A Comparison of Deep Q Networks

Dylan Trollope

Supervised by Prof. CJ Watkins

2022

1 Abstract

Inspired by Google DeepMind's *Playing Atari with Deep Reinforcement Learning*[1], this project examines how Deep Neural Networks can be used in conjunction with Q-Learning to solve Reinforcement Learning problems - a form of learning called Deep Q Learning. We present the challenges of applying Deep Neural Networks to traditional Q Learning problems and how these challenges are overcome. A base Deep Q network is presented in the CartPole environment and iteratively improved upon to demonstrate the effects of different architectures used to solve the problem presented by CartPole. The network takes as input the state information currently available to the agent and outputs the action which the network predicts to maximise the value function of the agent. A set of experiments is done to find the best parameters for the Deep Q Network and performance using these parameters is recorded. We observe the dynamics of the agent in this environment, to understand the relationship that exists between the exploration and exploitation of the agent compared to the values of states. We observe that mechanisms such Experience Replay and making use of a *Target Network*, the agent is able to successfully learn where good states exist in order to solve the environment.

We also examine the effects of "reward engineering": incentivising the agent to stay in states that may be thought of as *ideal* states - states if in which the agent continually remained, would solve the environment.

Try to apply the same architecture to other environments to see how generalisable the network is.

Contents

List of Tables

List of Figures

2 Introduction

Reinforcement Learning (RL) is learning from interaction - a type of learning in which an agent (or learner) seeks to reach a goal state within an environment. Rather than being told which action to take, the agent discovers which action is best to take, given its current state in the environment, by maximising some reward signal or function.

Recent advances in the field of Deep Learning have made it possible to extract high level features from high-dimensional input data, such as the creation of the convolutional neural network for image recognition and the transformer for speech recognition and language processing [references].

The fields of Deep Learning and Reinforcement Learning can be combined into a type of learning called Deep Reinforcement Learning, which has shown to give agents the ability to make decisions from large amounts of input data (such as a snapshot of an Atari game) in order to maximise their reward in order to reach a goal[1].

In this report, several Deep Reinforcement Learning architectures, known as Deep Q Networks, will be implemented and compared in a number of Environments available in OpenAI's Gym library[2]. The aim of these comparisons will be to experiment with and discuss the performance of these networks in different environments and understand why these differences might occur.

3 Literature Review

3.1 Reinforcement Learning: An Introduction

Reinforcement Learning: An Introduction[3] introduces reinforcement learning as learning from experience and rewards. It frames learning problems through the lens of dynamical systems, specifically learning to control Markov Decision Processes (MDPs) that exist in incomplete and stochastic environments. For the purpose of this project, it is used to formalise the idea of reinforcement learning problems as series of MDPs that terminate under specific conditions. Furthermore, this book is used to introduce the ideas of goals, rewards, policies and (Q) value functions and how to calculate and optimise for these functions. We also use the book to examine dynamic programming in form of value iteration and how we can use its concepts as the foundation for allowing an agent to learn in an environment. Lastly, we take a look at off-policy Temporal Difference (TD) learning - specifically Q Learning, in which the learned Q function directly approximates the optimal action-value function, regardless of the followed policy.

3.2 Playing Atari with Deep Reinforcement Learning

This paper[1] is the main inspiration for this project. It presents a cutting-edge model that was generalised to be able to ply several Atari games to an expert level by combining the fields of Reinforcement Learning and Deep Learning, known as "Deep Reinforcement Learning." This paper discusses the challenges of applying Deep Learning to Reinforcement Learning, discusses the Convolutional Neural Network architecture used during the training process and importantly, draws attention to Experience Replay as the mechanism by which learning is enabled and optimised. It's also observed that a similar problem had been solved using similar ideas: *TD-gammon*, an agent that learned to play backgammon using reinforcement learning and self-play (the act of learning to play a game by playing against yourself), was able to play the game to an expert level. Similarly to the work presented in this report, TD-gammon used a model-free reinforcement learning algorithm similar to Q Learning to achieve such high level play.

3.3 Notes on Q Learning

Notes on Q Learning was used to consolidate my knowledge of Q Learning, specifically the maths underpinning the ideas of rewards, return, value functions and the how the Bellman equation can be used to understand when a policy is optimal. It also helped in understanding how Q Learning differs from other TD methods and importantly, noting that the agent need not be aware of transition probabilities or expected rewards given its current state.

4 Introduction to Reinforcement Learning

4.1 Components of Reinforcement Learning

Besides the agent and environment, there are several components that make up a whole reinforcement learning problem[3], namely:

- A Policy dictates the agent's decision-making process at any given time. A policy is roughly a function that maps perceived states in an environment to actions to be taken when in those states. Policies can range from simple functions or choosing a value from a table to more complex, computationally expensive functions that may include Neural Networks. In general a policy will, given a state, assign probabilities to the actions to take in that state.

- A reward signal: At each time step, the environment sends the agent a reward for the action it has taken. The reward signal defines the goal of a reinforcement learning agent - the agent seeks to maximise the reward it accumulates over time.

- A value function: While the reward signal indicates what are *good* actions to take in the short tern, the value function tells the agent which actions are most valuable in the long term. In RL, the value of a state estimates how much reward the agent will accumulate in the future, starting from that state. The value function takes into account the states that are most likely to follow the current state and the rewards available at those states. For example, tt is possible that a state yields a low immediate reward but offers a high value as the states that follow all yield high rewards. The reverse may also be true.

  Values may be seen as secondary to rewards, as without rewards there are no values to estimate but the function of the value is to achieve more reward. Yet, it is the value function that must be considered when deciding which action is best to take as this will maximise the reward the agent receives over the long run, and ultimately bring the agent closer to the goal state. It will become clear later then, that finding methods to accurately estimate the value function is crucial in succeeding in RL problems.

- Lastly, RL problems may be comprised of a model of the environment: This is used to mimic the environment the agent finds itself in and can be used to make inferences about how the environment will behave. This allows predictions to be made about the environment: given a state and an action, the model may be able to predict the next state and reward from that action. This allows the agent to *plan* - the agent may be able to take a course of action made from these predictions rather than only estimating what the next best state may be.

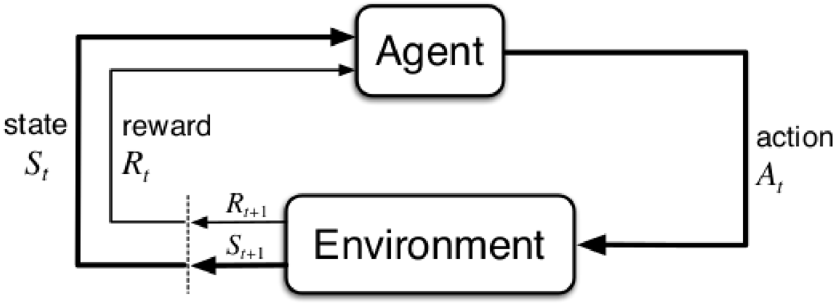


Figure 1: The Agent-Environment Interaction Cycle[3]

4.2 Markov Decision Process

Markov Decision Processes (MDPs) can be used to formalise sequential decision making.

