Deep Double Q Learning

An Le
504915864
UCLA Electrical and Computer Engineering
`binhanle@ucla.edu`

Ying-Huan Chen
005430127
UCLA Electrical and Computer Engineering
`yinghuanchen@g.ucla.edu`

Loic Maxwell
504422999
UCLA Electrical and Computer Engineering
`loicmaxwell17@ucla.edu`

Scott Bauersfeld
604813474
UCLA Department of Computer Science
`sbauersfeld@g.ucla.edu`

Abstract

Deep Q-learning has achieved remarkable success in a variety of reinforcement learning applications.in particular, Deep Q-learning agents can play Atari games at superhuman levels. However; Q-learning methods have been proven to consistently overestimate Q-values, which may impair agent performance. This limitation led van Hasselt et al. [2015] to introduce Double Deep Q-learning, which mitigates Q-value overestimation. In this paper, we examine the benefits of Double Deep Q-learning over traditional Deep Q-learning with respect to agent performance and training speed. After evaluating both optimization strategies on simple Markovian Decision Processes and a variety of Atari games, we determine that Double Deep Q-learning tends to improve agent performance, reduce training time, and increase training stability.

1 Introduction

Reinforcement learning attempts to learn a policy that can maximize the cumulative reward of an agent interacting with an environment through sequential decisions. One of the most popular reinforcement learning algorithms is Deep Q-learning [Mnih et al., 2015]. However, Deep Q-learning is based upon Q-learning, which is known to overestimate the value of state-action pairs due to inflexible function approximation [Thrun and Schwartz, 1993] and noise [Hasselt, 2010].

Value overestimation does not necessarily impair agent performance. In fact, value overestimation in the face of uncertainty is a widely used exploration technique [Kaelbling et al., 1996]. In addition, uniformly increasing the value of all state-action pairs would not change an ϵ-greedy policy. However, if the overestimation is not concentrated towards states and actions for which we wish to gather more information, the resulting policy may not be optimal. In order to prevent Q-value overestimation and test whether value overestimation impairs performance, van Hasselt et al. [2015] apply the double Q-learning algorithm [Hasselt, 2010] to training deep learning models for function approximation.

We implement Deep Q Networks (DQN) and Double Deep Q Networks (DDQN) in several MDP and Atari environments in order to compare their performance. The results of our experiments show that the Double Deep Q-learning algorithm produces better function approximation estimates than traditional deep Q-learning methods and increases performance in several learning tasks.

2 Background

An optimal Q-value is defined as the expected cumulative return if action A_0 is played in state S_0 and the optimal policy is played thereafter:

$$Q(s,a) = E\left[\sum_{t=0}^{\infty} \gamma^t * R_t \,\middle|\, a = A_0, s = S_0\right] \tag{1}$$

where γ is a discount factor between 0 and 1. In order to maximize the expected cumulative reward, a good policy should select the action with the maximum Q-value for any given state. As long as the Q-values are optimal, then this greedy policy will also be optimal.

The Q-learning algorithm proposed by Watkins and Dayan [1991] estimates optimal Q-values by assuming that the Bellman equation for Q-values can be expressed as:

$$Q(s_t, a_t) = R_{t+1} + \gamma * \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{2}$$

However, many environments have too many state-action pairs for all Q-values to be enumerated, so a set of parameters are used to approximate Q-values as $Q(s,a;\theta_t)$. To update these parameters, the agent takes an action A_t in state S_t and observes reward R_t and state S_{t+1}. The parameters are then updated as:

$$\theta_{t+1} = \theta_t + \alpha * (Y_t^Q - Q(s_t, a_t; \theta_t)\nabla_{\theta_t} Q(s_t, a_t; \theta_t) \tag{3}$$

where α is the learning rate and the target Y_t^Q is defined as:

$$Y_t^Q \equiv R_{t+1} + \gamma * \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_t) \tag{4}$$

2.1 Deep Q Learning

Deep Q-learning uses a deep neural network with parameters θ to estimate the optimal Q-value, $Q(s,a;\theta)$, for each action when given a state as input. In order to effectively train a DQN, Mnih et al. [2015] used experience replay buffers and target networks. The experience replay buffer stores the history of transitions, $(s_t, a_t, s_{t+1}, r_{t+1})$, between states so that the network parameters can be updated by uniformly sampling transitions from the replay buffer. This allows the network to train on each transition multiple times and increases the independence between data samples, which improves training efficiency and stability.

