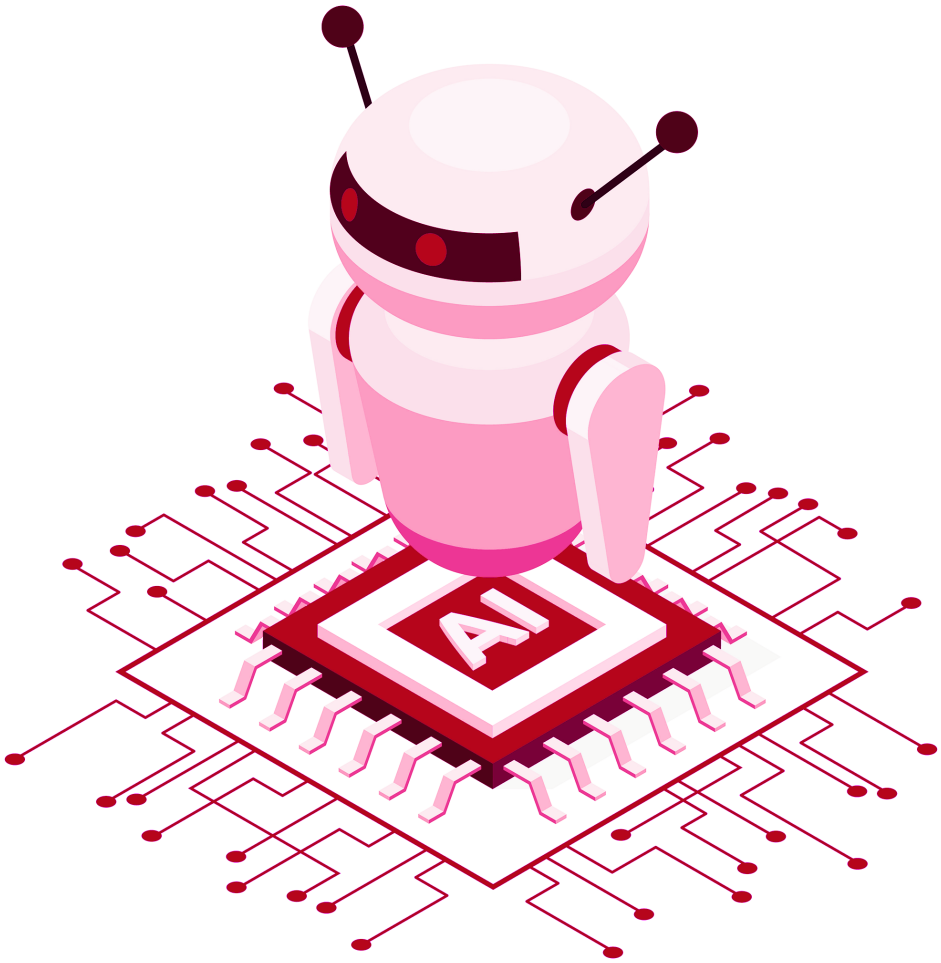

Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 2:

Value-Based Methods

By:

Danial Parnian

401110307

Spring 2025

Contents

Grading

The grading will be based on the following criteria, with a total of 100 points:

Task	Points
Task 1: Epsilon Greedy & N-step Sarsa/Q-learning	40
Jupyter Notebook	25
Analysis and Deduction	15
Task 2: DQN vs. DDQN	50
Jupyter Notebook	30
Analysis and Deduction	20
Clarity and Quality of Code	5
Clarity and Quality of Report	5
Bonus 1: Writing your report in Latex	10

Notes:

- Include well-commented code and relevant plots in your notebook.
- Clearly present all comparisons and analyses in your report.
- Ensure reproducibility by specifying all dependencies and configurations.

1 Epsilon Greedy

1.1 Epsilon 0.1 initially has a high regret rate but decreases quickly. Why is that? [2.5-points]

At first, the agent explores suboptimal actions and due to low epsilon, exploits them. But soon it finds more optimal actions and performs better. In figure 1, you can see this effect where cumulative regret decreases over time, and the slope decreases quickly.

