

# COMP3702 Artificial Intelligence (Semester 2, 2024)

## Assignment 3: Reinforcement Learning

Name: Harsha Joshi

Student ID: 45561623

Student email: [s4556162@uq.edu.au](mailto:s4556162@uq.edu.au)

Note: Please edit the name, student ID number and student email to reflect your identity. When submitting to Gradescope, ensure you mark the start of each question.

---

### Question 1 (Complete your full answer to Question 1 on the remainder of page 1)

- a) There are two key similarities between Value Iteration and Q-learning. Firstly, both algorithms aim to find the optimal policy for an agent in the Markov Decision Process (MDP) by “maximising cumulative rewards”. Secondly, both algorithms update value estimates iteratively to improve decision making over time, refining the understanding of “state-action” values (AutomaticAddison, 2019).
- b) The key difference between Value Iteration and Q-learning is that: Value Iteration requires a “complete model” of the environment (transitional probabilities and rewards), while Q-learning is a model-free reinforcement learning algorithm that learns directly from experiences – taking “an action  $a$ , from a state  $s$  and then receives a reward  $r$ ”, without needing any prior knowledge of the environment (AutomaticAddison, 2019).

## Question 2 (Complete your full answer to Question 2 on page 2)

- a) The following code snippet is the implementation of how R100 value vs Episode number was plotted (Paszke & Towers, 2024). This was added to the existing dqn\_gym.py file. Similarly, changes were made to the “while True” loop’s if is\_done section to account for the episode number being plotted from 0, further an elif statement was added to handle values beyond episode number reaching 100. In the elif, the program must take the average of the previous 100 rewards to determine the R100 values.

```
def plot_r100_vs_episode(env_name, ep_number, r100_values):
    episodes = list(range(ep_number))
    r100_values = [0] * (len(episodes) - len(r100_values)) + r100_values

    plt.figure(figsize=(12, 6))
    plt.plot(episodes, r100_values[:len(episodes)], label=f'{env_name} - R100 Value', color='green')
    plt.xlabel('Episodes')
    plt.ylabel('Reward / R100 Value')
    plt.title(f'R100 Value vs. Episode Number for {env_name}')
    plt.legend(loc='lower right')
    plt.grid(True)

    plt.savefig(f"Q2_{env_name}_Plot.png")
    plt.tight_layout()
    plt.show()
```

### **R100 Calculation (visible in both if statements):**

$r100 = \text{np.mean}(\text{all\_rewards}[-100:])$   
 $r100\_values.append(r100)$

where:

- $r100\_values$  is initialised as  $[0] * 10$  (for padding purposes)
- $\text{np}$  refers to Python’s inbuilt numpy library

