Deep Q-Learning on the CartPole Environment: Assignment 2 for Reinforcement Learning 2022

Kutay Nazli [1] **Filip Jatelnicki** [2] **Ivan Horokhovskyi** [2] **God of Probability** [3]

Abstract

As the systems in Reinforcement Learning become high-dimensional, continuous and have exponential branching factor, the methodology moves away from the tabular approach into implementing deep learning methods to reach solutions. We consider the simple *CartPole-v1* environment in *Gym* and implement a Deep Q-Network (DQN) to perform Q-value iteration to solve the environment, along with an automated parallelizable framework for solving reinforcement learning problems with HPO and subsequent analysis. We present the methodology for the network architecture, the hyperparameters, and optimization strategies, and perform and discuss the results of an ablation study comparing the performance impact of various features of the DQN such as exploration strategies, experience replay and a target network.

1. Introduction

1.1. Deep Reinforcement Learning

The field of Reinforcement Learning (RL) describes the use of the active interactions of an agent with its environment to learn the optimal behavior or achieve a specific goal in that environment. RL problems in general can be formulated as an MDP, a tuple $< S, A, T, R, \gamma >$ where S is the set of states s in the environment, A the set of allowed actions a, $T : S \times A \times S' \to [0, 1]$ is the transition function, $R : S \times A \times S' \to \mathbb{R}$ is the reward function and γ is the discount factor. We seek an optimal policy for the agent to follow, defined as $\pi^* : S \to A$ that maximizes the expected discounted sum of the future rewards $Q(s, a) = \mathbb{E}_\pi[\sum_i \gamma^i r_{t+i+1} | s_t = s, s_t = a]$.

In most simple problems, it is possible to store the Q-values of all the state-action pairs. These methods are therefore called *tabular* methods. However, in problems with continuous or high-dimensional S or A, it is highly computationally expensive or outright impossible to store such an amount of Q-values in memory as the search space branches exponentially with successive actions. The idea of Deep RL (DRL) then comes from utilizing neural networks not to explicitly calculate the Q-values of states, but to do *function approximation* to come up with an approximate Q-value function to return the values of a given state-action pair when needed. The main challenge of such an approximation comes from the limited amount of state-action pairs visited during training of the net, and thus having to generalize the output of the approximate Q-value function to pairs that are yet to be visited.

1.2. CartPole Environment

Gym is a suite of simple physics-based environments with a collection of games from legacy consoles such as Atari developed and distributed by the OpenAI that provides a great test-bed for developing DRL algorithms. The *CartPole-v1* environment consists of a cart that freely moves on a single horizontal axis on a frictionless surface, and a rigid pole attached to the cart by an un-actuated joint. The goal of the agent is to apply force to the car to then attempt to keep the pole upright as long as possible without the pole tipping more than $15°$ from the vertical or the cart getting further than 2.4 units from the center.[1] Every timestep where the pole is kept upright returns a reward of 1, and the environment terminates after 500 successful steps. The solution is considered to be achieved if the average of past 100 runs is above an episode reward of 475. The state space S for this problem consists of the tuple $< x, \dot{x}, \theta, \dot{\theta} >$, the horizontal position and velocity of the cart and the angular position and velocity of the pole respectively. Since all of these four parameters are continuous spaces, the state space for this problem is infinite, and therefore it is highly inefficient to practice tabular methods and makes it interesting for a DRL application.

[1]Leiden Observatory, Leiden University, Leiden, The Netherlands [2]LIACS, Leiden University, Leiden, The Netherlands [3]Random State University, MC, Monte Carlo. Correspondence to: <>.

