Deep Double Q-learning

Prabhat Nagarajan, Martha White[1], Marlos C. Machado[1]
Department of Computing Science
University of Alberta
Alberta Machine Intelligence Institute
[1] Canada CIFAR AI Chair
Edmonton, AB, Canada
nagarajan@ualberta.ca, whitem@ualberta.ca, machado@ualberta.ca

Abstract

Double Q-learning is a classical control algorithm that mitigates the maximization bias of Q-learning. To do so, it explicitly trains two independent action-value functions and uses them to decouple action-selection from action-evaluation when computing bootstrap targets. Double DQN adapts this decoupling to deep reinforcement learning (RL) but directly trains only a single action-value function and does not fully decouple its estimators. Consequently, the two estimators remain correlated, and overestimation persists. In this paper, we introduce Deep Double Q-learning (DDQL), a deep RL algorithm that explicitly trains two Q-functions through the Double Q-learning objective. DDQL stabilizes training through a combination of techniques, including lower replay ratios, longer target network update intervals, and shared layers. Across 57 Atari 2600 games, DDQL improves aggregate performance over Double DQN, outperforming it on 47 games while further reducing overestimation. We study key design choices of DDQL, including its network architecture, replay ratio, and minibatch sampling strategies.

1 Introduction

In the classical Q-learning algorithm [42], the agent updates the value estimate of a state-action pair in part towards the value of the maximum estimated action-value in the subsequent state. Formally, it updates its current estimate, $Q(s, a)$, towards the *bootstrap target* $G = r + \gamma \max_{a'} Q(s', a')$, where r is the received reward, $\gamma \in [0, 1)$ is the discount factor, and $Q(s', a')$ is the estimated value of state-action pair (s', a'). The computation of G can be rewritten as

$$G = r + \gamma Q_{\text{eval}}\Big(s', \arg\max_{a'} Q_{\text{sel}}(s', a')\Big), \tag{1}$$

where $Q_{\text{eval}} = Q_{\text{sel}} = Q$. That is, the same Q is used to both select the greedy action whose value should be bootstrapped and then evaluate that greedy action to produce the bootstrap target.

This coupling of action-selection and action-evaluation in the bootstrap target, combined with the maximization operation, gives rise to the so-called *optimizer's curse* [29], or *maximization bias* [31]. If we estimate the highest action-value by using a single estimator to select and evaluate the best action, then this estimate is non-negatively biased [29, 34]. This bias causes overestimation, where the estimated value (i.e., discounted return) of a state-action pair exceeds its true value. These biased updates create a feedback loop where overestimated values are propagated backward to other action-value estimates.

Double Q-learning [34] is a classical algorithm that explicitly addresses the maximization bias in Q-learning. Double Q-learning maintains two independent estimators of the action-value function, Q_1 and Q_2, which are used to decouple action-selection and action-evaluation in the bootstrap target.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

Specifically, these Q-functions are trained through *reciprocal bootstrapping*, that is, the action-value functions bootstrap from one another. If a Q-function, say Q_1, is being updated, that Q-function is used to select the value-maximizing action, $\hat{a}^* = \arg\max_{a'} Q_1(s', a')$. The other Q-function, Q_2, evaluates the selected action, $Q_2(s', \hat{a}^*)$, to assign credit backward through bootstrapping. By doing so, Double Q-learning mitigates the maximization bias and consequent overestimation of Q-learning.

The Deep Q-network (DQN) algorithm [22], the analog of Q-learning in deep RL, also suffers from overestimation, and Double DQN [35] was introduced as a solution. Like DQN, Double DQN trains a single Q-network to estimate action-values. To address the maximization bias, it leverages the idea of using two different estimators in the bootstrap target by using the target network, a time-delayed copy of the Q-network, as its second estimator. However, the target network remains correlated to the Q-network by virtue of being its time-delayed copy, and thus overestimation persists. Although Double DQN partially mitigates the maximization bias by integrating the ideas of double learning into DQN, adapting Double Q-learning to deep RL should reduce overestimation further. While a variety of approaches have been proposed to reduce overestimation [4, 8, 25, 38, 39, 43], including the use of two Q-functions [7, 11], none have adapted Double Q-learning to deep RL.

In this paper, we introduce Deep Double Q-learning (DDQL), the first adaptation of Double Q-learning to value-based deep RL that learns two action-value functions through reciprocal bootstrapping. DDQL outperforms Double DQN in aggregate across 57 Atari 2600 games [5, 21] and reduces overestimation. In addition, we study the impact of different design choices in adapting Double Q-learning to deep RL, including the network architecture, maintaining separate replay buffers for each Q-function, and the replay ratio.

2 Preliminaries

We formulate the task as a Markov decision process (MDP). A finite MDP is a tuple, $(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, \gamma)$, where $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ are finite sets of states, actions, and scalar rewards, respectively. P specifies the probability $P(s'|s, a)$ of transitioning from state s to s' after taking action a. At each transition, the agent receives a scalar reward $r \in \mathcal{R}$. The value $\gamma \in [0, 1)$ is the discount factor.

A policy π specifies the probability, $\pi(a|s)$, of taking action a in state s. The *expected discounted return* of following π after taking action a in state s is $q_\pi(s, a) := \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s, A_0 = a, A_{t\geq 1} \sim \pi\right]$, where R_t, S_t and A_t are random variables representing the reward, state, and action at time t, respectively. We call q_π the *action-value function* or Q-function. An optimal policy is denoted π^*. Its Q-function is denoted by q^*.

Q-learning [41, 42] maintains an estimated Q-function Q to learn q^* through experience. Given an arbitrary transition (s, a, r, s'), it performs the update $Q(s, a) \leftarrow Q(s, a) + \alpha\left[G - Q(s, a)\right]$, for step size $\alpha > 0$ and bootstrap target G. Q-learning is known to suffer from *overestimation* [32, 34]. Formally, for some policy π, given an estimate Q of q_π, we say that Q overestimates the action-value of state-action pair (s, a) when $Q(s, a) > q_\pi(s, a)$. Q-learning is a special case of off-policy Expected Sarsa [15, 31], which learns a Q-function for *target policy* π with the modified target $G' = r + \gamma \sum_{a'} \pi(a'|s') Q(s', a')$. When the target policy is the *greedy policy with respect to the current estimated action-value function*, Q, we recover Q-learning. Thus, if $\mathfrak{g}_Q$ denotes the greedy policy with respect to Q, then overestimation in Q-learning is present when $Q(s, a) > q_{\mathfrak{g}_Q}(s, a)$.

Q-learning's systematic overestimation stems from the maximization bias in its target computation when bootstrapping [34]. Suppose that for some s', all actions a' have the same true action-value $q_{\mathfrak{g}_Q}(s', a')$. As estimates are likely inexact, each $Q(s', a')$ can be written $Q(s', a') = q_{\mathfrak{g}_Q}(s', a') + \epsilon_{a'}$, for some error $\epsilon_{a'}$. If for some a', $\epsilon_{a'}$ is positive, then $Q(s', a') > q_{\mathfrak{g}_Q}(s', a')$ is an overestimate. Combined with maximization, the estimate of the maximum action-value is an overestimate. That is, $\max_{a'} Q(s', a') > \max_{a'} q_{\mathfrak{g}_Q}(s', a')$. Consequently, when $\max_{a'} Q(s', a')$ is bootstrapped, the update is biased towards an overestimated value. This biased update on $Q(s, a)$ then has downstream overestimation effects when predecessors to s are updated.