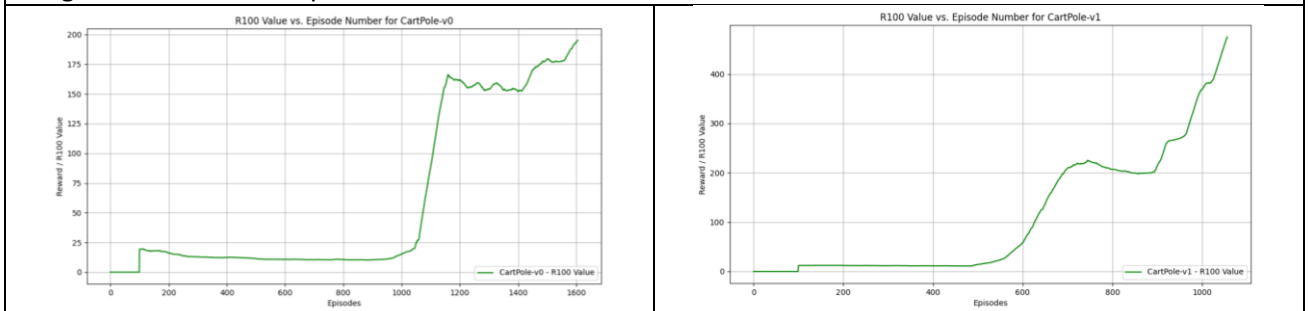
```
if len(all_rewards) > 0 and len(losses) > 0:
    r100 = np.mean(all_rewards[-100:])
    r100_values.append(r100)
    r_variance = np.var(all_rewards[-100:])
    l100 = np.mean(losses[-100:])
    fps = (frame_idx - episode_frame) / (time.time() - episode_start)
    print(f"Frame: {frame_idx} Episode: {episode_no}, R100: {r100:.2f}, MaxR: {max_reward:.2f}, R: {episode_reward:.2f}, FPS: {fps:.1f}, L100: {l100:.2f}, Epsilon: {epsilon:.4f}")

    # visualize the training when reached 95% of the target R100
    if not visualizer_on and r100 > 0.95 * params['stopping_reward']:
        env = gym.make(args.env, render_mode='human')
        env.reset()
        env.render()
        visualizer_on = True

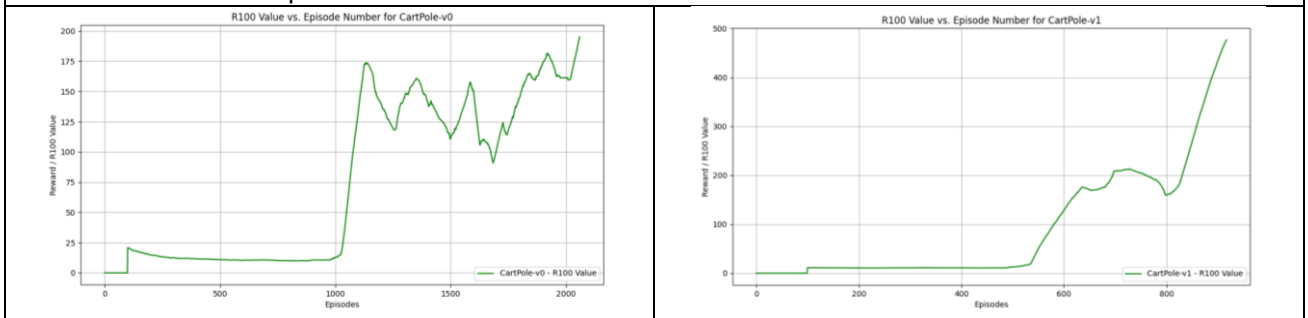
elif len(all_rewards) > 100:
    r100 = np.mean(all_rewards[-100:])
    r100_values.append(r100)
    r_variance = np.var(all_rewards[-100:])
    l100 = np.mean(losses[-100:]) if len(losses) >= 100 else np.nan # Handling any edge cases
    if time.time() - episode_start > 0: # Checking to prevent division by zero
        fps = (frame_idx - episode_frame) / (time.time() - episode_start)
    else:
        fps = 0 # Setting fps as 0
    print(f"Frame: {frame_idx} Episode: {episode_no}, R100: {r100:.2f}, MaxR: {max_reward:.2f}, R: {episode_reward:.2f}, FPS: {fps:.1f}, L100: {l100:.2f}, Epsilon: {epsilon:.4f}")
```

- b) Produced Plots: note that ‘single-hidden’ refers to a single hidden layer, while ‘two-hidden’ refers to a double hidden layer in a Neural Network

### **‘single-hidden’ Plot Comparison: CartPole-v0 vs CartPole-v1**



### **‘two-hidden’ Plot Comparison: CartPole-v0 vs CartPole-v1**



- c) In referring to the two videos on Blackboard, it is evident as a first impression that CartPole-v1 has about 6 seconds additional runtime – this indicates that this model is more complex as compared to CartPole-v0. Based on the plots above, CartPole-v1 is the more stable and robust environment because it can perform similarly well whether under a single hidden or double hidden layer, as shown by the curve from 600-800 episodes remaining consistent under both circumstances. CartPole-v0 is significantly “noiser” (Rolnick et. al, 2018) after 1000 episodes particularly under double hidden layer. Further, it is evident that while CartPole-v1 is the more complex model based on runtime, it takes significantly smaller number of episodes to converge – between 800-1100 episodes compared to CartPole-v0’s 1600-2000 episodes. All these factors cause `max_episode_steps` and `reward_threshold` to be increased from v0 to v1.

