Deep Reinforcement Learning
– Assignment 2: Deep Q-Learning –

Lukas Welzel [1] **Rens Kievit** [1] **Benjamin Silk** [1]

Abstract

In this report we attempt to apply and study a Deep Q-Learning algorithm to the cartpole environment. Using an ablation study as well as multiple grid-search like approaches we search the hyper parameter space for the optimal configuration. We find that Deep Q-Networks rely heavily on experience replay and target networks for stability. Furthermore, we find that a major improvement of the Deep Q-Networks performance can be achieved through soft target network updates. Finally, we discuss a novel strategy for prioritized experience replay.

1. Introduction

In this section we present the two main components of this project. In Section 1.1 we provide a brief description of the environment, and how the agent can interact with it. In Section 1.2 we describe the baseline version of our Deep Q-Learning neural network, and the learning algorithm which we apply to it.

1.1. Environment

The environment used in this report is that of a simple cartpole first described by Barto et al. (1983), and implemented by Brockman et al. (2016) on `OpenAI Gym` as `CartPole-v1`. The cartpole consists of a rectangular cart which can move along a horizontal axis, and a pole, attached with a single joint to the cart. This entire system is described by four parameters: the cart position x, the cart velocity $\dot{x}$, the pole angle θ and the pole angular velocity $\dot{\theta}$. These are all contained in the state vector s_t. The goal of the agent is to balance the pole upright ($\theta \in [-12^o, +12^o]$) for as long as possible without deviating too far from the center ($x \in [-2.4, +2.4]$).

The agent interacts with the environment by applying a fixed force, either to the right or left, at each timestep to the cart. It receives a reward of $+1$ for each survived timesteps, and it has 'won' an episode once a total reward of 500 is achieved. For most studies we have limited the maximum reward to 200 to keep the runtime of the studies in an acceptable range. In the plots we indicate the maximum rewards using dashed lines.

1.2. Method

In this report we implement a Deep Q-Learning neural network of which several parts can be en-/disabled for an ablation study. The general algorithm for training the agent is shown in algorithm 1. After initializing the agent, its replay buffer is filled through interacting with the environment $\mathcal{E}$. Importantly, the agent does not start learning until the buffer $\mathcal{D}$ is completely filled. Afterwards the agent starts learning by interacting with the environment for $N_{episodes}$. One important issue with this kind of simple training is that there is no selection for agents who have "finished" training. Furthermore, agents who perform poorly complete the learning much faster. This can mean that, if a study is stopped prematurely due to its runtime, the study is unbiased up-to the episode where the agent with the lowest $N_{episodes}$ completed and biased to poorly performing agents for any $episode > \min_{agent}(N_{episodes})$.

One major issue of the studies we performed is that the different DQN were not trained on the same number of timesteps. A DQN which learns quickly and reaches a high mean reward after few episodes will have more total timesteps available to train on than a DQN which initially returns low rewards. This is intended as slow learning DQN (if not improving after a sufficient number of episodes) are of less interest for this environment and, take up CPU/GPU time that could better be spend on quicker learning agents. The difference between the amount of timesteps for quick and slow learning DQN is around an order of magnitude.

1.2.1. Deep Q-Learning Agent

Architecture The neural network itself takes the state of the environment as inputs (x, $\dot{x}$, θ, $\dot{\theta}$) and outputs a Q-value for each of the two possible actions the agent can take (push left or push right). By default (see Table 1), the network

[1]Faculty of Science, Leiden University, The Netherlands. Correspondence to: L. Welzel <welzel@strw.leidenuniv.nl>, R. Kievit <kievit@strw.leidenuniv.nl>, B. Silk <silk@strw.leidenuniv.nl>.

contains 3 hidden layers with nodes of size [512, 256, 64]. Each hidden layer uses a ReLU activation function and the weights are initialized using the He Uniform scaling variance initializer. The MSE of the losses is optimized by the network during training using Root Mean Square Propagation. Some of these hyperparameters (e.g. hidden layer architecture) were varied and compared (see Section 2).

In the absence of a Target Network, introduced in the next paragraph, this Deep Q Network serves a similar function as a Q-Table in Tabular Q-Learning. For any state an agent finds itself in, the DQN finds the Q-value for each action, and the agent can select an action according to a given exploration strategy. Similarly to how the Q-table is updated with new Q-values, the DQN is updated with new weights.

Target Network In the algorithm as described in algorithm 1, the network from which the optimal next action is sampled is the same network that is updated after each episode. This can introduce a large instability factor in the learning process because the current knowledge of the network is biased towards the most recent states it observed. Thus, an agent that starts slanted to the right in a few consecutive runs, might overwrite all its knowledge of what it must do when the pole is slanted to the left.

To mitigate this instability, we create a second, identical, network which we call the **Target Network** (TN). From now on, we will refer to the existing network as the **Main**, or online, **Network**. Just as before, we still update the main network after each episode. However, instead of sampling from this quickly varying network, we sample our actions and optimal next action from the target network. In order to still ensure behavioural change of the agent, we copy the weights from the main network to the target network after every episode.