[1]Definition as provided by OpenAI: https://gym.openai.com/envs/CartPole-v1/

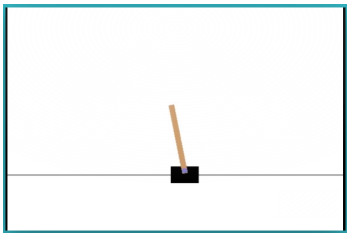

Figure 1. The *CartPole-v1* environment used in this report. Note the aforementioned horizontal track, the cart and the attached pole able to swing from side to side. Source: *Gym*, OpenAI

1.3. System settings

For the reason of experiments reproducibility and fairness, all experiments were made on a single computer. However, the program was tested that it works on other operating systems like Linux x64 and macOS. Setup (environment) is provided:

- Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
- Windows 10 (x64)
- Python 3.9.7
- Conda 4.11.0 (Torch 1.11.0, Numpy 1.22.3, Gym 0.23.1, Matplotlib 3.5.1, WandB (optional))

2. Methodology

2.1. Deep Q-Network

This work uses the PyTorch framework to implement the neural net (NN) used to predict Q-values ($q \in [0; 500] \subset \mathbb{R}$) for a given input, which defines a regression problem. The input and output dimensions are 4 and 2 respectively to match the aforementioned state space S and action space A of the environment, and posits a simple problem in terms of complexity. Since the input of the NN are floats of positions and velocities and not images, graphs or sequences, it was decided not to use convolutional layers or recurrent NNs and to use a fully-connected NN, with single hidden layer with 64 nodes, ReLU activation function, MSE loss-function, Adam optimizer and its parameters adopted as tune-able hyperparameters. Regarding network regularization, dropout for fully connected layers was tested, but not adopted as not many neurons and layers are used. Hence, in this case dropout would not have led to any performance improvement but could nevertheless be considered as a hyperparameter in further studies. Batch-normalization was also considered, but given the simplicity of the *CartPole* environment and its solution, it would only slow the network down and was therefore left out.

2.1.1. Exploration

Although the DQN methodology is able to come up with approximations for the Q-values for state-action pairs, the generalization of the solutions is a challenge that needs to be overcome. One of the possible solutions is to adequately diversify the training space for the model to ensure that the network does not overfit to the training states but provides equal performance on states never encountered. To this end, it is valuable to go off-policy and explore the state space rather than exploit or learn on policy all the time.

Methods such as ϵ-*greedy* and *Boltzmann temperature* distributions allow for probabilistic tune-able exploration, but the new states visited are picked at random and therefore might not be informative. Methods such as density or novelty distributions aim to overcome this problem by making sure that exploration of a state is more likely or more rewarding if the state is "interesting" or "novel". While this allows for more intelligent exploration, the mathematical implementations of novelty or density distributions become non-analytical or unobtainable in high-dimensional or continuous spaces in DRL. This requires the use of a separate NN or Markov tree search methods to come up with approximations. Even though these methods have been proven to improve the efficiency and performance of DQNs, in a simpler problem such as the *CartPole*, attempting to implement a more intelligent exploration method would be creating a more complicated problem to solve a simpler one.

Hence this work implements *annealed* ϵ-*greedy* exploration with linear function annealing, which allows for the NN to explore the space initially, and then to exploit and converge on a solution. The implementation and results will be further discussed in Section 3, while the optimization of this decay is discussed in 2.2.

2.1.2. Experience Replay

Since state-action samples in the *CartPole* environment are generated sequentially, they are highly correlated within training episodes. Training from these samples would inevitably lead to high variance. Another issue with training fully on-policy is that the samples used to update the weights of the network are generated using those very same weights. This could cause very high bias towards sub-optimal solutions or cause the network to diverge away from the optimal solution. To get away from this, once a step in the MDP is generated, an experience tuple $< s_t, a_t, r_t, s_{t+1} >$ is saved in the *experience memory*. The NN is then instead trained not on the sequential episodes generated from the environment, but on randomly sampled batches of experiences from the experience memory. This breaks the aforementioned variance from the sequentiality and the bias that would be incurred from updating the net solely with state-

action pairs generated from its current weights. This work implements an experience memory with a finite maximum size that stores only the most recent experiences and samples batches uniformly from them in a size that is tuned to provide optimal results. Attaching a form of novelty or density distribution as discussed in the previous section to these experiences and how they are sampled could provide for more intelligent use of the experiences for training, but that is beyond the scope of this work because of the reasons also elaborated on the previous section.