The target network has parameters θ', which are copied from the online network's parameters every τ steps and otherwise remain constant. The target Q-value is then computed as:

$$Y_t^{DQN} \equiv R_{t+1} + \gamma * \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta'_t) \tag{5}$$

By using a separate target network to compute the target value, Mnih et al. [2015] further improves training stability by ensuring that the approximation for the target Q-value remains stable even as the online network is changing.

2.2 Q-value Overestimation

The main weakness of Q-learning is Q-value overestimation, which can arise from noise in Q-values and function approximation. To illustrate the first case, Sutton and Barto [2018] specify an MDP with two states A and B, where A is the starting state (see Figure 4a). Taking the right action from A leads to the terminal state with reward 0, while taking the left action takes the agent to B with reward 0. From B, the agent has a choice of many actions all leading to the terminal state with a reward taken from a Gaussian distribution with mean -0.1 and variance 1. Evidently, the expected reward of taking the left action from A is -0.1, making the right action the superior option. However, during exploration, the agent is likely to encounter a positive reward from state B. The expression $\max_a Q_t(s,a)$, evaluated at state A only needs one of the actions from B to be positive for itself to be positive. In fact, van Hasselt et al. [2015] proved that $\max_a Q_t(s,a)$ exceeds $V_*(s)$ by at least $\sqrt{\frac{C}{m-1}}$, where m is the number of actions and $C = \frac{1}{m}\sum_a (Q_t(s,a) - V_*(s))^2$. Thus, the maximization of Q-values over actions is prone to overestimation.

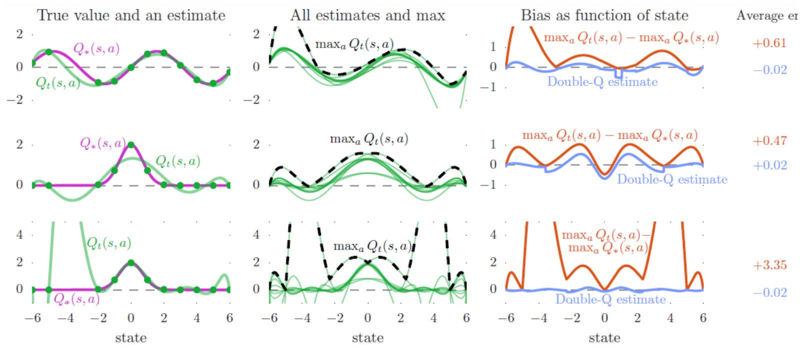


Figure 1: Q-value overestimation in function approximation as demonstrated by van Hasselt et al. [2015]. Left column: The true values $Q_*(s,a)$ (purple) and their polynomial estimates $Q_t(s,a)$ (green). The polynomials are fitted to the true values of the same 10 states. Middle column: The estimated values for 10 actions (green) and the maximum of the estimates $\max_a Q_t(s,a)$ (dashed black). Right column: The error between $\max_a Q_t(s,a)$ and $\max_a Q_*(s,a)$, used in Q-learning (orange); and the estimate derived from Double Q-learning (blue)

In addition to variance from MDPs, maximization bias also emerges from function approximation, even if the environment is deterministic. In an example provided by van Hasselt et al. [2015] in Figure 1, the functions $Q_*(s,a) = \sin(s)$ and $Q_*(s,a) = 2\exp(-s^2)$ are approximated by degree-6 and degree-9 polynomials, respectively (Q_t). Errors appear at unsampled states, whether or not the Q-value approximations are accurate at sampled states. For similar reasons as the stochastic MDP case, the maximum operator magnifies the overall error between $Q_t(s,a)$ and $Q_*(s,a)$. Therefore, since DQNs are universal function approximators, they are not immune to Q-value overestimation.