Given a process is in a state s, the *decision maker* within the process chooses an action a that is available to it in state s. At the next time step, the process moves into the state, s', as a result of action a from the previous time step and also gives the decision maker a reward, $R_a(s, s')$, corresponding to its new state.

The probability that a process moves into state s' is dependent on the action chosen at the previous state, and is given by the state transition function $P_a(s, s')$ - the current state depends on the previous state s and the action a taken in the previous state.

Formally, an MDP can be defined as a 4-tuple $\langle S, A, P_a, R_a \rangle$, where:

- S is the set of states - the state space
- A is the set of actions - the action space, where A_s is the set of actions available in state s.
- $P_a(s, s')$ is the probability that action a in state s will lead to state s'
- $R_a(s, s')$ is the reward received from after transitioning to state s' from state s after taking action a

4.3 Interaction between Agent and Environment

In RL, the agent (or *learner*) is placed inside an environment, comprised of everything outside the agent. The agent selects varying actions to perform, which present new situations (or states) to the agent. Every action taken in an environment also produces a reward - a numerical value that the agent aims to maximise over time by the selecting the *best* actions[3].

Formally, the agent and environment are in interaction with each other in a sequence of (discrete) time steps, $0 < t < T$. At each time step, t, the agent receives partial (or full) information about the environment's current state, $S_t \in \mathcal{S}$ and selects an action a at time t from the complete action space, $A_t \in \mathcal{A}(s)$.

At the next time step, $t+1$, the agent receives a numerical reward as a result of the action it took at the previous time step, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and is then in a new state, S_{t+1}.

Together, the MDP and agent-environment interaction form a sequence as follows:

$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3 \cdots$

4.3.1 Rewards and Return

Over the Reinforcement Learning process, rewards are accumulated as the agent takes actions in sequential states of the environment. These rewards can be positive or negative.

In Environments where rewards are positive, the agent most likely aims to maximise these rewards. In Environments where the rewards are negative (also referred to as costs), the agent may seek to minimise these rewards (costs)[4].

In an MDP, the agent seeks to maximise its **return**. The return may be thought of as an *average reward* over a long run of decisions. The maximisation of the return will be considered the *goal* of the agent, which is a distinct feature of reinforcement learning, compared to other forms of learning.

Given that an agent receives rewards at sequential time steps,

$R_{t+1}, R_{t+2}, R_{t+3}, R_{t+4}, \cdots, R_T$, the simplest return of the agent can be calculated as:

$$\rho_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} + \cdots + R_T \tag{1}$$

This return function is used when the RL problem is episodic in nature as it expects a final state at time T, which is the terminal state.

As can be seen from above, the return function is linear - immediate rewards and future rewards are *weighted* equivalently, there is no incentive for the agent to take actions that would maximise reward over the long run or the short run, which for certain environments may be desirable.

By introducing a discount factor into the return function, the agent can account for maximising more immediate or future rewards[4]:

$$\rho_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2}$$

where $0 \leq \gamma \leq 1$ is the discount rate and determines the value of future rewards that the agent receives. If the agent receives a reward k time steps in the future, it is only worth γ^{k-1} times what it would be worth if it were received immediately.

With γ close to 0, the agent is very short-sighted and only concerned with maximising immediate rewards - it will choose an action A_t to only maximise R_{t+1}.

With γ approaching 1, the return function takes future rewards into account more significantly and the agent is more farsighted.

Using equation (2) above, it can be shown how return functions at successive time steps are related[3][4]:

$$\begin{aligned}\rho_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ...) \\ &= R_{t+1} + \gamma \rho_{t+1} \qquad (3)\end{aligned}$$

γ is used to define a *soft* time horizon (Watkins, 2021). The greater the amount of time steps away from the current state, the less the return calculated from those time steps

4.4 Policy and the Value Function

4.4.1 Policy

Most RL algorithms are concerned with estimating a value function - a function that estimates how *good* it is for an agent to be in a state or how *good* it is to perform a given action in a given state. A state is *good* if the expected reward, or more specifically, expected return from that state to the terminal state is maximised[3]. Because future expected return is dependent on the sequence of actions taken by the agent in consecutive time steps, the value function is defined in terms of the behaviour of the agent, called a **policy**.

Formally a **policy** is a function that maps states to the probability of selecting the possible actions in those states and is denoted by π. In general, RL algorithms will attempt to find the optimal policy. An optimal policy is the policy that will maximise the expected return, starting in any state.

If an agent is following policy π at time t, then $\pi(a|s)$ is the probability that $A_t = a$ and $S_t = s$. The policy defines a probability distribution over $a \in A(s)$ for each $s \in S$.

RL algorithms specify how the agent's behaviour changes as a result of its experience.

Deterministic vs stochastic policies and policy space

4.4.2 Value Function

The value function of a state s under policy π, denoted $v_\pi(s)$, is the expected return from state s to the terminal state under policy π.

From the perspective of an MDP, the value function can be defined as follows[4][3]:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[\rho_t|S_t = s, \pi] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, \pi\right] \end{aligned} \tag{4}$$

for all $s \in \mathcal{S}$, where $\mathbb{E}_\pi$ denotes the expected value of return given the agent follows policy π at any time step t.

The function v_π is known as the *state-value function*.

As will be useful later, Equation (4) can be rearranged as follows[5]:

$$v_\pi(s) = R(s, \pi(s)) + \gamma \mathbb{E}_{s' \sim P(s'|s,\pi(s))}\left[v_\pi(s')\right] \tag{5}$$

where the first term in the equation is the immediate reward and the second term is the return gained from the successor states when starting at state s and following policy π.

4.4.3 Q Value

It is also possible to define the *action-value function* under policy π, which is the expected return starting from s, taking action a then continuing under policy π[4][3]:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[\rho_t|S_t = s, A_t = a, \pi] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a, \pi\right] \end{aligned} \tag{6}$$

It can be seen that the summation term above is the same as the summation term in value function in Equation (4) above, with the only difference being the action specified in the Q value calculation, i.e. The Q value can be written in terms of the value function.

Just as Equation (4) can be rewritten in Equation (5), the equation in (6) above can be represented in the following way[5]:

$$q_\pi(s, a) = R(s, a) + \gamma \mathbb{E}_{s' \sim P(s'|s,a)}\left[v_\pi(s')\right] \tag{7}$$

The functions v_π or q_π can be estimated over time:

Given an agent follows policy π and maintains an average of the expected return for each state encountered, then the average return for each state will converge to that state's value $v_\pi(s)$ as the number of times it is encountered approaches infinity[4].

If averages are kept for each unique action taken in each state, then the averages will also converge to the expected return of the action-value function, q_π[3].

