Stabilizing Deep Q Learning

Emre Uğur [1]

Abstract

This paper explores the implementation of Deep Q-Learning (DQL) with a target network and replay buffer. Both components have been shown to stabilize DQL learning, allowing it to learn faster and helping agents adapt better to the environment. Optimum performance is achieved with the combination of both the target network and the replay buffer.

1. Introduction

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make decisions in an environment. Agents explore the environment in a trial-and-error fashion and try to find a policy that yields the maximum cumulative reward over sequential actions.

In the simplest terms, traditional RL uses a table to store the values of each state or each state-action pair. Later, it decides on the optimal action for a particular state by choosing the action that leads to the state with the highest value. However, with large or continuous state spaces, this tabular method becomes unfeasible due to memory limitations. In such scenarios, the tables used to store state or state-action values are approximated, often by a neural network—this approach is known as Deep Reinforcement Learning (DRL). DRL has recently gained popularity due to its successful applications in self-driving cars and game playing (e.g., AlphaGo).

However, naïve implementations of DRL that directly mimic the tabular case often result in unstable learning. This paper explores two techniques—Replay Buffers and Target Networks—to stabilize the learning of Deep Q-Learning (DQL).

2. Theory

2.1. Q-learning

Q-learning is a model-free reinforcement learning algorithm used to find the optimal action-selection policy for a given environment. It is based on learning a Q-value function, which estimates the expected cumulative reward for taking a certain action in a given state.

2.1.1. Key Components

- **Q-table (Q-values, Q(s, a))**: A table storing the expected rewards for each state-action pair.
- **Bellman Equation**: Updates Q-values iteratively based on rewards and future estimates:

$$Q(s,a) \leftarrow Q(s,a)+\alpha[r+\gamma \max_{a'} Q(s',a')-Q(s,a)] \quad (1)$$

where:

- α (learning rate): How much new information overrides the old.
- γ (discount factor): How much future rewards matter.
- r (reward): Immediate feedback from the environment.
- s' is the next state,
- a' represents possible actions in the next state.

2.1.2. Learning Process

1. Start with an empty Q-table.
2. Select actions using an exploration-exploitation strategy (e.g., ε-greedy).
3. Observe reward and next state.
4. Update Q-values using the Bellman equation.
5. Repeat until convergence.

2.2. Deep Q-Learning

In **Deep Q-Learning** (DQL), the **Q-table** is approximated using a neural network parameterized by θ, denoted as $Q^\theta(s,a)$ (1).

The Q-update rule in Deep Q-Learning is given by:

$$Q^\theta(s,a) \leftarrow Q^\theta(s,a)+\alpha \left[r+\gamma \max_{a'} Q^\theta(s',a')-Q^\theta(s,a)\right]$$

where:θ represents the neural network parameters (weights)

However, the update cannot be performed as in the tabular case, where we can assign a specific value to the $Q(s,a)$ entry. Our Q-network has fewer free parameters than the number of state-action pairs, unlike the tabular case, which has the same number of free parameters. Therefore, some weights in the network must be shared across different (s,a), meaning that modifying one (s,a) entry will affect other entries. More importantly, our network is a black box, and we do not know which free parameter corresponds to a particular entry.

To guide the network, we provide it with a loss function, as done in supervised learning. This loss is computed as:

$$r + \gamma \max_{a'} Q^\theta(s',a') - Q^\theta(s,a)$$

where $r + \gamma \max_{a'} Q^\theta(s',a')$ is the value we want our network to output, and $Q^\theta(s,a)$ is our current prediction. The difference between these two values serves as a loss signal for backpropagation.

2.3. Stability Problems & Improvements

The naive implementation described above has several problems that can result in suboptimal training.

2.3.1. Correlation Between Samples

In the naive implementation, Q-network updates are performed on consecutive experiences, resulting in correlation between samples. This violates a key assumption in neural network training—that samples should be independent and identically distributed (i.i.d.)—and causes instability in learning.

2.3.2. Moving Target Problem

When updating $Q^\theta(s,a)$, we use $Q^\theta(s',a')$ in our target value computation. However, when an update is performed to improve $Q^\theta(s,a)$, the value of $Q^\theta(s',a')$, which was part of our target value, also gets modified. This creates a moving target problem, leading to unstable learning.

