Coursework 2

November 24, 2023

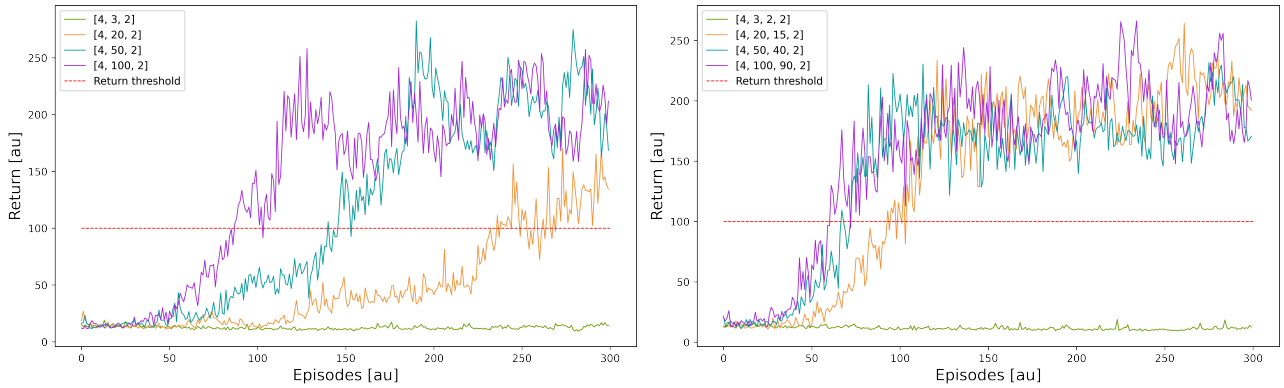
Sara Capdevila Solé
CID: 01727810
sc3719

1 Question 1

1.a DQN Hyperparameter search

In this section, we identify the hyperparameters that need optimisation and outline the ranges and methods to be considered. We maintain a fixed discount factor, $\gamma = 1$. Although this choice may require heightened exploration, fine-tuning other hyperparameters can effectively address any resulting differences. In this context, γ serves as an intrinsic component of the problem itself rather than a variable to be optimised. Additionally, all experiments are conducted for 300 episodes and 10 runs. This configuration strikes a balance, allowing sufficient computational time for the agent to learn and converge (achieving the goal of 50 consecutive episodes with rewards exceeding 100 - the 'Return threshold') while ensuring a representative variance across different initialisations.

DQN architecture: This involves identifying the optimal number of layers and their respective sizes in a DQN. Recent research [1] highlights the necessity for larger NNs than traditionally assumed to circumvent fundamental issues. However, the approach of indefinitely increasing layer sizes poses difficulties, particularly in terms of computational power and training time. Additionally, other studies suggest that smaller models are not only faster but also more manageable for training and hyperparameter fine-tuning [2].



(a) One Layer

(b) Two Layers

Figure 1: **DQN architecture variation:** Mean DQN rewards over 10 runs, with 1 and 2 hidden layers of different sizes. The DQN with 2 hidden layers converges in fewer episodes than the DQNs with 1 hidden layer. The remaining DQN parameters are kept constant with [ϵ-function: Linear, ϵ: 0.5, Activation: ReLu, Buffer size: 2000, α: 0.001, Batch size: 20, Optimiser: Adam, TDQN Update: 8].

The introduction of more hidden layers amplifies the problem's complexity, as optimising a more intricate composition of functions becomes challenging, leading to increased back-propagation time (the further a layer is from the output, the slower it learns). While a preference for deeper rather than wider DQNs exists to learn features at different levels of abstraction rather than memorising, the goal remains to achieve the smallest network capable of delivering satisfactory results. However, our decision to employ two hidden layers stems from practical considerations. While typically, one hidden layer suffices for the majority of problems, the introduction of a secondary hidden layer offers less computational complexity and greater convergence speed [3]. This observation is further supported by the outcomes of our DQN runs, as illustrated in Fig. 1b. Furthermore, based on this analysis, we expect that DQN of width ~ 100, maximises the stability and rewards.