**Question 3** (Complete your full answer to Question 3 on page 3)

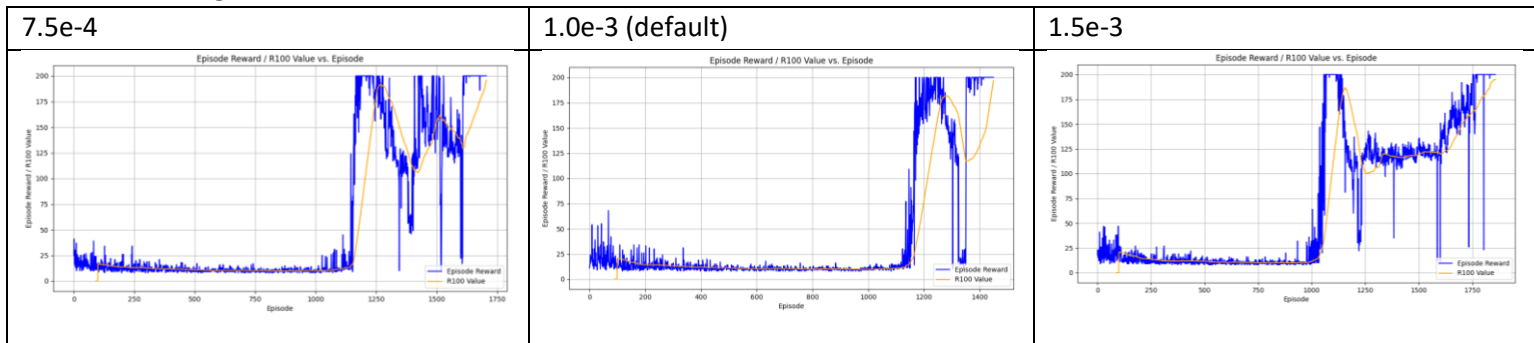
- a) TD-learning “is a method used to estimate the value of states” in a Markov Decision Process (MDP). It updates estimates based on the “temporal difference” between the estimated values of two successive states” (SaturnCloud, 2023). The loss function for training the DQN is based on the difference between the predicted Q-values and the target Q-values (Ankit, 2023). It can be expressed as:  $L = \left( r + \gamma \max_a \hat{Q}(s, a) - Q(s, a') \right)^2$  (Patel, 2017), where:
- Target =  $r + \gamma \max_a \hat{Q}(s, a')$
  - Prediction =  $Q(s, a)$
- b) From the code, referencing the function `calculate_loss()` in `dqn_gym`:
- Target Values: “`state_action_values`”. It is calculated by gathering the Q-values from the output of the network (`Q_s`) using the actions taken (`actions_v`).
  - Current State-Action Value Predictions/Estimates: “`expected_state_action_values`”. It is calculated using the rewards received (`rewards_v`), the maximum Q-value for the next states (`next_state_values`), and the discount factor (`params[ 'gamma' ]`). Additionally, (`1 - dones_v`) is a tensor which indicates whether episode has ended after the current action – this value is not included in mathematical formula itself (Paszke & Towers, 2024).
- c) A separate target network is created to “stabilises the learning process” so that agents are enabled “to learn complex behaviours from raw data” (Liu, 2024), with the overall goal being the improve performance (as measured by the rate of convergence in the environment – CartPole-v0 in this case). The target network is structured the same as the main network, with slower updating weights as it copies the weights of the main network every few training steps – this encourages decoupled learning through periodic updates (Liu, 2024). According to Kim et. al, this is done through the following minibatch equation:  $\theta \leftarrow \theta + \alpha(r + \gamma \max_a \hat{Q}(s', a'; \theta) - Q(s, a; \theta)) \nabla_{\theta} Q(s, a; \theta)$ , where  $\theta$  represents the relevant parameters. These periodic updates “are supposed to help reduce correlation between consecutive samples”, as this can negatively affect “gradient-based methods” (Kim, et. al, 2019).
- d) Using CartPole-v0, both periodic target network versus soft updates were trained on ‘single-hidden’, maintaining that this setting is most appropriate for the environment from Question 2. It is to be noted that in the `dqn.yaml` file provided, of the three relevant parameters (`alpha_sync`, `target_net_sync`, `tau`), only the specified ones were used to gain the results. In default mode, all three of these are used.