2.3.3. Replay Buffers

Instead of updating the network immediately after collecting an experience, experiences from episodes are stored in a buffer. Random samples from this buffer are later used to update the Q-network, which breaks the correlation between samples by randomizing their order and allows batch training, improving efficiency. Both of these factors contribute to more stable training.(2)

2.3.4. Target Networks

While calculating the target value for $Q^\theta(s,a)$, the value $Q^\theta(s',a')$ is used. Note that both are obtained from the same neural network parameterized by θ. This can lead to instability, as the target values keep shifting due to continuous updates to the network. (2) To address this issue, a separate Q-network, called the target network, is introduced. The target network, with parameters θ^-, is a periodically updated copy of the main Q-network. Instead of using $Q^\theta(s',a')$, the target network provides a more stable target $Q^{\theta^-}(s',a')$, as it is not affected by updates to $Q^\theta(s,a)$. This results in the following update rule:

$$Q^\theta(s,a) \leftarrow Q^\theta(s,a) + \alpha \left[r + \gamma \max_{a'} Q^{\theta^-}(s',a') - Q^\theta(s,a) \right]$$

where θ^- represents the parameters of the target network, which are updated periodically ($\theta^- \leftarrow \theta$ after a fixed number of steps).

2.3.5. Combined Pseudocode

Input: Learning rate α, discount factor γ, update interval m, replay buffer size N, total budget, batch size n.

```
1. **Initialize** Q-network $Q^\theta(s,a)$
2. **Initialize** target network $Q^{\theta^-}(s,a) \leftarrow Q^\theta(s,a)$
3. **Initialize** replay buffer $\mathcal{D}$
4. **Sample** initial state $s$
5. **while** budget do:
   (a) **if** at k*$m^{\text{th}}$ iteration(k: integer):
       i. $Q^{\theta^-} \leftarrow Q^\theta$     /* Update target network */
   (b) **Sample** action $a$ using policy (e.g., $\epsilon$-greedy)
   (c) **Simulate environment** to obtain reward and next state:
       i. $r, s' \sim p(r, s' \mid s, a)$
   (d) **Store experience** $(s, a, s', r)$ in replay buffer $\mathcal{D}$
   (e) **Sample** $n$ experiences from replay buffer
   (f) **Perform Q-update** using sampled batch:
       $Q^\theta(s,a) \leftarrow Q^\theta(s,a) + \alpha \left[ r + \gamma \max_{a'} Q^{\theta^-}(s',a') - Q^\theta(s,a) \right]$
   (g) **Set** $s \leftarrow s'$
```

Return: $Q^\theta(s,a)$

Setup

Environments

The Gymnasium CartPole environment has a 4D continuous state space consisting of cart position, cart velocity, pole angle, and pole angular velocity. The action space is discrete with two actions: moving the cart left (0) or right (1). The maximum number of steps is set to 150 before the environment truncates. At each timestep, the environment returns a reward of 1, meaning the longer the agent balances the pole, the more rewards it accumulates. (3)

Q Network

The Q-network is structured as an MLP with 5 inputs and a single output, where values < 0 indicate moving left and > 0 indicate moving right. The number of hidden layers and their dimensions are tunable. The DQL algorithm is limited to 5000 steps, which was empirically shown to be sufficient for learning while remaining computationally manageable. Every 250 steps, the system is evaluated by running 30 independent episodes using the current policy and averaging the accumulated returns. The final training score is obtained by averaging the last 10 evaluations, promoting stability and early goal learning.

Results

Naive DQL

The naive DQL is parameterized by the following hyperparameters. Each hyperparameter is tested with four values:

- `gamma_values` = [0.90, 0.95, 0.99, 1.0]
- `learning_rate_values` = [0.01, 0.05, 0.1, 0.2]
- `hidden_dim_values` = [8, 16, 32, 64]
- `num_hidden_layers_values` = [1, 2, 3, 4]
- `policy_values` = ['egreedy', 'softmax']
- `epsilon_values` = [0.05, 0.1, 0.2, 0.3]
- `temp_values` = [0.5, 1.0, 1.5, 2.0]

The entire configuration space is explored using grid search to find the optimal hyperparameters and assess their relative importance.

Below is the optimal configuration obtained from the grid search and the corresponding learning curve Figure 1.