4.5 Optimal Polices and Value Functions

For finite MDPs, it is possible to define precisely what it means for a policy to be optimal[3].

Value functions define a partial ordering over policies. A Policy π is defined to be equal to or better than a policy π' if its expected return is greater than or equal to that of π' for all states:

$\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$.

The *optimal policy* is that policy which is greater than or equal to all other policies - the optimal policy is not necessarily a unique policy and is denoted by π_*.

The optimal state-value function can be formalised as:

$$v_*(s) = \max_\pi v_\pi(s) \tag{8}$$

for all $s \in \mathcal{S}$.

Similarly, the optimal action-value function, q_* can be formalised as:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{9}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

4.6 Optimality and the Bellman Equation

A key problem to consider is exactly *how* to know when a policy is optimal.

Given that the optimal policy π_* is the policy that maximises $v_\pi(s)$ for every $s \in \mathcal{S}$, the optimal return function, also known as a Bellman equation can be defined as follows[3]:

$$v_*(s) = v_{\pi_*}(s) = \underset{a}{max}(R(s,a) + \gamma \mathbb{E}_{s' \sim P(s'|s,a)}\left[v_{\pi_*}(s')\right]) \tag{10}$$

$$\pi_*(s) = \underset{a}{argmax}(R(s,a) + \gamma \mathbb{E}_{s' \sim P(s'|s,a)}\left[v_{\pi_*}(s')\right]) \tag{11}$$

It is possible to define the optimal value function and optimal policy in terms of the optimal q value:

From Equation (5) and Equation (10), it can be derived that:

$$v_*(s) = \underset{a}{max}(q_*(s,a)) \tag{12}$$

and also from Equation (7) and (11),

$$\pi_*(s) = \underset{a}{argmax}(q_*(s,a)) \tag{13}$$

These equations, known as Bellman equations, must hold for every state of the RL problem and give the conditions for a policy and its associated value function to be optimal.

5 Q Learning

Now that it has been underlined that it is possible to know that a policy is optimal, *Q Learning* (Watkins, 1989) is introduced as a method to calculate the optimal Q Values and subsequently the optimal policy for an RL problem[6]. As will be shown later, Q Learning will be the main algorithm built upon to perform Deep Reinforcement learning.

Q Learning is a model-free RL algorithm that is used to learn the *Q Value* of an action in a particular state. It is a model-free algorithm in the sense that:

- the algorithm does not need to have a model of the environment to learn the best action to take in a given state, and
- the algorithm can learn Q Values without knowing what the next state is, what the expected return of the subsequent states will be or what the transition probabilities from any state s to s' will be.

It may seem intuitive that state transition probabilities need to be learned but this is not the case. Using Q-Learning it is possible for the agent to learn an optimal policy without ever learning state transition probabilities or expected rewards from states[4].

5.1 Initial Setup

The only data structure required to perform Q Learning is a table to store estimated Q values for every state and every action which is incrementally updated to approach the optimal policy.

For every state-action pair there is a $Q(s, a)$ value.

5.2 Experience

Q-Learning relies on episodes of experience to learn the optimal policy.

An experience is a 4-tuple $\langle s, a, s', r\rangle$ and describes the reward r received when transitioning from state s to s' by taking action a.

5.3 Updates

With the Q table initialised, Q values are updated until *convergence*, as follows:

```
for each episode do
    Initialise s
    for each step of episode do
        Choose $a$ from $s$ using policy from Q
        Take action $a$ in $s$ and observe $r$ and $s'$
        $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \underset{a'}{max} Q(s',a'))$
        $s \leftarrow s'$
    end for
    Until $s$ is terminal
end for
```

5.4 Off Policy Learning

Q Learning is **off policy** RL algorithm, in that it needs only the experience 4-tuple $\langle s, a, s', r\rangle$ to learn the relevant $Q(s,a)$ values. The agent does not need sequential experiences in order to learn the optimal policy.

However, to converge to the optimal Q values, the algorithm does need to explore every combination of states and actions "repeatedly and thoroughly" (Watkins, 2021). It can be seen that Q-Learning requires very little awareness of the environment it is in and the algorithm is rather simple. All that is required is to repeatedly update the Q table using the algorithm above.

5.5 Converging to the Optimal Policy

Q Learning uses approximate value iteration to find the optimal policy. This means that the learned policy "will become arbitrarily close to the optimal policy after an arbitrarily long period of time" (Daley, 2020). Using the Bellman equation as an iterative update (defined above), the value iteration algorithm converges to the optimal action-value function, $q_t \rightarrow q_*$ as $t \rightarrow \infty$. For this to occur, two conditions need to be met[1][4]:

1. The learning rate must approach 0.
2. Every state-action pair must be visited infinitely often. This is achieved as long as every action in a state has a non-zero probability of being selected.

5.6 Exploration vs Exploitation

In RL, the agent tends to take actions which it has tried in the past and found successful in adding maximising its Q value. But, in order to discover these state-action pairs, the agent must take actions that it has not attempted before. This means that the agent needs to *exploit* what it has already learned but it also needs to *explore* new actions in previously unseen states in order to maximise the Q values. The problem that an RL agent faces is that if it were to only employ one strategy, it would almost certainly fall short of its goal, and so a balance between exploration and exploitation needs to be found in order to continually approach the optimal Q values and ultimately the optimal policy.

5.6.1 Trade-off

When the agent explores, it can improve its current knowledge and gain better rewards in the long run if its exploration leads to the valuable states. However, when the agent exploits its current knowledge, it gets more immediate reward even if the behaviour of the agent is suboptimal in the long run.

As the agent can not explore and exploit at the same time, it must make a trade-off between exploring the environment (improving its knowledge in the long run) and exploitation (using its current knowledge to gain reward in the short term).

5.6.2 The ϵ-greedy Search Strategy

It is essential to find the correct balance of exploration and exploitation. Initially the agent is unfamiliar with its environment and needs to explore its surrounding states. As the agent becomes more familiar with its environment and discovers valuable states, it needs to reach those states and explore its new frontier of unknown states.

The ϵ-greedy approach selects the action with the highest estimated value most of the time by exploiting what is already knows, but with a small probability ϵ, the agent will explore the environment to observe valuable states that it has not yet discovered.

As will be seen, in practice, the epsilon value is made to decay - the more the agent has discovered about the environment, the less likely it is to explore states that it has not seen before. This is because after a long period of learning, the agent will have encountered most states and does not need to explore as much as in the initial stages of learning.

6 From RL to Deep RL

Deep Learning, a form of supervised machine learning, which uses neural networks to transform sets of (possibly high-dimensional) inputs to outputs, has shown significant progress in the fields of computer vision (such as YOLOv3 (Redmon, 2018)), natural language processing (such as BERT (Devlin et al., 2018)) and Bioinformatics in the form of AlphaFold (Jumper et al., 2021).