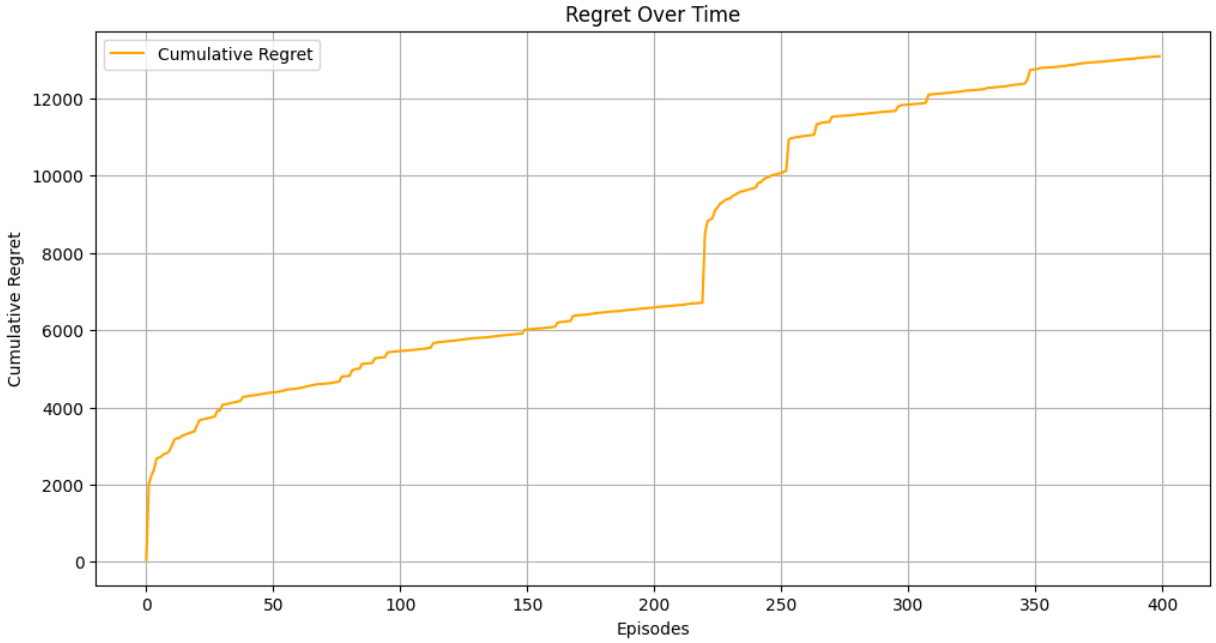

Figure 1: Cumulative regret over time for SARSA with $\epsilon = 0.1$

1.2 Both epsilon 0.1 and 0.5 show jumps. What is the reason for this? [2.5-points]

In the beginning, both tend to exploit a suboptimal policy. However, this effect is less for $\epsilon = 0.5$ since higher exploration prevents strong early exploitation and we see a smaller jump. Then they learn to act based on a better policy which leads to less regret (The second one still takes random actions 50% of the time, so this effect is less noticeable)

1.3 Epsilon 0.9 changes linearly. Why? [2.5-points]

In this case the agent selects almost random actions, and even learning a good policy can't reduce regret growth, as it barely affects action selection. Therefore, the slope remains nearly constant as you can see in figure 2.

