Reinforcement Learning 2024 Assignment 2: Deep Q Learning

Sherry Usman, Qin Zhipei, Meghan[1]

Abstract

This paper explores the use of Deep Neural Networks (DQNs) to solve sequential decision problems, such as the CartPole problem, by exploiting the advantages of function approximation. We focus on two main exploration strategies - epsilon-greedy and softmax/Boltzmann and find out that the optimal value of epsilon ϵ for e-greedy is 1.0 and the optimal value of temperature τ for softmax is 0.1. In addition, we contrast and compare performances of various hyper-parameters such as learning rate, batch size and even go as far as to modify the neural network architecture to investigate its effects on model performance. In addition to this, we conduct an extensive ablation study to compare and contrast the contributions of various components of a full DQN (with experience replay and target network) to model performance and conclude that the effects of removing components such as experience replay and target that a full DQN network performs much better than a DQN with missing components due to its ability to learn from previous experiences, brought about by experience replay, and its stable target, brought about by a target network.

1. Introduction

Since the rise of artificial intelligence (AI) and machine learning (ML), the advent of different deep reinforcement learning algorithms has fundamentally changed our understanding of reinforcement learning and sequential decision problems. Among this plethora of algorithms are deep Q-learning neural networks (DQNs), which stand out as a set of important paradigms used to solve complex sequential decision problems, while staying computationally efficient, lightweight and flexible. Applications of deep neural networks can be seen in a variety of different contexts, from playing video games such as Atari (Mnih et al., 2018) to even in adaptive robotics like teaching a robot cheetah how to jump (Wang et al., 2019).

In this paper we delve into the implementation of DQN in the CartPole environment. The CartPole is a classical reinforcement learning environment where the goal is to balance a pole on the cart, while applying forces to the cart at the bottom (as shown in Figure 1) (Brockman et al., 2016). While a number of different techniques can be used to solve this problem, our paper specifically concentrates on the use of DQNs. Through this paper, we hope to provide the specifications of various DQN models while also including the various exploration strategies that provide us the best exploration-to-exploitation ratio and also optimal hyper-parameters that will yield the best results. Furthermore, we hope to provide an extensive ablation study to help identify the impact of adding and removing certain components (specifically, experience replay and target network) on DQN's performance.

2. Framework

The CartPole environment poses unique challenges compared to the environments we have encountered previously. While the action space A is still discrete with two potential actions (push left and push right), there is a continuous observation space/state space S which includes the **cart position**, **cart velocity**, **pole angle** and **pole angular velocity**. A high-dimensional state space such as this makes the creation of a Q table/array storing the values for every state-action pair quickly infeasible. Thus, to tackle this problem we use DQNs which leverages the advantages of *function approximation* to obtain q values.

Learning with *function approximation* allows for

1. A more compact representation of the solution using a value function that can be stored in its approximate form.
2. Exploiting similarities between states by using common features between them to approximate their q-values (Plaat, 2023).

2.1. Exploration Strategies

Exploration strategies play a crucial role in reinforcement learning algorithms by providing a framework for policies the agent can use to maximise their rewards over time. Ex-

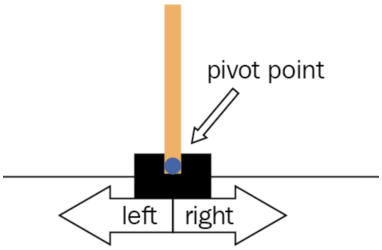

Figure 1. A visualisation of the Cartpole problem

ploration strategies do this by tweaking the exploration to exploitation ratio: how much the agent explores random unfamiliar actions versus how much the agent goes for tried and tested actions that provide high rewards. For this particular project we will explore two different exploration/exploitation policies - Epsilon-Greedy and Boltzmann/ Softmax Policy (although there are many others).

Epsilon-Greedy Policy (e-greedy policy): The e-greedy policy is an action selection policy that uses an epsilon parameter ϵ to determine the ratio of exploration to exploitation. An ϵ value of 0 gives a greedy policy while an epsilon value of 1 gives a uniform/random policy. We take a small random value n between 0 and 1.

- If n is less than ϵ, we take a random action from the state space.
- Otherwise we take a greedy action or an action which returns the highest Q(s,a) for the individual state-action pair.

This policy is shown in detail in the algorithm below.