Double Q-learning [34] is designed to address the maximization bias in Q-learning. It learns two Q-functions, denoted Q_1 and Q_2. At each timestep, one of these Q-functions is selected uniformly at random to be updated, and the other is used to compute the target of the update. Formally, if Q_1 is

chosen to be updated, then the update is:

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha \left[r + \gamma\, Q_2\Big(s', \operatorname*{argmax}_{a'} Q_1(s',a')\Big) - Q_1(s,a) \right]. \tag{2}$$

Q_1 is updated to be closer to the target value, where $Q_{\text{sel}} = Q_1$ selects an action greedily for evaluation, and $Q_{\text{eval}} = Q_2$ evaluates the selected action. Q_2 is updated similarly, where Q_2 is used for action selection and Q_1 for evaluation in Eq. 2. By having separate value functions selecting and evaluating the action in the bootstrap target, Double Q-learning is able to ameliorate overestimation. Double Q-learning also converges under the same standard technical conditions as Q-learning [34].

The Deep Q-network (DQN) algorithm [22] casts Q-learning as a sequence of deep supervised regression problems. The agent stores its recent interactions in an experience replay buffer [18, 19], which serves as a dataset. Minibatches are sampled from this buffer to train a Q-function, represented as a neural network with parameters $\boldsymbol{\theta}$, through a regression loss on the semi-gradient temporal difference (TD) error [30]. In this paper, we use the squared error loss $(y - \hat{y})^2$.

To train the Q-network parameters $\boldsymbol{\theta}$ on transition (s, a, r, s'), the prediction $\hat{y}$ is the Q-network's output $Q(s, a; \boldsymbol{\theta})$ and the target follows G from Eq. 1, $y = r + \gamma Q(s', \text{argmax}_{a'} Q(s', a'; \boldsymbol{\theta}_{\text{sel}}); \boldsymbol{\theta}_{\text{eval}})$. DQN computes these targets using $\boldsymbol{\theta}_{\text{sel}} = \boldsymbol{\theta}_{\text{eval}} = \boldsymbol{\theta}^-$, where $\boldsymbol{\theta}^-$ are the parameters of the *target network*. The target network's parameters are *not* directly learned. They are periodically copied from the Q-network, $\boldsymbol{\theta}^- \leftarrow \boldsymbol{\theta}$, and held fixed for a *target network update interval* (TNU interval). By remaining fixed for an interval, target networks provide stable, stationary regression targets.

Like Q-learning, DQN also suffers from the maximization bias [35], since $\boldsymbol{\theta}_{\text{sel}} = \boldsymbol{\theta}_{\text{eval}}$. Double DQN [35] is mostly identical to DQN except that it uses $\boldsymbol{\theta}_{\text{sel}} = \boldsymbol{\theta}$. In other words, instead of having the target network evaluate its own greedy action, Double DQN has the target network evaluate the Q-network's greedy action, acting analogously to Q_2 in Eq. 2. By decoupling action selection and action evaluation in the bootstrap target, Double DQN partially mitigates the maximization bias.

3 Deep Double Q-learning

This section introduces Deep Double Q-learning (DDQL), our adaptation of Double Q-learning to value-based deep RL. DDQL differs from Double DQN in that it adapts Double Q-learning to deep RL, whereas Double DQN integrates double updating into DQN. Desirably, DDQL introduces no additional hyperparameters. Appendix B describes how DDQL can be implemented. Appendix A describes how DDQL relates to other methods in the literature, including clipped double Q-learning, which has been popularized by actor-critic methods like TD3 [11] and Soft Actor-Critic [13].

To adapt Double Q-learning to deep RL, we identify three defining features of Double Q-learning, which stem from the double estimator [34] upon which Double Q-learning is built. These features are (1) *target bootstrap decoupling*, (2) *double estimation with reciprocal bootstrapping*, and (3) *dataset partitioning*. Each defining feature can be viewed as an incremental step upon the previous towards decoupling action-selection and action-evaluation in the bootstrap target. The first feature, target bootstrap decoupling, simply refers to the use of different Q-functions Q_{sel} and Q_{eval} in Eq. 1, i.e., $Q_{\text{sel}} \neq Q_{\text{eval}}$. This feature is the only feature of the three integrated by Double DQN, with $Q_{\text{sel}} = Q(\cdot, \cdot; \boldsymbol{\theta})$ and $Q_{\text{eval}} = Q(\cdot, \cdot; \boldsymbol{\theta}^-)$. DDQL additionally integrates the latter two features.

3.1 Double estimation and reciprocal bootstrapping

Double estimation with reciprocal bootstrapping refers to the explicit training of two Q-functions where one is used to select actions in the bootstrap target and the other is used to evaluate the selected action. DDQL does this by maintaining two sets of Q-network parameters, $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, which are both trained. As in most DQN-style setups [6, 14, 22, 27, 35, 40], we also maintain target networks for stability, with parameters $\boldsymbol{\theta}_1^-$ and $\boldsymbol{\theta}_2^-$. Suppose that $\boldsymbol{\theta}_1$ is updated for some transition (s, a, r, s'). Then the DDQL target is:

$$y_1(s') = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \boldsymbol{\theta}_1^-); \boldsymbol{\theta}_2^-). \tag{3}$$

If parameters $\boldsymbol{\theta}_1$ are being updated, their corresponding target network parameters $\boldsymbol{\theta}_1^-$ are used to select a greedy action, and parameters $\boldsymbol{\theta}_2^-$ are used to evaluate this action to produce the target value.

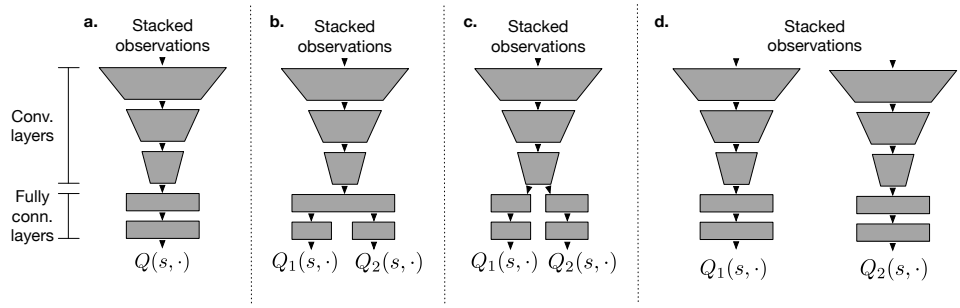


Figure 1: **DDQL architectures**. **a)** DQN/Double DQN, **b)** DDQL, **c)** DM-DDQL, and **d)** DN-DDQL.

We use target networks both for action-selection and action-evaluation in the bootstrap target to ensure fixed, stationary targets for the duration of the TNU interval, as is done in DQN. The presence of two Q-networks and two target networks creates several alternative ways to use two different networks in the bootstrap target. These alternatives are discussed in Appendix D.