TDQN update: The original update policy involves updating the target network with the weights of the main network after a certain number of episodes [4]. Updating the target network

too frequently, e.g. after every episode, can nullify the benefits of using a target network, reducing stability. Conversely, updating after too many episodes may lead to a reduction in the accuracy of the target DQN (TDQN), as it no longer accurately mirrors our trained DQN. A balance is crucial, and for our DQN with a total episode cycle of 300, the optimal frequency is expected to be around $\sim 2-15$ episodes, drawing inspiration from literature, e.g., [5].
Soft policy updates were also considered. This is when in each episode a ratio of weights, parameterised by τ, are retained in the TDQN[6][7]. However, after experimentation, we opted to adhere to the hard update. This ensures that our TDQN swiftly adapts to policy changes, particularly crucial for rapid convergence at the beginning of episodes. Which is well-suited for non-highly dynamic environments like CartPole.

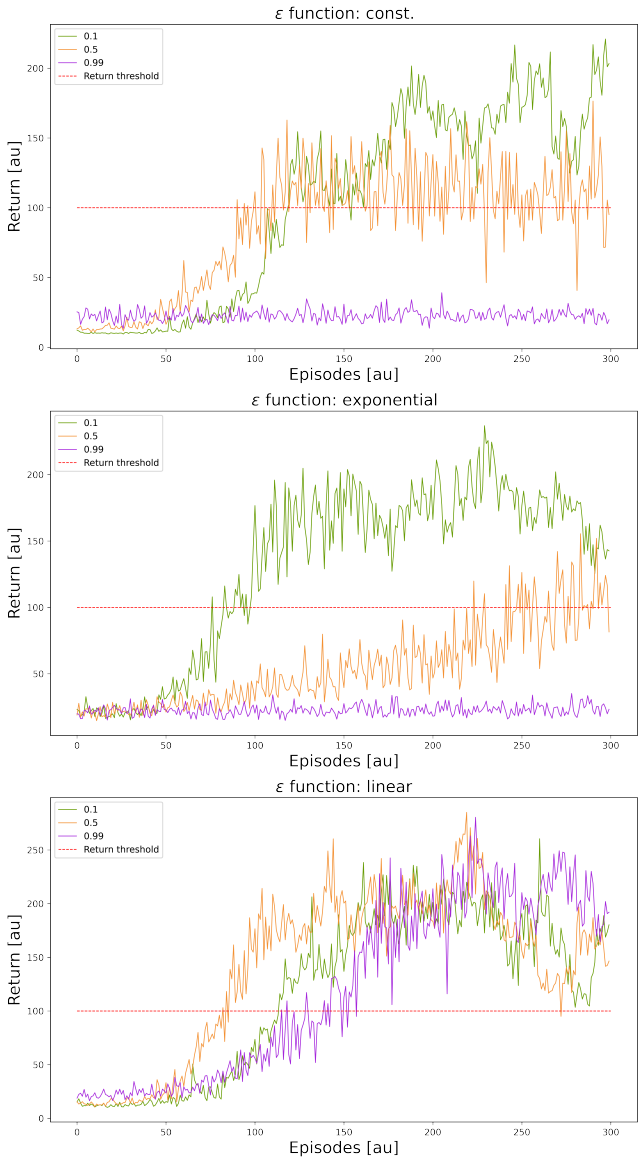
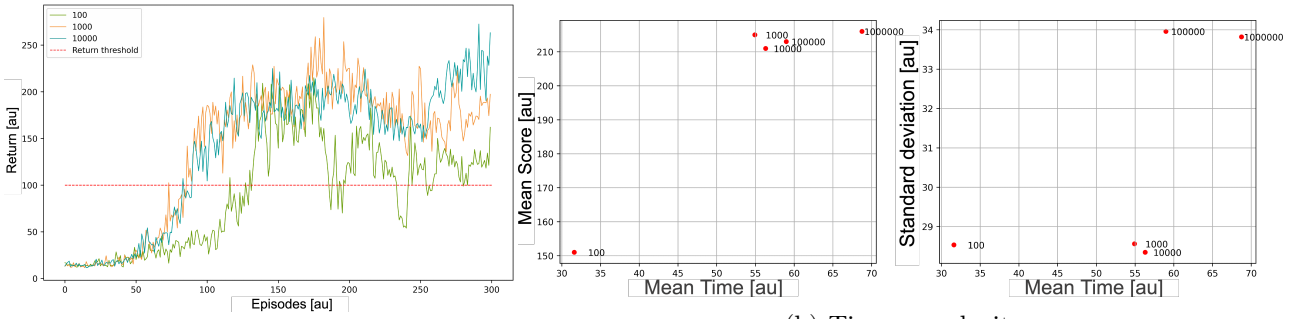