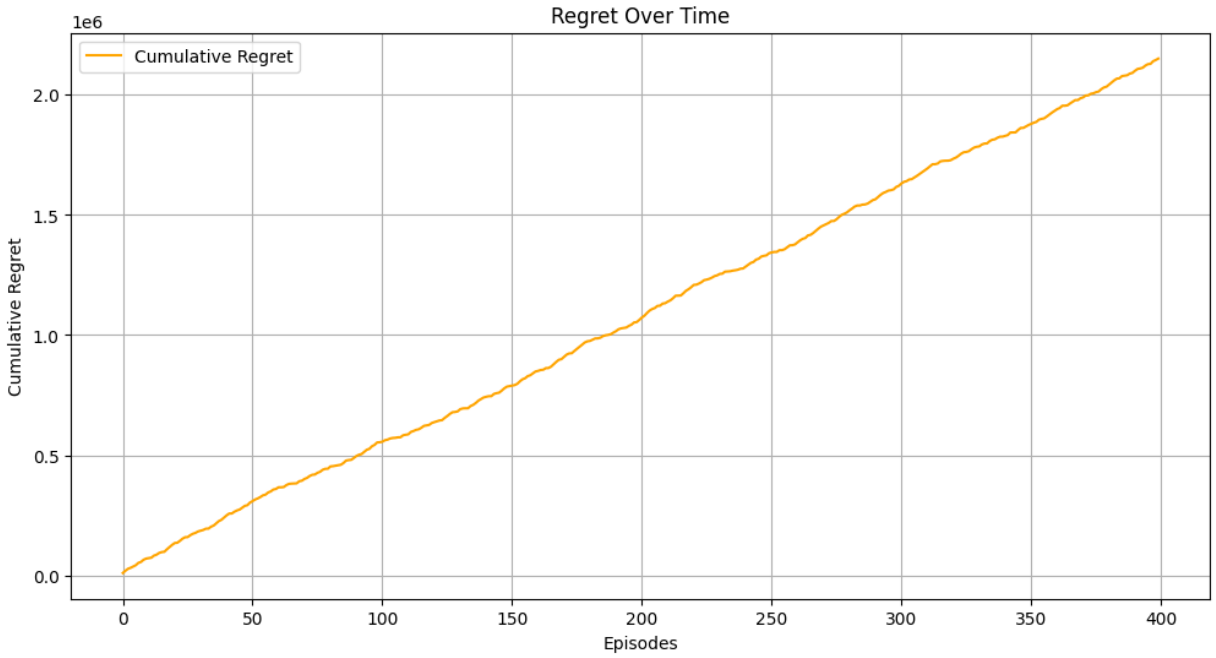

Figure 2: Cumulative regret over time for SARSA with $\epsilon = 0.9$

1.4 Compare the policy for epsilon values 0.1 and 0.9. How do they differ, and why do they look different? [2.5-points]

This is my expectation:
For 0.1 epsilon, the policy is more stable, as the agent mostly exploits the learned optimal actions with occasional exploration. This results in a structured path toward the goal. For 0.9, the policy appears less structured since the agent explores heavily and action selection is almost random. As a result, it doesn't find a clear optimal path and we see a less efficient policy.
This is what really happened in practice:
For 0.1 epsilon agent finds a path that leads to the goal and exploits it. for 0.9 epsilon due to more exploration, it sees all states equally and finds best actions for most of states. Overall, the policy of 0.9 epsilon is better than 0.1 and optimal. The reason is due to an implementation mistake in Notebook (which the TAs said not to change) where the epsilon value used to determine A' based on S' is 0.1, even when we set epsilon to 0.9, since we don't pass the epsilon as an argument to learning_rule function. So the algorithm is more like Q-Learning, rather than SARSA.

1.5 In the epsilon decay section, analyze the optimal policy for the row adjacent to the cliff (the lowest row). Then, compare the different learned policies and their corresponding rewards. [2.5-points]

In my implementation, the fast decay model failed to converge to an acceptable policy because epsilon decayed too quickly. As a result, even in the lowest row, the agent sometimes made suboptimal moves (moving left). However, increasing the value of alpha resolved this issue, leading to a policy similar to standard SARSA with 0.1 epsilon, where moves near the cliff are mostly upwards. Note that a higher alpha caused the medium and slow decay models to perform poorly, that's why I used a small alpha.

The medium and slow decay models performed similarly (after being trained for 1,000 episodes). In most states in the lowest row, the agent moved upwards, and both models achieved the goal in 17 moves. The results are highly dependent on hyperparameters, making it hard to provide a concise answer. You can see the final policies in Figures 3, 4, and 5.