For DDQL, we use the squared error loss, although in principle any loss that is a function of the TD error can be used. Given a minibatch $\mathcal{B} = \{(s_i, a_i, r_i, s'_i)\}_{i=1}^{N}$ of N transitions, we have the loss

$$\mathcal{L}_1(\mathcal{B}) = \frac{1}{N} \sum_{i=1}^{N} \big(y_1(s'_i) - Q(s_i, a_i; \boldsymbol{\theta}_1)\big)^2. \tag{4}$$

When updating $\boldsymbol{\theta}_2$, $y_2(s')$ and $\mathcal{L}_2$ are defined analogously to Equations 3 and 4, respectively.

In tabular Double Q-learning, only one of the two Q-functions is updated at a time using the online transition. When sampling from a replay buffer, which is relatively offline and stationary, we can instead sample two separate minibatches from the buffer and update each Q-function simultaneously in a single gradient update by minimizing the loss $\mathcal{L}_{\text{DDQL}} = \mathcal{L}_1 + \mathcal{L}_2$.

3.2 Dataset partitioning

In Double Q-learning, each experience transition is used to update only one of the two Q-functions, i.e., the sets of transitions used to train each Q-function are non-overlapping. We refer to this feature of Double Q-learning as *dataset partitioning*. Dataset partitioning further decouples the two Q-functions by separating their training data. To approximate dataset partitioning, we sample distinct minibatches from a single buffer to train each Q-function. Section 5.2 explores a strict form of dataset partitioning that uses separate replay buffers to train each Q-function.

3.3 Network architecture

When bridging the gap from tabular to deep RL, we transition from two lookup-table Q-functions to neural networks. DDQL uses a single network with two output heads, each representing a Q-function. Doing so allows our two Q-functions to benefit from a shared representation and potentially benefit from auxiliary task effects [20] while still maintaining separate parameters.

Table 1: Different adaptations of Double Q-learning to deep RL in terms of its defining features.

	Double Q-learning Defining Feature	Double DQN	DDQL & DM-DDQL	DN-DDQL
(1)	Target bootstrap decoupling	✓	✓	✓
(2)	Double estimation	-	✓	✓
(2a)	Exclusive parameters	-	✗	✓
(3)	Dataset partitioning	-	✓	✓
(3a)	Separate buffers	-	✗	✗

We explore two additional architectures, which we call double-MLP DDQL (DM-DDQL) and double-network DDQL (DN-DDQL). DM-DDQL uses a single shared trunk of convolution layers, branching into two two-layer multi-layer perceptrons (MLPs), each representing a Q-function. In DN-DDQL, the two Q-functions are parameterized by separate networks. In all DDQL variants, $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ refer to the parameters of the respective Q-functions, even when parameters are shared, as in DDQL and

Algorithm 1: Deep Double Q-learning

Input: target network update interval C, gradient update frequency k

Initialize replay buffer $\mathcal{D}$, Q-network parameters $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, *reset* $\leftarrow$ True;
for $t = 0$ *to* T **do**
  if *reset* $==$ *True* **then**
    $s_t \leftarrow$ sample start state;
  if $t \pmod{kC} == 0$ **then**
    $\boldsymbol{\theta}_1^- \leftarrow \boldsymbol{\theta}_1, \boldsymbol{\theta}_2^- \leftarrow \boldsymbol{\theta}_2$;
  Select and execute action a_t and observe s_{t+1}, r_{t+1};
  Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$;
  reset $\leftarrow (s_{t+1}$ is terminal$)$;
  if $t \pmod{k} == 0$ **then**
    Sample distinct minibatches $\mathcal{B}_1, \mathcal{B}_2$ from $\mathcal{D}$;
    Update $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$ through minibatch gradient descent on $\mathcal{L}_{\text{DDQL}}$;

DM-DDQL. Fig. 1 depicts the three DDQL architectures. Algorithm 1 provides the pseudocode for DDQL. Table 1 highlights how different algorithms and architectures incorporate the defining ideas of Double Q-learning.

3.4 Key algorithm details

We **initialize** the independent parts of the Q-functions identically. That is, in DDQL, two output heads share the same initialization; in DM-DDQL, the two MLPs are initialized identically; and in DN-DDQL, the networks are initialized identically. The reason for this is twofold. First, in tabular Double Q-learning, the Q-functions are typically both initialized to zeros. Second, there is no apparent benefit to having different initial action-value predictions, as the two Q-functions will, by design, deviate as they train on different experiences.

The **replay ratio** is the number of gradient updates per environment transition [10]. Our DDQL agents use a replay ratio of $1/8$, as opposed to DQN and Double DQN's $1/4$. Specifically, both Q-functions are updated *simultaneously* every eight timesteps. Though this replay ratio induces half as many parameter updates as Double DQN, it ensures that the total number of updates to individual Q-functions matches Q-learning, as two Q-functions are updated simultaneously. This ratio is also consistent with Double Q-learning, which on average updates each Q-function half as often as a Q-learner. The use of simultaneous updates is relevant for DDQL to mitigate gradient oscillation. Section 5.3 studies the role of the replay ratio in DDQL.

The **target network update interval** (TNU interval) is another important hyperparameter. While typically described in terms of environment timesteps, this interval should be a function of the number of parameter updates, as noted by van Hasselt et al. [37]. DDQL uses a TNU interval of 7,500 updates, inherited from the tuned version of Double DQN [35], which uses a 3x longer interval than DQN. As DDQL performs gradient updates half as often as Double DQN, DDQL refreshes its target network half as often in terms of environment timesteps. Despite doubling the batch size, halving the update frequency results in minimal slowdowns. DDQL and DN-DDQL are respectively 11% and 25% slower than Double DQN.

3.5 Double DQN versus Deep Double Q-learning

The core similarity between Double DQN and DDQL is that they both use target bootstrap decoupling. That is, they both use one Q-function to select the highest-valued action in the bootstrap target and another Q-function to evaluate the selected action. This use of two Q-functions in DQN's bootstrap target gives the name Double Deep Q-networks. Double DQN borrows one particular idea of Double Q-learning and applies it to a Q-learning style algorithm, DQN.

By contrast, DDQL takes the Double Q-learning algorithm itself and adapts it to deep RL. That is, DDQL is Double Q-learning using replay buffers and target networks. Double DQN only performs gradient updates on a single Q-function with a single loss function, and uses the target network as

the second Q-function in the bootstrap target. Importantly, DDQL trains two Q-functions, each with its own loss, and uses each Q-function in the target computation of the other Q-function (reciprocal bootstrapping). It also trains the two Q-functions on different transitions (dataset partitioning). Training two Q-functions introduces several design choices, including the choice of network architecture and the minibatch sampling strategy. We integrate the three defining features of Double Q-learning into our algorithm, and accordingly name it Deep Double Q-learning.

4 Evaluating Deep Double Q-learning

This section summarizes our empirical methodology, which is comprehensively outlined in Appendix C. We then show that DDQL reduces overestimation and outperforms Double DQN.