We implement the TN update as shown in algorithm 3. We implement two options for softened updates to reinforce the strength of using the TN. These soft updates use a softness parameter τ to soften the change so that the TN "remembers" its previous weights. The single-step linear update results in an exponential decay of the previous weights as the TN is re-updated n times with $\theta'_n \propto \tau^n$, where θ' are the frozen network weights of the TN. The other option we implement is a soft update with a varying τ, based linearly on the performance of the agent (measure via the reward) in the most recent episode.

Experience Replay The buffer itself is a cyclic tensor with capacity (depth) $n_{\mathcal{D}}$, storing $n_{\mathcal{D}}$ transitions $(s, a, r, s', d)_{\mathcal{T}}$ so that the oldest transition inside it is always $n_{\mathcal{D}}$ timesteps old. When new transition are added they always overwrite the oldest transition. The agent can sample a batch of $m_{\mathcal{D}} \leq n_{\mathcal{D}}$ transitions from the buffer. The sampled batch is neither ordered nor sequential and has no repeated elements. We implement no prioritization of transitions. The replay buffer is implemented to enable experience replay (ER) and can be understood as the agents "memory". ER has two primary functions:

1. **Convergence & Stability:** A major problem for DQN is stability. We train the network based on transitions and it then goes on to choose the next action by predicting the next rewards. This means that the network quickly becomes very specialized on predicting the rewards in the local state space. Hence, when leaving this region, the network becomes very bad at predicting rewards as it is lacking generality. This problem becomes worse the more specialized the network is for predicting rewards in the local region $|s_{agent} - \mathcal{S}_{local}| < \epsilon$, when moving to any region sufficiently far removed from $\mathcal{S}_{local}$ (see Figure 9). When the network is updated not only on the current state but also on states in its memory, given that the buffers capacity is high enough to contain a sufficient number of transitions that can sufficiently well approximate the Q-function. (Fedus et al., 2020) show that what is "sufficient" in this context is a non-trivial question, however a large capacity is essential, and a larger capacity is often preferable as the benefits of ER often persist until the buffer contains too many off-policy transitions.

2. **Learning efficiency:** Interacting with the environment can be expensive (e.g. in computational time or risk) whereas interacting with the model is generally inexpensive and risk-free. Without ER each transition is only used for learning once and discarded afterwards. However, Q-learning updates its Q-value function incrementally so that the agent does not fully learn the transition. Intuitively then, transitions can be exploited more than once for learning. This means that the potentially expensive transition can be more fully learned without having to re-experience it which is more efficient. Naturally, this is especially efficient if the local Q-value function is smooth (i.e. locally low variance in the transitions).

ER is implemented for the agent as shown in algorithm 2. Computing the loss and the network update are handled internally by `tensorflow/keras`. We implemented an efficient algorithm for computing the Bellman optimality equation that does not iterate over the batch using `numpy` and the bool-to-float casting of `numpy.arrays`. The ER works either with only the main network or the main and target network. We use the algorithm 2 also where ER is disabled, with $n_{\mathcal{D}} = m_{\mathcal{D}} = 1$, i.e. a buffer with only the most recent experience. Furthermore, we adjust the reward

that gets saved into the ER, if the episode ends without reaching the maximum reward (the agent fails the balancing) we push a transition into the buffer where $r = -0.5r_{max}$ so that the agent has extreme prejudice against transitions that have previously ended an episode prematurely.

1.2.2. EXPLORATION

During the learning process, we want the agent to both explore the environment searching for the optimal policy, and exploiting actions it knows are good to reinforce those strategies. In this report we use the ϵ-greedy approach, described by Equation 1, however in Section 2.3 we explore the effect of a different exploration strategy, the Boltzmann approach, described by Equation 2.

The **ϵ-greedy policy** is described by the exploration parameter $\epsilon \in [0, 1]$. It takes a random action a fraction of ϵ of the times, and otherwise takes the optimal action.

$$\pi(a|s) = \begin{cases} 1.0 - \epsilon + \epsilon/(|\mathcal{A}|), & \text{if } a = \arg\max_{b \in \mathcal{A}} \mathcal{X}(s, b) \\ \epsilon/(|\mathcal{A}|), & \text{otherwise} \end{cases} \tag{1}$$

The **Boltzmann policy**, also referred to as **softmax** is described by the temperature $\tau \in [0, \infty)$. It creates a probability distribution over the actions using the exponential of the network estimated action reward. For $\tau \to \infty$, this distribution becomes uniform.

$$\pi(a|s) = \frac{e^{\mathcal{X}(s,a)/\tau}}{\sum_{b \in \mathcal{A}} e^{\mathcal{X}(s,b)/\tau}} \tag{2}$$

Additionally, one would ideally want the agent to explore a lot at the beginning of its learning process, where it has little to no knowledge of the environment, and exploit as much as possible once we think the learning process has converged to an optimal value. In this report, we do this using an exponential decay factor r, diminishing the exploration factors (ϵ, τ) after each episode.

2. Results

To ensure optimal performance of our Deep Q-Learning algorithm, we vary a number of the hyperparameters present used in the learning process. We perform this search by choosing a reasonable value for each of the hyperparameters, presented in Table 1, and then varying only one of them over a few selected additional values. Each of these settings is run for 250 episodes, each with a maximum duration of 200 timesteps. This is enough to observe differences in learning speeds between different settings, but ensures that if agents get too smart, they do not take up too much processing time. In the following figures the shaded regions demarcate regions withing a confidence interval of 1σ based on the not smoothed episode rewards.

