**Introduction to Reinforcement Learning and Acrobot v1**

Reinforcement Learning involves machines and software agents to automatically observe the proper behaviour of a specific context to best optimise its performance through trial-and-error. The autonomous agent observes a condition in the environment and responds to it by acting, with the goal of maximising its reward. In a reinforcement learning model, the reasoning behind an action that the agent takes is called the policy, notated as , where denotes the parameters of the neural network, and and refer to the action and observation of each timestep. We chose to implement the Acrobot game which is centred around a robotic arm composed of two joints, two links where the actuated joint between the two links is used. The game’s initial state begins with the links dangling downward. The task's objective is to raise the lower link's end to a certain height in an upright position. The Acrobot's arm learns to navigate and build enough momentum for its two-link inverted pendulum system to swing the second link of its arm from its dangling start position to an upright position and keep it there. To complete the objective of the task, the Acrobot robot develops momentum by swinging itself backward and forward to move upright. In reinforcement learning, the robot arm is considered the agent who observes the environment and decides on action. As it makes interaction, the environment shifts to a new state. This returns an observation and reward for the action, reflecting the action’s efficacy in achieving the objective. Actions resulting in greater or optimal rewards are reinforced in further succession until arriving at the terminal state. A series of observations, action, next observations and rewards forms a single episode. Therefore, if the session is running for a longer return, this signals that the agent is performing optimally by acquiring maximal returns.

Using PyTorch, we implemented Deep Q-Learning which uses a deterministic estimate policy focused on future return and is an off-policy learning method. It centers around an observation array that receives four observations from the array, of which a hidden layer passes in where the inputs are finally consolidated into a single neuron. The state is the condition of the agent upon interacting with the environment. At each state, there are six possible information pieces that can be extracted from observations; the angular velocities of the first and second joint alongside the sine and cosine of the two rotational joint angles each to determine the speed and momentum of the joint swing. Within a state, the agent’s action values are dependent on the joints it decides to apply torque on. A positive or negative torque may be applied by the robot to the first joint and a positive or negative torque can be applied to the second joint. The robot's second link may be swung in various directions using these torques. The torques cause a swinging motion to the robot’s second link enabling a range of motions to sway using the torques. A state, signifying the arm’s position and angular velocity, is returned when the agent performs an action in the game environment. The agent begins at a position and velocity of 0, before deciding on an action. After learning the optimal parameters, it builds an approximate representation of the value function. Then, the agent’s performance evaluation occurs for each state to assist in supervised learning and improve its decision-making process. Ideally it would be optimised when a torque is applied to stop the second link from swinging backwards and stabilise it in the upright position when the robot has gained enough velocity to swing it over the pivot point.

**The Action Space and Reward Function**

The action space of the Acrobot’s environment pertains to the distinct collection of actions, where each action is directly related to the exertion of tension on the robot's two joints. The precise mapping of actions to torques in the action space is defined by three discrete values; **-1, 0,** and **1.** Depending on the action the robotic arm chooses to act upon, an action value of +1 will be given to reinforce positive torque actions. Conversely, it is punished with a -1 negative value in each time step. An action 0 represents cases where zero torques are assigned to both joints or when the agent decides to not take an action. Since the agent provides no torque to either joints in this implementation, action 0 stands for performing no action, allowing the robot to continue moving at its present speed and momentum.

The reward function returns a number between -500 and 0, with -500 denoting the lowest possible reward as the environment is precedented by running for 500 time-steps for each episode. The agent is not given specified instructions on how to carry out the task, nor given any heuristic hints. It relies on information about its environmental state, its position, and velocity, to improve its policy in order to maximise the total reward it receives, which is the total reward sum for each episode.

The robot arm learns to optimise its decisions that generate tension to the two joints to develop the appropriate amount of velocity where a reward is assigned to further its goal of controlling the arm to an upright position and stabilise the position using torque on the second link. The reward is quantified as a value to discern how well the action was performed, where the cumulative total of all expected future rewards updates the value in a given state. The agent’s experience from observing the environment’s information is fed into a multi-layer perceptron, a fully-connected network. In return, it obtains information about the status and reward to the environment. This includes six alternative states; the angular velocities of the joints, the sine and cosine rotational angles at each joint, torque to dictate the speed and momentum, and a boolean variable to signal if the objective function was reached, alongside the three possible actions values associated with reward values.

The network is continually trained to anticipate the expected value of the next action based on the current input state. Typically, the chosen action carries the max expected value after calculating the expected value of the future states. However, with the implementation of policy gradients, the highest probability is not automatically chosen but rather there is a sample probability to explore new actions or exploit already-known actions. This is factored in for cases when the value function is incorrect and does not significantly impair the model's ability to learn further. Exploitation (taking the action with the highest value) and exploration (performing random activities) are therefore balanced decisions, which we aim to reinforce in our algorithm.