Data: DQN, state s, epsilon ϵ
Result: Selected action
$n \leftarrow$ uniform random floating point between 0 and 1
if $n \leq \epsilon$ **then**
  $a \leftarrow$ random action from the action space A
else
  $a \leftarrow$ maxQ(s,)
end
Return selected action a;

As the best value of ϵ can vary depending on the environment and the specific neural network architecture, it is necessary to experiment with different epsilon values to find the optimal e.g. 0.01, 0.1, 0.5, 0.8 and 1.0.

Boltzmann or Softmax Policy: The Boltzmann/Softmax policy uses a temperature parameter τ instead. This allows a more flexible scaling between exploration and exploitation as instead of the value being between 0 to 1, the value lies between 0 and ∞. When $\tau \leftarrow 0$, the policy is greedy or exploitative and when $\tau \leftarrow \infty$, the policy becomes explorative. The formula for Boltzmann / Softmax policy is given below.

$$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{b \in A} e^{Q(s,b)/\tau}} \tag{1}$$

Similar to epsilon, the optimal value of τ is heavily influenced by the type of neural network architecture and environment. Thus, we experiment with an array of different temperature values to find the optimal e.g. 0.01, 0.1, 1, 5 and 1.0.

2.2. Simple DQN (DQN-TN-ER)

Before proceeding further it is important to understand the architecture of a simple DQN. A simple Deep Q-Learning Network (DQN) is a type of neural network architecture which attempts to approximate the Q-value function. It was initially created by DeepMind and has been used in the a number of different reinforcement learning environments since then such as the Atari game (DeepMind, 2016).

In a simple DQN, the Q-learning network is first initialised with random weights. The network then takes the current state s of the environment as input and outputs Q(s,a) values for each action (a push left and push right). Based on the agent's action selection policy (epsilon-greedy or Boltzmann policy in this case), the agent then decides which action to take. For all episode steps e_t, the agent accrues an experience $e_t = s_t, a_t, r_t, s_{t+1}$ where s_t is current state, a_t is the action undertaken by the agent, r_t is the reward for that action and s_{t+1} is the next state and additionally a variable that tells us is the agent was successful in keeping the pole upright for a number of timesteps, $done$.

The target Q-value is then calculate using the r, s_{t+1} and $terminated$ values as shown below:

1. If the episode terminates, the target is equal to the immediate reward. Otherwise, it is the sum of the immediate reward and the discounted maximum Q-value of the next state using the target Q-network (target net).
2. Then the loss between Q-value and target is computed.
3. This loss is then backpropagated to the weights of the online Q-network (online net).

The algorithm for simple DQN is shown as labeled (**Algorithm 1**).

2.3. DQN with Experience Replay (DQN-TN)

DQN with experience replay is another type of neural network architecture which incorporates a replay memory

Algorithm 1 Simple DQN Algorithm

Data: DQN, state s, exploration rate ϵ, discount factor γ, maximum timesteps M, number of episodes E
Result: Selected action
Initialise Q value function with random weights
for $i = 1$ **to** E **do**
  Initialise state s
  $T_r = 0$
  for $t = 1$ **to** M **do**
    Select action a using an ϵ-greedy policy or Boltzmann policy
    Receive next state s' and reward r by taking action a
    if *done* **then**
      $T_r = r$
    else
      $T_r = r + \gamma \times \max_{a'}(Q_(s_{t+1}, a'))$
    end
  end
end
Return selected action a

buffer, a veritable store of all past experiences of the agent. In a given run, for all episode steps E, experiences accrued by the agent during that step (s_t, a_t, r_t, s_{t+1}) are stored in the replay buffer/memory D with a certain maximum capacity N. During Q-learning, random batch of samples are retrieved from this buffer D. This minibatch is used to train the neural network model further and helps the agent to learn from a number of random previous experiences to inform their next action selection. The algorithm for DQN with experience replay can be seen below. [1]

Algorithm 2 DQN with Experience Replay

Data: DQN, replay memory buffer D, maximum buffer length N, state s, exploration rate ϵ, discount factor γ, maximum timesteps M, batch size b
Result: Selected action
Initialise replay memory D with a maximum length N
Initialise DQN with random weights θ
for $i = 1$ **to** E **do**
  $T_r = 0$
  for $t = 1$ **to** M **do**
    Select action a using an ϵ-greedy or Boltzmann policy
    Receive next state s_{t+1}, reward r_t and *done* by taking action a
    Store transition (s_t, a_t, r_t, s_{t+1}, *done*) in D
    Get a random minibatch of transitions of batch size b
    if *done* **then**
      $T_r = r_t$
    else
      $T_r = r_t + \gamma \times \max_{a'}(Q_(s_{t+1}, a'))$
    end
    Perform a gradient descent on $(T_r - Q(s_t, a))$ with respect to Q network parameters
  end
end