- **gamma:** 0.9
- **learning_rate:** 0.01
- **hidden_dim:** 8
- **num_hidden_layers:** 4
- **policy:** "softmax"
- **temp:** 2.0

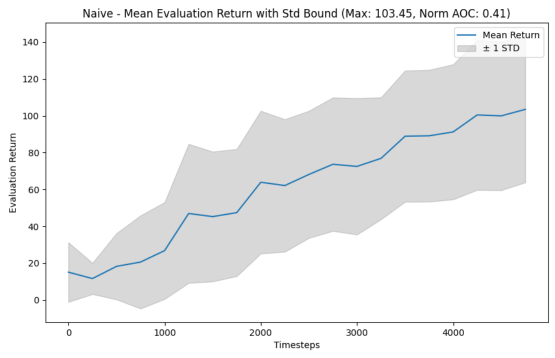

Figure 1. Learning curve of the naive implementation averaged over 50 runs

To gain an insight of the parameter importance, training scores with respect to each hyperparameter are computed. Figure 2

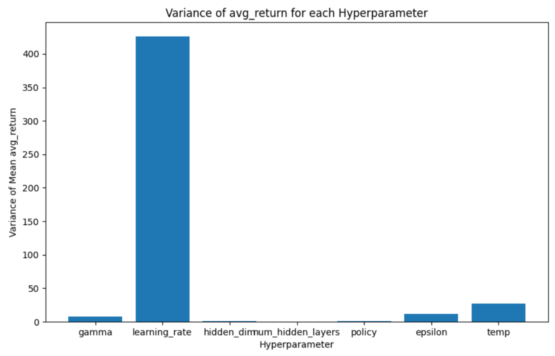

Figure 2. Variance of training score with respect to each hyper parameter to gain insight on parameter importance

Replay Buffer

Introducing a replay buffer adds two hyperparameters: buffer size and batch size. The replay buffer is implemented using a deque data structure with a maximum buffer size. Batch size determines the number of experiences sampled from the buffer at each training step. Given that DQL runs for 5000 steps, the maximum buffer size considered is 5000.

Smaller buffer sizes increase the sampling probability of recent experiences, equalizing their sampling chances relative to older experiences.

- `capacity_values` = [500, 1000, 2000, 5000]
- `batch_size_values` = [4, 8, 16, 32]

These hyperparameters are tested while fixing the previously determined optimal values, where the grid search identified a capacity of 5000 and a batch size of 32 as optimal.

Figure 3 shows the learning curve corresponding to this configuration:

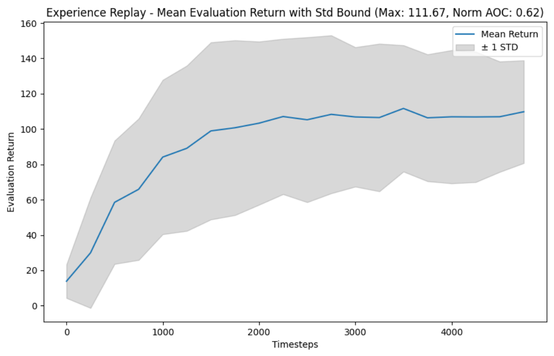

Figure 3. Learning curve of DQL using a replay buffer averaged over 50 runs

The variance attributed to each hyperparameter is shown in Figure 4.

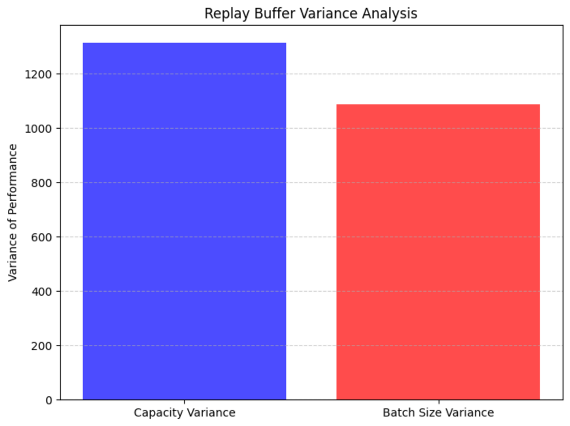

Figure 4. Variance of training score with respect to each replay buffer parameter

Target Network

Using a target network introduces another hyperparameter: the update interval, specifying how frequently the target network parameters are updated. The following intervals were tested [10, 50, 100, 200, 500, 1000]. Again, previous hyperparameters were fixed, and only the target update interval was varied. The grid search revealed that updating the target network every 50 timesteps is optimal for this task. Figure 5 is the corresponding training curve.

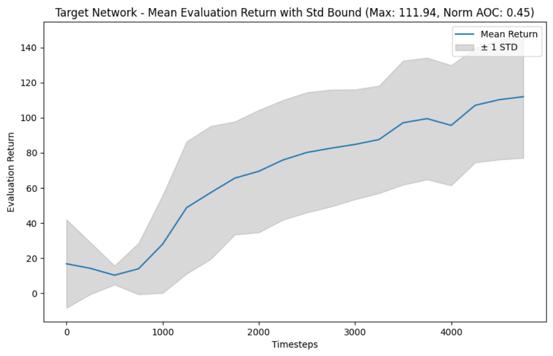

Figure 5. Learning curve of DQL using a target network averaged over 50 runs.

