University of Liverpool, Dept of Computer Science

Machine Learning and Bioinspired Optimisation: CA2

April 2023

Name	**Email**	**Student ID**
Rhys Cooper	sgrcoop2@liverpool.ac.uk	201680340
Liam Francis	sglfran5@liverpool.ac.uk	201690632
Ryan Maxwell	sgrmaxwe@liverpool.ac.uk	201328336
David Tate	sgdtate@liverpool.ac.uk	201688052
Thomas Williams	sgtwill6@liverpool.ac.uk	201450922

Table 1: Student Information

1 Introduction

A deep reinforcement learning agent using the Deep Q-Learning algorithm is deployed to play the "Blackjack-v1" game from the Open AI gymnasium. The aim is for the agent to learn an optimal policy that maximises its chances of winning against a dealer for each round played. Pytorch is used to build and train the Deep neural network used within the Deep Q-Learning algorithm.

2 Environment and Game Description

The value of the two cards dealt to a player is their hand, with cards 1 through 10 worth their face value, whilst royal cards are worth 10 points each. This is the 'player hand value sum' in the observation space table of Table 2 (left). Each ace can be used as either a 1 or 11 at the agents discretion, represented as 'usable' ace in the observation space table of Table 2 (left).

A player can 'hit', shown in the Action Space in Table 2 (middle), to draw another card to increase the value of their hand, with the goal being to achieve a score as close to 21 as possible without going over. If it goes over or the dealer ends with a higher valued hand, the agent loses that game and the environment returns a reward of -1. If the player's hand is worth more than the dealer's the reward is +1, and if the agent and dealer's hands are equal the environment returns a reward value of 0.

The game of Blackjack is by nature stochastic and so any strategy is limited in a probabilistic sense. For example, even with a strong hand, the dealer could still win. Although Blackjack is normally played with multiple decks, this game has been simulated with an infinite number of decks and is played until the specified number of rounds is met.

Observation	**Min**	**Max**
Player hand value sum	4	12
Dealer showing card value	2	11
Usable ace	0	1
Round has ended	True	False

Action	Value
Stick	0
Hit	1

Reward	Value
Loss	-1
Draw	0
Win	1

Table 2: Observation Space, Action Space, Reward Space

3 Deep Q-Learning

We deploy Deep Q-Learning, a reinforcement learning algorithm that uses a deep neural network to estimate the Q-values of each possible action in a given state, which refers to the best possible score at the end of an episode after performing action a in state s and proceeding optimally thereafter. Formally, this is represented as $Q(s_t, a_t)$, where Q is the Q-function, s_t is the state at time t, and a_t the action taken at time t. This Q-value represents the discounted future reward associated to action a for a given state, and is given by the addition of the immediate reward received, r_{t+1}, and the discounted, highest Q-value of the subsequent state s_{t+1}. This is all shown in equation (1) below.

$$Q(s_t, a_t) = E(r_{t+1} + \gamma max_{(a')} Q(S_{t+1}, a_{t+1}) \mid s_t, a_t) \tag{1}$$

Rather than storing the Q-values in a tabular form of a Q-table, the Deep Q-Learning internalises an approximation of the Q-function into a Deep Q-Learning neural network (DQN). Taking the current state as input, the DQN returns the expected Q-value for each action in the action space; forming its estimate of the Q-value function using the parameters of the DQN model (ϕ). This is shown in equation (2) below.

$$Q(s_t, s_t; \phi) \approx Q^*(s, a) \tag{2}$$

The DQN is then trained using a supervised learning approach; the prediction is the output of the DQN for a given state, $Q(s_t, a_t; \phi)$, and the target is the value calculated according to equation (1). The loss is given by the squared difference between the predicted and target value, which is then used to update the parameters of the DQN in the backwards propagation pass.