Figure 2: **DQN ϵ-function and ϵ variation:** Mean DQN rewards over 10 runs, with ϵ-functions: [Linear, Exponential, Constant] and ϵ : $[0.1, 0.5, 0.99]$. The reward threshold at 100 is plotted in red. The remaining DQN parameters are kept constant with [Activation: ReLu, Buffer size: 2000, α: 0.001, Batch size: 20, Optimiser: Adam, Architecture: $[4, 32, 16, 2]$, TDQN Update: 8].

ϵ-function and ϵ: We consider three different functions for ϵ-decay with an ϵ-greedy policy. The first is an ϵ-function annealed linearly from an initial value of ϵ to a final value of 0.01, drawing inspiration from the original DQN paper [4]. Building upon this, we implement a different ϵ-function, annealed exponentially from an initial value of 1 to a final value of ϵ. Finally, we consider a constant ϵ-function.
These functions exhibit distinct dependencies on the choice of ϵ, underscoring the importance of visualising its variations, as depicted in Fig. 2. The most stable performance overall is achieved with the Linearly decaying ϵ-function, showing optimal results at approximately $\epsilon \approx 0.5$.

Buffer size: While existing literature often suggests a fixed replay buffer size of 10^6 (e.g. [4], [8], [9]), we instead draw inspiration from Sutton & Zhang's paper [10] and decide to explore the impact of buffer size on DQNs complexity. We assess the time complexity increase while varying the buffer size. To achieve this, we compute the mean count and variance above the threshold and run time. Both metrics are plotted in Fig. 3b. As illustrated, as the memory capacity of the agent increases, it's exposed to a higher number of state-action pairs to batch from, but this improvement comes at the cost of higher training time. For clarity, fig. 3a focuses on visualising the first three orders of magnitude of batch sizes.
From these plots, a buffer size of approximately 10^4 is preferable, to achieve a balance between rewards, stability, and run time. Increasing the experience replay size enhances the stability of the DQN and prevents *Catastrophic forgetting* [11]. However, beyond a

threshold, the buffer becomes too large, resulting in unstable learning - the agent encounters more random transitions.



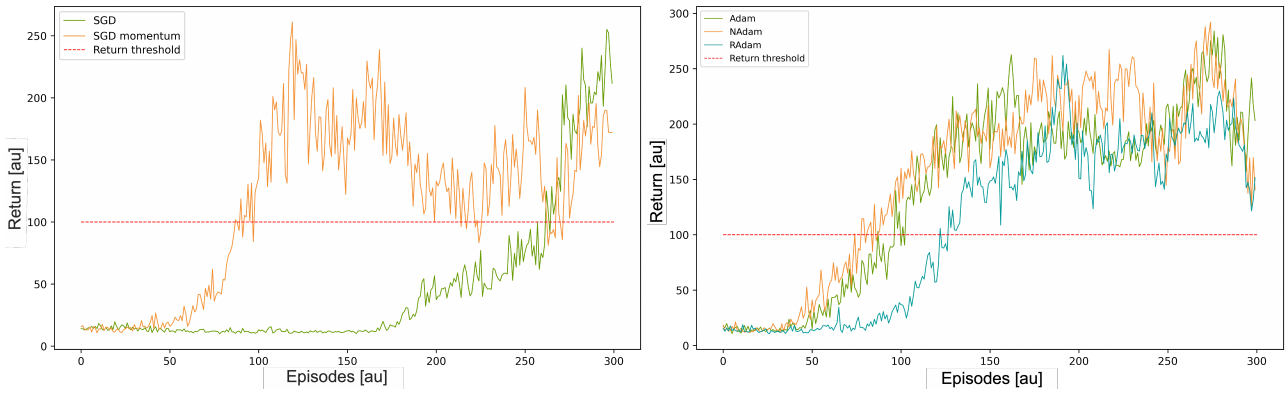
(a) Mean DQN rewards

(b) Time complexity

Figure 3: **DQNs performance variation with different Buffer size:** a) Mean DQN rewards over 10 runs, with Buffer sizes: $[10^2, 10^3, 10^4]$, and b) Time complexity per run plotted against the mean and variance over 10 runs, using Buffer sizes: $[10^2, 10^3, 10^4, 10^5, 10^6]$. The remaining DQN parameters are kept constant with [ϵ-function: Linear, ϵ: 0.5, Activation: ReLu, α: 0.001, Batch size: 20, Optimiser: Adam, Architecture: $[4, 32, 16, 2]$, TDQN Update: 8].