2.3 Double Q-learning

Hasselt [2010] introduced Double Q-learning to mitigate the maximization bias of Q-learning. Double Q-learning utilizes two separate estimators Q^A and Q^B: one to select the maximizing action from the next state and the other to evaluate its value. The online greedy action is based on $Q^A + Q^B$. In each iteration of Double Q-learning, either Q^A or Q^B is randomly selected for updating. The update formulae for Q^A and Q^b can be expressed as:

$$Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(r + \gamma Q^B(s', a^*) - Q^A(s,a)) \quad (6)$$

$$Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(r + \gamma Q^A(s', b^*) - Q^B(s,a)) \quad (7)$$

where $a^* = \text{argmax}_a Q^A(s',a)$ and $b^* = \text{argmax}_a Q^B(s',a)$. Since only one estimator is updated at a time, the two estimators learn from different sets of samples. $Q^B(s',a^*)$ is therefore an unbiased estimate of $Q^A(s',a^*) = \max_a Q^A(s',a)$ and vice versa. Thus, Double Q-learning learns Q-values more reliably than ordinary Q-learning.

3 Double Deep Q Learning

Double Deep Q-Learning applies the Double Q-learning algorithm to training deep neural networks. The intuition behind this algorithm is to combine the powerful Q-function approximation utility provided by deep neural networks, while applying double Q-learning in an attempt to mitigate the Q-function overestimation and the maximization bias of Q-learning. Both DQN and Double DQN update an online network with parameters θ in order to satisfy the Bellman equation $Q(s_t, a; \theta) = Y_t^Q$ where the target Y_t^Q represents $R + \gamma V(s_{t+1})$. In addition, both DQN and Double DQN use a target network with parameters θ' to estimate the value of the next state. The only difference between DQN and Double DQN lies in the calculation of the target. The DQN target in equation 5 can be expressed as:

$$Y_t^{DQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \theta_t^{'}); \theta_t^{'}) \tag{8}$$

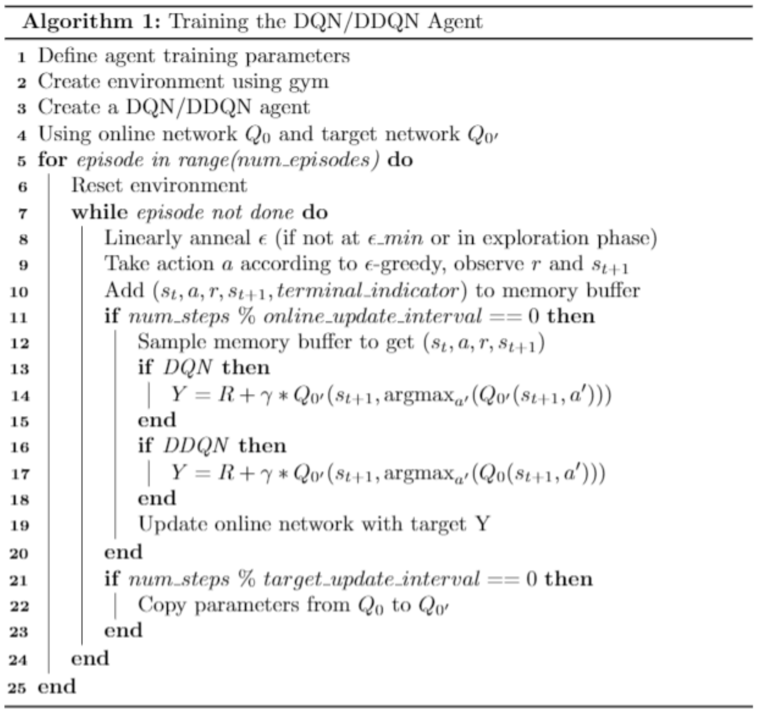
The Double DQN target is:

$$Y_t^{DDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \theta_t); \theta_t^{'}) \tag{9}$$