2.1.3. Target Network

The emergence of variance in RL NNs have already been aluded to in the previous subsection. Another related possible reason for variance in DQNs is the codependency of the Q-values returned by the NN and the target Q-values used to update the network. In a general target update formulation such as $(r_t + \max_a Q(s_{t+1}, a : \theta) - Q(s_t, a_t : \theta))^2$ it is easy to notice that both the current Q-values and the update values depend on the same weights of the network, θ. This causes the targets to shift with every update of the NN weights, and will lead to highly variant solutions or even total divergence in worst case scenarios. To combat this, every some number of episodes in the environment (or equivalently, updates of the NN), the weights of the now named *policy network*, are frozen and copied over to a network with the same exact architecture, the *target network*. Essentially, the target network serves as a slow-updating copy of the policy network to keep the target updates from fluctuating rapidly and causing unwanted variance. This in turn transforms the main MDP into smaller supervised learning problems that the policy network can tackle one at a time to ultimately reach a stable solution. Having said all this, the number of steps the policy NN needs to update before the weights will be copied over to the target network is called the *target fraction* in this work and is a hyperparameter that is tuned to optimize performance.

2.1.4. Training and Evaluation

Considering the architecture and features discussed in this section, the DQN follows Algorithm 1 to train.

As an addendum to Algorithm 1, the initial network training algorithm was written to update at the end of every action step in the environment. When the NN was trained this way instead, the number of total episodes required to train the net were less, however the overall performance of the agent was noticeably less and did not solve the environment. Instead updating at the end of every episode allows the experience memory to be meaningfully different from the past update and reduces the probability of resampling the same experiences, especially in the earlier training epochs.

Algorithm 1 DQN with Experience Replay and Target NN

```
Initialize Policy and Target NNs with random weights.
Initialize Memory M with max size N.
for episode in budget do
    s ~ p_0(s)                          ▷ Reset the environment.
    while True do
        a ~ π(a|s)                      ▷ Sample action from policy.
        r, s_{t+1} ~ T(r, s_{t+1}|s, a)
        Add < s_t, a_t, r_t, s_{t+1} > to M.
        s ← s_{t+1}
        if s_{t+1} is terminal then
            if episode/target fraction= 0 then
                Update the Target NN.
            end if
            Sample batch of experiences from M.
            Update the Policy NN with the batch.
            Update the policy π.
            break
        end if
    end while
end for
```

During initial testing and training, it was also realized that training the policy NN too much (as in continuing to train after the agent has returned cumulative reward values above the solution threshold for the environment (discussed in Section 1.2)) causes the NN to deteriorate in performance and deviate from the solution to restart the learning process from scratch. Even though experience replay and the target network are implemented to dampen this behaviour somewhat, it was also found that concluding training once the policy NN has been sufficiently trained does provide promising results. To this end, this work implements a criterion for stopping training, an average score of 475 over the last 5 episodes causes training to stop and the weights of the policy NN are saved and validated in the environment to test the generalizability of the solution. This is done through a validation process that initializes an agent with the saved weights, performs test runs in the environment with random starting conditions and its performance over 1000 episodes is then logged and analysed to see if it truly solves the environment. The results of this method will be discussed in the Ablation Study in Section 3.

2.2. Hyperparameters and Optimization

Given the nature of NNs, hyperparameter optimization (HPO) is considered to have substantial impact on the performance of machine learning deep learning algorithms. Although it is vital for optimal performance, the shortcoming of HPO is that with increasingly complex problems, the search space for hyperparameters also increases in size and can include continuous parameters or be highly non-

Hyperparameter	Range	Distribution
learning rate	$[10^{-3}; 10^{-5}]$	log
batch size	[500; 2000]	uniform
ϵ-decay	[0.995; 0.9999]	log
target fraction	[10; 155]	uniform

Table 1. Hyperparameters selected for optimization and their range distributions. Learning rate refers to the step size in the loss-function update for the NN, the batch size is as discussed in Section 2.1.2, epsilon decay as discussed in 2.1.1 and the target fraction as in 2.1.3.

linear. Given the limited time and computational resources available, it may not be possible to fully explore this hyperparameter search space. Hence, the multi-armed bandit rears its head again, but this can be tackled with various space-search methods or meta-learning.