Periodic Target Network ( <code>alpha_sync = false</code> , <code>target_net_sync = 1000</code> )	Soft Updates ( <code>alpha_sync = true</code> , <code>tau = 0.005</code> )
<ul style="list-style-type: none"> <li>• Average Reward over last 100 episodes: 195.07</li> <li>• Learning Speed (Episodes to reach 195 reward): 1207</li> <li>• Training Stability (Variance in last 100 episode rewards): 143.89</li> </ul>	<ul style="list-style-type: none"> <li>• Average Reward over last 100 episodes: 195.17</li> <li>• Learning Speed (Episodes to reach 195 reward): 887</li> <li>• Training Stability (Variance in last 100 episode rewards): 138.00</li> </ul>

To ensure the performances of these two settings could be numerically compared, statistical values were outputted for both settings. The average reward per 100 episodes for both remains consistent around 195, as that is the `stopping_reward` for the environment, with soft updates performing 0.10 better. However, the learning speed is significantly lower for soft updates being 320 episodes slower when compared to periodic target network. Finally, the training stability as represented through variance is slightly higher for period target network at 143.89, demonstrating more instability, as compared to soft updates’ 138 demonstrating that it is more consistent and converges more smoothly in the context of the environment. Soft updates, function similarly to target networks, though they are not as aggressive in their update thresholds (McKusick et. al, 1999). This means other factors such as learning speed and training stability can be optimised because the function is not spending excessive computational power on storage. Soft updates are more suited to smaller, less complex datasets and hence it is well aligned with CartPole-v0 (McKusick et. al, 1999).

#### Question 4 (Complete your full answer to Question 4 on page 4)

- a) A full-scaled version of the graph will be attached to the code submission for more clarity. The first row represents the three distinct learning rates (ordered from smallest to greatest) for CartPole-v0 trained on 'single-hidden' mode.



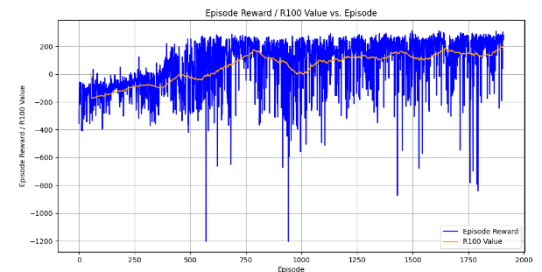
These learning rates were chosen based on the knowledge that training DQN models often has an optimal threshold by which they best operate (Messaoud et. al, 2020). It has also been mentioned that 1.0e-3 is a good learning rate (DataStackExchange, 2019). Despite this knowledge, it can be difficult to choose learning rates because each “run” of training this environment has the potential to produce different results (Messaoud et. al, 2020). Slight, yet significant enough, deviations from the default were chosen. Further, to ensure that there is a baseline the default learning rate, which was provided as 1.0e-3 was also used (Nair).

- b) Performance stability, which is typically greater than 1000 episodes for this environment, is demonstrated in the plot by the point just before the values spike upwards. It is evident that with increasing learning rate, the performance stabilises much sooner with the greatest learning rate (1.5e-3) stabilising at just after 1000 episodes, while it takes the default learning rate 1.0e-3 about 1180 episodes and the smallest learning rate (7.5e-4) approximately 1200 episodes. Similarly, the second distinct factor of these graphs is that the zig-zag quality of the last one-third of each plot. This demonstrates how well CartPole-v0 deals with convergence to a specific value (MaxR/max\_episode\_steps = 200). The frequency of the zigzags is to be noted and more “waves” demonstrate the struggle for that learning rate to ultimately converge the environment (“noisiness” of the data) (Chen, 2024). The default learning rate 1.0e-3 converges the fastest (about 270 episodes) with a very distinct sinusoidal wave pattern. The smallest-valued learning rate 7.5e-4 converges second-fastest (about 500 episodes) with a slightly less distinct sinusoidal double wave pattern, while the largest-valued learning rate 1.5 e-3 converges the slowest (about 770 episodes) with an extremely erratic wave pattern attempting to replicate a traditional sinusoidal wave. It is to be noted that, the episode values to convergence are calculated by subtracting the final episode value with the performance stability value determined earlier.