[1] Sutton, Richard S and Barto, Andrew G - Reinforcement learning: An Introduction - Second Edition

2.4. DQN with a Target Network (DQN-ER)

While it is possible to create a DQN with a single network, this poses a potential problem. With each step, the agent iteratively learns about the best action to take based on observed actions and resulting rewards. This means that the Q network gets updated every time and Q values predicted for the next state keep changing. This creates the moving target problem where the best Q values the network is striving towards keep changing. This problem is solved by a target network.

A target network is a second untrained/less frequently trained neural network that provides a stable log of the target Q values over a short period. This reduces the instability caused by changing Q targets. There is often a compromise made as a secondary network that is not consistently trained will contain Q values that are sub-optimal and training the main network using such a target network may mean we run the risk of training the main network using sub-optimal Q values. However, we curb this by copying the weights of the main network to the target network after a certain number of timesteps called the target update frequency.

Algorithm 3 DQN with a Target Network

Data: replay memory buffer D, maximum buffer length N, state s, exploration rate ϵ, discount factor γ, maximum timesteps M, target update frequency C
Result: Selected action
Initialise replay memory D with a maximum length N
Initialise online network Q with random weights θ
Initialise target network $\hat{Q}$ with random weights θ' = θ
for $i = 1$ **to** E **do**
  $T_r = 0$
  for $t = 1$ **to** M **do**
    Select action a using an ϵ-greedy or Boltzmann policy
    Receive next state s_{t+1}, reward r_t and *done* by taking action a
    if *done* **then**
      $T_r = r$
    else
      $T_r = r + \gamma \times \max_{a'}(Q_(s_{t+1}, a'))$
    end
    Perform a gradient descent on $(T_r - Q(s_t, a))$ with respect to Q network parameters
    Every C steps update $\hat{Q}$ = Q
  end
end

2.5. DQN with a Target Network and Experience Replay

We can say that a DQN with a target network and experience replay combines the benefits of both components in one neural network. It allows the DQN or the agent to benefit

from past experiences while also allowing it to keep a stable target, evading the shifting target problem. The algorithm can be seen as labeled (**Algorithm 4**).

Algorithm 4 DQN with Target Network and a Experience Replay

Data: Replay memory buffer D, maximum buffer length N, state s, exploration rate ϵ, discount factor γ, maximum timesteps M, batch size b
Result: Selected action
Initialise replay memory D with a maximum length N
  Initialise online network Q with random weights θ
  Initialise target network $\hat{Q}$ with random weights θ' = θ
  Initialise DQN with random weights θ

for $i = 1$ **to** E **do**
  $T_r = 0$
  for $t = 1$ **to** M **do**
    Select action a using an ϵ-greedy or Boltzmann policy
    Receive next state s_{t+1}, reward r_t and $done$ by taking action a
    Store transition (s_t, a_t, r_t, s_{t+1}, $done$) in D
    Get a random minibatch of transitions of batch size b
    if *done* **then**
      $T_r = r_t$
    else
      $T_r = r_t + \gamma \times \max_{a'}(Q_(s_{t+1}, a'))$
    end
    Perform a gradient descent on $(T_r - Q(s_t, a))$ with respect to Q network parameters
    Every C steps update $\hat{Q}$ = Q
  end
end
Return selected action a

3. Initial Neural Network Architecture and Hyper-Parameter Tuning

For our implementation of DQNs, we used the Python **Pytorch** package. Our initial DQN structure is as follows: an input layer of size 4 corresponding to the size of the observation space. This input layer feeds into a first hidden layer with 128 nodes, followed by a ReLU activation function. This then feeds into another hidden layer comprising of 128 nodes which finally outputs to a layer comprising of 2 nodes, each representing one possible action (either push left or push right). In this section we experiment with various hyper-parameters such as batch size, learning rates and more while also fine-tuning our neural network by modifying the number of hidden layers and neurons in each layer. Since this environment is also highly stochastic, we de-noise by testing over 1000 episodes and averaging our results over every 10 episodes.