HPO becomes even more crucial in RL, due to the additional hyperparameters present for the RL agent, like for instance policy or agent training loop details on top of the model hyperparameters present in general machine or deep learning. Even in a simple environment such as the *CartPole*, more than 20 hyperparameters were used. And since it was not possible to optimize all of them properly and do so with statistical significance, it was decided to optimize only 4 most important of them. The selected parameters, their ranges, distribution, and short description are present in Table 1.

Since HPO is not the main scope of this work, it does not implement the most recent state-of-the-art methods due to their inherent complications in implementation or integration. Methods of grid search, random search and Bayesian optimization were therefore considered for the HPO. Grid search was eliminated because searches the space ineffectively looking for similar parameters. Bayesian optimization cannot be parallelized efficiently and has its hyperparameters (such as the acquisition function and kernel of the Gaussian processor) which need to be selected separately. Random search is a simple but very efficient method and was picked for this problem.

Through this random search process, 59 different hyperparameter sets were chosen. To make experiments statistically significant, but at the same time to try an adequate amount of hyperparameter sets with a limited budget, each set was run for 5 repetitions. For further experimentation, it was decided to distil a smaller set of the best performers. Out of performed experiments, only 8 of the set were able to solve the environment 3 out of 5 times, while the other 2 times the respective models trained (always having average score above 300) but did not solve the environment. Even though the training was stopped for these sets, if the training was continued after failed validation - they could still converge within the given budget. Allowing training to continue after failed validation was not implemented in this work, but can be considered as a future upgrade. Nevertheless, the settings that got 3 solves are presented in Table 2.

#	learning rate	batch size	ϵ-decay	target fract.	Valid. avg. reward	Solved out of 15 runs
1	0.05	1750	0.9996	55	437.46	5
2	0.05	1500	0.9994	155	426.48	6
3	0.05	1000	0.999	10	392.66	4
4	0.0005	1750	0.9996	10	444.55	5
5	0.0005	2000	0.9996	10	448.65	6
6	0.0005	2000	0.999	55	438.84	7
7	0.0001	1500	0.9996	155	**449.16**	6
8	0.0001	2000	0.9994	155	427.51	5

Table 2. The hyperparameter sets that validated the environment thrice in 5 training runs. Further experimentation and results are based on these sets of parameters. Table also includes average reward from 15 repetitions of validation phase.

In an effort to find reliable and stable solutions, further experimentation was conducted. In this experiment, 10 repetitions for each setting from Table 2 were performed. The learning stage performance for these is presented in Figure 2. With this experiment, one best solution has been chosen by comparing the mean reward values achieved during ten repetitions of validation phase. The final hyperparameters set contains: learning rate: 0.0001, batch size: 1500, ϵ-decay: 0.9996, target fraction: 155 and with this set network was able to achieve average reward of 449.16 during validation over the total 15 runs. Further experiments will use these settings.

3. Ablation Study

An ablation study was performed to better understand and quantify the performance impact of various features such as the experience replay and the target network as discussed in Section 2.

3.1. No Experience Replay, No Target Network

This version of the DQN possesses no memory to perform experience replays and no target network to stabilize the variance of the Q-value target updates. Instead, the weights are updated at the end of every episode using the $< s_t, a_t, r_t, s_{t+1} >$ tuples gathered during that very same episode. And since there is no target network, the policy

