ASSIGNMENT 4: A2C AND DEEP Q-LEARNING

Ozan Ilhan
s3559513

Qiu van Leeuwen
s3547566

1 INTRODUCTION

In this assignment we have coded two different models, which are Advantage Actor Critic (A2C) and Deep Q-Learning (DQL). A2C uses two different neural networks, while DQL uses one neural network. A2C is an on-policy algorithm, which means it learns from the actions it takes while following its current strategy. DQL is an off-policy algorithm, which means it learns from the actions it takes from different strategies. This allows it to learn from past experiences, which are stored in the replay buffer. This buffer is like a memory bank for the agent. Actions are stored in here and the DQL agent can randomly sample from this memory bank to use it for learning. A2C cannot use this because A2C is an on-policy algorithm.

As inspiration and guidance to create these algorithms, since these were not taught to us in class, we used (1) and (3).

We have chosen an environment from the Gym (2), there are a few environments we could choose from and we ended up choosing the blackjack environment. We chose this because this one is very quick to train, thus allowing us to run big tests. In this environment, the agent plays against the dealer. The observation space consists of three things. First, the player's current sum, the dealer's one showing card and whether or not the agent holds an usable ace. The action space consists of two actions, which are stick and hit. This results in 704 unique states and 2 actions. If the agent wins the game, the reward is +1. If the agent loses the game, the reward is -1, and lastly, if it is a draw then the reward is 0.

2 RANDOMIZED TESTING

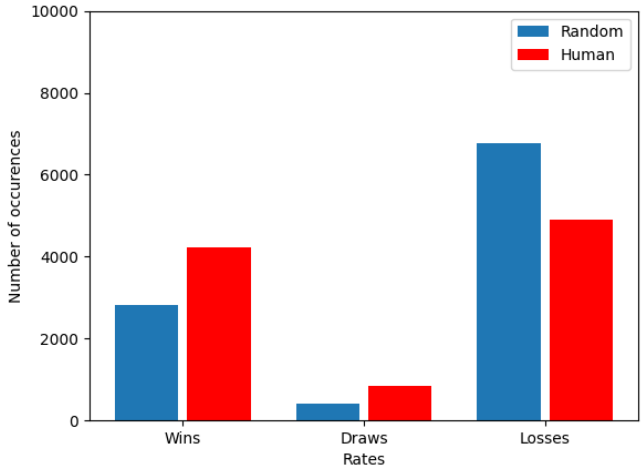


Figure 1: Randomly sampling an action from the BlackJack environment 10000 times over 10 times and taking the average of this, while comparing it to real human win, draw and loss rates.

We first decided to run a model where the agent just randomly executes actions. We ran multiple tests and averaged these results and eventually compared them to real human rates, as you can see in figure 1.

The random agent only wins 2800 games, which means it wins 28% of games. It loses around 68% of games and has a draw rate at around 4% of games. The human rates are, 42.22% win rate, 49.1% lose rate and 8.42% draw rate. The randomized actions are far off from being near human level. We tried to make a RL agent that beats and surpasses humans at blackjack.

3 MODELS

3.1 HYPERPARAMETER TUNING

In our experiment we compare A2C and DQL models with varying learning rates. To get a well-performing model, we had to tune the gamma value and hidden layer size of these two models. We first ran multiple tests where we evaluated models with different gamma values. We tested the models with the following gamma values: 0.3, 0.5, 0.7, 0.9 and 0.99. For A2C, the best performing models had a gamma value of 0.99. For DQL, the best performing models had a gamma value of 0.3.

Next, we tried to find a good hidden layer size. We trained our models with the optimal gamma values we just said and varying hidden layer sizes, such as: 8, 16, 32 and 64 and 128 hidden nodes. For A2C, the best choice would be 32 hidden nodes, but for DQL the best choice would be 128 hidden nodes.

3.2 A2C

Refer to listing 1 in the appendix for the code. Our first algorithm is A2C. This uses a combination of reinforcement learning with deep learning, also called Deep RL. There are two neural networks. One for the actor and one for the critic. They both have the observation space as the input, which is three different observations. They also both have a hidden layer of 32 hidden nodes. The actor is responsible for choosing actions. We apply soft-max on its output to turn the raw values into a probability distribution over all the possible actions. These probabilities sum up to 1. This also allows for some exploration, since even actions with a low probability of being chosen will be explored, since they have a chance of being selected. The other neural network is the critic. This neural network has a single output. This output is the value estimation. It estimates the value of being in a certain state, which is basically a prediction of future rewards. This is done to guide the actor's policy improvement. By comparing the critic's value estimation with the actual rewards received, the algorithm can calculate the advantage. The advantage indicates how much better or worse an action was compared to what was expected and this guides the actor with improving its policy because the actor can use this information to adjust its policy, which will lead to better actions being chosen.