6.1 Applying Deep Learning to RL

Using Deep Learning, agents can make decisions from large amounts of (unstructured) input data without being aware of the state space. It has been shown possible to play games like Atari Breakout to an expert level using Deep Reinforcement Learning (Mnih et al., 2013). Using a variant of Q-Learning, later described as a Deep Q Network, the model is able to learn policies directly from input data (4 consecutive frames of Breakout gameplay) using a Convolutional Neural Network and output a value function to predict future rewards.

6.2 Challenges to Consider when Applying Deep Learning to RL

There are some challenges to consider in the application of Deep Neural Networks to problems that have been traditionally thought of as Reinforcement Learning Problems:

- Firstly, Deep Learning (a type of *supervised* learning) applications are able to find accurate, generalised models by using large volumes of hand-labelled training data. In comparison, RL algorithms tend to learn from a scalar reward function that can be *noisy* and *delayed* - delayed in the sense that the true reward for an action in a state may only be seen thousands of time steps after that action was taken. This type of delay does not exist in Deep Learning as, once the network is trained, there is an (almost) instant prediction between an input and its predicted label.

- Secondly, in deep learning, most networks are assumed to take as input, samples that are i.i.d (Independently and Identically Distributed). In comparison, RL encounters sequences of states that are highly correlated to each other - i.e. Getting to state s' from s may be completely dependent on taking action a. There may also be a distribution shift in the RL distribution as the algorithm learns new behaviours, which may present challenges to deep learning algorithms that assume the data distribution is fixed.

6.3 Neural Network as a Function Approximator

When outlining the conditions needed for Q Learning to converge, it is stated that:

$q_t \to q_*$ as $t \to \infty$.

In practice however, it is impractical to use this approach as $q(s, a)$ is estimated for every sequence (episode) and so there is no generalisation between many sequences. To remedy this, a neural network with weights θ is introduced as a non-linear function approximation in order to estimate $q(s, a, \theta) \approx q_*(s, a)$. This network is known as a Q-Network (Mnih et al., 2013).

6.3.1 Training a Q-Network

The Q-Network is trained by minimising a sequence of loss functions, $\mathcal{L}_i(\theta_i)$ that changes for every iteration,

$$\mathcal{L}_i(\theta_i) = \mathbb{E}_{s,a\sim p(s,a)} \left[(y_i - q(s, a, \theta_i))^2\right] \tag{14}$$

where

$$y_i = \mathbb{E}_{s'\sim\mathcal{S}} \left[r + \gamma \max_{a'} q(s', a', \theta_{i-1})|s, a\right] \tag{15}$$

is the target for iteration i and $p(s, a)$ is the probability distribution over sequences of states and actions, also known as the *behaviour distribution.* The parameters of the previous iteration θ_{i-1} are kept fixed for the current iteration of the loss function $\mathcal{L}_i(\theta_i)$. It is important to note that, unlike in supervised learning where targets are fixed before learning, the targets for $\mathcal{L}_i(\theta_i)$ are dependent on the current network weights.

Differentiating the loss function with respect to the weights, the gradient is as follows:

$$\nabla_{\theta_i}\mathcal{L}_i(\theta_i) = \mathbb{E}_{s,a\sim p(s,a);s'\sim\mathcal{S}} \left[(r + \gamma \max_{a'} q(s', a', \theta_{i-1}) - q(s, a, \theta_i))\nabla_{\theta_i}\ q(s, a, \theta_i)\right] \tag{16}$$

Because it is computationally expensive to compute the full expectations of the above equation, stochastic gradient descent can be used to optimise this loss function. If there are updates after every time step and the expectations are replaced by single samples from the behaviour distribution $p(s, a)$ and $\mathcal{S}$ respectively, we arrive at the Q-Learning algorithm (Mnih et al., 2013).

6.4 Model Free and Off-Policy

Just as in Q-Learning, it is important to note that Deep Q Networks are *Model Free* and adopt *Off-policy* learning.

It is model-free in that it solves the RL problem directly by using samples from the state-space $\mathcal{S}$ without explicitly learning about the environment in which the agent is placed. It is off-policy in that it learns a *greedy* strategy $a = \max_a q(s, a, \theta)$ while ensuring adequate exploration of the state space[4][1].

In practice, the behaviour distribution is selected by an ϵ-greedy strategy that follows the strategy with probability 1 - ϵ and selects a random action with probability ϵ[1].

6.5 Experience Replay in Deep Q Networks

Q-Learning makes use of episodes of experience, where the experience at time t takes the form $e_t = \langle s, a, s', r \rangle$.

In Deep Q Networks, we want to store these experiences $\mathcal{D} = e_1, e_2, ..., e_T$ in memory called an *experience replay.* When training the Deep Q Network, the Q-Learning updates are applied to samples of the experience memory, $e \sim D$, drawn at random from the store of experiences, $\mathcal{D}$[1].

Upon performing experience replay, the agent selects an action according to an ϵ-greedy policy. A fixed-length history of experience is used in order to train the neural network.

6.5.1 Advantages of Using Randomly Sampled Experience Replay

The Experience Replay approach described above provides some efficiencies during training[1]:

- Each step of experience may be used in many weight updates, providing greater data efficiency

- As sequential experiences are highly correlated, using random samples ignores this correlation, reducing the variance in training over a long run

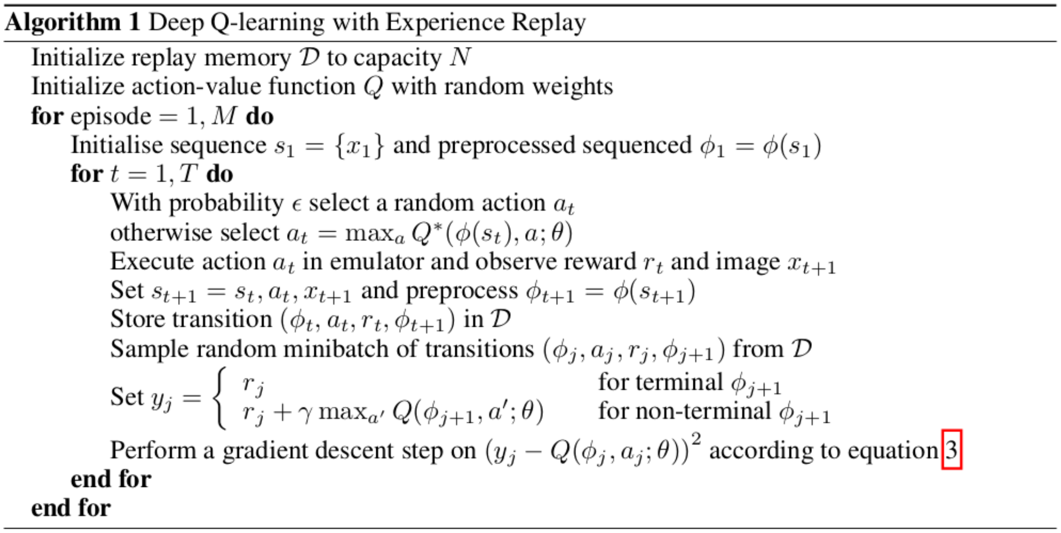
- Lastly, when learning *on-policy*, the current parameters are highly influential on the next selected sample. For example, if the *maximising* action is for the agent to move to the left, subsequent samples will be from the left hand side and likewise, if the maximising action moves the agent to the right, the next samples will be from the right. By using experience replay and off-policy learning, the behaviour distribution is averaged over many previous states and actions, avoiding big oscillations or divergence during learning[1].