Batch size: In the training process, a random batch of transitions is sampled from the buffer, disrupting the temporal correlation between consecutive samples and improving learning stability. A smaller batch size introduces more variability in updates, potentially facilitating faster convergence but with increased variance. Conversely, excessively large batch sizes may decrease variance but could impede the learning process.
While ensuring the batch size is always smaller than the buffer size, we explore values between $10 - 100$. According to [12], increasing the batch size to around ~ 500 yields optimal performance in terms of convergence and stability for the agent, albeit at the expense of computational time. After experimenting with different batch sizes, we find that values between $10-60$ strike a favourable compromise between time complexity and convergence. Larger batch sizes are not considered, as we aim for the smallest batch size that enhances the agent's learning and stability, given that excessively large batch sizes can degrade model quality [13].

Optimisation function and α: The optimisation function is sometimes considered to not be of great importance, as the optimiser's parameters (such as learning rate, α, and weight decay) can be fine-tuned to maximise its performance in each problem-setting.
Gradient descent is the most well-known optimiser owing to its simplicity [14]. Over time, accelerated versions have been introduced, incorporating stochasticity (SGD) or momentum. The Adam optimiser, utilises the first and second moments of gradients to adaptively scale learning rates (α), it has demonstrated faster convergence and increased robustness to noisy gradients, particularly at the beginning of training [15]. Introducing Nesterov's accelerated gradient (NAG) momentum, a widely used technique in the deep learning community [16], gives rise to a variant known as NAdam. RAdam, another variation of Adam, distinguishes itself by incorporating a rectified α.
Experimenting with our DQN, we plot the variation of agents learning with SGD with and without momentum in Fig 4a, and Adam and its two variations NAdam and RAdam in Fig. 4b, all at a fixed $\alpha = 0.001$. These results clearly justify the literature. We will proceed to use NAdam as our chosen optimiser, due to its convergence speed and lower variance. Nonetheless, these plots all ran on a fixed α which could bias our choice. However, this choice has been justified by literature and in the next section we will adjust the rest of the hyperparameters to be optimised for the performance of NAdam, using $\alpha \in [1e-1, 1e-4]$ [17], [6]. Finally, the weight decay of the optimiser was kept at 0, as the risk of overfitting to noise is relatively low in this environment, and introducing it degrades the agents learning.



(a) SGD optimiser (b) Adam optimiser

Figure 4: **Optimiser functions:** DQN performance with difference optimising functions; SGD and SGD with momentum in a), and Adam, NAdam and RAdam in b). The remaining DQN parameters are kept constant with [ϵ-function: Linear, ϵ: 0.5, Activation: ReLu, Buffer size: 2000, α: 0.001, Batch size: 20, Architecture: $[4, 32, 16, 2]$, TDQN Update: 8].

Activation function: Although predicting the optimal policy involves classification, in the context of DQNs, it predicts the optimal policy from the Q-values. Therefore, we consider the Rectified Linear Unit (ReLU) and its variations.
ReLu is a common activation function as it introduces some non-linearity to DQNs, allowing them to learn more complex functions. However, it suffers from the problem of dying neurons [18]. Leaky ReLU has been introduced as a solution to address this issue. In contrast to ReLU, Leaky ReLU assigns a non-zero slope to the negative part, preventing complete dropout of the negative values.
In the studies presented in [19] and [20], various modifications of ReLU were found to consistently outperform the original ReLU. In these respective studies, Leaky ReLU demonstrated an average performance improvement of approximately 20% and 35.4% compared to ReLU across various networks. This performance was measured by assessing variance and validation errors.