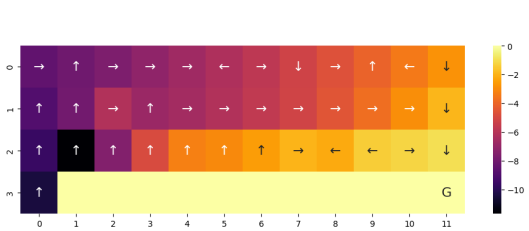

Figure 3: Final policy of SARSA with fast decay

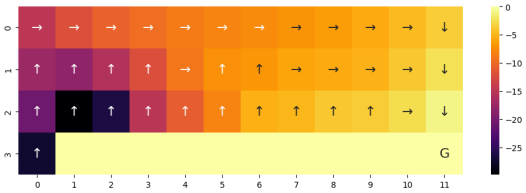

Figure 4: Final policy of SARSA with medium decay

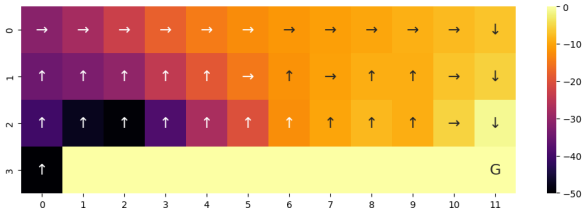

Figure 5: Final policy of SARSA with slow decay

2 N-step Sarsa and N-step Q-learning

2.1 What is the difference between Q-learning and sarsa? [2.5-points]

The main difference is in update rule for $Q(S, A)$. in SARSA we choose the next action based on the current policy. The update rule is $Q(S, A) := Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$ where $A' = \pi(S')$.

While in Q-Learning we choose the optimal next action and the update rule is $Q(S, A) := Q(S, A) + \alpha(R + \gamma \max_{A'} Q(S', A') - Q(S, A))$.

2.2 Compare how different values of n affect each algorithm's performance separately. [2.5-points]

Increasing n in SARSA leads to more stable training and faster convergence. Also the resulting policies achieve better rewards.

In contrast, for Q-Learning, increasing n leads to worse performance. Specifically, the total reward of the final policies decreases as n increases. In my implementation, the total reward was -13 for $n = 1$, -15 for $n = 3$, and -17 for $n = 5$. This happens due to Q-Learning's off-policy nature and overestimation errors when n is large.

2.3 Is a Higher or Lower n Always Better? Explain the advantages and disadvantages of both low and high n values. [2.5-points]

No, We should adjust n for the best result. A low n leads to faster updates and lower variance, but it might take longer to converge. A higher n leads to faster convergence (for on-policy methods), but it introduces higher variance, overestimation in off-policy methods, and increased computational cost. So we should set n based on the environment and algorithm we are using.

3 DQN vs. DDQN

3.1 Which algorithm performs better and why? [3-points]

DDQN performs better because it designed to address the original DQN's issues! In DQN, the network tries to adjust Q-values to match labels that are computed by the same network, so these labels are dynamic. Since the labels change frequently, the model struggles to converge to optimal values, leading to instability. DDQN solves this by using a target network to predict the labels. This target network updates less frequently, allowing the policy network to gradually converge toward more accurate labels, resulting in better performance and stability.

3.2 Which algorithm has a tighter upper and lower bound for rewards. [2-points]

DDQN is more stable, and as shown in the related plots (see Figure 6 for example) the DDQN agents provide tighter bounds. Different agents lead to more consistent results compared to DQN, where the results vary significantly.

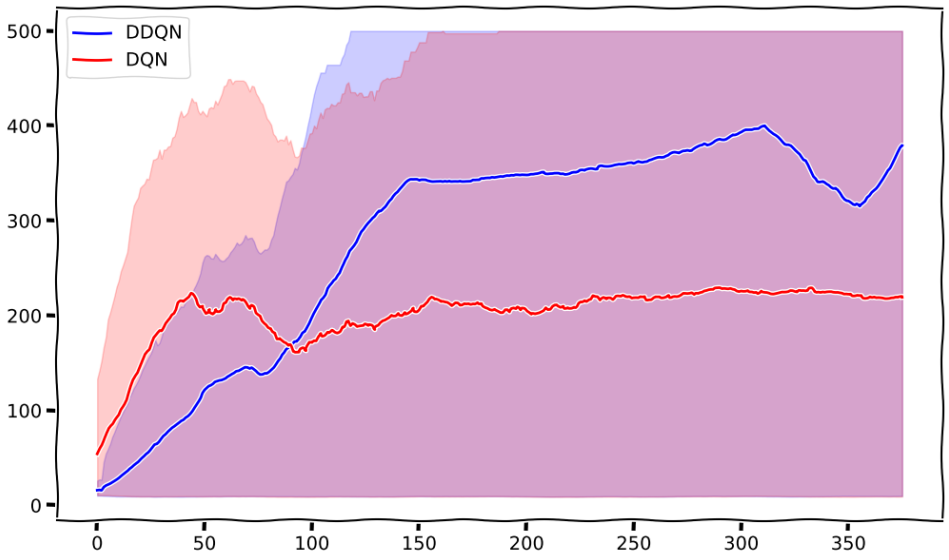

Figure 6: plot provided by plot_smooth function, comparing bounds and moving average of algorithms

3.3 Based on your previous answer, can we conclude that this algorithm exhibits greater stability in learning? Explain your reasoning. [2-points]

Yes, DDQN exhibits greater stability. Tighter bounds indicate that DDQN's performance is less variable accross different runs, which implies that it is more stable in its learning process.