6.5.2 Drawbacks of Randomly Sampled Experience Replay

Because of finite memory constraints, Experience Replay only stores the last $\mathcal{N}$ experience tuples, and tuples are sampled uniformly from $\mathcal{D}$. This means that all transitions between states are weighted of equal importance, when in practice there exist transitions that contribute substantially more to the long term return than other transitions[1].

6.6 Deep Q-Learning Algorithm

The full algorithm for Deep Q Learning is presented below



```
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 $\epsilon$ 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
```

Figure 2: Deep Q Network Algorithm [1]

6.7 Deep Q-Learning Architecture

The architecture consists of a fully connected Neural Network where:

- The input layer, i consists of x nodes, where x is the number of dimensions that the agent observes in the environment
- The second layer, h_1, consists of 64 fully connected nodes
- The third layer, h_2, consists of 128 fully connected nodes
- The output layer, o has y nodes, where y is the number of actions available to the agent within the environment.

7 The CartPole Environment

The Environment in which the Deep Q Networks are tested is the CartPole Environment[2], which entails a pole balancing atop a cart which moves in the (frictionless) horizontal plane. The pole is attached to the cart by an unactuated joint. The goal of the environment is to balance the cart on the pole for as long as possible by moving the cart from side to side. The Environment ends when the Cart moves out of (horizontal) bounds (+/- 4.8 units from the centre of the environment) or if the pole falls more than 24 degrees from the centre. The Environment will also terminate if the reward reaches 500.

The CartPole observation space is four-dimensional and is defined as follows:

Index	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-24°	24°
3	Pole Angular Velocity	-Inf	Inf

Table 1: Table of CartPole Observation Space

The action space is simple and two-dimensional:

Action Number	Action
0	Push cart to the left
1	Push cart to the right

Table 2: Table of CartPole Action Space

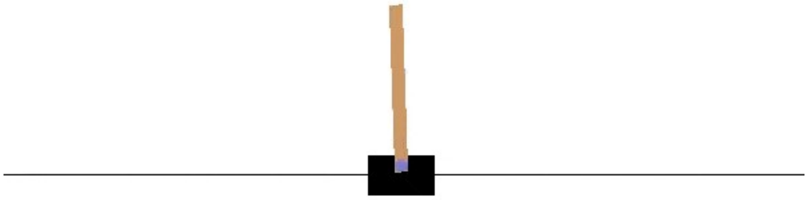


Figure 3: The CartPole Environment

To maximise the difference in architectures used, the following experimental parameters are used and kept constant between experiments:

Variable	Value
episodes	150
learning rate	0.001
γ	0.9
ϵ	0.3
decay	0.99

Table 3: Experimental Parameters

8 Experiments

The experiments that follow were implemented using a host of Python libraries. The CartPole environment is accessed through the use of OpenAI's Gym library[2]. The Neural Networks compared are created and trained using PyTorch[7]. Lastly, the experiments were run in Jupyter Notebooks[8] and plots rendered in Matplotlib[9]. Throughout the experiments section, references will be made to snippets of the implementation of the experiments to bridge the understanding between Reinforcement Learning Theory and its Practical Implementation. The code used to implement many of the following experiments was adapted from several sources, namely [1][10].

8.1 Random Strategy

To create a base line for obtained reward and to later observe the effects of mechanisms such using a Neural Network as a function approximator and Experience Replay, we observe an agent trying to solve CartPole with a *Random Strategy*. A random strategy means that the agent employs a policy where, at every time step, it takes a random action. It is expected that the agent fails completely to solve the environment, as no learning occurs and no information about prior states is used to inform its action in its current state.

```
for _ in range(episodes):
    state = env.reset()
    done = False
    total = 0
    while not done:
        action = env.action_space.sample()
        next_state, reward, done, _ = env.step(action)
        # env.render()
        total += reward
        if done:
            break
    reward_list.append(total)
```

As can be seen from the implementation above, the action at every step is sampled randomly from the action space - there is no knowledge or experience considered when choosing an action. The format of this plot is inspired by [10].

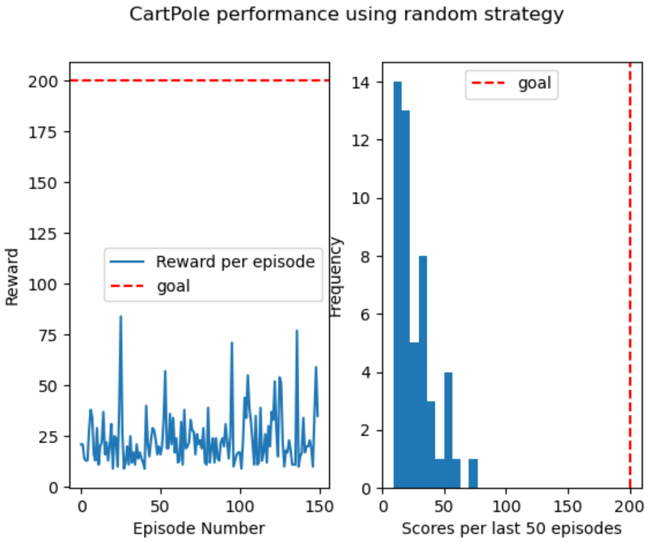


Figure 4: Reward vs Episode for the random strategy in the CartPole environment

Using a reward score of 200 as the *goal* reward, it can be seen that employing a random strategy is not effective in achieving the goal, as expected. The highest reward not being more than 100 points and so using a random strategy is not sufficient in achieving the goal state of the environment.

As can be seen from the histogram on the right, the performance of the agent does not improve over time. Rewards over time do not approach the goal reward as the policy employed by the agent does not have any learning element.

Plotting the agent's performance over 3 runs - where a run is an agent retrained to solve the environment - produces the same observations: selecting a random action at every step does not allow the agent to solve the environment.

