and low convergence, similar to other policy gradient algorithms. The stochastic policy may take different actions in different episodes, and one wrong action can completely alter the episode result. Perhaps the learning rate was too high and it could not successfully converge through the

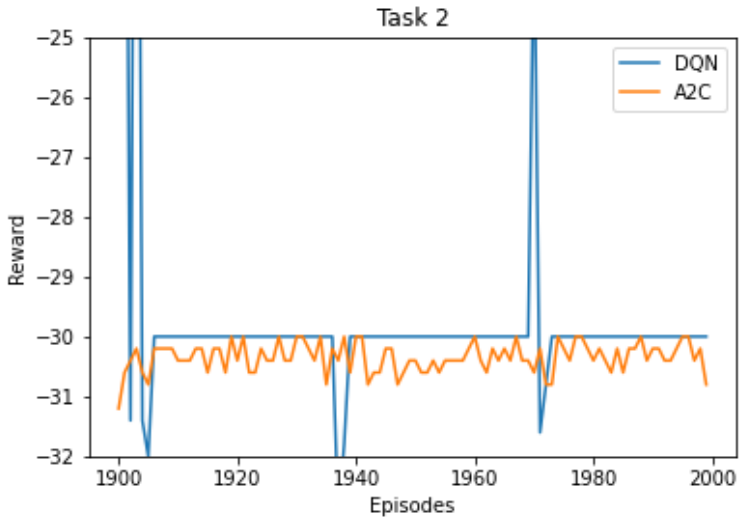
gradient. Tuning the hyperparameters and performing more experiments might have led to better performance for A2C in Task 1. Training for more episodes could help as well.

One problem with the plot above is that the A2C graph covers up the majority of the DQN graph so I do not really know exactly how fast and how well DQN converged.

Compared to Sarsa(Lambda) and Expected Sarsa, both A2C and DQN perform worse in Task 1. Expected Sarsa and Sarsa(Lambda) converge very fast in Task 1 to obtain the maximum reward of +0.3 for much less computational resources required and much easier implementation if Stable Baselines was not allowed. Moreover, Sarsa(Lambda) and Expected Sarsa required very little hyperparameter tuning compared to these Deep RL algorithms. Sarsa(Lambda) and Expected Sarsa also had much lower variances over the last 100 episodes with 0.03 and 0.01 respectively. This can be seen by comparing the plots above to the plots from Assignment 2 Report.

Task 2



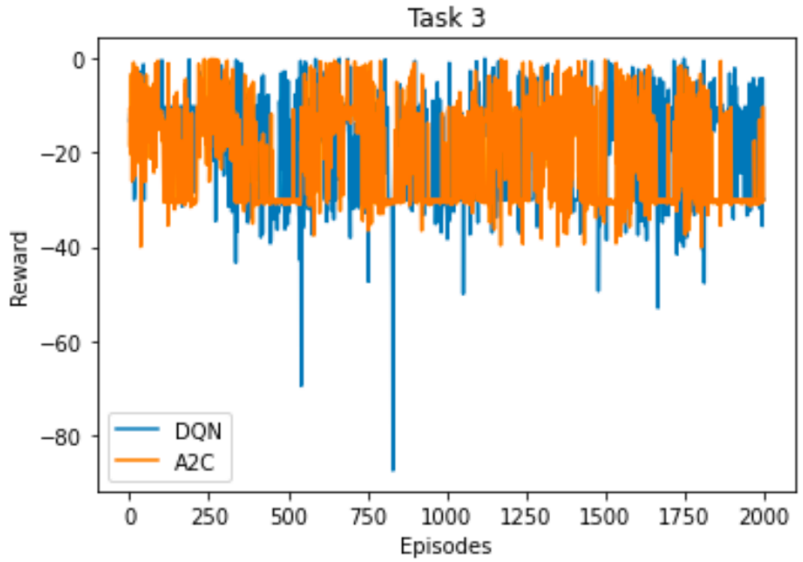


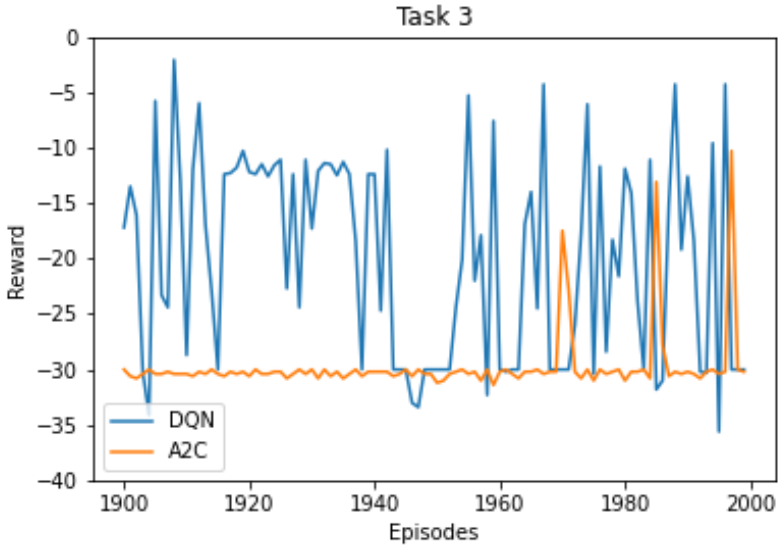
As we can see from the plots above, both DQN and A2C struggle a lot in Task 2. Over the last 100 episodes, neither of these algorithms is able to consistently obtain a reward that is anywhere near the maximum reward of -0.1 from this task. It does seem like A2C has less variance over the entire 2000 episodes compared to DQN though. It seems like both converge at around a reward of -30, with spikes up and down for both algorithms, with A2C having less very negative spikes compared to DQN. Interestingly, DQN and A2C are able to obtain maximum rewards throughout the entire 2000 episodes of -0.4 and -0.2 respectively. The variance over the final 100 episodes for DQN is approximately 6.8 whereas the variance for A2C is 0.06. From the variance, we can see that A2C was more stable, but it wasn't stable at a great reward so it's not super helpful.