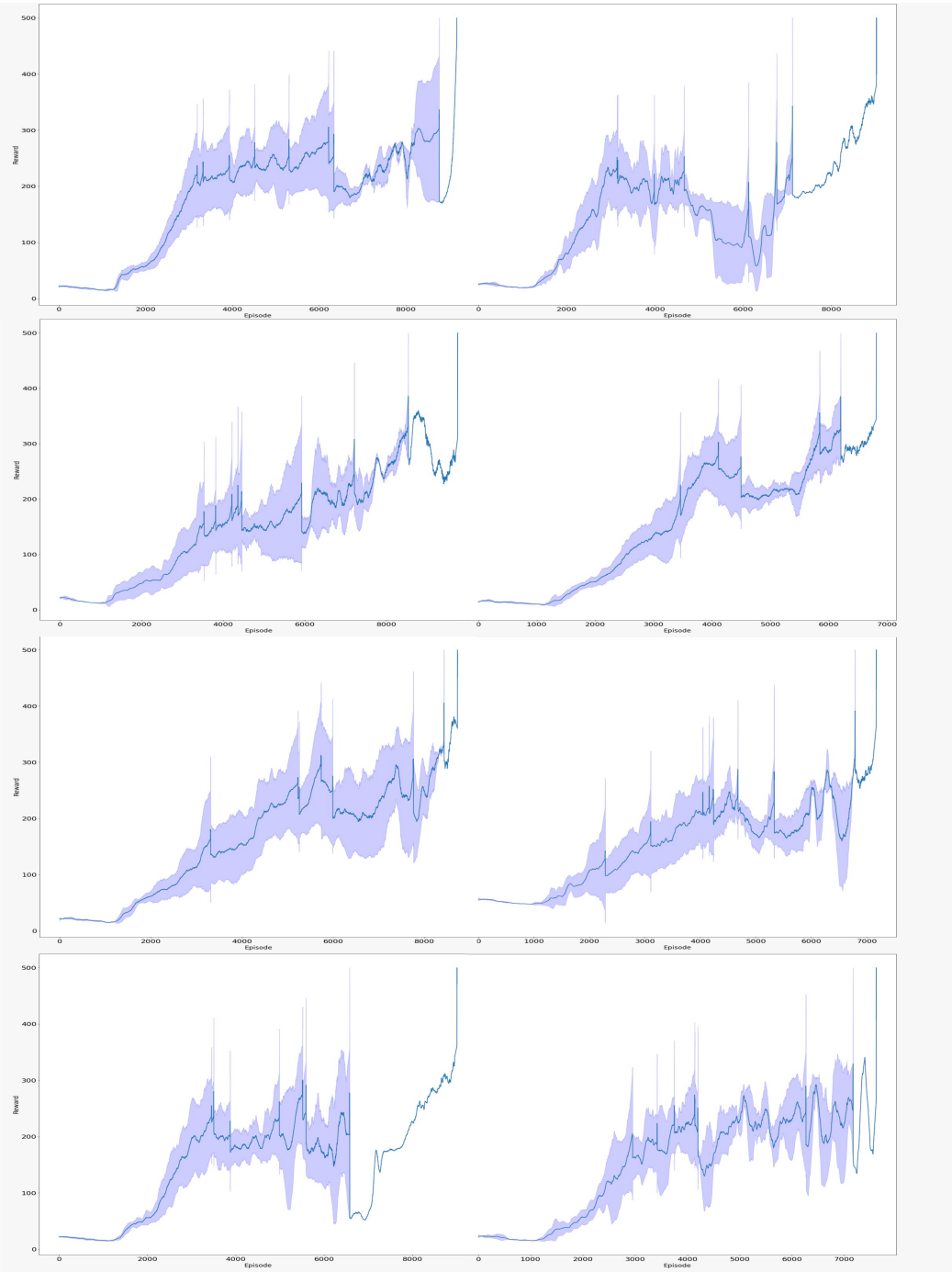

Figure 2. Figures represent reward values achieved during learning process of each hyperparameter set in Table 2, starting from set 1 in *top right* and going *right* then *down* for successive sets. Results are averaged over 15 total runs and smoothed with the Savitzky-Golay filter (window length = 100 and polynomial order = 1), with shaded regions representing the 1-σ error intervals. Realize that for some subplots the intervals disappear while the mean curves continue. This is due to all runs ending before this time during training and only a single run continuing. The sharp peaks of the σ shades comes from runs learning quickly at those points and increasing in rewards and achieving the training goal. The sharp peaks at the ends of the mean curves come from re-concatenating the mean values from the runs that where smoothed out due to the filter. Overall, in terms of success in achieving the training goal (the number of episodes it takes to achieve and the spread of runs), run number 7 was considered to be the best performer.

network is also used to calculate the target rewards for the net update. It can be noticed that the agent starts off having behavior akin to an agent that executes random actions and returns similar results, however slowly converges to an even poorer "solution", and the results can be seen in Figure 3. This is very likely to be due to the high bias incurred from updating on the same episodes generated by the same weights.