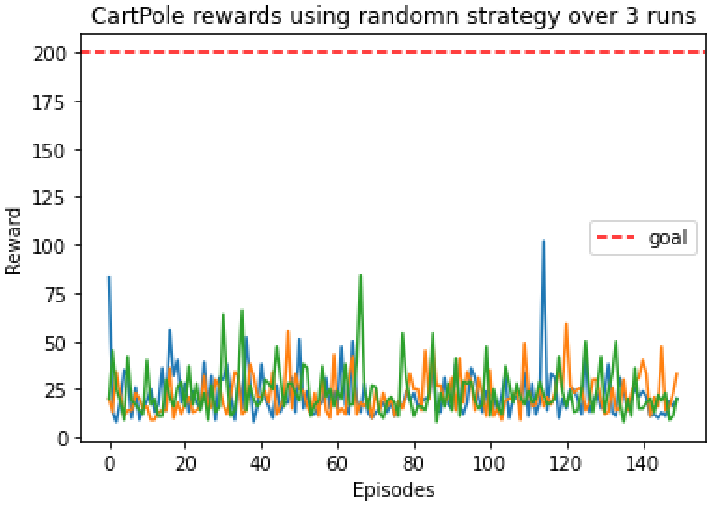


Figure 5: CartPole Performance over 3 runs with Random Strategy

8.2 Learning only from the Most Recent Action

We now introduce the Deep Q Network as the mechanism by which the agent learns, explores and exploits its environment. Before introducing Experience Replay, we observe how well the agent performs by taking only the most recent action into account when performing its current action[1].

8.2.1 The Network

```
SIZE = 64
self.nn = nn.Sequential(
            torch.nn.Linear(state_dim, SIZE),
            nn.LeakyReLU(),
            nn.Linear(SIZE, SIZE * 2),
            nn.LeakyReLU(),
            nn.Linear(SIZE * 2, action_dim)
    )
    self.loss = nn.MSELoss()
    self.optimiser = torch.optim.Adam(self.nn.parameters(), lr)
```

The snippet above defines the Neural Network used for Deep Q Learning. As stated before, it is a fully connected network that takes the observation space size as input, outputs Q Values for each action in the environment and has two hidden layers of size 64 and 128 respectively.

[1] The choice of optimiser, loss function, activation function and learning rate parameters will be motivated by later experiments

8.2.2 The Update Function

The function below is used to update the Q Network given a training sample. It computes the loss given the current state's predicted Q value and the actual Q value produced by the network and then does a weight update using the ADAM Optimiser.

```
def update(self, state, y):
    y_pred = self.nn(torch.Tensor(state))
    loss = self.loss(y_pred, Variable(torch.Tensor(y)))
    self.optimiser.zero_grad()
    loss.backward()
    self.optimiser.step()
```

8.2.3 The Predict Function

The predict function returns the Q values of the actions predicted by the network from the current state of the agent.

```
def predict(self, state):
    with torch.no_grad():
        return self.nn(torch.Tensor(state))
```

8.2.4 Taking the Most Recent Action into Account

In the training loop, we predict the Q value of the next state, then we assign the Q Value for each action given the estimate of Q values for the next state. Finally we update the model given the current state and the predicted Q value of the next state.

```
q_values_next = model.predict(next_state)
q_values[action] = reward + gamma * torch.max(q_values_next).item()
model.update(state, q_values)
```

8.2.5 Performance Evaluation

Plotting the performance of the agent over 150 episodes, the agent has been able to learn from the environment over time and slightly improve its performance - there is a clear upward trend in performance. However, it was only able to reach the goal score of 200 during one episode.

Using a Deep Q Network to remember the most recent action improves the agent's performance but given the 150 episode constraint, it has not done very well.

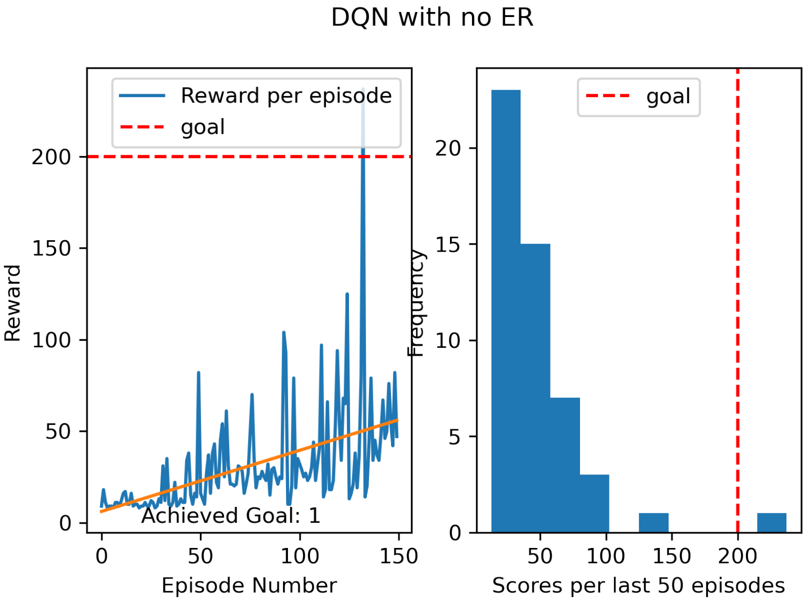


Figure 6: Deep Q Learning without Experience Replay in the CartPole environment

Evaluating the performance of the agent over 3 runs, where a *run* is the retraining of the Deep Q Network, the same agent behaviour emerges: there is an upward trend in the agent's performance but it only reaches the goal state once in all three runs.

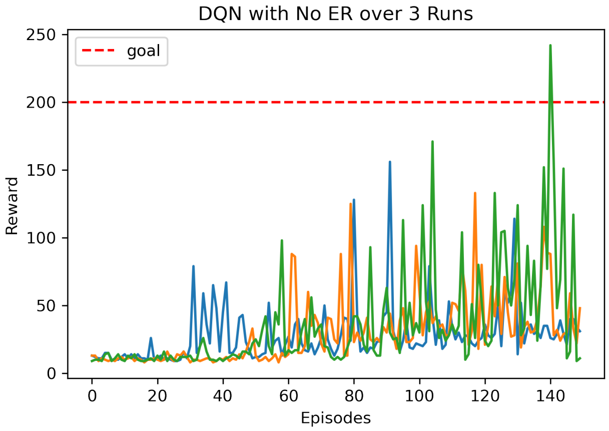


Figure 7: Deep Q Learning with no ER for 3 runs

8.3 Introducing Experience Replay

To see how effective Experience Replay is in the context of Deep Q Networks, the below experiment is conducted using a Deep Q Network with Experience Replay (ER). The implementation of ER is below:

8.3.1 The replay function

```
def replay(self, memory, size, gamma):
    if len(memory) >= size:
        states = []
        targets = []
        batch = random.sample(memory, size)

        for experience in batch:
            state, action, next_state, reward, done = experience
            states.append(state)
            q_vals = self.predict(state).tolist()
            if done:
                q_vals[action] = reward
            else:
                q_vals_next = self.predict(next_state)
                q_vals[action] = reward + gamma * torch.max(q_vals_next).item()
            targets.append(q_vals)
        self.update(states, targets)
```

When there are sufficient samples in the agent's memory, it samples experiences in order to update the Q Network. For every experience, the network tries to predict the Q value of the next state, given the state of the experience, similarly to the network without ER. Then, we assign Q values to each action using the next state's predicted Q values. We then use the states in the agent's memory and its predicted Q values to do a model update.