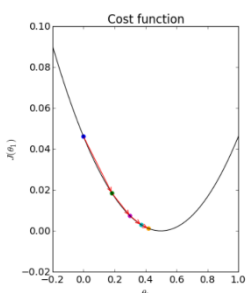
Hence, it is evident that the default learning rate performs the best as it converges the fastest and stabilises steadily. This allows for 1.0e-3 to be the optimal learning rate for CartPole-v0 overall.

- c) LunarLander-v3 under 'single-hidden' was chosen to demonstrate this. Like with part a, ChatGPT helped choose a too high learning rate of 5.0e-3 (faster than the default 1.0e-3). A learning rate that is too high can cause fast training in the start of a run, slowing down rapidly as the data becomes noisier with increased complexity (Chen, 2024).

Hence, it is seen that there an attempt to stabilise until 300 episodes. The too high learning rate seems to converge at 200 by chance rather than following a clear pattern towards convergence. The plot for gradient descent



demonstrates a clear wave like trend towards some quantity. Further, the “noisiness” of the data is demonstrated distinctly by the zigzag feature (Chen, 2024). There are large rewards gained at specific points – 550, 950, 1450, 1800 – due to the increased learning rate. In the extreme haste to train the model fast because of the learning rate’s too high feature, the model struggles to establish a clear, sustainable path to convergence from the very first episode.



**Question 5** (Complete your full answer to Question 5 on page 5)

- a) Epsilon-Greedy ( $\epsilon$ ) is a “simple method to balance exploration and exploitation by choosing between exploration and exploitation randomly”, whereby reinforcement learning attempts to controls the balance between exploration and exploitation. During training, an agent needs to explore the environment (try new actions) to discover optimal strategies, but it also needs to exploit known information (choose the best-known actions). The exploration and exploitation values are known as epsilon hyperparameters. As this is used in  $\epsilon$ -greedy policies, where with probability  $\epsilon$ , the agent explores random actions (first epsilon hyperparameter), and with probability  $1-\epsilon$ , it exploits the action with the highest estimated reward (second epsilon hyperparameter) (GeeksforGeeks, 2023)
- b) In the implementation, the value of epsilon is updated dynamically during training based on the current frame index. As such, it is evident that as the frame index gets larger the final epsilon\_by\_frame value will be less.

```
def epsilon_by_frame(frame_idx, params):  
    return params['epsilon_final'] + (params['epsilon_start'] - params['epsilon_final']) *  
    math.exp(-1.0 * frame_idx / params['epsilon_decay'])
```

- Epsilon Start: This is the initial value of epsilon, params['epsilon\_start']. It determines the amount of exploration at the start of training. A higher value encourages more random actions, ensuring a wide exploration of the environment.
- Epsilon Decay: The parameter params['epsilon\_decay'] controls the rate at which epsilon decays as training progresses. It influences how quickly the agent transitions from exploration to exploitation. A slower decay (larger value) means the agent explores for a longer period, which may improve performance in complex environments.
- Epsilon Final: This is the minimum value epsilon can reach, params['epsilon\_final']. It ensures that the agent continues to explore even in later stages of training, preventing the agent from being stuck in suboptimal strategies by maintaining some level of exploration.

This decay mechanism provides a smooth transition from exploration to exploitation, allowing the agent to gather sufficient information early on and then gradually shift towards exploiting the best-known actions.

## Question 6 (Complete your full answer to Question 6 on page 6)

### Problem with Vanilla DQN Learning:

In Vanilla DQN, each state-action pair is evaluated indiscriminately, even when different actions in certain states lead to similar results. The lack of distinction results in inefficient learning, especially in environments where many actions in numerous states produce similar outcomes. As such, the agent may experience slower convergence and ineffective decision-making. For example, in games like Atari, RL agent's decision to move left or right may not always have a significant impact on the overall performance. Despite this, the Vanilla DQN calculates separate Q-values for each action, which leads to unnecessary computations and increased noise in the learning process.