4.1 Experimental setup

Evaluation. We conduct experiments in the Arcade Learning Environment [5, 21], the platform on which Double DQN was originally evaluated.[1] We evaluate Double DQN, DDQL, DM-DDQL, and DN-DDQL across the Atari-57 set of environments [35]. To study specific aspects of DDQL, we perform additional ablations on a set of games we call *Ablation-11*, which combine the Atari-5 games [3]—a representative subset of Atari-57—and six additional games that have been used to study overestimation in DQN [4, 35]. We use the environment settings proposed by Machado et al. [21]. That is, we use sticky actions, the full action set, and game-over termination.

Double DQN. All algorithms[2] are built upon the PFRL library [12]. Our Double DQN uses the same architecture and hyperparameters as the original paper's tuned Double DQN [35]. Reflecting recent advances [2, 23], we use the Adam optimizer [16] with the squared error loss instead of the original RMSprop with the Huber loss. Our settings for Adam follow those of Hessel et al. [14], following standard practice in the DQN literature [1, 2, 9, 23].

DDQL. Following van Hasselt et al. [35]'s approach of maintaining the same hyperparameters as the baseline, all DDQL variants share Double DQN's hyperparameters, unless stated otherwise in Section 3. DDQL's Q-functions share Double DQN's hidden layers and branch into two architecturally identical output heads that match Double DQN's output layer. DM-DDQL's Q-functions share Double DQN's convolutional layers with two architecturally identical two-layer MLPs that match Double DQN's final two layers. Each of DN-DDQL's two networks matches Double DQN's architecture. The behavior policy for the DDQL variants during training is an ϵ-greedy policy with respect to the average of the two Q-functions: $\frac{1}{2}Q(s,a;\boldsymbol{\theta}_1) + \frac{1}{2}Q(s,a;\boldsymbol{\theta}_2)$, following Double DQN's ϵ-decay schedule. Each algorithm is run for five seeds for each evaluated game unless stated otherwise.

Measuring Overestimation. Double DQN learns a Q-function that estimates the expected discounted return of the greedy policy with respect to that very Q-function. Thus, overestimation occurs when the Q-value estimate exceeds the expected return of its greedy policy. Similar to van Hasselt et al. [35], we measure overestimation during periodic evaluation phases. For all state-action pairs of all completed episodes in this evaluation phase, we compute the average difference between the action-value predictions and their corresponding achieved returns to compute overestimation. For DDQL, the action-value prediction is the average of its two Q-values. Appendix C.2 outlines our overestimation computation, including how we handle episodes that are truncated due to time limits.

4.2 Results: overestimation and performance

Fig. 2 depicts the average final overestimation of Double DQN and DDQL across games (per-game overestimation curves are in Fig. 17 in Appendix F). While the degree of overestimation is game-dependent, as the scale of returns can vary between games, Double DQN overestimates in all games other than Montezuma's Revenge. This overestimation suggests that its target bootstrap decoupling with the Q-network and target network is insufficient for eliminating overestimation. This result is to be expected: because the target network is a time-delayed copy of the Q-network, the two networks remain heavily correlated. DDQL also overestimates, but it reduces overestimation compared to Double DQN in 42 out of 57 environments.

[1] Each run takes 2 to 3.5 GPU days on average. We estimate our experiments require between 8 and 13 GPU years.
[2] github.com/prabhatnagarajan/ddql.

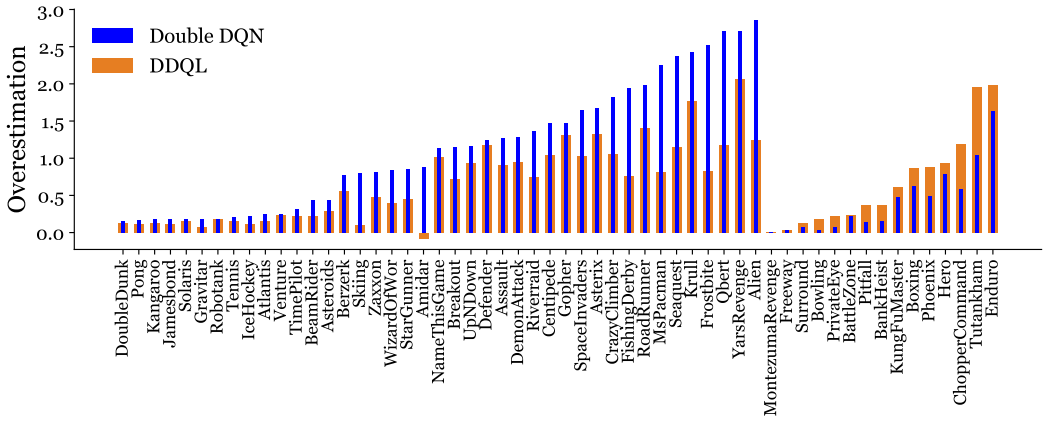


Figure 2: **Per-game DDQL overestimation.** Final overestimations averaged across five seeds of Double DQN and DDQL (Video Pinball is omitted for visibility).

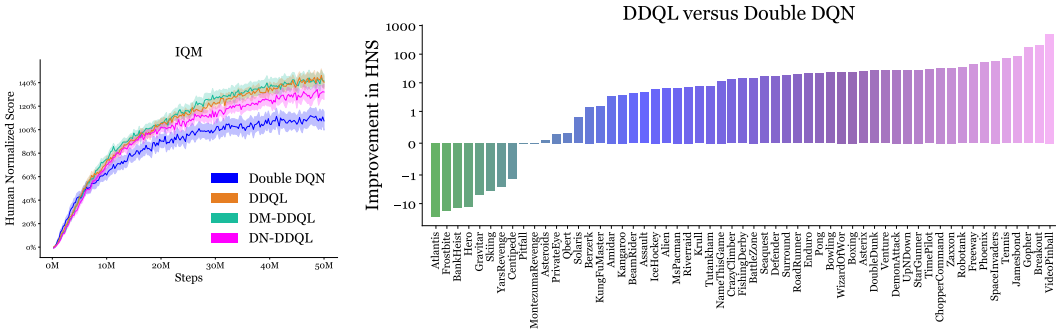


Figure 3: **Aggregate learning curves and per-game performance improvements.** (*left*) Interquartile mean of the HNS throughout training. The shaded region represents a 95% stratified bootstrap confidence interval [2]. (*right*) Per-game improvement in HNS of DDQL over Double DQN, calculated as the average area under the curve across five seeds. The y-axis is on a log-scale.

Fig. 3 depicts DDQL's performance improvements over Double DQN. Fig. 3 (*left*) plots the standard metric of interquartile mean (IQM) [2] of the human-normalized score (HNS) [35] across the 57 Atari games. In Appendix F, we show the per-game learning curves in Fig. 19 and the per-game scores in Table 7. As a percentage of Double DQN's IQM of the HNS, DDQL performs 31% better than Double DQN. Fig. 3 (*right*) shows the per-game improvements, in terms of average area under the curve (AUC) of the HNS of DDQL over Double DQN. DDQL outperforms Double DQN in 47 out of 57 games. The log-scale y-axis shows that performance decreases are often small, while the gains can be quite high.

5 Understanding Deep Double Q-learning

In this section, we study the impact of different design choices when adapting Double Q-learning to deep RL. In particular, we study the impact of maintaining separate parameters, of using two separate buffers for dataset partitioning, and of varying the replay ratio.