8.3.2 Performance Evaluation

Compared to the Deep Q Network that only samples the most recent action, the Network with Experience Replay performs significantly better. The agent reaches the goal for the first time after approximately 50 episodes and goes on to reach the goal 72 times in total, almost half of the total episodes.

It is interesting to note the main feature of the Reward vs Episode graph, in which the agent makes a clear improvement at approximately the 55th episode. From this episode onwards, the agent can be considered as having *solved* the environment. However, from approximately episode 100, there is a decreasing trend in the agent's performance. The agent revisits states that allow it to achieve the goal but not necessarily continually improve its performance over episodes.

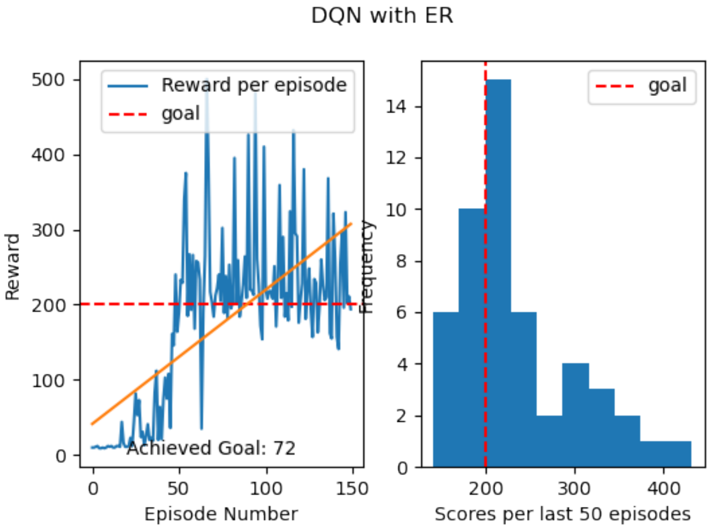


Figure 8: Deep Q Learning with Experience Replay for memory size $\mathcal{N} = 10$

From observing the agent's behaviour, this is because of two reasons:

1. When the agent (the cart in CartPole) has a ”breakthrough” episode, the steep uptrend in the performance graph, it learns to keep the pole upright near the edge of the environment. This means that any small oscillations in the movement of the pole and subsequent cart reaction often means the cart leaves the environment, terminating the episode.

2. The experiences sampled from the replay mechanism are experiences in which the agent had a low Q value predicted and so the model is (perhaps incorrectly) trained to take these actions that force the agent into lower performance scores. Later, we experiment with memory sizes to see what effect the amount of memory sampled has on the performance has on the agent.

Again, over 3 runs, the agent is able to reach the goal state 75 times on average - half of the amount of available episodes.

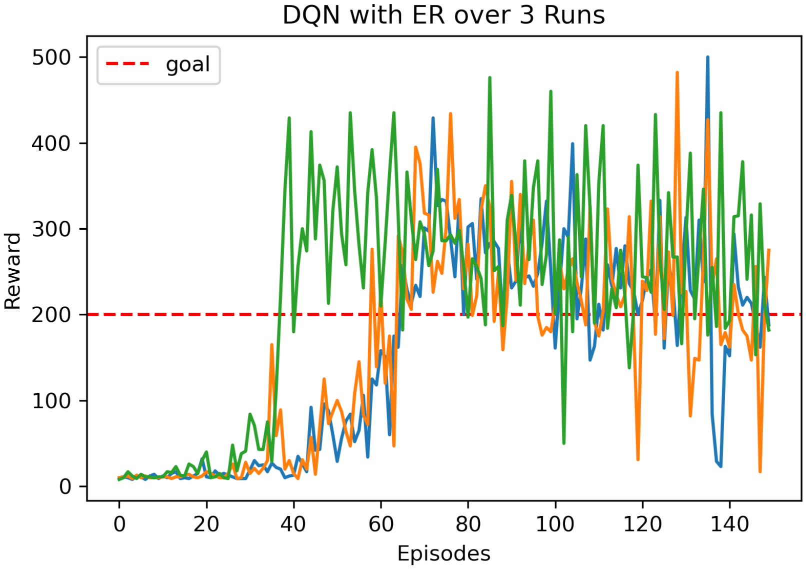


Figure 9: Deep Q Learning with Experience Replay over 3 runs

From the above experiments, it is clear to see that the agent is capable of learning to solve the environment given it uses the Experience Replay mechanism to learn.

The iterative improvement from random strategy to ”single action replay” to Experience Replay has shown improvement in the agent’s ability to learn and the agent that uses Experience Replay has the most success in reaching the goal state in the environment.

8.4 A Deep Dive into Deep Q Networks with ER

We now conduct some experiments to analyse the performance of the Deep Q Network with Experience Replay and observe how different parameters might affect the performance of the network and subsequently, the agent.

8.4.1 Starting ϵ and Exploration vs Exploitation

It is interesting to note how much exploration (choosing a new action from a given state) or exploitation (choosing actions that are known to produce high reward) the agent conducts and how this influences the performance of the agent.

For values of epsilon in the range 0.3 to 0.7, we show how many times the agent reaches the goal state[2]

[2] The parameters for these experiments are the same as in 3

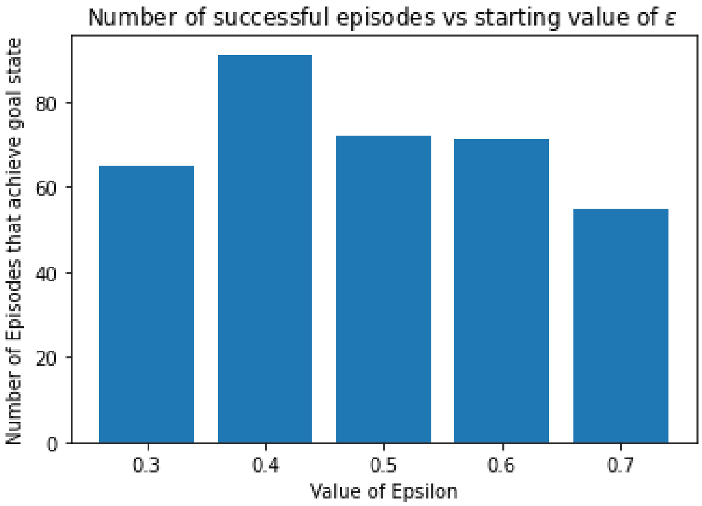


Figure 10: Exploration vs Exploitation

Value of ϵ	# of goal states reached
0.3	65
0.4	91
0.5	72
0.6	71
0.7	55

Table 4: Amount of times the goal state was reached for different values of ϵ

Although all values of ϵ produced reasonably good results, there is a clear difference in the choice of epsilon that performs best: 0.4. An ϵ of 0.4 is a *sweet spot* for finding the balance of exploration to exploitation. A smaller value - in this case, 0.3 - possibly does not give the agent enough time to explore the environment before having to rely on its knowledge to find the best states. This means that the agent has stopped exploring the environment prematurely and not encountered states that are highly valuable.