### How Duelling DQN Fixes This Problem:

Dueling DQN addresses the inefficiencies of the original architecture by introducing a decomposition of the Q-value function into two distinct components: the state-value function ( $V(s)$ ) and the advantage function ( $A(s, a)$ ). The state-value function represents the value of being in a particular state, regardless of the action taken, while the advantage function measures the relative benefit of selecting a specific action compared to other possible actions in the same state. This decomposition allows the network to distinguish between states where the choice of action is critical (high advantage difference between actions) and those where the action has little impact (small advantage difference). Rather than directly predicting Q-values for every state-action pair, the duelling architecture predicts the value of the state ( $V(s)$ ) and the advantage of each action ( $A(s, a)$ ), combining them as:

$$Q(s, a) = V(s) + A(s, a) \text{ (Equation 3)}$$

This approach enables the network to make more accurate distinctions between critical and non-critical actions, resulting in more efficient learning.

### Duelling Combats Vanilla:

The duelling network structure enhances learning efficiency by enabling the agent to focus on the state-value function in situations where the choice of action is less important. This helps to reduce the noise in Q-value estimates when different actions have similar outcomes. At the same time, the agent can prioritise learning the advantage of actions only in states where the choice of action significantly affects the outcome.

Hence, this optimisation is particularly beneficial in complex environments, where some actions can drastically change the outcome. This is not done by Vanilla DQN, as it calculates Q-value updates for trivial actions. Thus, Duelling DQN accelerating learning in environments.

### Improvements:

Dueling DQN offers several improvements over vanilla DQN:

- **Faster Convergence:** by reducing unnecessary learning in states where the choice of action does not matter. This leads to more efficient updates and a quicker learning process.
- **Better Learning Efficiency:** by focusing on state-value learning independently from the action. This is particularly useful in environments where many actions result in similar rewards.
- **Improved Stability** (due to duelling's architecture): by reducing the variance in Q-value updates for less critical states.
- **For Complex Environments:** where certain decisions have a greater impact on the outcome; Duelling DQN enhances performance by effectively learning the importance of states and actions.

In summary, Dueling DQN improves on vanilla DQN by decomposing Q-values into state-values and advantages, leading to faster, more efficient learning, better stability, and improved decision-making in complex tasks.

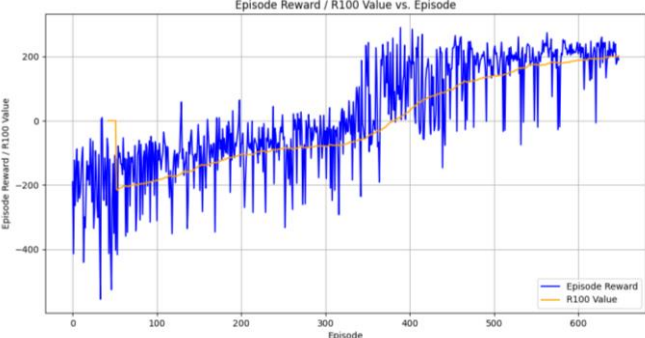
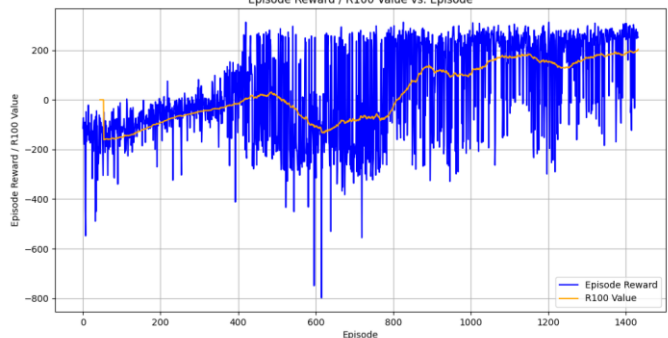
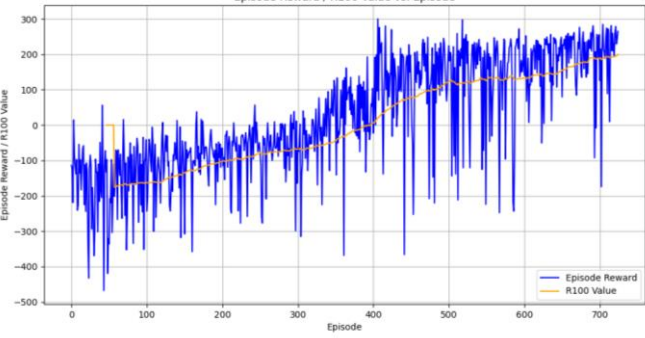
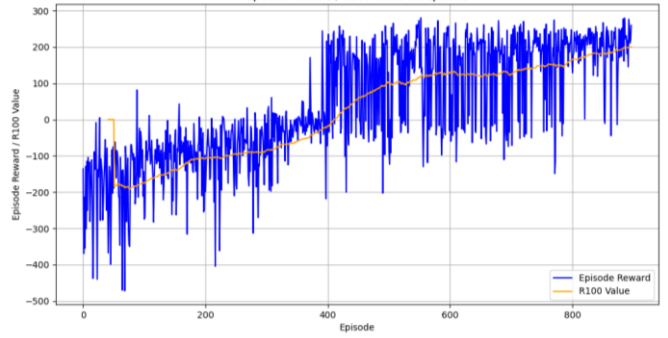
## Question 7 (Complete your full answer to Question 7 on page 7)