For Double DQN, the action selection for state s_{t+1} is still based on the online network (θ_t), so DDQN estimates the value of the greedy policy according to the online network. However, the target network ($\theta_t^{'}$) evaluates the value of current greedy policy. This is analogous to Double Q-learning, in which one Q-value estimator is used to select the action for a given state and another Q-value estimator evaluates the state-action pair.

3.1 Paper Results and Algorithm Pseudocode

In their paper, van Hasselt et al. [2015] applied the DQN and DDQN algorithms to a variety of Atari game environments. The pseudocode for these algorithms is shown below:



```
Algorithm 1: Training the DQN/DDQN Agent
Define agent training parameters
Create environment using gym
Create a DQN/DDQN agent
Using online network $Q_0$ and target network $Q_{0'}$
for episode in range(num_episodes) do
    Reset environment
    while episode not done do
        Linearly anneal $\epsilon$ (if not at $\epsilon\_min$ or in exploration phase)
        Take action $a$ according to $\epsilon$-greedy, observe $r$ and $s_{t+1}$
        Add ($s_t, a, r, s_{t+1}, terminal\_indicator$) to memory buffer
        if num_steps % online_update_interval == 0 then
            Sample memory buffer to get ($s_t, a, r, s_{t+1}$)
            if DQN then
                $Y = R + \gamma * Q_{0'}(s_{t+1}, \text{argmax}_{a'}(Q_{0'}(s_{t+1}, a')))$
            end
            if DDQN then
                $Y = R + \gamma * Q_{0'}(s_{t+1}, \text{argmax}_{a'}(Q_0(s_{t+1}, a')))$
            end
            Update online network with target Y
        end
        if num_steps % target_update_interval == 0 then
            Copy parameters from $Q_0$ to $Q_{0'}$
        end
    end
end
```

Figure 2: Pseudocode for training the DQN and DDQN agents for playing Atari games.

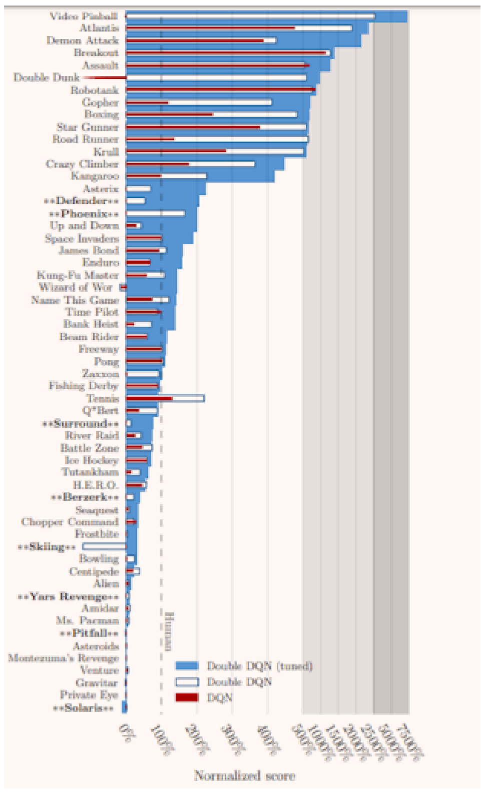
The training process starts by defining a set of hyperparameters. The training hyperparameters include, but are not limited to, the replay memory buffer size, the target/online network update intervals, initial exploration phase length, start/end ϵ value and the ϵ anneal rate. The network hyperparameters relating to the online/target network training include, but are not limited to, the optimizer (SGD with momentum, Adam, RMSProp, etc.), the learning rate, and the batch size.