Learning Rate α	0.01
Discount Factor γ	0.90
Starting Exploration Factors ϵ	1.00
τ	1.00
Exploration Strategy	ϵ-greedy
Annealing Method	Exponential
Decay Factor r	0.99
Target Network Update Frequency	1/episode
Buffer Depth	2000
Batch sample Size	100
No. Hidden Layers	3
No. Nodes per Hidden Layer	512, 256, 64
Hidden Layer Activation Function	ReLU
Weight Initialisation	HeUniform
Optimizer	RMSprop

Table 1. All hyperparameters used in the Deep Q-learning algorithm, with selected reasonable default values.

2.1. Ablation Study

The results of the ablation study are shown in Figure 1. The full DQN easily outperforms the other DQN and the ER seems to be more important than the TN, even though the DQN can not sufficiently exploit its memory without a TN. The main issue for DQN without ER is that they easily overfit and, when in a state far away from the recent state space region, fail to generalize.

2.2. Learning Rate & Discount Factor

We vary the learning rate α and discount factor γ to compare how our Agent performs for different values. We present the learning curves for the learning rates (0.01, 0.1, 0.25, 0.5) as well as the discount factors (0.1, 0.3, 0.6, 0.9).

Although each result is imprecise with just 3 runs to compare for each setting, we do see a clear correlation between agent performance and discount factor. For γ closer to 1.0, the average total reward is larger. The same cannot be said for the learning rates we have tested. Although $\alpha = 0.25$ seems to have a slight edge over the others, with the small sample size of runs it seems premature to conclude that this learning rate will consistently give better performance.

2.3. Exploration

As described in subsubsection 1.2.2, as a baseline we use an ϵ-greedy approach for exploration. In this section we present a comparison in the learning curves between ϵ-greedy- and Boltzmann-exploration. We make use of exponential annealing of the exploration factor in both cases. As an addition, we also provide an ϵ-greedy approach where we subtract $\epsilon/N_{episodes}$ from ϵ after each episode. The results, averaged over three learning iterations are presented in Figure 4.

From these results, we observe that the performance of both exploration strategies is approximately equal. After approx-

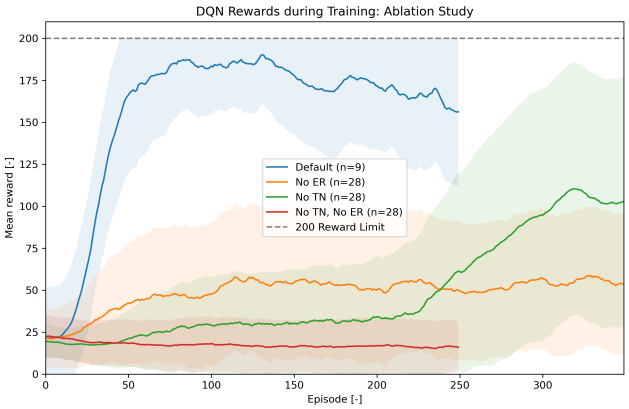

Figure 1. **Ablation Study:** Comparison between the average rewards over time, smoothed over a 21 episode window, for different configurations of the Deep Q Agent: removing the Experience Replay Buffer and/or Target Network. The shaded areas show the 1σ confidence intervals derived from n training runs. The DQN without ER perform very poorly and show no or very limited persistent learning in the episodes that were sampled. Instead, the agent learns (somewhat) over a few episodes but then suffers from overfitting and then returns to an underfitted state after some short episodes. The DQN without ER but with TN quickly reaches an equilibrium state where the TN compensates somewhat for the missing ER since it retains the previous main network. If ER is enabled but there is no TN, learning is initially very low, but slowly improves as the buffer is filled with transitions that are closer to on-policy. Nevertheless, without a TN the DQN cannot fully exploit its memory. Finally, the full DQN learns very fast and reaches (probably) equilibrium near the maximum reward. Not shown here is a full DQN trained on an environment with a maximum reward of 10 000 where it reaches its equilibrium around 300 - 600 after 100 episodes and is somewhat stable up to 250 episodes where the study was terminated.

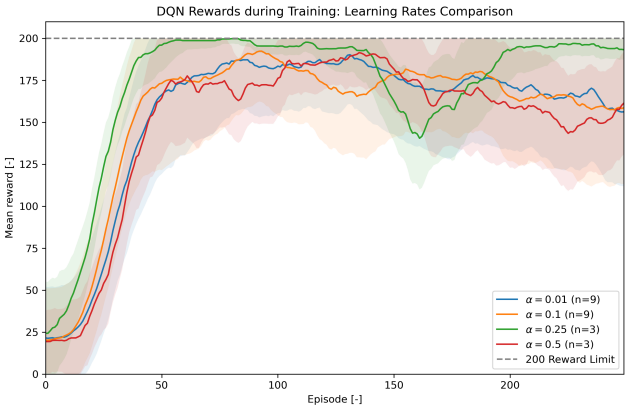

Figure 2. **Learning Rates:** Comparison between the average rewards over time, smoothed over a 21 episode window, for different learning rates. The shaded areas show the 1σ confidence intervals derived from n training runs.