3.4 What are the general issues with DQN? [2-points]

The main issue with DQN is overestimation bias, which leads to instability and slower convergence. As explained earlier, using a single network to predict Q-values for both the current and next states (for target labels) causes overestimation problems. This issue is addressed by DDQN. Additionally, DQN requires a large replay buffer for complex problems, leading to high memory usage and computational costs. While DDQN improves stability, it still struggles with these challenges, a problem that algorithms like PPO aim to resolve.

Another issue is catastrophic forgetting. Since DQN relies on replay buffer to train on past experiences, it can still struggle with retaining knowledge from older experiences when new data is pushed. This can lead to the model forgetting previously learned strategies, especially in non-stationary environments or when the replay buffer is not managed effectively.

3.5 How can some of these issues be mitigated? (You may refer to external sources such as research papers and blog posts be sure to cite them properly.) [3-points]

DDQN Addresses overestimation bias by decoupling action selection from Q-value estimation. The online network selects the action, while the target network evaluates it [6].

Another method reduces catastrophic forgetting by replaying important experiences more frequently. Transitions with higher TD error are given higher priority [7].

3.6 Based on the plotted values in the notebook, can the main purpose of DDQN be observed in the results? [2-points]

Yes. As you can see in figures 7 and 8, the overestimation bias problem is resolved and q-values are more consistent in DDQN comparing to DQN. Note that only models with average reward more than 450 are drawn. If we include all trained agents, it is even more visible.

3.7 The DDQN paper states that different environments influence the algorithm in various ways. Explain these characteristics (e.g., complexity, dynamics of the environment) and their impact on DDQN's performance. Then, compare them to the CartPole environment. Does CartPole exhibit these characteristics or not? [4-points]

The performance of DDQN is influenced by environmental characteristics like complexity, stochasticity, and reward sparsity. In complex or stochastic environments, DDQN reduces overestimation bias, leading to more stable and robust learning. In environments with sparse rewards, DDQN's conservative Q-value estimates help avoid misleading overoptimistic policies, improving learning efficiency [8].

However, CartPole is a simple, deterministic environment with dense rewards. While DDQN still improves over DQN, CartPole does not fully exhibit the characteristics where DDQN's advantages are significant.