5.1 Independent parameters

By evaluating DM-DDQL and DN-DDQL, we can examine the impact of parameter sharing on overestimation. DN-DDQL decouples the two Q-functions further by representing them using separate neural networks. The result of this decoupling is large reductions in overestimation, to the point of underestimation, a phenomenon known to occur in Double Q-learning [34]. This underestimation is shown in Fig. 4, which shows the per-game overestimations of all the DDQL

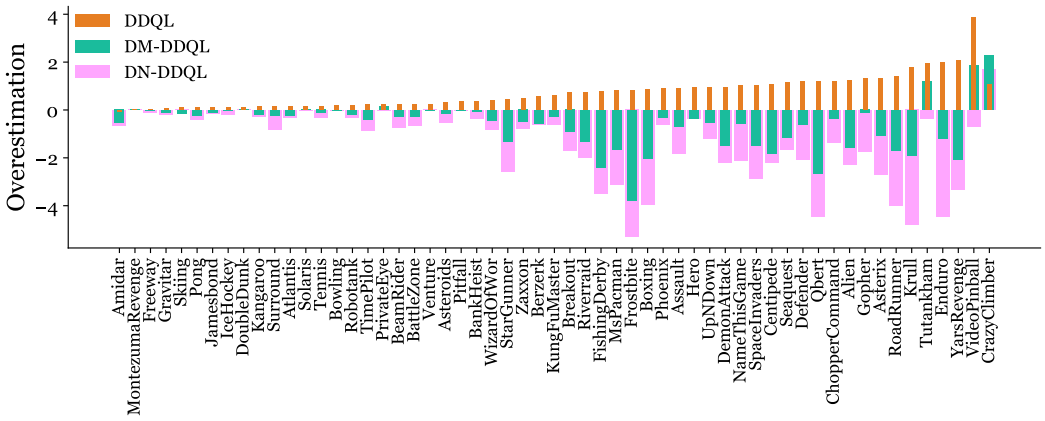


Figure 4: **Per-game overestimation for the DDQL variants.** Final overestimations averaged across five seeds per algorithm on each of the 57 games. DDQL generally overestimates the most, followed by DM-DDQL, then DN-DDQL. The latter two often underestimate.

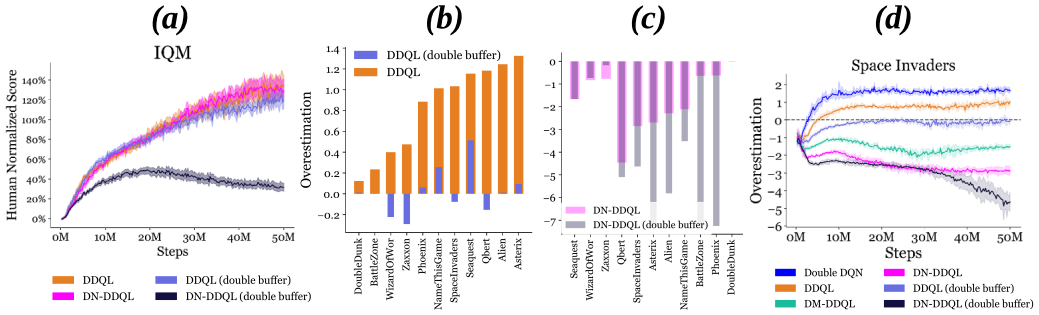


Figure 5: **Dataset partitioning experiments.** (*a*) The IQM of the HNS on the Ablation-11 games of DDQL and its double-buffer counterparts. Shaded region: 95% stratified bootstrap confidence interval. (*b* & *c*) Final overestimation, averaged over five seeds of DDQL and DN-DDQL, respectively, and their double-buffer counterparts. (*d*) Overestimation throughout training for different algorithms averaged across five seeds on Space Invaders. Translucent lines depict individual seeds.

variants. The large gap in overestimation between DDQL and DN-DDQL suggests that the sharing of the hidden layers in the network of DDQL contributes substantially to overestimation. This finding is reinforced by DM-DDQL, which represents an intermediate architecture between DDQL and DN-DDQL. Like DDQL, DM-DDQL shares parameters, but fewer. Correspondingly, DM-DDQL exhibits an intermediate amount of overestimation between DDQL and DN-DDQL. These results suggest that structural independence in the Q-functions de-correlates the action-value functions.

Fig. 3 (*left*) depicts DM-DDQL and DN-DDQL's performance improvements over Double DQN in aggregate across the 57 games (per-game improvements in Fig. 18 in Appendix F). As a percentage of Double DQN's IQM of the HNS, DM-DDQL and DN-DDQL perform 32% and 23% better than Double DQN, respectively. Though DN-DDQL closely embodies the original Double Q-learning algorithm, using separate networks yields lower performance than partial parameter sharing. DM-DDQL, by contrast, is just as effective as DDQL.

Sharing hidden layers, as in DDQL and DM-DDQL, strikes a middle ground between reducing overestimation and retaining good performance. That said, all three architectures perform well, demonstrating that the DDQL algorithm is compatible with different architectures and effective under different degrees of parameter-sharing. If overestimation is a major concern, DM-DDQL may be the best option. In general we recommend DDQL due to its simplicity and generality.

5.2 Partitioned datasets

The third defining feature of Double Q-learning is dataset partitioning, where each experience transition is used to train only a single Q-function. DDQL maintains a single buffer and samples distinct minibatches to update each Q-function. Sampling distinct minibatches, however, does not ensure strict dataset partitioning. Experiences are sampled multiple times from the buffer and so transitions are inevitably used to train both Q-functions.

To study the impact of strict dataset partitioning, we modify DDQL to maintain two separate buffers with half the capacity of a standard replay buffer, leaving the algorithm otherwise unchanged. Transitions are added uniformly at random to one of the two buffers. Each buffer generates minibatches exclusively for one of the two Q-functions to ensure that transitions are used to train only one Q-function. By separating the data used to train each Q-function, we further decouple the two Q-functions. Thus, we hypothesize reduced overestimation in this setting.

We compare DDQL to its double-buffer counterparts on the Ablation-11 environments in Fig. 5 (*a*), reporting the IQM of the HNS. DDQL is slightly better than its double-buffer counterpart. Using separate buffers in DN-DDQL substantially harms performance. Training on different buffers may require better generalization from the Q-networks, a requirement that may be softer when parameters are shared. Fig. 10 and Fig. 11 in Appendix F depict the per-game learning curves for DDQL and DN-DDQL respectively.

Fig. 5 (*b*) and (*c*) show the average final overestimation of DDQL and DN-DDQL and their double-buffer counterparts (per-game curves for DDQL shown in Fig. 12 in Appendix F). As hypothesized, the use of separate buffers reduces overestimation substantially for both DDQL variants. DDQL's overestimation goes closer to 0, providing further evidence that separating the data used to train each estimator, as is done in Double Q-learning, reduces overestimation. Applied to DN-DDQL, which already separates parameters, it reduces overestimation even further, albeit at the cost of poor performance.