We train our model for 5000 repetitions with a gamma value of 0.99 and varying learning rates of 0.0001, 0.0005 and 0.001 for both the critic and the actor. The gamma value of 0.99 ensures that the model takes into account long-term rewards, since this is important in the blackjack environment. A high gamma value is also important for actor-critic algorithms, since the two models have to learn stable policies. A high gamma value helps with faster convergence and learning. During the training, we call the model with the current state and we select an action using the probability distribution given by the actor. We then execute this action in the environment and calculate the advantage. We use this advantage to calculate the actor and critic loss, to optimize the policy. The equation of the advantage, actor loss and critic loss are the following, where *A* is the advantage, *r* is the reward, *v* is the value of the current state, *v'* is the value of the next state, *d* is 1 if the current state is a terminal state and lastly, $\pi(a)$ is the probability distribution of the current policy:

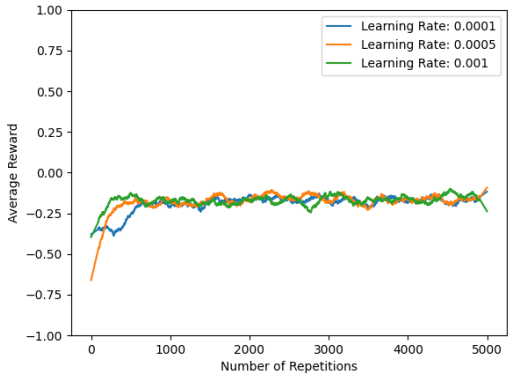
$$\mathrm{A} = r + (\gamma \cdot v' \cdot (1 - d)) - v \tag{1}$$

$$\mathcal{L}_{\text{actor}} = -\log(\pi(a)) \cdot \mathrm{A} \tag{2}$$

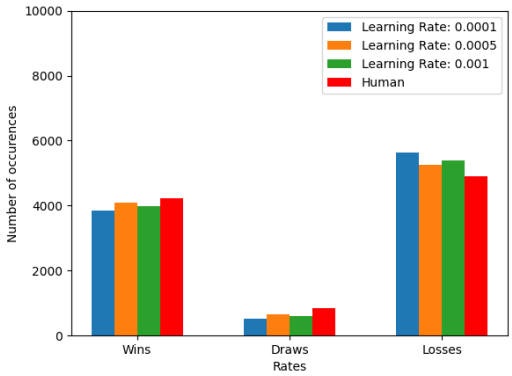
$$\mathcal{L}_{\text{critic}} = \mathrm{A}^2 \tag{3}$$

These loss values are used to correctly update the model weights during back-propagation. This ensures that the A2C models learn and find an optimal policy.

To test the performance of our A2C algorithm, we trained different models with varying learning rates. We ran 10 tests per learning rate and averaged these results. We kept track of the rewards and used this to plot a learning curve, as seen in figure 2a. This figure shows us how the agent starts to learn how to play blackjack for every learning rate. All three curves are kind of similar. The curve with the learning rate of 0.0001 performs okay. It increases till 800 repetitions, where it kind of converges at a reward of around -0.2. The curve of the learning rate of 0.0005 performs better. It converges at an average reward of -0.2 as well at around 800 repetitions, but this time the curve stays a bit higher than the previous one. The final curve, which is the curve of the learning rate of 0.01 converges at an average reward of around -0.2 as well. It increases quickly and converges at around 800 repetitions as well. All three curves act very similarly. The one that would be the most optimal would be the curve of the learning rate of 0.001 as this one starts at a higher average reward than the rest and converges quickly. This reward they all converged on indicates that the agents still lose more than they win. This is actually a sign that it plays similarly to humans, since the house eventually always wins. The reason the other curves are slightly less optimal could be due to a lower learning rate resulting in the agent learning slightly slower and therefore having an overall lower average reward.



(a) Average reward over 10 training tests per learning rate.