After initialising a Deep Q-Network in the Agent `__init__` method, the main structure of the code takes place in two core methods: `play_rounds()` and `train()`.

```python
for round in range(self.n_rounds):
    self.epsilon *= 1 / (1 + self.eps_decay_rate * round)
    state = env.reset()
    next_state = self.observation_formater(state)
    ended_round = False

    while not ended_round:
        state = next_state
        action = self.get_action(state)
        next_observation = (env.step(action))
        next_observation = self.observation_formater(next_observation)
        mem_clip = [state, action, next_observation]
        self.add_to_memory(mem_clip)
        next_state = next_observation[0]
        reward = next_observation[1]
        ended_round = next_observation[2]

        if len(self.memory) > self.batch_size:
            loss = self.train().item()
            loss_dict[round] = loss

    self.optimal_strategy(state, action)
    accuracy_dict[round] = self.get_accuracy(round+1)
```

Listing 1: play_rounds() snippet

An overview of the implementation is included here as a precursor to the discussion of results. The agent uses a DQN comprised of three fully connected layers, each with a ReLU activation function. It takes as input an observation from the environment according to the observation space in Table 2 (left). The maximum of the Q-values determines the action the agent takes. In `play_rounds()`, the agent plays a fixed number of rounds of Blackjack, and for each round, it uses an decaying epsilon-greedy policy to select actions. After each action, the agent stores the current state, action, next state, and reward in a replay memory buffer(Mnih et al., 2013). Once the replay buffer reaches a certain size, the agent randomly samples a batch of state transitions from the buffer and updates its neural network parameters using the DQN loss function. After each round, the agent calculates and records the loss and accuracy (relative to the optimal strategy) of its learned policy. In the `train()` method, the current and next state Q-values are then calculated using the DQN. The target Q-values are then used to create a target Q-value tensor. The loss is calculated between the target Q-value tensor and the current state Q-value tensor. The optimiser's gradients are then set to zero, and the backward pass is calculated using the loss tensor. Finally, the optimiser's step method is called to update the weights of the network, and the loss is returned for logging purposes.

3.1 Double Deep Q-Learning

After successfully implementing a Deep Q-Learning Agent, we extended our code to incorporate a Double Deep Q-Learning (DDQL) agent. The aim of this extension was to overcome the overestimation bias that commonly occurs when using DQL for games (Van Hasselt, Guez and Silver, 2015b). DDQL involves two Deep Q-Networks: the policy network and the target network. As with DQL, the policy network determines the action based on the maximum Q-values for a given state, which determines the agent's subsequent reward. Whereas Deep Q-Learning uses the same network to estimate the next state Q-values, as seen in equation (1), in DDQL the next state Q-values are instead estimated using the target network. The weights of the policy network are then updated based on the difference between the predicted Q-value and the target Q-value.

Finally, in Double Deep Q-Learning the weights of the policy network are periodically transferred to the target network. The idea is that since the target network's weights are only updated periodically, rather than every time a new training example is encountered, this will prevent the network from overestimating Q-values that are seen early, and reduce overestimation bias.

3.2 Hyperparameter Tuning

To find the optimal hyperparameters, we searched through the hyperparameter combinations. The combination that produces the highest win rate percentage is shown below.

batch size	ϵ	min ϵ	ϵ decay rate	learning rate	γ	update freq	win rate
175	1	0.0125	0.0001	0.001	0.5	12	43.25%