Since neither of these algorithms converged at the maximum reward possible, but seemingly did converge at approximately -30, it's quite likely that they got stuck in local minima/maxima. Perhaps a lower learning rate might have helped them learn better, along with entropy-based or curiosity-based exploration methods to encourage them to explore and reach the goal state in the optimal manner. Perhaps more hyperparameter tuning would have helped these Deep RL algorithms perform better as well, but due to time constraints, I did not have the capacity to experiment as much as I would have liked.

Compared to Sarsa(Lambda) and Expected Sarsa, both A2C and DQN perform worse in Task 2. Expected Sarsa and Sarsa(Lambda) converge very fast in Task 2 to obtain the maximum reward of -0.1 for much less computational resources required and much easier implementation if Stable Baselines was not allowed. Moreover, Sarsa(Lambda) and Expected Sarsa required very little hyperparameter tuning compared to these Deep RL algorithms. Sarsa(Lambda) and Expected Sarsa also had similar variances over the last 100 episodes with 0.3 and 0.06 respectively. This can be seen by comparing the plots above to the plots from Assignment 2 Report.

Task 3





As we can see from the plots above, both DQN and A2C struggle a lot in Task 3. Over the last 100 episodes, neither of these algorithms is able to consistently obtain a reward that is anywhere near the maximum reward of -0.5 from this task. It seems like A2C converges around -30 over the last 100 episodes, whereas DQN does not seem to converge at all. DQN and A2C are able to obtain maximum rewards throughout the entire 2000 episodes of -0.9 and -1 respectively, which is somewhat close to the maximum reward but not quite. The variance over the final 100 episodes for DQN is approximately 82.9 whereas the variance for A2C is 9.1. From the variance, we can see that A2C was more stable, but it wasn't stable at a great reward so it's not super helpful. DQN just did not converge at all as we can see from the variance.

Since neither of these algorithms converged at the maximum reward possible, but A2C seemingly did converge at approximately -30, it's quite likely A2C got stuck in local minima/maxima. For DQN, it seems as if it had maybe too large of a learning rate and could not traverse the gradient properly. Perhaps a lower learning rate might have helped them learn better, along with entropy-based or curiosity-based exploration methods to encourage them to explore and reach the goal state in the optimal manner. Perhaps more hyperparameter tuning would have helped these Deep RL algorithms perform better as well, but due to time constraints, I did not have the capacity to experiment as much as I would have liked.

Compared to Sarsa(Lambda) and Expected Sarsa, both A2C and DQN perform worse in Task 3. Expected Sarsa and Sarsa(Lambda) converge very fast in Task 3 to obtain the maximum reward of -0.5 for much less computational resources required and much easier implementation if Stable Baselines was not allowed. Moreover, Sarsa(Lambda) and Expected Sarsa required very little hyperparameter tuning compared to these Deep RL algorithms. Sarsa(Lambda) and Expected Sarsa also had much lower variances over the last 100 episodes with 2.3 and 3.5 respectively. This can be seen by comparing the plots above to the plots from Assignment 2 Report.

Overall Analysis

Both algorithms described in this report are computationally very expensive due to the deep neural network architectures along with the numerous floating point operations that are required to obtain the optimal weights for the neurons. Neither of these algorithms is able to consistently converge at the maximum reward that can be obtained across all environments (even though DQN performs well in Task 1, it does not do well at all in Tasks 2 and 3). Moreover, these algorithms would be very hard to implement if Stable Baselines was not allowed.

As seen through the tasks above and the analysis of Expected Sarsa and Sarsa(Lambda) from assignment 2, the Deep RL algorithms are not recommended for use in Maze World as the Expected Sarsa and Sarsa(Lambda) algorithms perform much better consistently across all of the three different tasks for much less computational expense, and are much easier to implement generally (especially if Stable Baselines is not allowed). Thus, Expected Sarsa and Sarsa(Lambda) are recommended over these Deep RL algorithms for Maze World in these tasks. Perhaps these Deep RL algorithms would perform better in much more complex environments as they are most likely able to handle lots of information through the DNN architectures.

As per the analysis from Assignment 2, Expected Sarsa is recommended over Sarsa(Lambda). Please refer there for the analysis.

If I had more time, I would have loved to measure the amount of CPU and Memory usage for these Deep RL algorithms and compare them to Expected Sarsa and Sarsa(Lambda). This information would have been very insightful in determining just how much more computationally expensive Deep RL algorithms can be.