The impact of integrating Double Q-learning's defining features on overestimation is demonstrated in Fig. 5 (*d*), which shows individual overestimation for different algorithms in the game Space Invaders. Double DQN, which simply decouples estimators in the bootstrap target using its Q-network and target network, has the most overestimation. DDQL, DM-DDQL, and DN-DDQL, the methods which integrate double estimation with reciprocal bootstrapping and a weaker form of dataset partitioning, reduce overestimation. The gaps between DDQL, DM-DDQL, and DN-DDQL demonstrate that the lower the degree of parameter-sharing, the greater the reduction in overestimation. Lastly, the double-buffer variants show that strict dataset partitioning, where we use two buffers to explicitly separate the data used to train each Q-function, reduces overestimation further yet.

5.3 Replay ratio

DDQL uses a replay ratio of $1/8$ instead of the $1/4$ used by DQN and Double DQN on Atari 2600 games. To study the importance of this difference, we evaluate DDQL and DN-DDQL with double the replay ratio at $1/4$ on the Ablation-11 environments. This change ensures that each Q-function receives as many updates as the single Q-function in Double DQN. All other algorithm details, hyperparameters, and experiment configurations are unchanged.

Fig. 6 (*left*) shows the IQM of the human-normalized score across the Ablation-11 games. Doubling the replay ratio reduces performance in both DDQL variants. The lower replay ratio performs better.

To understand why this occurs, consider that each parameter update can change the behavior policy, usually substantially [10, 28]. A lower replay ratio implies less frequent updates, ensuring that the buffer gets more transitions per policy. Thus, transitions are re-sampled fewer times, reducing potential overfitting. Moreover, the buffer has data from fewer and more recent policies in terms of parameter updates, slowing the change of its data distribution. In short, lower replay ratios create a more stationary learning problem. DDQL benefits from this stationarity, and is able to outperform Double DQN with half as many parameter updates.

Fig. 6 (*right*) contrasts Double DQN's overestimation to that of DDQL with double the replay ratio. As DDQL with twice as many updates still reduces overestimation over Double DQN, the argument that DDQL's reduced overestimation is solely a consequence of having fewer updates—for example, by slowing the compounding of overestimation errors—is eliminated. Additionally, we ran three seeds

of Double DQN on all 57 environments with a replay ratio of $1/8$, shown in Fig. 16 in Appendix E. In short, we find that the lower replay ratio does not benefit Double DQN's performance, indicating that DDQL's improved performance is not solely attributable to having a lower replay ratio.

5.4 On initializations and targets

Appendix E contains additional results and discussion on other design choices that contribute to performance. Appendix E.2 investigates the impact of initializing the two Q-functions identically. We find that in DN-DDQL with shorter TNU intervals, identical initializations can be critical for good performance. Appendix E.3 investigates the benefits of using a target network for action-selection in the bootstrap target rather than a Q-network, which is continually changing. We find that using stationary targets, that is, using two target networks as opposed to a target network and a Q-network, improves performance in some environments.

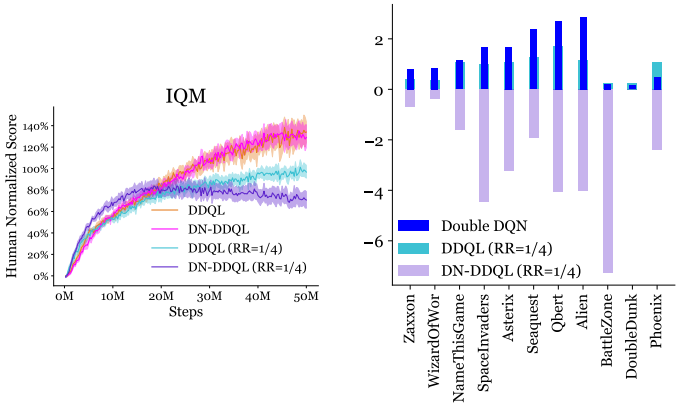


Figure 6: **Replay Ratio Experiments.** (*left*) IQM of the HNS across five seeds on the Ablation-11 games. Shaded region: 95% stratified bootstrap confidence interval. Increasing the replay ratio harms DDQL's performance. (*right*) Compares the final overestimation of DDQL with a replay ratio of $1/4$ to Double DQN, averaged over five seeds, on the Ablation-11 games.

6 Discussion and conclusion

In this paper, we introduced Deep Double Q-learning (DDQL), an adaptation of Double Q-learning featuring double estimation with reciprocal bootstrapping and dataset partitioning. DDQL reduces overestimation and improves performance over Double DQN, without additional hyperparameters. With DDQL, we show that Double Q-learning can be adapted into a high-performing algorithm in the value-based deep RL setting in a manner that closely resembles the original algorithm. We studied multiple aspects of DDQL, including three architectures, two kinds of dataset partitioning, and the importance of the replay ratio.

This paper has two main limitations. The first is that DDQL is restricted to discrete-action, value-based RL. Though in principle DDQL can be extended to continuous control actor-critic methods such as Soft Actor-Critic [13] or TD3 [11], the integration is not seamless. In Appendix A, we discuss how the Clipped Double Q-learning used in actor-critic methods differs substantially from Double Q-learning and DDQL. The second limitation is that it is yet unclear how DDQL integrates with other deep RL advances such as those in Rainbow [14].

Though a general study on the relationship between overestimation and performance is outside the scope of this paper, our results shed some light on this topic. Broadly, our results are consistent with other works that have shown that the impact of overestimation can be beneficial or harmful [17, 38], depending on the situation. We observed that the overestimation reduction accompanying Double Q-learning's defining features does not necessarily translate to performance improvements. Deep learning has several interacting parts. Thus, mechanisms to decouple the Q-functions must be balanced against their impact on the training process. Overestimation is impacted by the architecture, training data, and the replay ratio, yet these all impact learning dynamics differently.

In fact, this paper can largely be viewed as outlining how to stabilize training in spite of these decoupling strategies. Techniques to improve optimization and representation learning, such as identical initialization, simultaneous updates, and shared hidden layers, benefit learning. Moreover, stabilizing mechanisms to induce stationarity, including longer target network update intervals, lower replay ratios, and fixed targets, all contribute to stable learning and help realize better performance.

Acknowledgments and Disclosure of Funding

We thank the anonymous reviewers for their feedback, which improved the paper. We thank Andrew Patterson for useful discussions, Brett Daley for reviewing parts of the code, and Abhishek Naik for making helpful suggestions. We thank Anna Hakhverdyan and Alex Lewandowski for providing feedback on drafts of the paper. The authors especially thank Khurram Javed, Emma Jordan, Aditya Ganeshan, and Jens Tuyls for extensive discussions and feedback on the paper.

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Canada CIFAR AI Chair Program. Prabhat Nagarajan was supported by the Alberta Innovates Graduate Student Scholarship during the bulk of this research. Computational resources were provided in part by the Digital Research Alliance of Canada.

References

[1] R. Agarwal, D. Schuurmans, and M. Norouzi. An Optimistic Perspective on Offline Reinforcement Learning. In *International Conference on Machine Learning*, 2020.

[2] R. Agarwal, M. Schwarzer, P. S. Castro, A. C. Courville, and M. G. Bellemare. Deep Reinforcement Learning at the Edge of the Statistical Precipice. *Neural Information Processing Systems*, 2021.