Training is done by running a certain number of episodes and updating the online and target networks according to their respective update rates. For each step in an episode, ϵ is first linearly annealed as long as the exploration rate has elapsed and the minimum ϵ has not yet been reached.

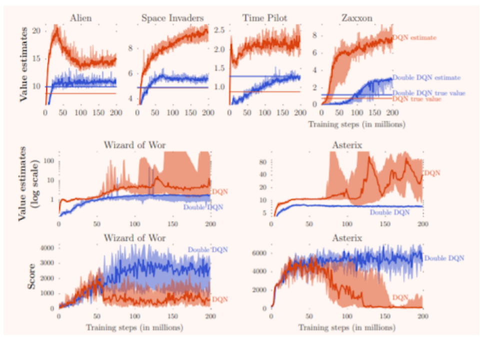
Then an action is selected according to the ϵ-greedy policy using the online network to calculate Q-values, and the resulting reward and next state are observed. The (state, action, reward, next state) tuple is then added to the replay memory. During an online network update, the memory is first sampled to obtain a (state, action, reward, next state) tuple, which is used to compute the target Q-value, Y.

The key difference between the DQN and DDQN agents lies in this computation. The target Q-value is used for optimizing the online network using one of the aforementioned optimizers while using the Huber loss metric. The DDQN agent uses the online network to select the greedy action for the next state while the target network evaluates this state-action pair. On the other hand, the DQN agent uses the target network to carry out both of these tasks. When selecting the action for a state with one network and evaluating its Q-value with a different network, we are able to diminish Q-value overestimation by decoupling action selection from value estimation. Therefore, DDQN is able to yield better results because it provides a better approximation of the true Q-value for any state-action pair.

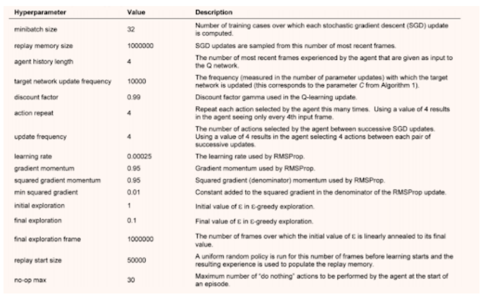
During a target network update, the parameters from the online network are simply copied over to the target network. By applying the above algorithms, van Hasselt et al. [2015] were able to produce the following results:



(a)



(b)



Hyperparameter	Value	Description
minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter C from Algorithm 1).
discount factor	0.99	Discount factor gamma used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
update frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	Gradient momentum used by RMSProp.
squared gradient momentum	0.95	Squared gradient (denominator) momentum used by RMSProp.
min squared gradient	0.01	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial value of ε in ε-greedy exploration.
final exploration	0.1	Final value of ε in ε-greedy exploration.
final exploration frame	1000000	The number of frames over which the initial value of ε is linearly annealed to its final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.

(c)

Figure 3: van Hasselt et al. [2015] results: (a) Agent score for each Atari game (b) Value function estimates for DQN/DDQN agents, across each timestep (c) Training and network hyperparameters for untuned network

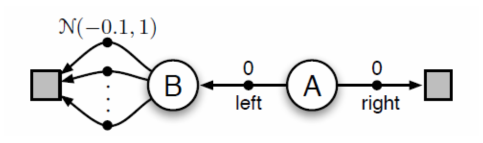
The reported performances of DQN and DDQN agents across many Atari games by van Hasselt et al. [2015] are depicted in Figure 3a. This figure clearly demonstrates the superior performance of DDQN over standard DQN. Figure 3b helps support these results, by demonstrating how double Q learning reduces the value function overestimation. The training parameters for the untuned DDQN and regular DQN are shown in Figure 3c. The tuned version of the DDQN network used a target network update frequency of 30,000 and a final ϵ value of 0.01. Otherwise, all other hyperparameters were left unchanged. Additionally, a single shared bias was used for all action values in the top layer of the network. In most cases, this tuning further improved the performance on each of the games.