(b) Average Wins, Draws and losses

Figure 2: A2C models with learning rates 0.0001, 0.0005 and 0.001 compared in a graph where we show the learning curves and a graph where we show how well the models perform when tested in the BlackJack environment.

We also tested all these models in the blackjack environment by making them play 10000 games each over 10 tests and averaging the results, so this makes them play 100000 games in total. We kept track of who won and we averaged these results and show them in figure 2b side by side with real human rates. We use the human rates as a metric to evaluate how well the models perform. If they surpass the human win rate, then this obviously means they play better than the average human. While the A2C models got close to surpassing humans, none of them eventually did surpass humans. The model with learning rate 0.0001 plays the worst out of all of them. It has around 3800 wins, which is a lot under human win rates. It also loses a lot and has little draws. The model with learning rate 0.0005 performs the best out of all of them. It wins around 4100 games, which is close to human win rates. It also loses the least games and has the most draws. The model with learning rate 0.001 plays slightly worse by having around 4000 wins. Overall, the model with learning rate 0.0005 seems to perform the best and gets close to human rates. The learning rate of 0.001 was supposed to be the best one according to figure 2a, but it performs slightly worse than the learning rate of 0.0005. This could be due to figure 2a showing the learning curve, not how optimally the models actually will work during the testing. All three models perform slightly worse than human rates though. This could be due to us not optimizing all the hyper-parameters well enough. We did do hyper-parameter tuning but maybe there was some specific gamma value and hidden layer size where the model would perform better or maybe it is simply impossible for A2C to perform better than humans. It is also possible they do not perform well simply because such an environment is not that well-suited for A2C.

3.3 DQL

Refer to listing 2 in the appendix for the code. Our second algorithm is DQL. This uses only one neural network, but it also uses an experience buffer, which is like a memory bank of the state it agent has been in together with the action taken, reward, the next state and whether or not the current state is a terminal state. The input and output of the neural network are setup similarly as the A2C neural networks, but this time the hidden layer has 128 nodes. This neural network outputs Q-values for every action in the state.

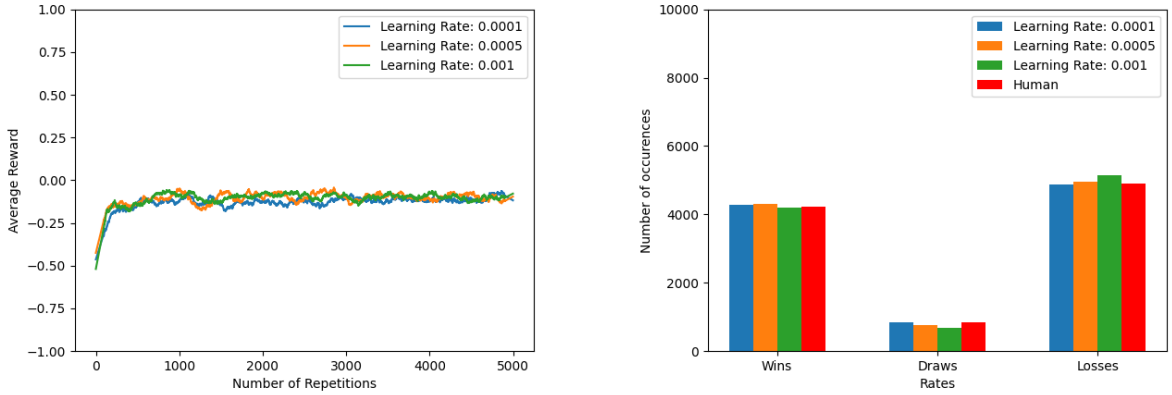
We trained this model similarly as A2C as well. We only changed the gamma value to 0.3 and hidden layer size to 128, as we figured out during hyper-parameter tuning that these seem to be the most optimal. During the training, we call the experience replay class with a batch size of 64 and use an e-greedy algorithm with epsilon 0.1 to select an action.

$$\text{action} = \begin{cases} \text{random action} & \text{if p} < \epsilon \\ \arg\max(Q(a)) & \text{if p} \geq \epsilon \end{cases} \tag{4}$$