The initial hyper-parameters we used are as follows.

Hyper-parameters	Value
Discount factor γ	0.97
Exploration rate ϵ	1.0
Batch size	32
Learning rate	0.01

Table 1. Hyper-parameters

3.1. Network Structure

We first begin by further tuning our initial neural network architecture. After an experiment with various layers of our neural network architecture, we concluded that a DQN with 1 or 2 hidden layers performs much better than a DQN with 3 hidden layers. We can see that the number of layers in a neural network is significantly correlated to the bias-variance trade-off. One potential reason for this is a more complex network while increasing its capacity to learn may create a high-variance model. This over-fits the data, leading to the network performing poorly on unseen data.

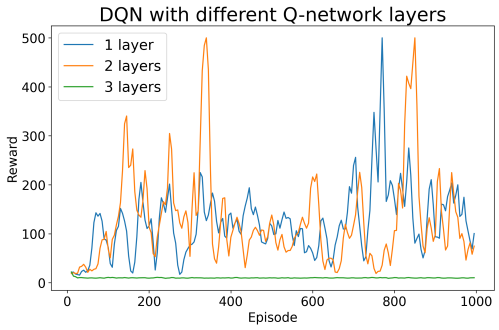

Figure 2. The figure shows a comparison of the performances of DQN architectures with a varying number of layers. As we can see above at the architecture with 1 or 2 layers significantly outperforms the architecture with 3 layers, even touching 500.

We also experimented with the number of neurons in our architecture and through repeated tests found that hidden layers with 64 neurons outperform those with 128 or 32 neurons. We kept this architecture for all future experiments.

3.2. Tuning Batch-Size

While larger batch sizes can lead to faster training times due to less frequent updates, they can also suffer from overfitting and lower accuracy. Smaller batch sizes on the other

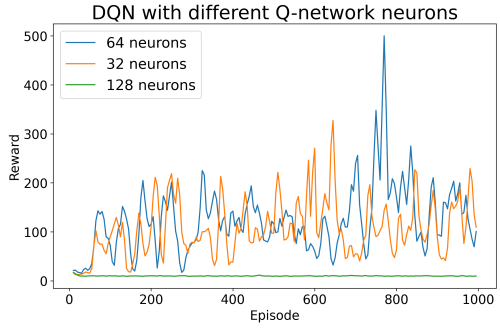

Figure 3. The figure hows a comparison of the performances of different DQN architectures with a varying number of neurons in the hidden layer. In this comparison we choose the DQN with just one hidden layer. As seen above, the architecture with 64 neurons outperforms the architecture with 128 or 32 neurons. Thus we will continue to use this architecture for our studies.

hand are much more time consuming and computationally expensive but because of their more frequent updates tend to be more accurate. However, it is important to note that there are a variety of other factors that can simultaneously influence model accuracy such as model architecture, the optimisation algorithm and so on. Thus, the relationship between batch size and model performance is not universally applicable and testing is needed to accurately compare the impact of batch size on model performance.

From Figure 4, it is evident that larger batch sizes produce higher returns than smaller batch sizes. This may be because larger batch sizes produce more stable generalisations as they are spread over a larger number of experiences. This also prevents over-fitting which is often seen in cases where the batch size is too small.

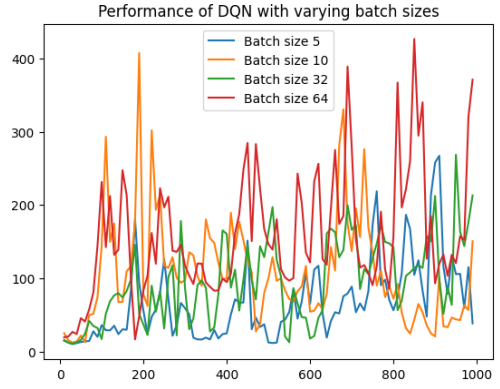

Figure 4. The figure shows comparison of epsilon-greedy performances for varying batch sizes in a DQN with Experience Replay and a Target Network (DQN). As seen above much higher values of batch size (e.g. 64) produce much higher but more unstable returns, peaking at a little over 400. In comparison lower values of batch size produce lesser returns but tend to be more stable. An example is the more or less steady rise of the batch size 5.