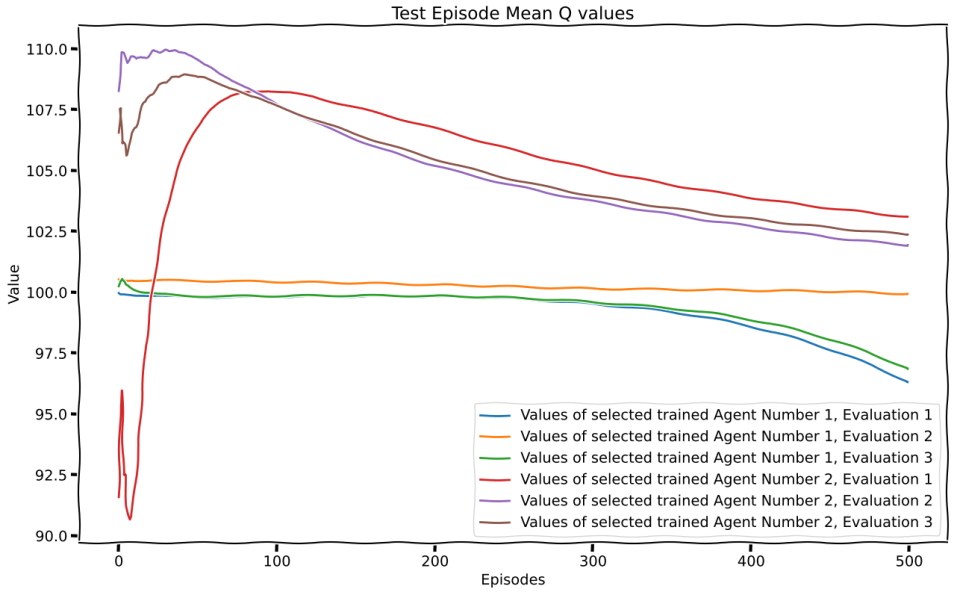

Figure 7: Mean Q values over episodes for some of the best DQN agents

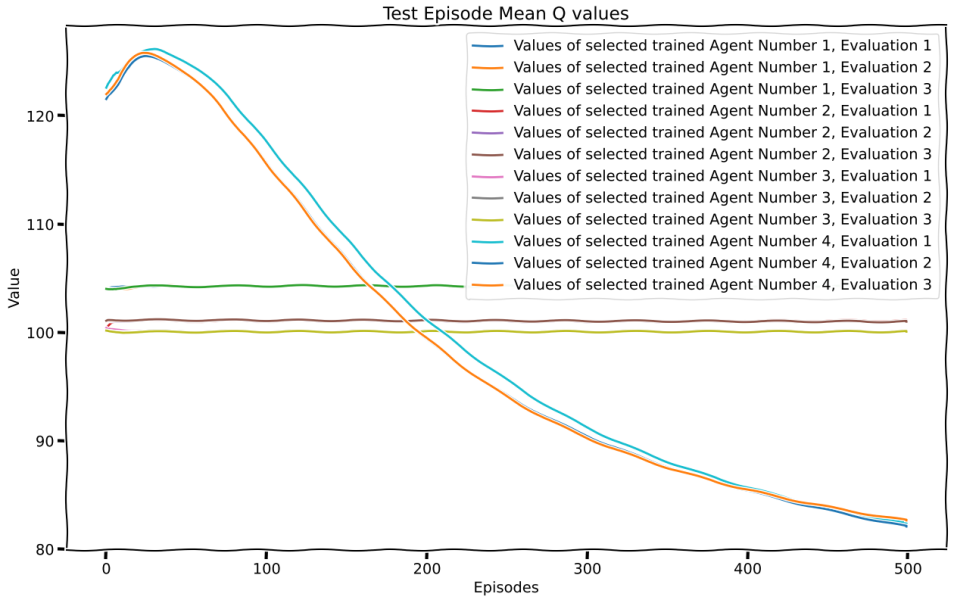

Figure 8: Mean Q values over episodes for some of the best DDQN agents

3.8 How do you think DQN can be further improved? (This question is for your own analysis, but you may refer to external sources such as research papers and blog posts be sure to cite them properly.) [2-points]

Techniques like DDQN or prioritized experience replay can be used as mentioned earlier [6, 7]. Also we can replace epsilon-greedy exploration with other methods which improves the algorithm specially in sparse reward environments [9]. Combining several of these methods (some of which are mentioned) together results in the best variation of DQN known as Rainbow DQN which outperforms standard DQN in almost all environments [10].

References

[1] R. Sutton and A. Barto, Reinforcement Learning: An Introduction, 2nd Edition, 2020. Available: http://incompleteideas.net/book/the-book-2nd.html.

[2] Gymnasium Documentation. Available: https://gymnasium.farama.org/

[3] Grokking Deep Reinforcement Learning. Available: https://www.manning.com/books/grokking-deep-reinforcement-learning

[4] Deep Reinforcement Learning with Double Q-learning. Available: https://arxiv.org/abs/1509.06461

[5] Cover image designed by freepik

[6] Van Hasselt, H., Guez, A., Silver, D. (2016). Deep Reinforcement Learning with Double Q-Learning. AAAI. Available: https://arxiv.org/abs/1509.06461.

[7] Schaul, T., Quan, J., Antonoglou, I., Silver, D. (2015). Prioritized Experience Replay. ICLR. Available: https://arxiv.org/abs/1511.05952.

[8] This paragraph was generated by ChatGPT (OpenAI, 2025).

[9] Fortunato, M., et al. (2017). Noisy Networks for Exploration. ICLR. Available: https://arxiv.org/abs/1706.10295.

[10] Hessel, M., et al. (2018). Rainbow: Combining Improvements in Deep Reinforcement Learning. AAAI. Available: https://arxiv.org/abs/1710.02298.