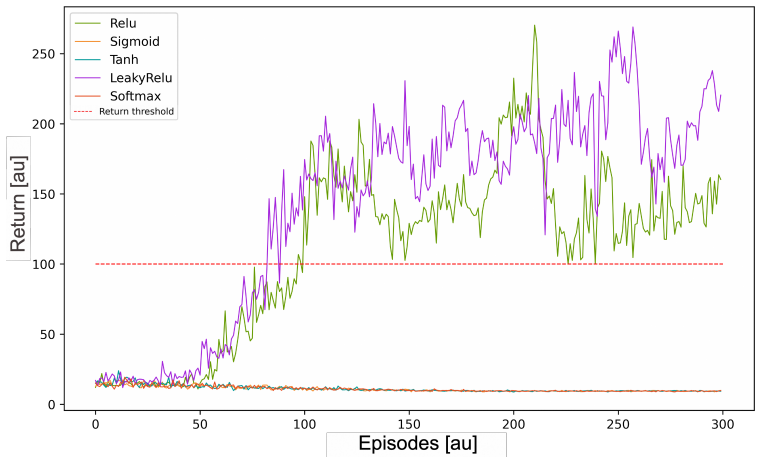


Figure 5: **Activation functions comparison:** Plot showing the mean over 10 runs of the DQN, for activation functions: [ReLu, LeakyReLu, Sigmoid, Tanh, Softmax], applied in each layer except the output. The remaining DQN parameters are kept constant with [ϵ-function: Linear, ϵ: 0.5, Optimiser: Adam, Buffer size: 2000, α: 0.001, Batch size: 20, Architecture: $[4, 32, 16, 2]$, TDQN Update: 8].

To assess this performance difference in our DQN, we plot in Fig. 5, the reward variation for each of these activation functions, applied to every layer. Similar trends were obtained when employing different activation functions in different layers, with LeakyReLu still outperforming others in mean rewards and variance. Also plotted in the figure is the performance of some activation functions used in classification problems, which fail terribly.

1.a.1 Optuna fine search

Leveraging insights from both literature and experimental findings, we have narrowed down our hyperparameters to a concise and refined range. To obtain the most optimum hyperparameter values, we will perform a probabilistic search using a 'Tree-structured Parzen Estimator' [21] by evaluating the performance of each hyperparameter on the DQN by defining two objective functions. The first one measures the number of episodes in which the mean of the DQN run obtains a reward surpassing 100, referred to as the 'count score'. Simultaneously, we compute the variance of the agent's learning over different runs. Maximising and minimising the first and second objective functions concurrently, results in an agent which has a stable and fast learning. To do this, we use an exisiting hyperparameter optimisation framework called `Optuna`. Given the hyperparameter ranges that have not been specified in the previous section, we optimise (ϵ, Buffer size, Learning rate, Batch size, TDQN Update) while keeping the other hyperparameters fixed. This is shown in Table 1.

Hyperparameter	Range
Size of hidden architecture	$[40] - [200]$
ϵ	$[0.2] - [0.8]$
Buffer size	$[1e4] - [9e4]$
Learning rate	$[1e-1] - [1e-4]$
Batch size	$[10] - [60]$
TDQN Update	$[2] - [10]$

Table 1: **Optuna search space:** Hyperparameters and their corresponding ranges, initialised for the `Optuna` search. The other hyperparameters of the DQN were fixed at [ϵ-function: Linear, Number of layers: 2, Optimiser: NAdam, Activation function: LeakyReLu].

1.b Optimised Hyperparameters and Learning curve

The learning curve of the optimised DQN, utilising the refined hyperparameters are outlined in Table 2. The resulting optimised learning curve is depicted in Fig. 6. As intended, the agent consistently attains rewards exceeding 100 within the initial 30 episodes. This marks a substantial improvement compared to the DQN employing unoptimised hyperparameters.