We execute this action in the environment and then put all the variables returned from the step function in the experience buffer. When the size of the experience buffer reaches the batch size, we randomly sample experiences and get the Q values of the current state and next state. We then calculate the target Q values, which are the expected future rewards for taking an action in a given state, based on whether the episode is done or not per experience in the experience buffer. This experience is *i* in the following equation:

$$\text{target}(i, a_i) = \begin{cases} \text{r}_i & \text{if } d \\ \text{r}_i + \gamma * max(Q'_i) & \text{if not } d \end{cases} \tag{5}$$

If the episode is done, the target Q value is just the reward. Otherwise, it is the reward plus the discounted maximum Q value of the next state. We calculate the loss between the current Q values and the target Q values using the Mean Squared Error loss. What the Mean Square Error calculates is the average difference between the Q values and the target Q values. Using this loss value the neural network does back-propagation.



(a) Average reward over 10 training tests.

(b) Average Wins, Draws and losses

Figure 3: DQL models with learning rates 0.0001, 0.0005 and 0.001 compared in a graph where we show the learning curves and a graph where we show how well the models perform when tested in the BlackJack environment.

To test the performance of the DQL algorithm we did the same experimental setup as A2C. We plotted these results in figure 3. As seen in figure 3a, the three models with different learning rates converge very quickly. The first one, with a learning rate of 0.0001 converges at around 800 repetitions at an average reward of -0.2 and stays around this average reward. The second curve,

which has a learning rate of 0.0005 increases immediately to an average reward of around -0.15 and stays here. The last curve, which is the curve of a learning rate of 0.001 increases immediately to an average reward of -0.18. All three curves behave similarly, but the one that stands out is the curve with learning rate 0.0005, as this one tried to keep a high average reward. This could be due to it learning faster with a higher learning rate and also exploring and exploiting the state-action space more aggressively due to its high learning rate, while a learning rate of 0.001 might cause it to just learn too quickly and get stuck in a sub-optimal policy. The curve with learning rate 0.0001 has the lowest average reward out of all 3 possibly due to it learning way too slow.

The other figure, figure 3b, shows how well every model performs when testing it in the blackjack environment. They all perform very similarly. The models at the learning rate of 0.0001 and 0.0005 have the highest win rate. The model at learning rate 0.0001 performs the best out all three because it has the highest number of wins and lowest number of losses. This one, together with the learning rate of 0.0005 both perform better than humans. The final one, the learning rate of 0.001 performs slightly worse than humans. In figure 3a, the learning rate of 0.0001 was performing the worst, but how is this possible? Well, figure 3a shows the learning curves. This does not indicate the performance it will have during testing. It just shows how the agent learned and if it has a bad start, then the average reward of learning curve might be lower overall. As you can see, at 5000 repetitions all three models are at around the same average reward. So, the best performing model is the model with learning rate 0.0001, which plays blackjack better than humans. This could be due to a low learning rate causing more stability and reducing the chances of overfitting. It is possible that the other two models overfit a bit, causing them to perform worse. A higher learning rate can also cause a model to converge quickly, as seen in figure 3a with learning rate 0.001, but this could also cause it to get stuck in a sub-optimal policy.

4 Comparison

Overall, DQL performs better than A2C, since two of the three models in DQL perform better than humans, while not a single model in A2C performs better than humans. The curves in figure 3a also increase and converge quickly, while the curves in figure 2a increase and converge slightly slower. All curves converge at around the same average reward. In figure 3b, the models perform better overall compared to figure 2b, as you can see two models perform better than humans in figure 3b.

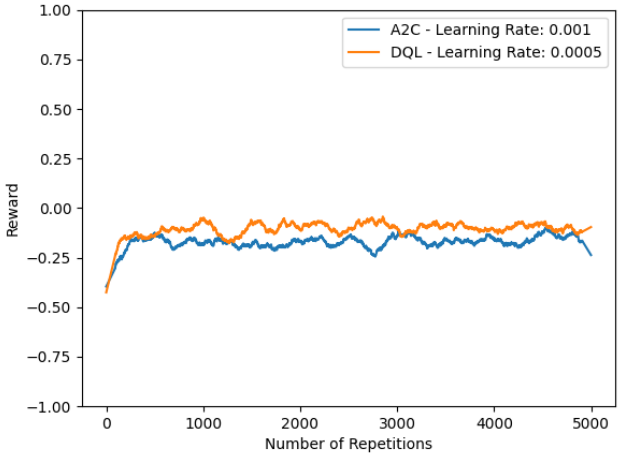


Figure 4: After running the A2C and DQL algorithms with learning rates 0.0001, 0.0005 and 0.001 we took the learning curves that had the highest average reward overall of both algorithms.