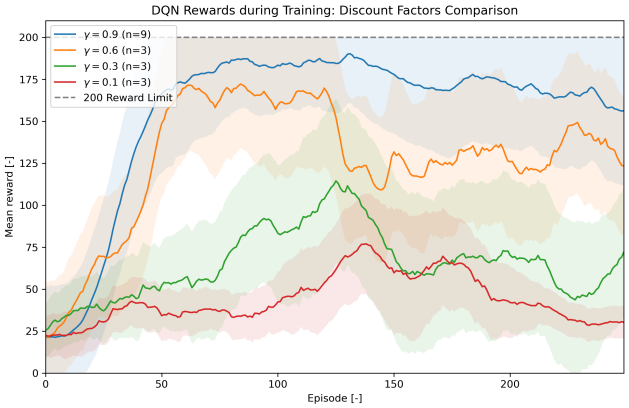

Figure 3. **Discount Factors:** Comparison between the average rewards over time, smoothed over a 21 episode window, for different discount factors. The shaded areas show the 1σ confidence intervals derived from n training runs.

imately 50 episodes, both methods approach the maximum possible reward, with softmax slightly favored. However, the difference is not very significant as the 1σ confidence intervals are quite wide due to the limited amount of runs. The large gap between both exponential decay strategies, and the ϵ-greedy linear anneal is very significant though. Clearly indicating the fact that, at least for the small space we investigated, the choice of annealing schedule is more important than the choice of exploration strategy.

2.4. Networks

2.4.1. DQN Architecture

The simplest adaptation we can make to the network architecture, is tuning both the amount of hidden layers, and the amount of nodes in each of these layers. As described in subsection 1.2, we opted for a relatively wide 3-layer network as the default. Here, we will look at the effect of narrowing and shallowing the network on the learning curve.

In Figure 5 we present the learning curves for the following layer configurations, where each number represents the amount of nodes in a hidden layer in sequential order: [64, 32], [128, 64], [256, 64], [256, 128, 64], [512, 256, 64].

In this figure, we see that at the end of a training run, all agents have either completed the environment, or are very close to winning. The difference between different agent in this timestep-limited environment mainly lies in the first $\sim$ 150 episodes. There, we see that both 3-layer agents learn very quickly and reach mean rewards of approximately 180, before plateauing. In the case of all three 2-layer networks, we see that they need 50-100 episodes more on average to reach the 200 reward limit. However, after 180 episodes, we see nearly no variation anymore in the mean reward of any of these three agents.

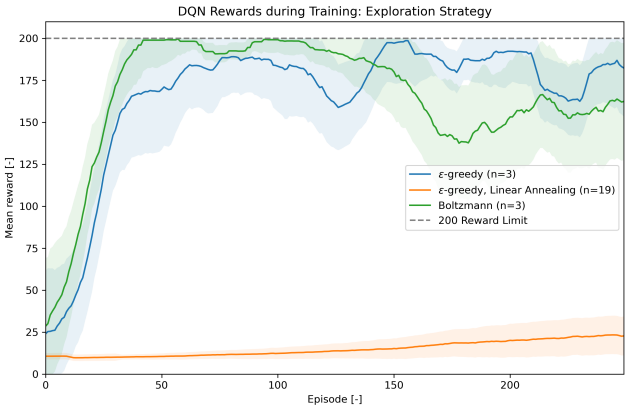

Figure 4. **Exploration:** Comparison between the average rewards over time, smoothed over a 21 episode window, between the two different exploration strategies introduced in subsubsection 1.2.2. The shaded areas show the 1σ confidence intervals derived from n training runs. Exponential decay performs much better than a linear anneal.

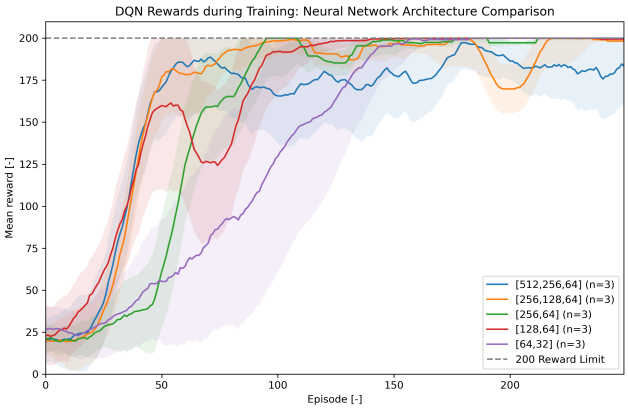

Figure 5. **Network Architecture:** Average reward over time, smoothed over a 21 episode window, for an agent with default settings from Table 1, except a variation over the number of hidden layers and nodes per layer. The shaded areas show the 1σ confidence intervals derived from n training runs. Exact architecture is given in the legend, where the numbers in square brackets represent the amount of nodes per layer in sequential order.