3.3. Tuning the E-greedy Policy for Different Epsilons

Note: We used an epsilon decay factor which steadily reduced epsilon over a number of episodes.

Figure 5 shows the performance of DQN neural network with the epsilon-greedy algorithm for epsilon = 1.0 is much better than the performance of other epsilons = 0.9 or 0.5. This implies that higher rates of exploration are more rewarding than lower rates of exploration, allowing the agent to explore more lucrative state-action pairs in the state space. However, on closer examination, we find that while higher values of epsilon perform better they also tend to be more unstable. Conversely, lower values of epsilon may provide poorer performance but tend to be more stable.

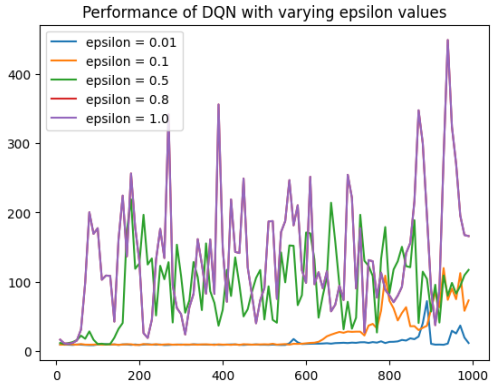

Figure 5. The figure shows a comparison of epsilon-greedy (with epsilon decay) performances in a DQN with Experience Replay and a Target Network (DQN). We can see that lower values of epsilon (e.g. 0.01) produce limited rewards while higher values such as 0.8 and 1 are much more suitable, peaking at over 400. It should be noted that an epsilon decay factor was used to vary exploration to exploitation ratio over time.

3.4. Tuning the Softmax Policy for Different Temperatures

Figure 6 shows the performance of DQN neural network with the Softmax policy for different values of temperature τ. This gives us an indication that a τ of 0.1 performs much better than τ of 0.01, 1, 5 or 10. Thus, we can argue that a smaller exploration to exploitation ratio tends to be more lucrative when using the Softmax Policy as such a ratio encourages more exploitation and reduces randomness. However, a too small τ (such as 0.01 in this experiment) might cause the algorithm to fixate on a local optimum early on. Additionally, it may result in the network lacking adaptability to states or environmental changes not encountered during the training process, leading to poor performance when confronted with new states.

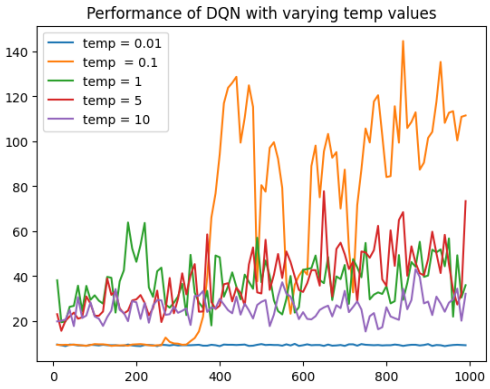

Figure 6. The figure shows the comparison of Boltzmann/softmax performances for varying temperature values in a DQN with Experience Replay and a Target Network (DQN). We can see that lower values of temperature (e.g. 0.01) produce limited rewards while much higher values (e.g. 5 and 10) are also are unsuitable, peaking at a little over 60. Middle values however such as 0.1 are much better suited and produce rewards as high as 140.

3.5. Tuning Learning Rates

When attempting to minimize the loss function of the Q-network, it is important to adjust the learning rate to determine the magnitude of parameter updates with each iteration. In this experiment, we experimented with three different learning rate settings: 0.001, 0.01, and 0.05. As can be seen from Figure 7, only when the learning rate is set to 0.01 do we obtain meaningful results.

We speculate that when the learning rate is too high, the Q-network does not converge at all, leading to training failure. On the other hand, when the learning rate is too low, it results in too small steps for parameter updates, requiring the Q-network to undergo more iterations to achieve convergence. This causes it to fail to obtain meaningful results in the early stages.

Our results show that an inappropriate value for the learning rate, whether too large or too small, can significantly impact the convergence of the model within a limited number of episodes. For our best value lr=0.1, it is high enough to allow the model to learn efficiently but is not so high that it prevents the model from refining its policy towards the optimal solution.

4. Ablation Study

In this section we conduct an ablation study of the various components of the DQN to understand the importance of each component and their impact on the overall productivity of our model.