### Justifying the Changed Hyperparameters:

From CartPole, it is evident that the hyperparameter thresholds need to be adhered to such that LunarLander-v3 can be trained appropriately. To understand the effect of each hyperparameter, the settings were changed one-by-one. For `learning_rate` and `tau`, slight, yet significant enough, deviations were implemented barring the default threshold and hyperparameters of LunarLander-v3, similar to Question 4 (Nair).

Despite not being previously discussed as much since Question 2, it was decided that the ‘two-hidden’ layers be one of the hyperparameters changed. Purely referencing the `dqn.yaml` file, it is seen that LunarLander-v3 is complex in comparison to CartPole-v1 with significantly larger `hidden_size` and `hidden_size2` values, despite most other hyperparameters being quite similar, hence using more hidden layers in training could potentially improve performance by allowing for more “abstract representations” of the environment (StackOverflow, 2023). It was predicted that like CartPole-v1, LunarLander-v3 would perform well even under the ‘two-hidden’ hyperparameter.

### Plots and Performance Values (Settings in Brackets):

<p>‘default’: (‘single-hidden’, 1.0e-3, 0.005)</p>  <ul style="list-style-type: none"><li>• Average Reward over last 100 episodes: 201.38</li><li>• Learning Speed (Episodes to reach 200 reward): 349</li><li>• Training Stability (Variance in last 100 episode rewards): 3044.74</li></ul>	<p>‘two-hidden’: (‘two-hidden’, 1.0e-3, 0.005)</p>  <ul style="list-style-type: none"><li>• Average Reward over last 100 episodes: 202.92</li><li>• Learning Speed (Episodes to reach 200 reward): 416</li><li>• Training Stability (Variance in last 100 episode rewards): 14916.21</li></ul>
<p>‘learning_rate’: (‘single-hidden’, 2.5e-3, 0.005)</p>  <ul style="list-style-type: none"><li>• Average Reward over last 200 episodes: 200.12</li><li>• Learning Speed (Episodes to reach 200 reward): 405</li><li>• Training Stability (Variance in last 100 episode rewards): 6265.77</li></ul>	<p>‘tau’: (‘single-hidden’, 1.0e-3, 0.003)</p>  <ul style="list-style-type: none"><li>• Average Reward over last 200 episodes: 200.45</li><li>• Learning Speed (Episodes to reach 200 reward): 392</li><li>• Training Stability (Variance in last 100 episode rewards): 3638.60</li></ul>

### Good Policy:

The definition can vary for a ‘good policy’ (StackOverflow, 2023), however according to Garza, a good policy is one that: “will maximise expected rewards”, which is concretely done through “training a neural network” as above (Garza, 2018). As such, it is evident that the ‘two-hidden’ gains this most successfully (although abstractly).