2.4.2. DQN Weight Updates

We observed three different kinds of weight updates for the DQN as shown in algorithm 3. A default option that directly sets the main network weights as the target network weights, an option that shifts the target weights by 10% towards the main network weights and an option that updates the target weights more aggressively the more successful the most recent episode was. The resulting study is shown in Figure 6.

2.5. Replay Buffer

Replay buffers without prioritization have two main parameters, buffer capacity and batch size. In Figure 7 we varied both parameters independently. We did not keep the replay ratio constant for these studies.

2.6. Environment

We performed one study on how the environment affects the learning. The physics of the environment are constant and typically not changeable but the maximum reward/-timestep/episode length can potentially be chosen arbitrarily. We suspected that a lot of learning is possible after the default 200 steps. Figure 8 shows the result of this study.

3. Discussion

3.1. Effect of Default Hyperparameters

It goes without saying that the choice of hyperparameter is extremely important for the performance of any machine learning algorithm. This is especially true for deep reinforcement learning problems, which in general tend to be very unstable (Dasagi et al. (2019), Mnih et al. (2015)). Additionally, the amount of hyperparameters present in any deep learning problem can quickly become quite large as can be seen from the presented list in Table 1. Considering the range of values each of these parameters can take, and the amount of time even a single, relatively short, training iteration takes on an average computer, one could imagine that a full hyperparameter grid-search is out of reach for a small group of students.

The approach taken in this report: deciding upon reasonable default values for all hyperparameters, and varying over a single one at a time, is a lot less computationally expensive, but can potentially introduce biases. However, the effects of a certain 'badly chosen' hyperparameter, if kept constant, would be similar independent of the variation in another hyperparameter. Thus, the main issue in our analysis would be potential instability introduced by bad parameters. This can mainly be eliminated by running the same setting for multiple full training sessions, as is done here. Therefore, the comparative analysis as done in this report makes sense.

A larger effect on the results comes from the imposed maximum reward limit. We observe many settings reach the 200 point ceiling, sometimes even within 50-100 episodes, such as in Figure 2 and Figure 5. In those cases, this limits the analysis to only the initial learning speed, and excludes any conclusions concerning differences in final performance.

The effect of the discount rate should be highlighted. As expected for balancing discounting future states leads to quickly deteriorating performance as any deviation from close-to-equilibrium states becomes more difficult to recover from and hence it is imperative to correct small errors

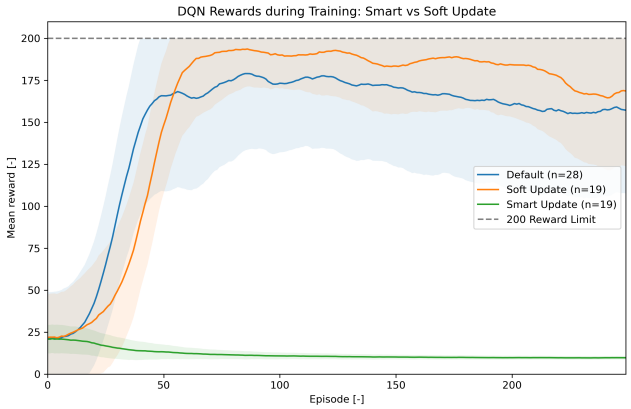

Figure 6. **Target Network Update:** While learning is slightly delayed, the soft target network update makes it possible to reach higher mean rewards and retain them for longer. The "Smart" target network update that we proposed, performs horrendously and has the worst performance overall.

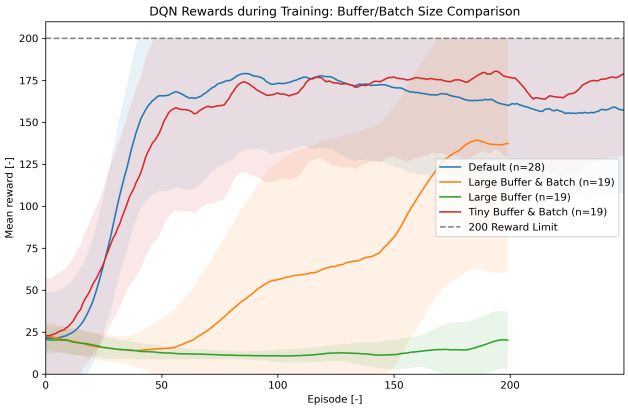

Figure 7. **Buffer settings:** The small buffer capacity performs comparable to the default buffer capacity, but can adapt quicker, is less stable but here still retains higher mean rewards for longer. The large buffer capacities hold mostly off-policy transitions and are hence extremely slow as most of the values in the sampled batch lag many episodes behind. Increasing the batch size allows the DQN to have more complete knowledge over a larger sub-state space and speeds up the learning compared to low replay ratios.

even if their effects only become apparent many steps later in this unstable system. **??** point out that n-step methods are a "key" to learn unstable and/or non-linear systems which also highlights the performance of a discount rate near 1.

3.1.1. DQN Weight Updates

subsubsection 2.4.2 shows an issue with naively approaching failure: The way in which the "smart" target network weight update is constructed leads to the the DQN only learning when the last run was successful compared to the maximum reward. This means in turn that the DQN does not learn how to avoid failure and gets continually overfitted to the tiny state-space in which it can excel. In combination this means that the DQN is trained to not recognize failure and hence consistently acts in a "reckless" way that can only pay-off in a very specific state space.[1] This result highlights the value of learning from failure.