4 Our Results

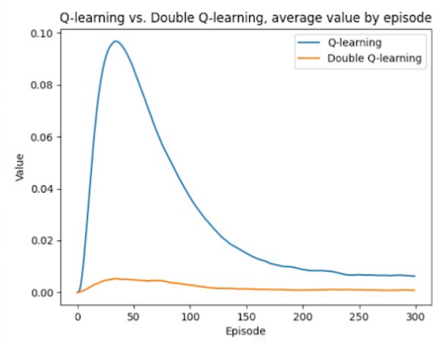
We first implemented DQN and DDQN on a simple Markov Decision Process to show that DDQN is less susceptible to overestimating Q-values and achieves better performance as a result. We then implemented DQN and DDQN on a variety of Atari games to show that reducing overestimation allows DDQN to achieve better performance and faster training times in complex environments.

4.1 Markov Decision Process

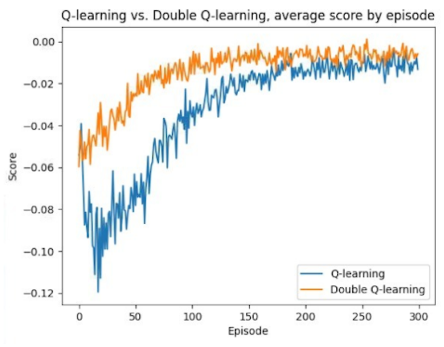
The effects of maximization bias are shown in Figure 4. In the first 50 episodes, maximization induces the Q-value of (Figure 4a) to be positive (Figure 4c), which causes the agent to favor the left action heavily (Figure 4d). As the agent continues to explore B, the average score dips. Once the agent finally learns that going left has an expected reward of -0.1, it gradually switches to the right action. As a result, the value estimates and average score stabilize at near zero (Figure 4b). In comparison to DQN, DDQN reduces overestimation of the left action and achieves an optimal policy much more quickly.



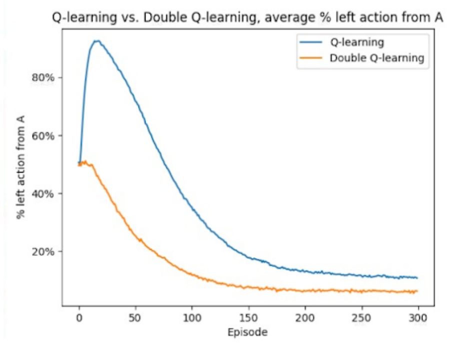
(a)



(c)



(b)



(d)

Figure 4: (a) MDP diagram (b) DQN and DDQN agent performance on this MDP across each episode (c) Average value function estimation across each episode (d) Chance of selecting the left action from A across each episode

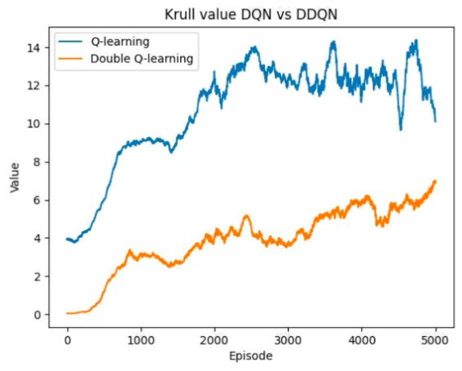
4.2 Atari Games

To further compare the performance and training speed of DQN and DDQN, we trained each agent on several Atari games: Krull, Pong, Breakout, Asterix, Alien, Boxer, and Roadrunner. In order to improve the performance of our agents on these games, we performed the following pre-processing steps as recommended by Mnih et al. [2013].