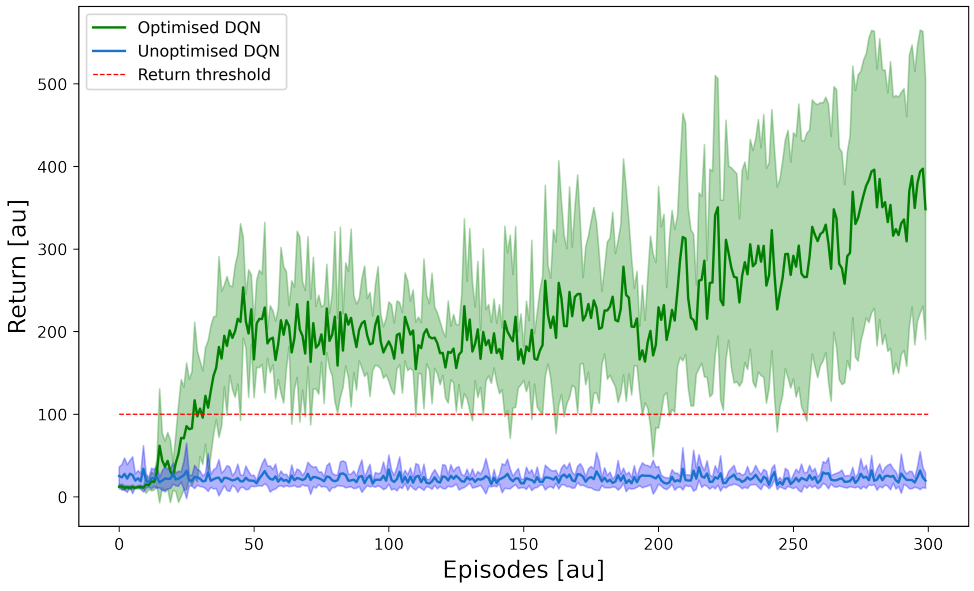


Figure 6: **Optimised Learning curve of the DQN:** The learning curve of the optimised DQN (green) uses hyperparameters from Table 2, showcasing significant improvement compared to the DQN with unoptimised hyperparameters (purple). The mean and standard deviation (surrounding fill) are plotted for each DQN over 10 runs and 300 episodes.

Hyperparameter	Value	Justification Summary
ϵ-function	Linear	Chosen for its higher stability in comparison to exponential and constant functions - see Fig. 2.
ϵ	0.3	Range identified using Fig. 2, and value optimised using Optuna for most stable learning.
Architecture	$[4, 140, 80, 2]$	Two layers (Fig. 1b) has faster convergence properties than one (Fig. 1a), reducing the time complexity of the problem, as justified in [3]. The width of the layers is optimised using Optuna.
Activation Function	LeakyReLu	Predicting Q-values requires an activation function with non-restricted ranges. We choose LeakyReLu over ReLu based on existing studies e.g. [19] & [20] and Fig. 5.
Buffer Size	60000	Range obtained based on a balance between obtaining a high mean reward and small learning variance from Fig. 3a. The precise value is optimised using Optuna.
Optimiser	NAdam	Chosen over Adam and its variations for better stability and mean rewards, as observed in Fig. 4, and also [16].
Batch Size	20	Range justified in [12] & [13], to prevent model degradation, and specified using the Optuna search.
α	0.001	Range extracted from [6] & [17] and optimised using the Optuna search.
TDQN Update	2	Optimised using the Optuna search, once range had been found experimentally which maximises rewards and minimises variance of learning.

Table 2: **Optimised DQN:** Hyperparameters, values, along with a summary justification.

2 Question 2

2.a Slices of the greedy policy action

Using the optimised hyperparameters detailed in Table 2, we can visualise slices of the greedy policy. By keeping the position of the cart fixed at 0 m, we can depict the actions taken by our agent as the angular velocity (ω) and pole angle from the normal (θ) change. This 2-dimensional plot is explored with various cart velocities, specifically $v = 0, 0.5, 1, 2$ ms^{-1}. The visualisation is presented in Fig. 7.

The observed patterns in these plots align with our expectations. When the cart velocity is at $v = 0$ ms^{-1}, we anticipate that positive values of both ω and θ would result in a clockwise torque on the pole. To counteract this force and maintain balance, pushing the cart to the right becomes the optimal action. This logic extends to situations with large values of θ and/or ω. At larger, constant velocities, the pole essentially remains in the same inertial frame of the cart, and the primary force is gravity. When θ and ω are both zero, the pole experiences no force, but the cart is on track to reach the environment's boundaries at $x = \pm 2.4$ m [22]. Consequently, the predominant action becomes pushing the cart to the right, generating a negative angular velocity, which is when the optimal action becomes pushing the cart to the left, to balance the

pole back upright. For instances with very large values of θ and ω, pushing the cart to the right no longer remains the optimal action, as this often results in a pole angle greater than $\theta > 0.2094\,\text{rad}$, which terminates the environment. Accordingly, these ranges exhibit smaller Q-values, as neither action gives the state an expected high future reward, as shown in Fig. 8.