[3] M. Aitchison, P. Sweetser, and M. Hutter. Atari-5: Distilling the Arcade Learning Environment down to Five Games. In *International Conference on Machine Learning*, pages 421–438, 2023.

[4] O. Anschel, N. Baram, and N. Shimkin. Averaged-DQN: Variance Reduction and Stabilization for Deep Reinforcement Learning. In *International Conference on Machine Learning*, 2017.

[5] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 2013.

[6] M. G. Bellemare, W. Dabney, and R. Munos. A Distributional Perspective on Reinforcement Learning. In *International Conference on Machine Learning*, 2017.

[7] G. Chen. Decorrelated Double Q-learning. *arXiv preprint arXiv:2006.06956*, 2020.

[8] X. Chen, C. Wang, Z. Zhou, and K. Ross. Randomized Ensembled Double Q-Learning: Learning Fast Without a Model. In *International Conference on Learning Representations*, 2021.

[9] J. Farebrother, J. Orbay, Q. Vuong, A. Ali Taiga, Y. Chebotar, T. Xiao, A. Irpan, S. Levine, P. S. Castro, A. Faust, A. Kumar, and R. Agarwal. Stop Regressing: Training Value Functions via Classification for Scalable Deep RL. In *International Conference on Machine Learning*, 2024.

[10] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney. Revisiting Fundamentals of Experience Replay. In *International Conference on Machine Learning*, 2020.

[11] S. Fujimoto, H. van Hoof, and D. Meger. Addressing Function Approximation Error in Actor-Critic Methods. In *International Conference on Machine Learning*, pages 1587–1596, 2018.

[12] Y. Fujita, P. Nagarajan, T. Kataoka, and T. Ishikawa. ChainerRL: A Deep Reinforcement Learning Library. *Journal of Machine Learning Research*, 2021.

[13] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, et al. Soft Actor-Critic Algorithms and Applications. *arXiv preprint arXiv:1812.05905*, 2018.

[14] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. In *AAAI Conference on Artificial Intelligence*, 2018.

[15] G. H. John. When the Best Move Isn't Optimal: Q-learning with Exploration. In *AAAI Conference on Artificial Intelligence*, 1994.

[16] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[17] Q. Lan, Y. Pan, A. Fyshe, and M. White. Maxmin Q-learning: Controlling the Estimation Bias of Q-learning. In *International Conference on Learning Representations*, 2020.

[18] L.-J. Lin. Programming robots using reinforcement learning and teaching. In *AAAI Conference on Artificial Intelligence*, 1991.

[19] L.-J. Lin. *Reinforcement Learning for Robots Using Neural Networks*. Carnegie Mellon University, 1992.

[20] C. Lyle, M. Rowland, G. Ostrovski, and W. Dabney. On the Effect of Auxiliary Tasks on Representation Dynamics. In *International Conference on Artificial Intelligence and Statistics*, 2021.

[21] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling. Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *Journal of Artificial Intelligence Research*, 2018.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[23] J. S. Obando-Ceron and P. S. Castro. Revisiting Rainbow: Promoting more Insightful and Inclusive Deep Reinforcement Learning Research. In *International Conference on Machine Learning*, 2021.

[24] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy. Deep Exploration via Bootstrapped DQN. *Neural Information Processing Systems*, 2016.

[25] O. Peer, C. Tessler, N. Merlis, and R. Meir. Ensemble Bootstrapping for Q-Learning. In *International Conference on Machine Learning*, 2021.

[26] J. Quan and G. Ostrovski. DQN Zoo: Reference implementations of DQN-based agents, 2020. URL `http://github.com/deepmind/dqn_zoo`.

[27] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized Experience Replay. In *International Conference on Learning Representations*, 2016.

[28] T. Schaul, A. Barreto, J. Quan, and G. Ostrovski. The Phenomenon of Policy Churn. *Neural Information Processing Systems*, 2022.

[29] J. E. Smith and R. L. Winkler. The Optimizer's Curse: Skepticism and Postdecision Surprise in Decision Analysis. *Management Science*, 2006.

[30] R. S. Sutton. Learning to Predict by the Methods of Temporal Differences. *Machine learning*, 1988.

[31] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[32] S. Thrun and A. Schwartz. Issues in Using Function Approximation for Reinforcement Learning. In *Connectionist Models Summer School*, 1993.

[33] M. Towers, A. Kwiatkowski, J. Terry, J. U. Balis, G. De Cola, T. Deleu, M. Goulão, A. Kallinteris, M. Krimmel, A. KG, et al. Gymnasium: A Standard Interface for Reinforcement Learning Environments. *arXiv preprint arXiv:2407.17032*, 2024.

[34] H. van Hasselt. Double Q-learning. *Neural Information Processing Systems*, 2010.

[35] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. In *AAAI Conference on Artificial Intelligence*, 2016.

[36] H. van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil. Deep Reinforcement Learning and the Deadly Triad. *arXiv preprint arXiv:1812.02648*, 2018.

[37] H. van Hasselt, M. Hessel, and J. Aslanides. When to use parametric models in reinforcement learning? In *Neural Information Processing Systems*, 2019.

[38] M. Waltz and O. Okhrin. Addressing maximization bias in reinforcement learning with two-sample testing. *Artificial Intelligence*, 2024.

[39] X. Wang and A. Vinel. Cross Learning in Deep Q-Networks. *arXiv preprint arXiv:2009.13780*, 2020.

[40] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. Dueling Network Architectures for Deep Reinforcement Learning. In *International Conference on Machine Learning*, 2016.

[41] C. J. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, Cambridge, UK, 1989.

[42] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 1992.

[43] R. Zhu and M. Rigotti. Self-correcting Q-learning. In *AAAI Conference on Artificial Intelligence*, 2021.

A Related work

As we study adaptations of Double Q-learning in this paper, measuring DDQL's impact on overestimation has naturally been a central point. Though DDQL indeed reduces overestimation, our objective was more to adapt Double Q-learning than it was to find the best algorithm that reduces overestimation. Nonetheless, we should acknowledge the broader literature in value-based deep RL that aims to address or manage overestimation. Averaged-DQN [4] reduces overestimation by performing Q-learning updates with the Q-learning target averaged over multiple historical target networks. Self-correcting DQN [43] aims to balance overestimation and underestimation, but does not implement double estimation. Like Double DQN, it too leverages the target network instead of a second independent Q-function.

There are value-based ensemble methods that aim to manage overestimation. Maxmin DQN [17] aims to allow and control for different amounts of overestimation or underestimation. Ensemble Bootstrapped Q-Learning (EBQL) [25] and cross Q-learning [39] both generalize Double Q-learning to ensembles. Both papers only investigate multi-head architectures, and only investigate ensembles of at least five members. Wang and Vinel [39]'s study is also restricted to the small environments of CartPole and Lunar Lander [33]. While EBQL is evaluated on Atari 2600 games, its evaluation is restricted to 11 random environments as opposed to the 57 games, and overestimation is not measured in these environments. Waltz and Okhrin [38] also work in the ensemble setting, building on Bootstrapped DQN [24], with the aim of balancing overestimation and underestimation.