3.2. Experience Replay

In Figure 1 we studied the effect of turning off two components used in the Deep Q-Learning algorithm: the Target Network (TN), and the Experience Replay Buffer (ER). From that figure we see that, at least in the relatively short training time we provided, turning off the experience replay has the biggest effect on the ability of the agent to learn. This highlights the importance of the buffer for real, long term improvement. In Figure 9, we show the learning curve of an agent without ER, combined with all the 28 unsmoothed individual runs that were merged to create the mean result as light-gray dashed lines. In those singular runs, a very clear oscillating pattern is visible. Throughout the entire training duration, we see agents having sometimes having a few very good runs, before completely dropping off and forgetting everything it learned.

This is because these agents are only able to learn from their previous step, instead of sampling from a wide range of previous experiences as is the case for the agents with ER. This inability to remember previous experiences causes the network to become very unstable, overfitting the network on small regions of the state space until randomness, either through exploration or the initialization, introduces a situation where the agent has no idea how to act. As outlined in subsection 1.2, this is the expected behaviour.

Interestingly enough, we see the performance of the agent increase for the first ~ 50 epochs, which corresponds roughly to the first time that the sudden peaks in performance hit the 200 reward limit. This might indicate that an agent without experience replay could perform slightly better than shown here if the ceiling was increased or removed, because it would progressively be able to propagate higher rewards back through its nodes. However we were not able to do the experiment to analyze this phenomenon.

The buffer study in subsection 2.5 largely agrees with the results of (Fedus et al., 2020) and reinforce the interwoven nature of ER with the DQN. The most interesting result is the success of the DQN with a small buffer. Smaller buffer capacities adapt quicker since they hold mostly on-policy transitions and the agent can hence maintain a high mean reward for longer than a more static agent, even though this might be an artifact of the maximum episode number we

[1] Super cool to see that RL agents also need to "fail fast" and move on.

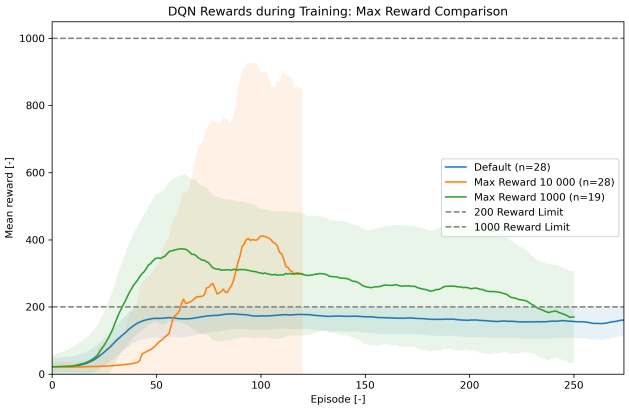

Figure 8. **Maximum episode length:** DQN which are not limited to 200 steps can reach higher maximum rewards. The maximum reward of our DQN approach seems to behave asymptotically, where at some point increasing the possible length of an episode does not lead to an agent that can continue to balance the cartpole. Furthermore, the different DQN seem to approach a equilibrium near a mean reward of 200 no matter what their maximum reward was.

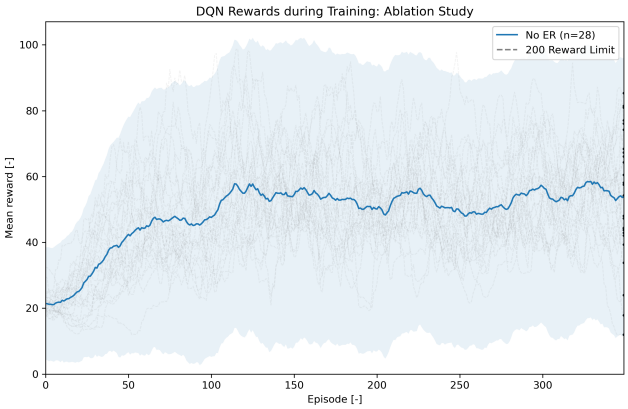

Figure 9. Averaged and smoothed learning curve of the network without experience replay from Figure 1, combined with all 28 individual unsmoothed runs. This plot shows rapid oscillation between an over- and underfitted approximant.

imposed here. The agents with high buffer capacity and low replay ratio perform very poorly as the buffer will contain almost exclusively off-policy samples until the agent has remained in a sub-state space long enough to replace most old transitions. It can then slowly improve and is very stable.

3.3. Further Research

As mentioned in previous sections, a lot of our analysis is limited simply by the amount of computing time available to us. Therefore, the simplest next steps, listed in order of increasing computational cost, would be first up increasing the amount of episodes an agent is allowed to train would allow us to know if the relations between settings we see now remain constant, or change over time. Secondly, increasing the amount of times we repeat the learning process for a single setting can decrease our confidence intervals, increasing the potential power of our analysis.