Experience Replay Buffer & Target Network Combined Results

The final training curve Figure 6, using both the optimal replay buffer and target network hyperparameters determined above

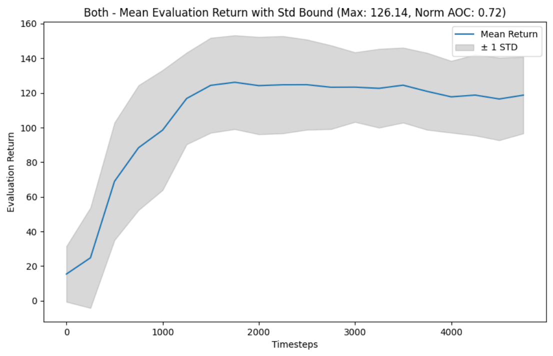

Figure 6. Learning curve of DQL using both a target network and a replay buffer averaged over 50 runs.

Comparison of all methods

Figure 7 shows each learning curve(mean returns averaged over 50 runs) in a single plot for comparison.

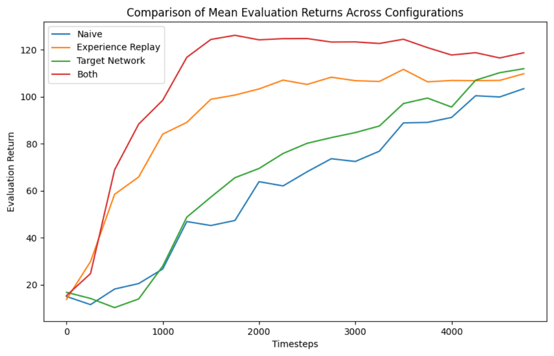

Figure 7. Results for Naive - Only TN - Only ER - TN & ER in a single graph

3. Conclusion

The learning curves are compared using AOC because it quantifies the total accumulated reward, indicating learning efficiency, while the variance within runs and their deviation from an exponential decay function are also inspected since a more stable learning algorithm is expected to show less deviation across runs and exhibit a shape resembling exponential decay.

The agent trained with naive DQL manages to adapt to its environment; however, the learning is unstable as the curve does not follow a standard exponential decay function, and the variance within runs is high, as seen in Figure 1. When the hyperparameter variances are examined (see Figure 2), the learning rate's importance dominates the other parameters. In many cases, tuning only this parameter will suffice. The exploration policy does not seem to have a major effect on performance, suggesting that the two methods, e-greedy and softmax, are practically identical in this context. However, their respective parameters do have an effect, as epsilon and temperature contribute to performance. The Q-network parameters, including hidden layer dimensions and the number of hidden layers, have almost no effect on performance, suggesting that all the possible network configurations are more than sufficient for this simple problem and that increasing the network's complexity does not yield significant improvement.

Introducing an experience replay buffer stabilizes learning, as the curve more closely resembles an exponential decay function, and the AOC of the training curve is significantly higher compared to the naive version (naive = 0.41, experience replay = 0.62) (see Figures 1,3). However, the variance within runs does not show an improvement. Inspection of the hyperparameter importance (Figure 4) for the two parameters introduced by the replay buffer suggests that both parameters are equally important.

The target network also stabilizes learning, as the variance within runs decreases and learning achieves a slightly higher AOC compared to the naive version (naive = 0.41, target network = 0.45) (see Figures 1, 5). However, the problem of deviating from the exponential decay curve persists.

When both components are used simultaneously, learning demonstrates its best performance, as the curve closely matches the exponential decay, and the variance within runs is significantly reduced (see Figures 6). The AOC is highest (combined = 0.72), indicating that the experience replay buffer and the target network address different aspects of stability and performance in the learning process.

Figure 7 provides a concise summary of the analysis. The naive implementation exhibits slow and unstable learning. Both the experience replay buffer and the target network enhance learning; however, the experience replay buffer yields a more dramatic improvement by accelerating the learning process. The combination of both achieves optimal performance, resulting in the fastest learning and producing an agent that adapts to its environment more effectively.

3.1. Improvements & Further Research

Further research can focus on more advanced environments to observe the effects of these improvements on more complex learning problems. The range of the gamma parameter can also be extended beyond [0.9, 1] to further assess its importance, as the limited range used in this paper may account for the low variance observed in Figure 2.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[2] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 32).

[3] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. *arXiv preprint arXiv:1606.01540*.