Table 3: Optimal Hyperparameter Combination

4 Results

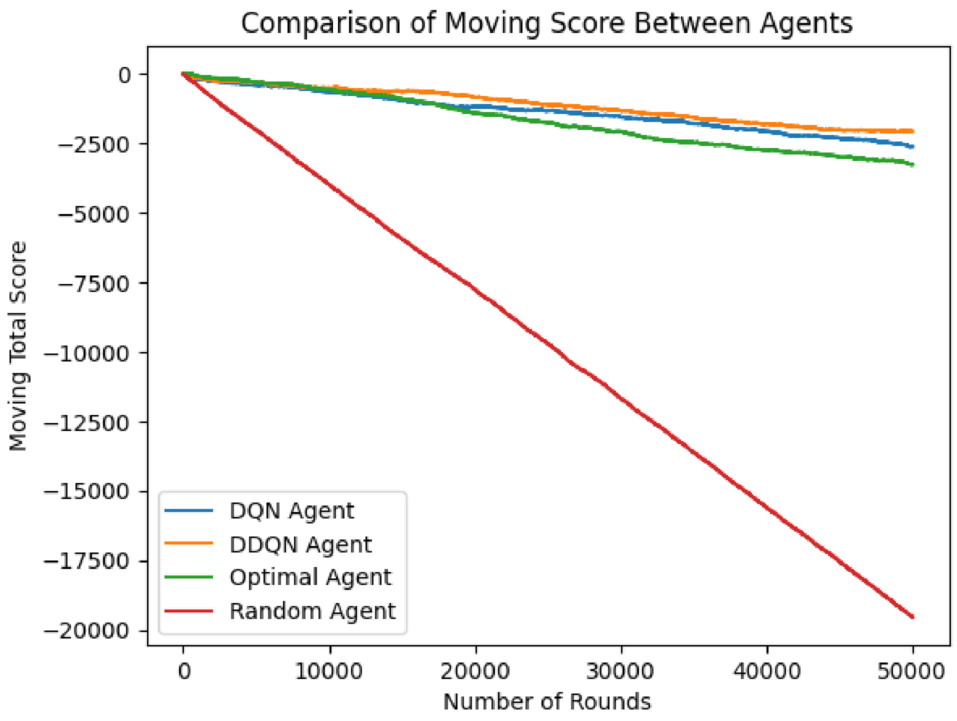

Figure 1: Comparison of Moving Total Score Between Agents

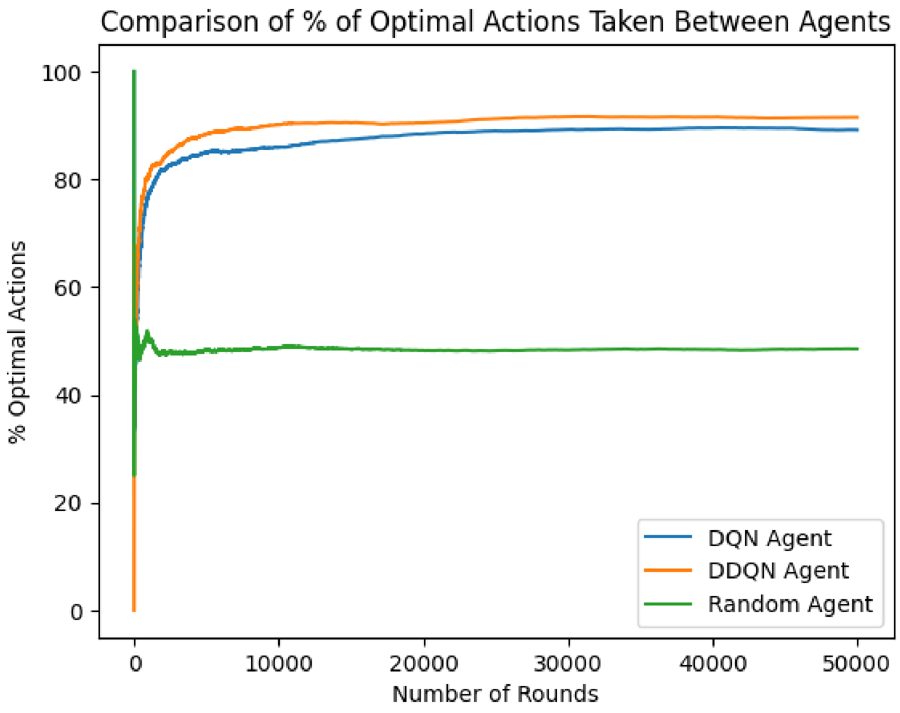

Figure 2: Comparison of % of Optimal Actions Taken Between Agents

-

Agent	Win Rate	Times optimal chosen
DQN	43.02%	89.14%
DDQN	43.64%	91.48%
Optimal	42.45%	100%
Random	28.42%	48.50%

Table 4: Agent Results After 50,000 Rounds

4.1 Discussion of results

In order to determine the success of the Deep Q-Learning agents, we compare the moving total score with two benchmark agents. First, the ‘Random Agent’ simply randomly chooses to hit or stick, whilst the second ‘Optimal Agent’ chooses its action based on Blackjack basic strategy. The Double Deep Q-Network agent is the best performing agent: Figure 1 shows that for 50,000 rounds, the DDQL agent finishes with the highest moving total score relative to the other agents with different policies, a result corroborated by Table 4 where DDQL has the highest win percentage rate of 43.64%.z

This expected result can be explained through Figure 2: the DDQL agent is able to learn a Q-Function that approximates 91.48% of Blackjack basic strategy – a greater extent than the DQL agent, who learns 89.14%. This therefore results in the DDQL agent achieving a higher win rate. Since the win rate for the DDQL is higher than the optimal agent, yet its Q-function only approximates 91.48% of the optimal agent, it could be theorised that the DDQL has learnt its own strategy for the game that slightly outperforms the blackjack basic strategy.