One novel method for which the time in this assignment was not sufficient is Fourier Transform Prioritized ER (FTPER). The existing methods for ER like proportional and rank-based intend to more efficiently replay and learn by prioritizing high temporal difference error transitions. However, this only accounts for single transitions. The agent however, does not act over a discontinuous state space since state histories are highly correlated through an episode. Hence, we can view the buffer as a time series vector/tensor and, to extract features larger than a single transition apply a Fourier transform to it. This can be used on its own or combined with other PER and lets us prioritize based on the frequency of ”transition tracks”. This would let us make even more efficient use of the replay buffer. The only study of ER similar to what we propose here is by (Li & Pathak, 2021). Based on their initial study of frequency-based functional regularization, learned Fourier features and by extension FTPER, seem like a low-cost high-return prioritization method which should be further investigated. Previous work on principal component analysis for PER could be used as an initial research direction (see e.g. Fyfe & Lai (2007); Monforte & Ficuciello (2020)). It is worth highlighting that we expect FTPER to become relatively more efficient for larger $\mathcal{S} \otimes \mathcal{A}$-spaces.

3.4. Conclusion

We find that both ER and TN are essential for DQN. The variance in performance from changing hyperparameters (in reasonable ranges) is largely overshadowed by ablating parts of the DQN. All but very simple networks perform well. The discount rate impacts especially unstable systems and needs to be close to unity for the DQN to learn and act on an unstable system. No matter what the maximum episode length is, our agents tend to a mean reward of 200.

References

Barto, A. G., Sutton, R. S., and Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. doi: 10.1109/TSMC.1983.6313077.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

Dasagi, V., Bruce, J., Peynot, T., and Leitner, J. Ctrl-z: Recovering from instability in reinforcement learning, 2019. URL https://arxiv.org/abs/1910.03732.

Fedus, W., Ramachandran, P., Agarwal, R., Bengio, Y., Larochelle, H., Rowland, M., and Dabney, W. Revisiting fundamentals of experience replay. In *International Conference on Machine Learning*, pp. 3061–3071. PMLR, 2020.

Fyfe, C. and Lai, P. L. Reinforcement learning reward functions for unsupervised learning. In *International Symposium on Neural Networks*, pp. 397–402. Springer, 2007.

Li, A. and Pathak, D. Functional regularization for reinforcement learning via learned fourier features. *Advances in Neural Information Processing Systems*, 34, 2021.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

Monforte, M. and Ficuciello, F. A reinforcement learning method using multifunctional principal component analysis for human-like grasping. *IEEE Transactions on Cognitive and Developmental Systems*, 13(1):132–140, 2020.

A. Algorithm Blocks

Algorithm 1 DQN Learning

Input: DQN agent, environment & learning settings
Result: Optimal policy or Q-value function approximation after N_{epochs}
Initialization:
DQN Agent with:
  circular replay Buffer $\mathcal{D}$:
  $\{\{s \mid s_T\}, \{a \mid a_T\}, \{r \mid r_T\}, \{s' \mid s'_T\}, \{d \mid d_T\}\}$
  with:
    randomly sampled batches $\delta \sim \mathcal{D}, \quad \delta \subseteq \mathcal{D}$,
  Network(s) $\mathcal{X}$: {
    online network $\mathcal{X}_O$ with θ,
    target network $\mathcal{X}_T$ with θ' (optional)},
  Policy π: $a \sim \pi(\hat{Q})$,
  Incentive function $\mathcal{I}$: $i \leftarrow \mathcal{I}(s', T; r)$,
  Anneal policy ξ: $x_\pi \leftarrow \xi(r, E, T; x_\pi)$,
  Weight update policy ω: $\theta \leftarrow \omega(\tau; \theta)$,
  Experience replay ϕ: $\theta_\mathcal{X} \leftarrow \phi(\delta, \mathcal{X})$,
Cartpole Environment $\mathcal{E}$
for *E in* $1 + N_{epochs}$ **do** /* fill buffer, train on N_{epochs} */
  $\mathcal{E} \leftarrow \mathcal{E}(s_0), \quad s_0 \in \mathcal{S}_0$; /* reset environment */
  $s \sim \mathcal{E}$; /* sample environment */
  $T \leftarrow 0$; /* initialize timestep */
  repeat /* act inside environment */
    $\hat{Q} \sim \mathcal{X}(s, \mathcal{A}_s), \quad \mathcal{A}_s \subseteq \mathcal{A}$; /* predict Q */
    $a \sim \pi(a|\hat{Q})$; /* select action under policy */
    $r, s' \sim \mathcal{E}(r, s' \mid s, a)$; /* take action */
    $r \leftarrow \mathcal{I}(s', T; r)$; /* adjust reward */
    $\mathcal{D}_i \leftarrow s, a, r, s'$; /* fill buffer */
    $s \leftarrow s'$; /* progress agent */
    $T \leftarrow T + 1$; /* next timestep */
    if $\mathcal{D}$ *is full* **then** /* learn */
      if $\mathcal{X}_T$ *exists &* s *is final* **then**
        $\theta' \leftarrow \omega(\tau; \theta)$; /* update target network */
      $x_\pi \leftarrow \xi(r, E, T; x_\pi)$; /* anneal policy */
      $\theta \leftarrow \phi(\delta, \mathcal{X})$; /* update and/or replay */
    else if s *is final* **then** // continue filling buffer
      $\mathcal{E} \leftarrow \mathcal{E}(s_0), \quad s_0 \in \mathcal{S}_0$; /* reset environment */
      $s \sim \mathcal{E}$; /* sample environment */
      $T \leftarrow 0$; /* reset timestep */
  until s *is final*; /* continue to next episode */
end