**DQN Algorithm**

The environment step() function returns any useful objects relevant to the agent, which is the joints of a robotic arm. The Acrobot environment contains expectations over the changes in stochastic transitions by employing -learning, a model-free reinforcement learning technique, to enhance the learning of optimum policies.-learning is an off-policy algorithm where a policy is learned by approximating the -values of the optimal policy, the action with the highest value without randomising. Every episode consists of experience transitions which are compounded together by state, action, reward, and future state. This is stored in the replay memory, a buffer which the neural network randomly samples from. A sampled experience enters the network and forms . The core foundation of the Q-learning function is that a policy can be created to maximise rewards by configuring a return function where an action is taken within a given state.

In the equation, the learning rate () determines the significance between the old and learned value. The neural network uses the current state as an input and generates -values for every action that may be taken based on the parameters which get updated based on the information received by the agent’s state . Then, a future state is fed into the network forming .

**Update Rule**

The training update rule ensures that every function complies with the Bellman equation, used for reinforcement learning to relate the value of a state or action to the values of its succeeding states or actions. This conveys how the value of a state or action is equivalent to the sum of the immediate reward acquired plus the expected value of the potential future rewards from the next state or action. Throughout the learning process, the algorithm updates its predictions by observing the environment to gauge the values of states or actions, with the objective to optimise its reward. The learned policy is computed for every timestep (s’) and future action (α’).

To evaluate the optimal actions, the target networks calculate the target value that gets subsequently assigned in the -learning updates in a separate duplicate of the neural network. The technique uses repeated updates of its estimates acquired from its observed rewards and current state, where an estimated decision for the best action-value function is chosen . The target network, therefore, is set as which is used to compare the performance of the agent’s current behaviour . The discount factor establishes the significance assigned to future rewards. In this instance, the environment is not known, but because neural networks are universal function approximators, they can be trained to behave identically to as the agent decides what is the best approximate action to take. Experience replays the neural network by randomly sampling from a buffer containing the agent's experiences (state, action, reward, and future state) to enhance the agent’s learning with backpropagation, where gradient descent is implemented to reduce the squared distance between and , the discounted value of the optimal value for future states. This is then used by the agent to improve its chances of receiving the optimal reward objective.

We created a feed-forward neural network that accounts for the variation between the current and prior batches. First, we specified the first hidden layer which will take in the batch sample and output a hidden number, with an activation function rectified linear unit. This enables the agent to approximate a real state action value as an output for each state, where a mean square error loss is implemented. The second layer will take in the hidden layer created previously and the same parameters as before. It generates two outputs, and , where represents the network’s input in order to predict the expected reward of each action with the information it currently has. To retrieve the -value of the reward in the given state, the neurons are activated by the identity function. This outputs a real number for each action to signify the expected total reward.

Our optimize\_model function computes for one optimization timestep. It extracts a single batch sample, concatenates the tensors together into a single vector to calculate and which merge them together to form the loss.

We set a loop to train the model for 1000 episodes, while the step size is the number of timesteps for each episode. The average step is an empty list, containing the results of each performance action to see the average performance of the neural network. For every episode, the observations of the environment are reset before we calculate the actions resulting from the probability sample for each timestep. We implemented a relay memory to the training of our DQN, which stores the previous experiences of the agent’s interaction with the environment for each loop. It returns four outputs; state-action-reward-next state tuples, necessary to approximate the -values. The loop restarts when the episode is complete and the model terminates. By using random sampling of batches, the agent’s transition batch becomes decorrelated from any presumed consecutive experiences and is used to update the parameters with stability. In our code, this consists of two classes: transition and relay memory. The transition tuple denotes the single transitional shift an agent makes in the environment. This transitional shift outputs a (state, action) pair which is mapped to the (next\_state, reward) output. The relay memory is a cyclic buffer of the bounded size used to store current observation transitions. The function includes random.sample() method which chooses a random batch of transitions to train.