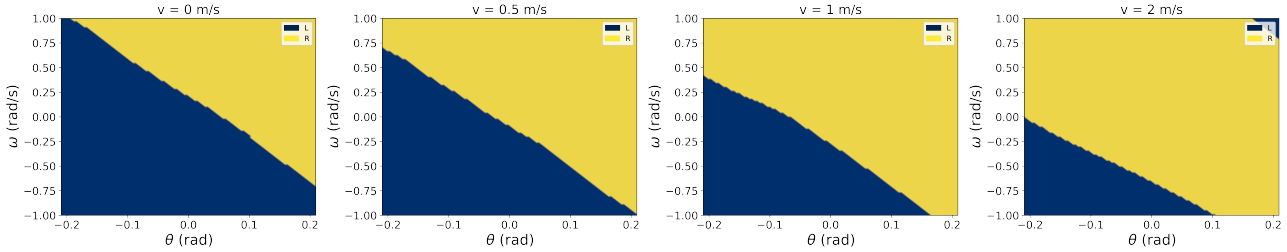


Figure 7: **Slices of the greedy policy action at different velocities (v) in** $\text{m}\,\text{s}^{-1}$**:** The optimal policy of pushing the cart 'Left' (L) or 'Right' (R) depending on the angular velocity (ω) and angle from the normal (θ), of the pole. The action L is depicted in Blue, and R is depicted in yellow. These plots have been computed using the mean Q-values over 10 runs.

At equilibrium, initiating the push to the right before the left is preferable, as this results in a larger acceleration of the cart and, consequently, a larger torque in the anticlockwise direction. Balancing this torque by subsequently pushing the cart to the left can, in some cases, lead to a temporary equilibrium at a negative θ, requiring another push to the left. Therefore, the initial action of pushing the cart to the right prolongs the duration of the environment - achieving a higher reward.

2.b Slices of the Q-values

The averaged Q-values over 10 runs for the same ranges are depicted in Fig. 8. The plot reveals a maximum Q-value of around 200 for the most optimal state-action pairs, consistent with the expected return illustrated in Fig. 6. As anticipated, the Q-values generally decrease with larger absolute values of v, θ and ω, as the agent is closer the the environment's edges. As the velocity increases, the most valued state-action pairs shift towards the negative θ and ω range, aligning with the analysis presented in the previous subsection. The highest Q-values align with an unstable equilibrium of the pole - where the policy changes from left to right in Fig. 7. This is intuitive, as v increases, the impact of having a large positive ω and θ becomes more significant, and a push to the right will lead to the environment terminating sooner.

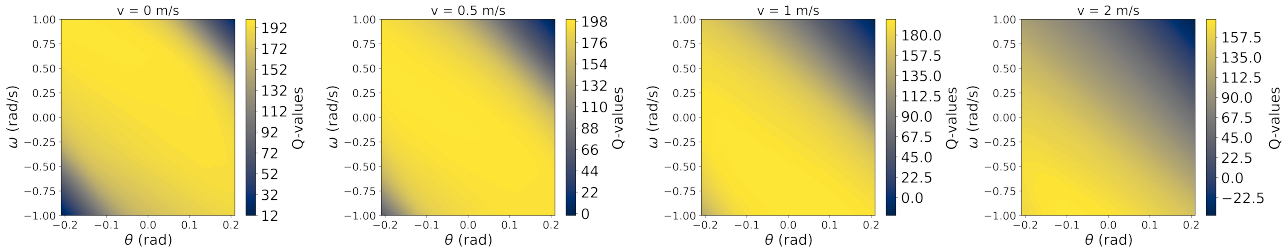


Figure 8: **Slices of the Q-values at different velocities (v) in** $\text{m}\,\text{s}^{-1}$**:** Plots showing the mean Q-values over 10 runs, plotted on a blue (low) to yellow (high) colour map scale, as a function of the angular velocity (ω) and angle from the normal (θ), of the pole.

The presence of the negative Q-values can be due to the choice of the NAdam optimiser, overshooting the loss minimisation. If this was a concern, a final layer activation function that ensures positive output values could be implemented, such as ReLu or a scaled Sigmoid function. Nonetheless, the same general relationship of the Q-values will still hold.