Algorithm 2 Experience Replay: $\phi(\delta, \mathcal{X})$

Input: Deep Q-learning network(s), buffer
Result: Updated Deep Q-learning network
Initialization: Filled, circular replay Buffer $\mathcal{D}$:
$\{\{s \mid s_T\}, \{a \mid a_T\}, \{r \mid r_T\}, \{s' \mid s'_T\}, \{d \mid d_T\}\}$
with:
randomly sampled batches $\delta \sim \mathcal{D}, \quad \delta \subseteq \mathcal{D}$,
Network(s) $\mathcal{X}$: {
online network $\mathcal{X}_O$ with θ,
target network $\mathcal{X}_T$ with θ' (optional)}

$\delta \sim \mathcal{D}$; /* sample batch from buffer */
$\{s, a, r, s'\}_{m_\mathcal{D}} \sim \delta$; /* take S.A.R.S from batch */
$d \leftarrow \begin{cases} 1 & \text{if } s' \text{ is final} \\ 0 & \text{else} \end{cases}$; /* terminal transitions */
$\hat{Q}_\theta(a|s)_{m_\mathcal{D}} \sim \mathcal{X}_O(s_{m_\mathcal{D}})$; /* batch predict Q */
for $\{s, a, r, s'\}$ *in* $\{s, a, r, s'\}_{m_\mathcal{D}}$ **do** /* iterate over batch */
  if $\mathcal{X}_T$ *exists* **then** /* if DDQN use TN */
    $\hat{Q}_\theta(s', a') \sim \mathcal{X}_O(s')$; /* predict Q(s') using ON */
    $\hat{Q}_{\theta'}(s', a') \sim \mathcal{X}_T(s')$; /* predict Q(s') using TN */
    $a'_\theta \leftarrow \arg\max_a \left[\hat{Q}_\theta(s', a')\right]$; /* select greedy a' using ON */
    $\hat{Q}^\star_\theta(a|s) \leftarrow r + (1 - d)\gamma \max\left[\hat{Q}_{\theta'}(s', a'_\theta)\right]$; /* Bellman optimality */
  else /* if single network */
    $\hat{Q}_\theta(s', a') \sim \mathcal{X}_O(s')$; /* predict Q(s') using ON */
    $\hat{Q}^\star_\theta(a|s) \leftarrow r + (1-d)\gamma \max\left[\hat{Q}(s', a')\right]$; /* Bellman optimality */
  end
end
$L(\theta) \leftarrow \frac{1}{m_\mathcal{D}} \sum_{i=1}^{m_\mathcal{D}} \left(r_i + \gamma \max_{a'} \hat{Q}_{\theta'}(s'_i, a') - \hat{Q}^\star_\theta(s_i, a_i)\right)^2$; /* find loss */
// Use RMSprop optimizer to update network (Hinton+12)
// This makes proper use of the batch
// see Riedmiller+93, Hinton+12
$\theta \leftarrow \texttt{RMSprop}\left[L(\theta)\right]$; /* ON update using RMSprop */

Algorithm 3 Target Network Update: $\omega(\tau; \theta)$

Input: Target Network, Online Network, episode reward r,
maximum possible reward r_{max}
Result: Updated Target Network
Initialization: Network(s) $\mathcal{X}$: {
online network $\mathcal{X}_O$ with θ,
target network $\mathcal{X}_T$ with θ'},
min, max update softness $\mathcal{T}_{min}, \mathcal{T}_{max} \in \mathbb{R}_{\in[0,1]}$,
softness parameter $\mathcal{T} \in \mathbb{R}_{\in[\mathcal{T}_{min}, \mathcal{T}_{max}]}$

if *smart update* **then** /* custom smart TN updates */
  $\theta'' \leftarrow \theta'$; /* init placeholder weights */
  if *soft update* **then** /* constant soft update */
    $\tau \leftarrow \mathcal{T}$; /* init softness parameter */
    for θ_i, θ'_i *in* θ, θ' **do** /* iterate over weights */
      $\theta''_i \leftarrow (1 - \tau)\theta'_i + \tau\theta_i$; /* proportional update using tau */
    end
    $\theta' \leftarrow \theta''$; /* update TN weights */
  else /* reward based soft update */
    // set hardness linearly depending on current DQN performance
    $\tau \leftarrow \mathcal{T}_{max} + (\mathcal{T}_{max} - \mathcal{T}_{min}) \cdot (r_{max} - r)/r_{max}$;
    /* proportional update using tau */
    for θ_i, θ'_i *in* θ, θ' **do** /* iterate over weights */
      $\theta''_i \leftarrow (1 - \tau)\theta'_i + \tau\theta_i$; /* proportional update using tau */
    end
    $\theta' \leftarrow \theta''$; /* update TN weights */
  end
else /* hard update */
  $\theta' \leftarrow \theta$; /* hard TN update */
end