For ϵ values > 0.4, the inverse problem occurs: the agent explores the environment thoroughly but also finds states that are not considered very valuable. When these states are then sampled during the replay mechanism and the network trained on these states, the agent learns to find states that are sub-optimal.

Below, this is examined further. We plot the number of times the agent explored or exploited the environment for different values of ϵ for every step over every episode. The blue bars represent the amount of times explored while the red bars represent the amount of times the agent exploited the environment.

As expected, the amount of exploration that occurred increases as the value of ϵ increases, seen by the increasing height of the blue bars. We expect that as the amount of exploration increases, the amount of exploitation decreases. From the graph, we observe that this is true for values 0.5 and above. However, when comparing $\epsilon = 0.3$ to $\epsilon = 0.4$, this trend is not followed. This is because for $\epsilon = 0.4$ and the nature of CartPole, the longer the agent can balance the pole, the more exploitation it can achieve.

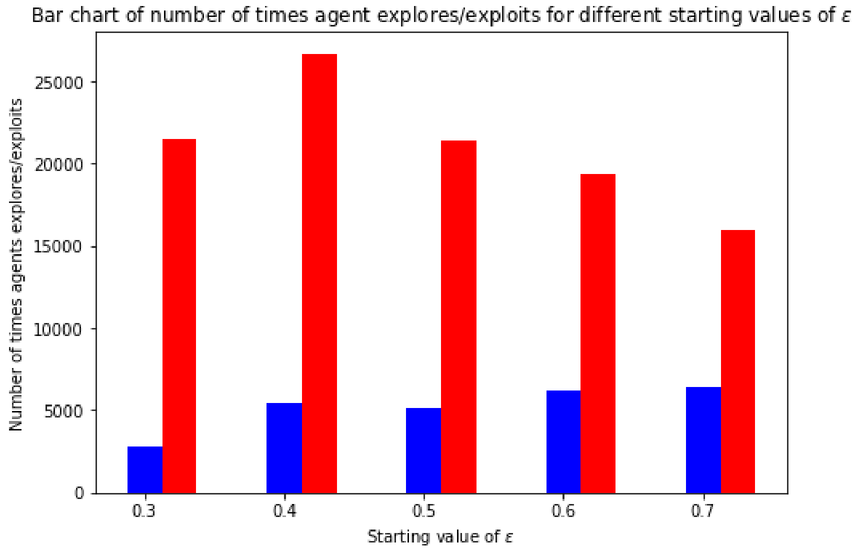


Figure 11: Exploration vs Exploitation

Value of ϵ	# explored	# exploited	Exploration/Exploitation (%)
0.3	2848	21549	12/88
0.4	5466	26717	17/83
0.5	5175	21457	19/81
0.6	6262	19343	24/76
0.7	6438	16001	29/71

Table 5: Ratio of exploration vs exploitation for different starting values of ϵ

8.4.2 An Interesting Point about CartPole

In many environments, Reinforcement Learning Specific and otherwise, an episode terminates when the goal state is reached. For example:

- In the classic RL environment, Mountain Car[2], the goal state as well as the termination state happen when the car reaches the flag atop the mountain.
- In the Atari game, Breakout, the game terminates when all the bricks have been destroyed (success) or if the player runs out of "lives."

CartPole is unlike these environments. CartPole terminates when it is unsuccessful (the cart moves off the screen in either direction or the pole falls over). For an agent to be successful in this environment then, it needs to avoid these terminal states (also referred to as *doomed* states) for as long as possible in order to maximise its reward. The longer the agent can avoid the set of doomed states, the higher its reward value and the better its performance.

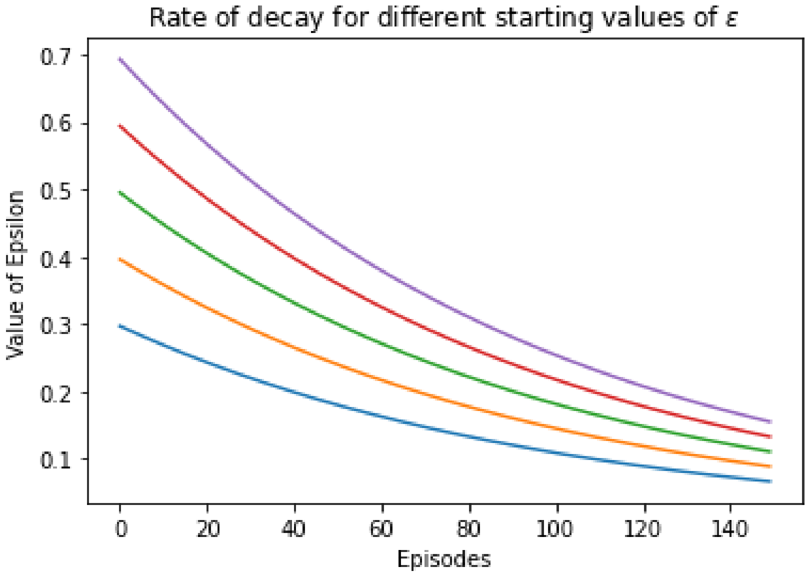


Figure 12: Exploration vs Exploitation

TODO:

Speak about code implementation for different architectures Justification of parameter choices Change experiments to be over 3 runs only Speak about best epsilon value Plot of Q values vs slices of states Graph of predicted y vs actual y

List of next experiments Optimisers: ADAM vs RMSProp vs ... Memory size vs average goal achieved

Learning rates

Average rewards vs episode

Lunar Lander

Q values

8.5 Research Questions

- Effect of learning rate on agent performance
- What effect do different optimisers have on the agent's performance/rate of learning?
- Q Values in relation to the states of the environment
- Does Double Deep Q Network perform better than a DQN with Experience replay?

9 Conclusions

10 Professional Issues

11 Self-Assessment

- Having to decrease environment count because of realisation that one environment had too much to analyse and not much more time to conduct full regiment of experiments on another environment
- Briefly attempting to solve different environments with the same architecture was futile - because of how CartPole accumulates rewards and terminates vs other environments. The closest

References

[1] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013). cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL: `http://arxiv.org/abs/1312.5602`.

[2] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[4] Christopher J. Watkins. *Notes on Q Learning*. 2021.

[5] Sergey Levine. *Reinforcement Learning*. UC Berkley CS W182/282A. 2021.

[6] C. J. C. H. Watkins. "Learning from Delayed Rewards". PhD thesis. King's College, Cambridge, 1989.

[7] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[8] Thomas Kluyver et al. "Jupyter Notebooks – a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press. 2016, pp. 87–90.

[9] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: `10.1109/MCSE.2007.55`.

[10] Rita Kurban. "Deep Q Learning for the CartPole". In: (2019).