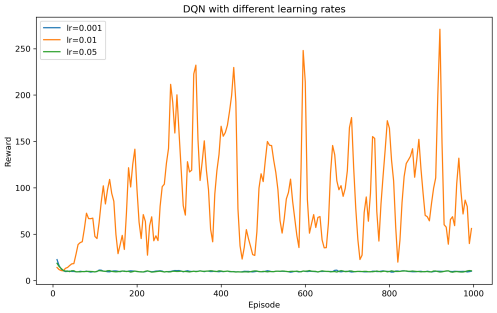

Figure 7. A comparison of the performance of DQN with different learning rates 0.001, 0.01 and 0.05 shows that the ideal learning rate is 0.01, giving us much higher returns on average than 0.05 and 0.001 which tend to flatline at just below 10.

4.1. DQN versus DQN-ER

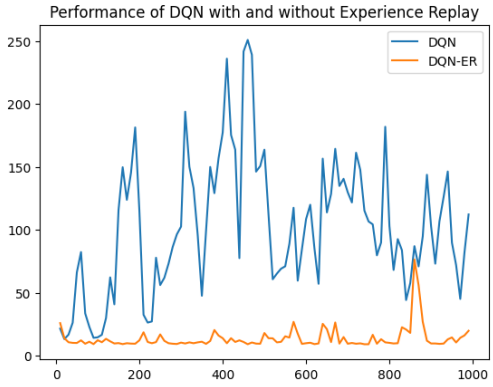

Figure 8. The figure shows a comparison of the performance of DQN with (DQN) and without experience replay (DQN-ER). As seen in the figure a DQN with experience replay performs much better (provides higher returns) than a DQN without experience replay.

The first part of the ablation study focuses on the performance of a DQN with and without an experience replay buffer with our fine-tuned parameters. As seen above, an DQN with an experience replay buffer performs much better than a DQN without one. This can be explained by the fact that a replay buffer provides a store of random samples taken from the agent's previous experiences, and this store helps to de-correlate the samples used for training preventing sub-optimal convergence (getting stuck in a local maxima) and allows the agent to use exploit more randomness in exploring past experiences, leading to better learning in the long run.

4.2. DQN versus DQN-TN

The second part of the ablation study compared the performance of a DQN with and without a target network. As we can see in Figure 9, a DQN with a target network performs much better than a DQN without a target network. The lack of a target network means that the target values during training will continuously change, leading to instability in training. Additionally, since the target values are closely linked to the updating network parameters, this may result in the Q-values being consistently overestimated or underestimated, thereby causing the training to diverge.

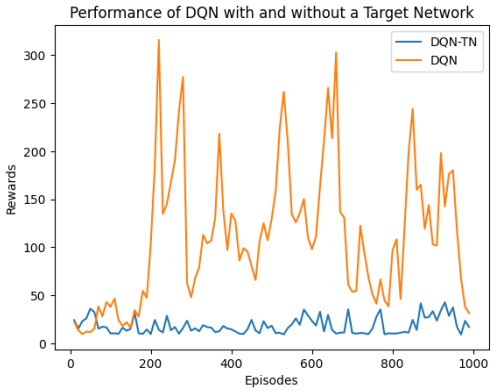

Figure 9. The figure shows the comparison of the performance of DQN with (DQN) and without a target network (DQN-TN). As seen in the figure a DQN with a target network provides much higher results than a DQN without a target network, peaking above 300.

4.3. DQN versus DQN-TN-ER

The third part of the ablation study compared the performance of a DQN with and without a target network and experience replay. Removing both the target network and the experience replay diminishes the output of our neural network. We can think of this as an accumulation of the individual effects of removing a target network and removing a replay buffer.

Discussion and Conclusion

In this section, we want to encapsulate our findings derived from a series of tests we conducted in this project.