References

[1] Sebastien Bubeck and Mark Sellke. "A Universal Law of Robustness via Isoperimetry". In: *Advances in Neural Information Processing Systems*. Ed. by A. Beygelzimer et al. 2021. URL: https://openreview.net/forum?id=z71OSKqTFh7.

[2] Andreas Loukas, Marinos Poiitis, and Stefanie Jegelka. "What training reveals about neural network complexity". In: *Advances in Neural Information Processing Systems*. Ed. by A. Beygelzimer et al. 2021. URL: https://openreview.net/forum?id=RcjW7p7z8aJ.

[3] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, May 2010, pp. 249–256. URL: https://proceedings.mlr.press/v9/glorot10a.html.

[4] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[5] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533. URL: https://api.semanticscholar.org/CorpusID:205242740.

[6] Nesma M. Ashraf et al. "Optimizing hyperparameters of deep reinforcement learning for autonomous driving based on whale optimization algorithm". In: *PLOS ONE* 16.6 (June 2021), pp. 1–24. DOI: 10.1371/journal.pone.0252754. URL: https://doi.org/10.1371/journal.pone.0252754.

[7] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2019. arXiv: 1509.02971 [cs.LG].

[8] Marcin Andrychowicz et al. *Hindsight Experience Replay*. 2018. arXiv: 1707.01495 [cs.LG].

[9] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279. ISSN: 1076-9757. DOI: 10.1613/jair.3912. URL: http://dx.doi.org/10.1613/jair.3912.

[10] Shangtong Zhang and Richard S. Sutton. *A Deeper Look at Experience Replay*. 2018. arXiv: 1712.01275 [cs.LG].

[11] James Kirkpatrick et al. "Overcoming catastrophic forgetting in neural networks". In: *Proceedings of the National Academy of Sciences* 114.13 (Mar. 2017), pp. 3521–3526. ISSN: 1091-6490. DOI: 10.1073/pnas.1611835114. URL: http://dx.doi.org/10.1073/pnas.1611835114.

[12] Adam Stooke and Pieter Abbeel. *Accelerated Methods for Deep Reinforcement Learning*. 2019. arXiv: 1803.02811 [cs.LG].

[13] Nitish Shirish Keskar et al. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. 2017. arXiv: 1609.04836 [cs.LG].

[14] Augustin Cauchy et al. "Méthode générale pour la résolution des systemes d'équations simultanées". In: *Comp. Rend. Sci. Paris* 25.1847 (1847), pp. 536–538.

[15] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[16] Yu E Nesterov. "A method for solving the convex programming problem with convergence rates". In: *Dokl. akad. nauk Sssr*. Vol. 269. 3. 1983, pp. 543–547.

[17] Vidhi Chugh. *Tuning Adam Optimizer Parameters in PyTorch*. Accessed on 20/11/2023. URL: https://www.kdnuggets.com/2022/12/tuning-adam-optimizer-parameters-pytorch.html#:~:text=An%20optimal%20learning%20rate%20value,in%20most%20of%20the%20cases..

[18] Lu Lu. "Dying ReLU and Initialization: Theory and Numerical Examples". In: *Communications in Computational Physics* 28.5 (June 2020), pp. 1671–1706. ISSN: 1991-7120. DOI: 10.4208/cicp.oa-2020-0165. URL: http://dx.doi.org/10.4208/cicp.OA-2020-0165.

[19] Bing Xu et al. *Empirical Evaluation of Rectified Activations in Convolutional Network*. 2015. arXiv: 1505.00853 [cs.LG].

[20] LeeDongcheul. "Comparison of Reinforcement Learning Activation Functions to Improve the Performance of the Racing Game Learning Agent". In: *Journal of Information Processing Systems* 16.5 (Oct. 2020), pp. 1074–1082.

[21] Shuhei Watanabe. *Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance*. 2023. arXiv: 2304.11127 [cs.LG].

[22] Filippo Valdettaro Charles Pert Prof Aldo Faisal. *Assessed Coursework 2 - Reinforcement Learning*. Accessed on 10/11/2023. URL: https://scientia.doc.ic.ac.uk/api/2324/70028/exercises/2/spec.