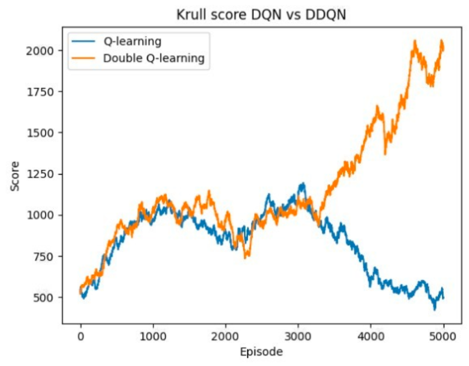
1. Input frames are divided into consecutive groups of 4.
2. The reward is summed across all 4 frames in the group and clipped between -1 and +1 to reduce variance in the rewards.
3. All frames except the maximum of the last two frames in each group of 4 are discarded, which effectively downsamples the input frames by a factor of 4.
4. The remaining frame is converted to an 84x84 grayscale image.
5. Four consecutive remaining frames are stacked to form an 4x84x84 data sample to allow the network to determine the velocity of moving objects.
6. Any loss of life triggers the done signal so the network learns to penalize all losses of life rather than only game over signals.

The performance of DQN and DDQN is shown in Figures 5 and 6. The hyperparameter values used for training each of these agents are shown in Table 1. For a given game, we used the same hyperparameter values for training the DQN and DDQN agents. In general, DDQN tends to achieve higher scores with fewer training episodes compared to DQN.

The instability of Deep Q-learning can be seen when using DQN on “Krull” (Figure 5). Although DQN reaches high scores initially, the score is unstable due to value overestimation in Q-learning. This problem eventually causes the score to fall. In comparison, when using double DQN, the learning became much more stable. The same effect can be seen in “Breakout”. DQN initially reached a higher score compared to DDQN, but the score started falling after 25000 episodes, and became 0 after 30,000 episodes.



(a)



(b)

Figure 5: (a) The Q-value estimated by DQN and DDQN on Krull. (b) DQN and DDQN score by episode on Krull.

Table 1: Hyperparameters used while training each agent. All other hyperparameter values not shown here matched the ones specified in Figure 3c. The hyperparameter values for Boxer and Roadrunner, not shown here, matched those used for Alien.

	Krull	Breakout	Pong	Alien	Asterix
Optimizer	Adam	Adam	Adam	Adam	Adam
Learning Rate	0.00025	0.0000625	0.00025	0.0000625	0.0000625
Replay memory buffer size	10,000	1,000,000	10,000	10,000	10,000
Target Network Update Frequency	30,000	30,000	10,000	10,000	10,000
Replay Start Size (Initial Exploration Phase Length)	50,000	50,000	5,000	50,000	50,000
ϵ decrement steps	1,000,000	1,000,000	100,000	1,000,000	1,000,000
Final ϵ value	0.01	0.01	0.01	0.01	0.01