## Appendix/References (Include references and usage of Generative AI on the last page)

Question 1:

automaticaddison. (2019, August 22). Value Iteration vs. Q-Learning Algorithm in Python Step-By-Step. Automatic Addison. <https://automaticaddison.com/value-iteration-vs-q-learning-algorithm-in-python-step-by-step/>

Question 2:

Paszke, A., & Towers, M. (2024). Reinforcement learning (Q-learning) tutorial. PyTorch. [https://pytorch.org/tutorials/intermediate/reinforcement\\_q\\_learning.html](https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html)

Rolnick, D., Veit, A., Belongie, S., & Shavit, N. (2018, February). *Deep learning is robust to massive label noise*. arXiv. <https://arxiv.org/pdf/1705.10694>

COMP3702 Tutorial 11 Code

Question 3:

Ankit. (2023). *Introduction to deep Q-learning: A beginner's guide to understanding DQNs*. <https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/>

Kim, S., Asadi, K., Littman, M., Konidaris, G. (2019). *Deep Mellow-DQN: The mellowmax operator for reinforcement learning*. Brown University. <https://cs.brown.edu/people/gdk/pubs/deepmellow.pdf>

Liu, H. (2024, March 24). *What is deep Q-network (DQN)?* Medium. <https://medium.com/@haleylui388/what-is-deep-q-network-dqn-215b5ff22c04>

McKusick, M. K. & Ganger, G. R. (1999, June 6-11). *The design and implementation of a log-structured file system*. USENIX. [https://www.usenix.org/legacy/publications/library/proceedings/usenix99/full\\_papers/mckusick/mckusick.pdf](https://www.usenix.org/legacy/publications/library/proceedings/usenix99/full_papers/mckusick/mckusick.pdf)

Patel, Y. (2017, July 30). *Reinforcement learning with Keras and OpenAI: Deep Q-learning (DQNs)*. Towards Data Science. <https://towardsdatascience.com/reinforcement-learning-w-keras-openai-dqns-1eed3a5338c>

Saturn Cloud. (2023, August 24). *Temporal difference learning*. <https://saturncloud.io/glossary/temporal-difference-learning/>

Question 4:

Chen, J. (2024, April 30). *Zig Zag indicator*. Investopedia. [https://www.investopedia.com/terms/z/zig\\_zag\\_indicator.asp](https://www.investopedia.com/terms/z/zig_zag_indicator.asp)

Christian. (2016, June 5). Visualizing the gradient descent method. <https://scipython.com/blog/visualizing-the-gradient-descent-method/>

Data Science Stack Exchange. (2019). *What is a minimal setup to solve the CartPole-v0 with DQN?* <https://datascience.stackexchange.com/questions/24513/what-is-a-minimal-setup-to-solve-the-cartpole-v0-with-dqn>

Messaoud, S., Bradai, A., Bukhari, S. H. R., Quang, P. T. A., Ahmed, O. B., & Atri, M. (2020). *A survey on machine learning in Internet of Things: Algorithms, strategies, and applications*. Science Direct. <https://www.sciencedirect.com/science/article/abs/pii/S2542660520301451>

Nair, S. (n.d.). DQN and hyperparameters: Studying the effects of the various hyperparameters. Saasha Nair. <https://saashanair.com/blog/blog-posts/dqn-and-hyperparameters-studying-the-effects-of-the-various-hyperparameters>

Question 5:

GeeksforGeeks. (2023, January 10). Epsilon-greedy algorithm in reinforcement learning. GeeksforGeeks. <https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/>

Question 6:

Wang, Z., Schaul, T., Hessel, M., Hasselt, H. V., Lanctot, M., & de Freitas, N. (2016). *Dueling network architectures for deep reinforcement learning*. Google DeepMind. <https://arxiv.org/pdf/1511.06581>

Question 7:

Garza, G. (2018, January 18). *Deep reinforcement learning: Policy gradients*. Medium. <https://medium.com/@gabogarza/deep-reinforcement-learning-policy-gradients-8f6df70404e6>

Stack Overflow. (2023). *About reward policy in a DQN model*. <https://stackoverflow.com/questions/76068218/about-reward-policy-in-a-dqn-model>