Based on the ablation study, it is evident that DQNs with experience replay buffer and/or a target network perform better than DQNs with such components removed. DQN algorithms with a target network and experience replay not only benefit from solving the shifting target problem but also help learn from past experiences, explaining their better results overall. We also see that DQN algorithms are much better than their tabular counterparts. They deal with high-dimensional state and action spaces using function

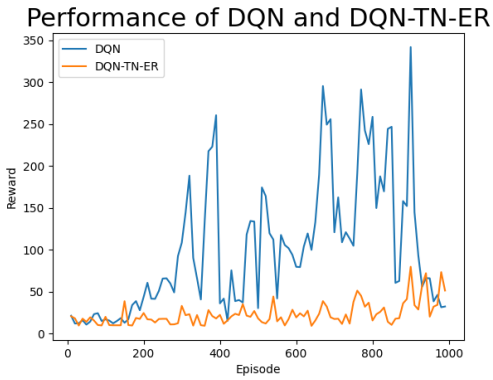

Figure 10. The figure shows the comparison of the performance of DQN and without either experience replay or a target network (DQN-TN-ER). As seen in the figure, the full DQN trained more quickly and achieved a significantly higher return than DQN-TN-ER, with its peak return surpassing 340.

approximation, which not only helps explore similarities between different state-action pairs but also allows the capturing of more complex and non-linear relationships between state-action pairs and their respective rewards. This allows DQNs to grasp more advanced patterns in data that are not possible in tabular reinforcement learning algorithms.

However, while deep neural network provide a strong framework to solving sequential decision problems they also suffer from weaknesses. Firstly, due to a highly stochastic initialisation process deep neural networks suffer from randomness and instability. Furthermore, there may be additional randomness in results due to variability in mini-batches selected. While a potential fix for this could be to use Prioritised Experience Replay (PER) to select experience samples from the batch, this may introduce the problem of over-fitting to the most recent experiences.

Secondly, while our project focuses on a fraction of exploration strategies and architectures, there are many other strategies such as Upper Confidence Bound (Auer et al., 2002) and Thompson Sampling (Chapelle & Li, 2011) and different kinds of architectures and associated activation functions that can be used to fine-tune our rewards. Thus, a significant amount of time and resources is needed to effectively investigate the ideal combination of exploration strategies, hyper-parameters and neural network architectures needed to produce the best results.

Thirdly, while our data is sufficient for the scope of our project, it is evident that a more comprehensive analysis of these algorithms is needed to fully understand the advantages and disadvantages of each and such an endeavour would not only be computationally expensive but also require a significant amount of memory due to the significant resources needed to train the agents.

Our extensive study also gives us a good outlook on the

various factors affecting our CartPole agent. For example, we discovered that for a relatively simple task like CartPole, a neural network with only one or two hidden layers, each containing tens of neurons, is sufficient for successful training. To reduce model complexity and computational costs, we ultimately used a Q-network with only one hidden layer.

In terms of exploration strategies, for epsilon-greedy, we discovered that it was necessary to set epsilon to the highest value (1.0) initially to promote exploration, but that this value should also be steadily decreased over a period of time to allow the agent to exploit the knowledge discovered in the later stages of training and to find the optimal strategy. For softmax policy, we discovered that a consistently high τ value was too random and thus did not yield good results, while a relative smaller τ (0.1) successfully achieves a balance between exploration and exploitation.

References

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite time analysis of multi-armed bandit problem. 2002. URL https://rdcu.be/dCLjy.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Olivier Chapelle and Lihong Li. An empirical evaluation of thompson sampling. 2011. URL https://papers.nips.cc/paper_files/paper/2011/hash/e53a0a2978c28872a4505bdb51db06dc-Abstract.html.

DeepMind. Deep reinforcement learning, 2016. URL https://deepmind.google/discover/blog/deep-reinforcement-learning/.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2018. URL https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf.

Aske Plaat. *Deep Reinforcement Learning*. 2023.

Tingwu Wang, Xuchan Bao, Ignasi Clavera, Jerrick Hoang, Yeming Wen, Eric Langlois, Shunshi Zhang, Guodong Zhang, Pieter Abbeel, and Jimmy Ba. Benchmarking model-based reinforcement learning. 2019. URL https://arxiv.org/pdf/1907.02057.pdf.

Summary of Notation

Symbol	Description
S	state space
A	action space
a	selected action from the action space
$Q(S, A)$	Q value of a state action pair
ϵ	An exploration rate epsilon ranging from 0 to 1
τ	temperature parameter used in Boltzmann policy ranging from 0 to ∞
$\pi(a\|s)$	probability of taking an action a in state s under stochastic policy
e_t	experience containing current state, next state, reward and done
M	maximum number of timesteps per episode
E	number of episodes
T_r	total reward
γ	discount factor
D	replay memory buffer
N	capacity of replay memory buffer
b	batch size
Q	online network
θ	weights of online network
Q'	target network
θ'	weights of target network
C	target update frequency