5 Exploration vs Exploitation Trade-off

The balance between exploration and exploitation is an important aspect of reinforcement learning in various contexts, including Deep Q-Learning. In the context of Blackjack, the agent’s main goal is to maximise its expected return. However, the agent has no knowledge of the Q-value function at the start of the game, and therefore needs to balance exploration and exploitation to determine the best action to take in each state. In this context, exploitation involves taking the action with the maximum Q-value (considering the current state), and exploration involves taking actions that do not necessarily have the highest Q-values (by randomly choosing whether to hit or stick) in order to gain a better understanding of the reward distribution for each action in each state.

One possible exploration method is the epsilon-greedy approach; this selects the action with the highest Q-value with probability $1 - \epsilon$, and selects a random action with probability ϵ. This approach balances exploration and exploitation by allowing the agent to explore randomly with a small probability whilst still exploiting the action with the highest Q-value the majority of the time.

In our implementation, we decided to build on this approach by using a sigmoid-decaying epsilon value as the number of rounds increased, until bottoming out at a pre-defined minimum value. At each round the agent follows the previously explained epsilon-greedy approach, but the varying epsilon value means the agent explores more frequently during earlier rounds, allowing it to build up a better knowledge of Q-values, and exploits more frequently in later rounds, utilising this knowledge. The minimum epsilon value ensures that even over many rounds there remains a small degree of exploration, which we found to be a beneficial addition from our hyperparameter tuning. We found this approach to be successful at balancing exploration and exploitation, allowing the agents to find the most optimal strategy for the problem at hand.

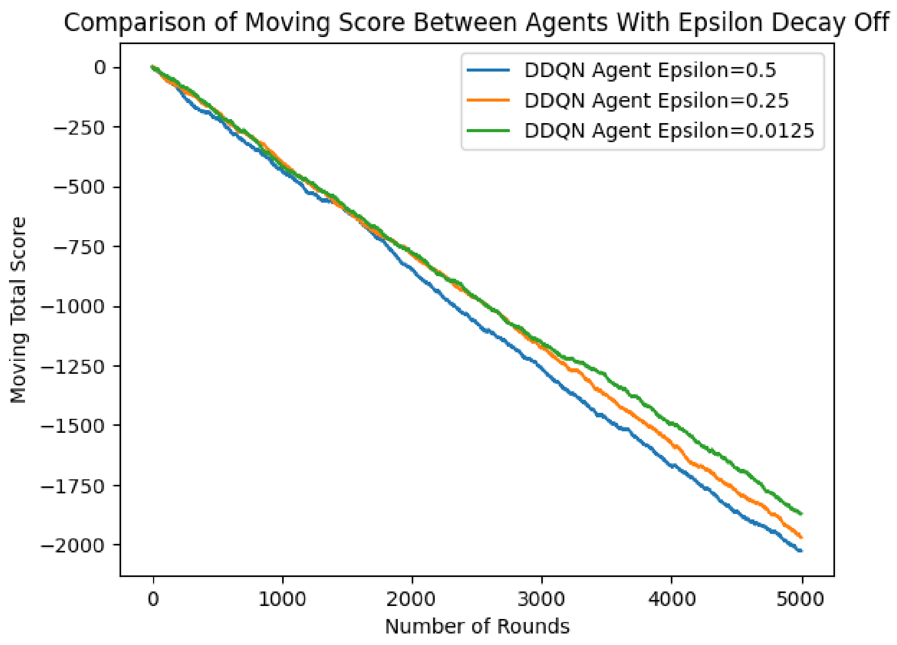

Figure 3: Comparison of Moving Total Score Between Agents

The above figure shows an output plot from our hyperparameter tuning function. In order to find the best combination of epsilon related hyperparameters (starting epsilon, epsilon decay rate, and minimum epsilon) that resulted in the most optimal balance of exploration and exploitation for our Agent, we initialised 3 Agents, each with different values for a selected hyperparameter, and ran the `play_rounds()` function for 5000 rounds. This output plot then displays the moving reward for each Agent, allowing us to easily compare which value looks to be most optimal, and incorporate this into our final Agent. From the above plot we can see that with epsilon decay off, the starting epsilon value on 0.0125 results in the optimal balance of exploration and exploitation in the long run.

6 Reference List

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. [online] Available at: https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf.

Van Hasselt, H., Guez, A. and Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. [online] Available at: https://arxiv.org/pdf/1509.06461.pdf.