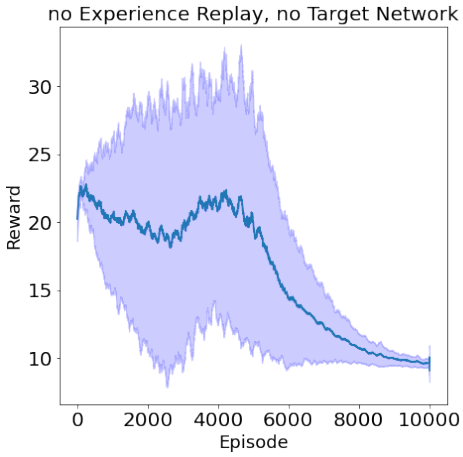

Figure 3. Smoothed rewards during training an agent in the absence of both the experience replay and the target network averaged over 10 trial runs with the 1-σ error regions. Realize that the overall performance of the bare DQN starts off as random and decays over time.

3.2. No Experience Replay

This version of the DQN does have the target network but does not have a memory for experience replay. This means that the updates are still calculated through a frozen copy network, but the updates are performed from the very same episode. There is less variance during learning due to target network stabilizing the process, but the overall performance of the net at the end of the allowed training budget is still around the same and worse than completely random actions. This is probably still due to the absence of experience replay, and the self-correlation present in the updates due to it. The performance is presented in Figure 4.

3.3. No Target Network

This final version of the DQN has an experience replay to sample mini-batches for the weight updates but the Q-value targets are still calculated with the same policy network. Of the three simpler architectures tested in this ablation study, this is the only one that shows signs of learning and provides results that can be compared to the full network with both

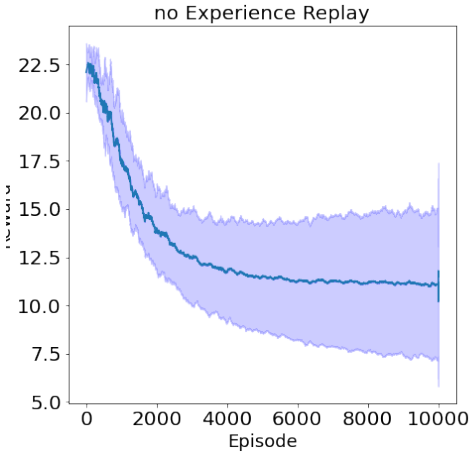

Figure 4. Smoothed rewards during training an agent in the absence of the experience replay averaged over 10 trial runs with the 1-σ error regions. Realize that the overall performance of the DQN starts off as random and decay rapidly to suboptimal solution.

features enabled. However still, when trained for 10 separate runs, while this architecture does provide good average rewards throughout (over 200), it never reached the training goal set for validation to be started and does therefore underperform compared to the full network with the same set of hyperparameters presented in Figure 2 subplot 7 which was able to train successfully multiple times and achieve solutions when validated.

4. Conclusions

4.1. Results

As the result of this project an automated parallelizable framework for solving reinforcement learning problems with HPO was built along with a validation script, automated results and artifact (model) logging, analytical notebook to provide various statistics and plots.