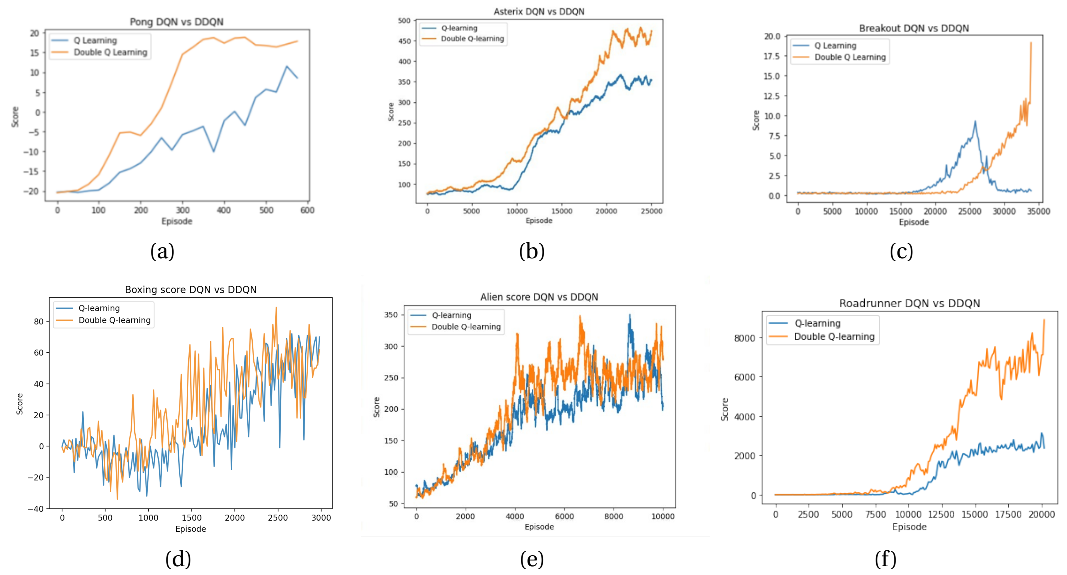


Figure 6: (a) Pong, (b) Asterix, (c) Breakout, (d) Boxing, (e) Alien, and (f) Roadrunner performance of DQN (blue) and DDQN (orange) agents.

5 Discussion

The score trends depicted in Figures 5 and 6 are as expected. DDQN was able to train more quickly and scored better on average than DQN. In the case of Krull, we also showed how DQN overestimates the value function to a greater degree than DDQN. However, van Hasselt et al. [2015] were able to train agents that achieved significantly higher scores on each of these games, with the exception of Pong. They trained each of their agents for about a week, which came out to be about 200M frames of game time per game. Due to our limited computational resources, we were only able to train each agent for about 4-8M frames each. Due to its simplicity however, our Pong agent trained significantly faster than each of the other games, and we were able to match the results from the paper.

Additionally, we briefly explored how the network depth affected the training time and agent performance. Mnih et al. [2015] used a network with 3 convolutional layers, followed by 2 fully connected layers. We trained a DDQN agent to play Asterix, on a network with 2 convolutional layers and another with 4 convolutional layers. After 30,000 episodes, neither was performing as well as our original agent (average scores of 300-350 compared to around 450). We suspect the shallower network was underfitting the data, while the deeper network simply trained slower. The deeper network has more potential, but it would take significantly more time to determine whether it could outperform the original network architecture used in van Hasselt et al. [2015] and Mnih et al. [2015].

6 Conclusion

In this paper, we investigated the benefits of Double Q-learning and DDQN over traditional Q-learning and DQN on MDPs and Atari games. We reproduced the Sutton and Barto [2018] plot comparing Q-learning and Double Q-learning, while also confirming the existence of value overestimation in stochastic, episodic MDPs. We also demonstrated the soundness of DDQN by implementing the algorithm on several Atari games. We observed that DDQN tends to estimate Q-values more accurately than DQN , which leads to faster and more stable training, as well as better performance on each game.

7 Team Member Contributions

An	Researched value overestimation and created stochastic MDP example Trained agents for Krull, Asterix, and Alien
Loic	Investigated the effects of network depth on DDQN training Implemented frame pre-processing and trained agent for Roadrunner
Scott	Implemented DQN and PyTorch code for training agents Trained agents for Pong and Breakout
Ying-Huan	Implemented DDQN and trained agent for Boxing Organized reports, poster, and presentation

References

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL `http://arxiv.org/abs/1509.06461`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, and et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015. doi: 10.1038/nature14236. URL `https://www.nature.com/articles/nature14236`.

Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *In Proceedings of the Fourth Connectionist Models Summer School*. Erlbaum, 1993.

Hado V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010. URL `http://papers.nips.cc/paper/3964-double-q-learning.pdf`.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996. URL `https://arxiv.org/abs/cs/9605103`.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning, Jan 1991. URL `https://link.springer.com/article/10.1007/BF00992698`.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL `http://arxiv.org/abs/1312.5602`.