We have put two curves in figure 4. These are the best performing models from A2C and DQL. The best performing model that uses the A2C algorithm has a learning rate of 0.001. The best performing model that uses the DQL algorithm has a learning rate of 0.0005. The curve of DQL increases very quickly. It also converges at around 200, while the curve of A2C increases kind

of slowly compared to DQL. It also converges at around 400 repetitions. There could be various reasons for DQL performing better than A2C. The first one is in how both algorithms differ from each other. DQL is an off-policy algorithm and uses an experience buffer. This allows it to learn from a diverse set of experiences. This increases stability and reduces variance. A2C is an on-policy algorithm and does not use an experience buffer. It updates the policy based on the current path it is taking. This can lead to higher variance in the updates and a slower convergence, because the model is constantly adapting to new experiences without using an experience buffer. Another reason could be the learning rate. DQL has a learning rate of 0.0005, while A2C has a learning rate of 0.001. A higher learning rate of course means that model learns quicker by making larger updates, which can cause instability and result in the agent getting stuck in a sub-optimal policy. A slighlty lower learning rate may make the learning process more stable causing the learning curve to increase and converge quickly. Another reason is in how both algorithms handle exploration and exploitation. DQL uses an e-greedy algorithm with an epsilon value of 0.1, which allows it to explore 10% of the time and exploit 90% of the time. A2C does not do this and simply relies on the current policy for exploration since it is an on-policy algorithm. It does not explore less-visited states as much as an off-policy algorithm like DQL does. The last reason is the use of the experience buffer. This buffer allows DQL to reuse past experiences multiple times. This allows it to be more efficient and learn more quickly, even if the best performing model has a lower learning rate than the best performing A2C model. A2C does not use such a technique and does not rely on past experiences to learn. This makes it less efficient and therefore makes it slower in learning. All these reasons together make DQL a far more superior algorithm in our environment. A2C would work better in a complex action space or a continuous action space. A2C adapts more quickly to changes in the state-action space due to it being on-policy and not relying on past experiences. Since our environment was just a large, discrete state-action space, DQL works better. The environment is stable and things will not just change frequently, allowing DQL to work better and perform better than humans.

5 Discussion

We have tested both algorithms and noticed that DQL performs better in the environment that we chose, due to it not being a complex state-action space. A2C would work better in a complex state-action space or a continuous action space. We now also understand the importance of hyper-parameter tuning, as our models would perform very badly and converge very slowly when our hyper-parameters were not optimized.

In the future, we could research another algorithm. Soft Actor-Critic (SAC) is kind of a mix between DQL and A2C, since it uses an experience buffer and an actor and critic. We could compare the algorithm to DQL and A2C in the blackjack environment to see if this algorithm would be more efficient. We do think this would be more efficient as it combines the two good parts of both algorithms. Another thing we could test out is comparing SAC in two different environments. This environment would be the Mountain Car and Mountain Car Continuous environments in the Gym (2). One has a discrete action space and one has a continuous action space. This would be a perfect opportunity to compare how SAC performs in a discrete and continuous action space.

6 Conclusion

In conclusion, we have created the Advantage Actor-Critic (A2C) and Deep Q-Learning (DQL) algorithms and have tested these in the BlackJack Gym (2) environment. A2C is an on-policy algorithm and uses two neural networks. One for the actor and one for the critic. These two work together to select an action. DQL is an off-policy algorithm that uses an e-greedy algorithm to select an action. It may sometimes explore actions or just exploit and select the current best action. In the environment that we chose, DQL performs better than A2C due to our environment being a discrete state-action space that is not too complex. Two of our DQL models also perform better than humans at blackjack. A2C performs more poorly since A2C is not made for the environment we used. A2C would perform better in a complex or continuous action space, instead of a discrete state-action space.

REFERENCES

[1] GeeksforGeeks. Actor-critic algorithm in reinforcement learning. *GeeksforGeeks*, 2023.

[2] Gym Documentation. Gym is a standard api for reinforcement learning, and a diverse collection of reference environments. `https://www.gymlibrary.dev/index.html#`.

[3] Aniruddha Sanyal. Develop your first ai agent: Deep q-learning. *Towards Data Science*, 2019.