**Results**

**Rewards of the model while training**

The agent was trained on the Acrobot-v1 environment for 1000 episodes on a GPU-enabled machine. Our DQN multi-layer perceptron comprises six fully connected layers, each with 64 units and ReLU activation. We set our hyperparameters at a learning rate of 0.001, the probability of selecting a random action using an exploration rate EPS\_START, set to 1.0, which depreciates linearly until it reaches 0.01 EPS\_END, as it is being monitored by EPS\_DECAY rate of 0.001. The plot\_durations plot the episode lengths of the average reward claimed over 2000 episodes. The closer the discount rate is to 1, the more weight future rewards will acquire. Whereas if the discount rate leans closer to 0, future expected rewards perform poorer than immediate rewards. As we are training over several episodes, we set our batch size of 64 and discount rate to 0.99, which gives future rewards more weight and aids the agent in learning to optimise long-term benefits. The experience (state, action, reward, next state) is stored in a replay buffer, and a random batch sample is extracted to update the -function. The target -value used for each update is computed using the target network, a copy of the -network, updated every 5 episodes. The network parameters, TARGET\_UPDATE\_WEIGHT, are then updated using an optimizer with a weight of 0.05 to improve the gradients of the loss function given by the parameters obtained from backpropagation. We chose to implement the Adaptive momentum (Adam) optimiser, to simplify the convergence rate. It assigns each parameter's learning rate during training, while also combining momentum-based updates by leveraging the average of the gradients’ second moments, the gradient’s uncentered variance, alongside the averages of the first moment. It is designed to handle optimisation involving non-stationary objective functions that change across different parameter spaces by adjusting the learning rate and momentum estimates. This makes it resistant to outliers when the estimates become noisy or encounter delayed reward issues. Each episode's total reward that the agent received was kept track of during training and plotted below. The agent was then put through 10 episodes during the testing stage to evaluate the algorithm’s generalised performance.



We can observe that at the beginning of training, the algorithm commonly takes random actions to learn and explore the environment to gauge the best rewards. As the agent claims more rewards from interacting with the environment, the -values are updated while training. The agent attains its goal once it makes accurate -value estimations for all states and actions to optimise its long-term cumulative rewards. In this case, the agent begins with a high value which diminishes over time until convergence will enable the agent to explore during the initial stages before progressively moving towards utilising the knowledge acquired to develop the best policy to exploit rewards in the long run better. The algorithm alternates between selecting an action and randomly sampling one uniformly.

During training, the agent improved its reward by episode ~100, by claiming an average reward of below -100. The reward accrued from training began to reach convergence after 100 episodes, with some deep drops, especially in episodes 329 and 496 where the agent was punished with rewards -500 and -130 respectively. This is indicative of the agent choosing to explore its options, first taking an action that is extremely far from its current policy and being punished with -500 rewards after already reaching stable convergence episodes prior. The average reward overall during training was -102.537, indicating that the agent had converged to a good policy before the end of training.

During testing, the agent was able to successfully reach its end position after 250 episodes, resulting in an average reward of -59.90559999999998 for each episode. The best reward it received early in episode 41, receiving a reward of -47.799999999999955. Additionally, the heaviest reward punishment was -102.19999999999976, slightly above -100. This indicates that the agent is deciding not to deviate away from the state action policy it identified, opting for exploitation.



**Conclusion:**

In conclusion, we have presented the implementation and evaluation of a reinforcement learning agent for the Acrobot environment using DQN. The agent was able to successfully learn to swing the joint in a way that his arm remains in an upright position using a neural network to approximate the Q-function. Additionally, we discussed the outcomes of the agent's testing and training in the Acrobot-v1 environment. The model was trained and tested using a specific reward function and the rewards achieved during training and testing are rewards not consistent and can vary between different runs. The initial reward during training was -500, which means that the model was initially performing poorly. However, after training, the average reward increased to around -100, indicating that the model improved its performance. The trained model was then tested again for 250 episodes, the average reward further increased to around -60, which is a significant improvement from the initial reward during training. This suggests that the model has learned from its training experience and is able to perform better when tested on a new set of episodes. Further analysis, such as testing the agent on different environments or comparing it to other reinforcement learning algorithms, would be necessary to fully evaluate the performance of the agent.

**Question 2: The Agent’s Exploration and Exploitation Policy**

Exploration is a policy where the agent attempts new actions that has not previously attempted in order to learn more about the environment and potentially better strategies for maximising reward. Exploitation is the process of performing the best-known action based on the information gathered thus far in order to maximise the reward. In other words, exploitation is the use of learned information to make decisions.

Contained within the environment are information-specific characteristics considered ‘environment observations’ where the step function is called, and action is passed to obtain the observation used to update the total reward for the current episode. The transition stores (state, action, next state, reward, done) pertain to the current position, velocities, and the state of the game. Pertaining to the Acrobot game, this information manifests as the velocity and the current position of the arm where the reward is another object returned from calling the step function. This is the reward sum accumulated from the agent’s previous action. The agent is motivated towards its goal of increasing the reward level subject, to the scale of its surrounding environment. Additional to the objects returned, ‘Done’ is a boolean variable passed into a while loop as actions are being passed for a set number of iterations. This indicates whether the environment needs resetting. In this instance, this can signify vaguely that the arm is not in the correct upright position or during the game-over stages when the agent reaches the terminal timestep for the particular episode. The done variable would pass as true to signal the agent to start over and reset. The epsilon represents the agent’s policy to optimise its goal of collecting the maximum rewards. By selecting random action, the agent can use exploration to increase current environmental information. Exploitation, on the other hand, chooses the "greedy" behaviour (epsilon-greedy policy) that will yield the most benefit which is initiated through select\_action.