After the initial architecture for the network was decided on and implemented, most of the time and effort for this work went into the optimization of the hyperparameters. Even though this hyperspace is enormously large and has a complicated geometry, this work is able to isolate several sets of hyperparameters that when repeatedly tested and validated can provide NN weights that solve the environment. Although this process has never achieved 100% success rate, the training of the net is fast enough that a solution can and almost always be found within a fraction of 20 minutes reliably, and this result is remarkable by itself considering the inherently random nature of DRL and DQNs in general. How the consistency of this method can be improved is

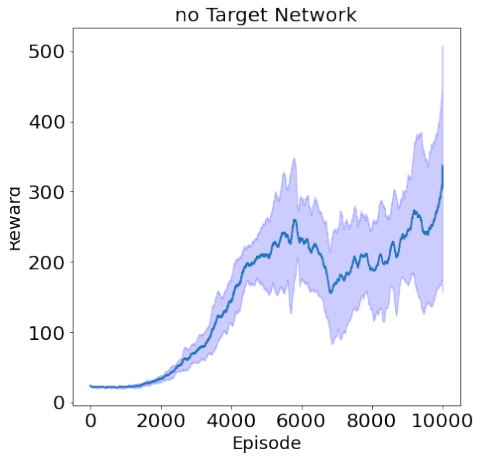

Figure 5. Smoothed rewards during training an agent in the absence of the target network averaged over 10 trial runs with the 1-σ error regions. While this net architecture does provide some result, it is still inferior to the full NN and does not make it to the validation phase.

discussed in Section 4.2.

When discussing the results of the ablation study, it is clear to see that the best performing architecture is the DQN with both experience replay and target network. As previously discussed in Section 2, both of these methods are implemented to overcome various inherent shortcomings of the DQN methodology, such as the high self-correlation of episodes and high variance of the target updates. Removing the experience replay means that the NN is only able to train on one past episode, and never learns to perform better than completely random actions therefore, and actually decays further due to biases incurred along the way. The semi-reliable performance of the experience-replay-only DQN also proves this point that utilization of experience replay is vital in DQNs. The absence of the target network definitely hinders performance and slows down learning, however the impact is not as noticeable as the absence of the experience replay.

4.2. Future work

During experiments and work, some decisions were made due to limited time budget. By choosing a more sophisticated hyperparameter optimization technique, the hyperparameter spaces could have been wider and more diverse. By choosing Random search, hyperparameter space had to be kept shallow enough to make statistically significant enough conclusions feasible. One note about the optimal solution presented in this work is that, for both the learning rate and the target fraction, the best performing values were at the extrema of their respective search spaces. This could be an indicator that prioritizing broadening these spaces in those directions could improve results reliably, and is probably the next natural step in improving the entire HPO process for this work.

Additionally, one of the biggest breakthroughs during experiments was to find some kind of stopping criteria for learning phase. At the beginning, carried out learning phases focused on maximizing collected rewards and solving environment during the learning phase itself. After realizing that a saved network that achieved couple very high rewards was able to solve the environment reliably, decision was made to try couple different but simple stopping criteria. Unfortunately, no significant improvement took place in this area. Future work could include some sophisticated mathematical or algorithmic solutions to ensure stability of learning results to further tighten the consistency of the training process. Another previously alluded addition could also be to allow sets that pass the training criteria but not the validation check to keep training for the rest of the available budget and see if they solve the environment that way. This could increase the amount of solves obtained by allowing the weights that get close (450 average rewards) to train the necessary smallest amount to possibly also achieve solution.

A. Appendix

A.1. Other hyperparameters

Hyperparameters considered in hyperparameter optimization were not the only ones that were used throughout experiments. Search space for HPO was intentionally limited to a set of the most impactful and significant. Among those which were not optimized, it is worth to mention settings of NN architecture and stop learning phase conditions. In experiments, different setting and shapes of NN were not tested. Network used in experiments could have different shapes and sizes, use different activation functions, normalization methods and optimizers, which could eventually lead to better performance. Second important criteria was the condition for the end of the learning phase. Aforementioned, a condition for the network to stop the learning phase was achieving five consecutive rewards, that averaged higher than 475. This condition enabled us to achieve reliable and repeatable results most of the time. Nevertheless, there is space for improvement, to ensure that the network would always be able to solve the environment after learning phase. Different conditions were tried, but no better and tighter condition was established.