The algorithm decides on an action based on the highest -value present of the two using greedy-action, which then gets updated at time *t*, . While training, the algorithm operates an greedy policy when simultaneously choosing actions, with the notion of picking the max reward. At -probability, the agent chooses to ‘explore’ by understanding the environment better or resort to implementing a greedy policy (1-) until it reaches its convergence target . As the value decays the longer the algorithm learns, the agent typically becomes dependent on the learned -values (0 ) and less likely to explore (1 ).

A discount rate or target -value helps adjust the learning parameters that determine the agent action’s effect on the rewards throughout its history performance, instead of only considering the reward for a single previous action. This mitigates issues regarding the assignment of credit problem when its difficult to discern when actions are given credit when the agent obtains a reward or is punished at time *t* for instances where the agent’s actions lead to a prolonged or have extended consequences on the final total outcome of reward it receives. For instance when the agent takes a step that results in a short-term reward but ultimately identifies the correct torque values to swing and maintain the arm at an upright position over the long run. The algorithm is designed to train a policy to optimise the discounted cumulative reward, evaluated by the return .

A separate target network is used by the algorithm to predict the prospective future reward for every state-action pair. Target networks compute the target -values to update the -learning rule and store the updates to a copy of the main neural network. It adds the immediate reward with the product obtained from multiplying the discount factor with the maximum -value for all possible actions in the next state to form the target -value . In calculating the discounted cumulative reward, the relative weight of immediate rewards against future rewards is determined by the discount factor. Future rewards are assigned more weight when the discount factor is greater, which incentivises the agents to collect rewards sooner rather than considering temporarily distant incentives as equivalent to each other. The immediate current state reward , is multiplied by the discount rate, then the next reward with a discount rate to the exponent of n+1

for the next given state.

Determining the optimal discount rate is solely reliant on whether the actions would cause short or long-term effects. With a discount rate skewed towards 1, the agent would expect to perform optimally on long-term actions, which places more weight on future rewards. With a discount rate of 0, this is assumed that immediate rewards are more important than future rewards, ideal for lower training intervals

To reduce prediction inaccuracy, we applied the mean squared error to calculate the difference between the predicted -values based on actions derived from all the rewards accumulated prior to the action, with the target -value. The immediate reward at the current state action is added to the discount factor that establishes the proportional importance of immediate rewards compared to future rewards , which get subtracted from the targeted -value. When the agent moves to a different state and receives an immediate reward after interacting with the environment, calculating the mean squared error returns a single scalarscore for a state-action pair. This value is used during the backpropagation stage to update the learning parameters, by computing the stochastic gradient descent (SGD) or "mini-batch gradient descent" where the neural network is trained to reduce the distance given by mean squared error. Specifically, our code utilises the Mean Squared Error loss function to calculate the gradient of the loss function between the predicted and targeted -value. It is implemented over a batch of transition, , derived from a sample of the replay memory buffer. The Mean Squared Error, implemented as nn.functional.mse\_loss calculates the mean squared error between two tensors; the current -value and the target -values.

where N is the batch size, and the current\_q\_values are the -value generated by the Bellman equation for the target state and action, whereas the target -values are the values produced by the policy network for the current state and action. A negative MSE score would indicate that present -value estimates are, on average, lower than the target -value estimates, indicating that it is not yet capable of reliably estimating the true values of the -function, otherwise it is underfitting. This calls for more hyperparameter tuning or restructuring the neural network. The policy network should be trained to minimise the MSE loss between the current Q-values and the desired Q-values in order to increase the accuracy of the Q-function estimates and increase potentiality of optimising the agent’s behaviour. Therefore, training the model over several timesteps ensures a better estimation of good actions taken by the average. For added stability to the learning process, the -values are normalised by subtracting the average of the -values accrued across all state-action pairs and dividing by the standard deviation for all -values.This streamlines the -values into a standardised normal distribution with a mean of 0 and standard deviation of 1, enabling the training time to render significantly better in complex environments over several timesteps.

| Name | Email Id | Student Id |
| --- | --- | --- |
| Vamshi Krishna Brungi | sgvbrung@liverpool.ac.uk | 201648205 |
| Tevy | [sgtkuch@](mailto:sgtkuch@liv.ac.uk)[liverpool.ac.uk](mailto:sgktallu@liverpool.ac.uk) | 201667837 |