Though our focus in this paper is on adapting Double Q-learning for value-based deep RL, it is important to note an adaptation commonly used in deep off-policy actor-critic settings known as Clipped Double Q-learning (CDQ) [11]. CDQ adapts some ideas of Double Q-learning and has been popularized by algorithms like TD3 [11] and Soft Actor-Critic [13]. CDQ is distinct from Double Q-learning and DDQL in several ways. First, CDQ computes a *single* target Q-value which is used to update both Q-functions, rather than having each Q-function update towards different targets. Second, this single target is computed by bootstrapping the minimum estimated Q-value across two Q-functions. Third, each Q-function evaluates the *same* target action from a single policy, rather than having two different policies select the actions for each respective Q-function. Fourth, these Q-functions are trained on the same minibatches, i.e., they are trained on the same data unlike Double Q-learning or DDQL. In the tabular setting, this is akin to updating both Q-functions with the same experience and same target value, which fundamentally departs from Double Q-learning. There are also other approaches for reducing overestimation that have been applied to the off-policy actor-critic setting. Decorrelated Double Q-learning [7] introduces a regularization term to de-correlate the two Q-functions. Randomized Ensembled Double Q-Learning [8] maintains an ensemble of Q-functions and minimizes across them to compute targets. These works all aim to mitigate overestimation, but they neither study variants that resemble the original Double Q-learning algorithm, nor do they study the pure value-based setting.

B DDQL implementation

When computing gradients for DDQL on two separate minibatches, one option is to use two separate forward and backward passes to compute the loss and gradients for each Q-function. Another option is to compute these simultaneously by masking minibatches. Rather than sampling two separate minibatches of size 32, we can first sample a larger minibatch of size 64. On this larger batch, we can compute the predictions and targets for each Q-function, giving us 128 total prediction-target pairs. Only 32 prediction-target pairs are relevant for Q_1 and a different 32 pairs are relevant for Q_2. We can simply mask the irrelevant pairs before computing the loss. Depending on how element-wise losses are aggregated, some scaling may be needed. For example, if using the mean squared error loss, the batch size is doubled with our masking, so the computed loss should be adjusted.

C Experimental details

In this appendix, we provide details regarding environments, training, evaluation, and overestimation measurement.

Table 2: Environment details, following Machado et al. [21].

Detail	Setting	Description
sticky action probability	0.25	Probability by which the simulator ignores the agent's selected action and repeats the action executed in the previous frame.
termination criterion	end-of-game	How episode termination is signaled during training (either end-of-game or loss-of-life).
action space	full	The full action space has 18 actions. The minimal action space uses the game-specific minimum required actions.

C.1 Environments, training, and evaluation

Environments In this paper, we use the environment settings proposed by Machado et al. [21]. All agents use the full action set of 18 actions for all games. We use "sticky" actions, where the simulator repeats the action executed at the previous frame with probability 0.25, regardless of the agent's selected action. We use the actual termination of the game as the termination signal to the learning agent. These details are outlined in Table 2.

The Ablation-11 set of environments contains the Atari-5 subset, a subset of Atari-57 games that are said to be predictive of Atari-57 performance in terms of median human-normalized score: BATTLEZONE, DOUBLEDUNK, NAMETHISGAME, PHOENIX, and QBERT [3]. It also contains six different Atari 2600 games that are used in the literature to study overestimation in DQN [4, 35]: ALIEN, ASTERIX, SEAQUEST, SPACEINVADERS, WIZARDOFWOR, and ZAXXON.

Training The agent is trained for 50M timesteps. When training, the rewards are clipped to be between -1 and 1. The agents deploy an ϵ-greedy policy during training, beginning at $\epsilon = 1.0$ and linearly annealing to 0.01 over 1M timesteps. The agent's replay buffer has a capacity of 1M, and network updates are only performed after the agent has completed 50k timesteps. The network architecture used is identical to the architecture used by van Hasselt et al. [35], i.e., the DQN architecture [22] with a single shared bias for all actions in the final layer as opposed to per-action biases. Agents use the Adam optimizer with the MSE loss using a step size of $6.25\text{e}{-5}$, $\epsilon = 1.5\text{e}{-4}$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$ [1, 9, 14, 23]. Target networks are copied from the main Q-networks every 7,500 gradient updates, which corresponds to 30k timesteps for Double DQN and 60k timesteps for DDQL. The preprocessing scheme follows that of Mnih et al. [22]. Table 3 outlines our hyperparameters and training details.

Evaluation Agents are evaluated for 125k timesteps after every 250k timesteps of training. Evaluation episodes are truncated at 30 minutes, or 27k timesteps. Agents deploy an ϵ-greedy policy during evaluation with $\epsilon = 0.001$. For DDQL agents, the ϵ-greedy policy is with respect to the average action-values of the two Q-functions. Table 4 outlines these details.

Human-normalized scores The human-normalized score [22] of an agent can be computed as

$$\text{score}_{\text{hns}} = \frac{\text{score}_{\text{agent}} - \text{score}_{\text{random}}}{\text{score}_{\text{human}} - \text{score}_{\text{random}}} \tag{5}$$

Our human scores and random scores used to compute Eq. 5 are taken from DQN Zoo [26]. Suppose we have N environments and M seeds per environment. Let $X_{j,k}$ refer to the human-normalized score of the agent on the jth environment and kth seed. To compute the interquartile mean of the human-normalized score, we follow Agarwal et al. [2]:

$$\text{IQM HNS} = \text{IQM}\left(\{X_{j,k} : j = 1, \ldots, N \quad \text{and} \quad k = 1, \ldots, M\}\right). \tag{6}$$

C.2 Measuring overestimation

As discussed in Section 4.1, we measure overestimation by comparing achieved returns of a greedy policy to the Q-value predictions of our networks. Recall the greedy policy with respect to Q, $\mathfrak{g}_Q$.

[3](30 mins = 30 minutes * 60 seconds/minute * 60 frames/second / 4 frames per timestep = 27,000 timesteps)

Table 3: Hyperparameters and training details.

Hyperparameter	Value	Description
minibatch size	32	Number of transitions used per update per Q-function.
replay memory size	1,000,000	Number of transitions stored in the replay buffer.
agent history length	4	Number of previous frames stacked in state representation.
target network update frequency	7,500	Frequency (in terms of parameter updates) of target network updates
discount factor	0.99	Value of γ used in the target computation.
action repeat	4	The number of simulator frames for which an action is repeated in a single timestep.
update frequency	8	Frequency (in timesteps) of parameter updates.
replay start size	50,000	Minimum number of transitions in the replay buffer required before parameter updates begin.
initial exploration	1.0	Initial value of ϵ used for ϵ-greedy exploration.
final exploration	0.01	Final value of ϵ used for ϵ-greedy exploration.
final exploration timestep	1,000,000	The number of timesteps over which ϵ is linearly annealed to its final ϵ.
maximum episode length	27,000	Timesteps after which an episode is truncated and the environment is reset.
step size	6.25e-5	Adam optimizer step size.
Adam ϵ	1.5e-4	The ϵ used by Adam.
Adam β_1	0.9	β_1 hyperparameter value in Adam.
Adam β_2	0.999	β_2 hyperparameter value in Adam.

Table 4: Evaluation details.

Detail	Value	Description
evaluation ϵ	0.001	The ϵ used for the ϵ-greedy policy used during evaluation.
maximum episode length	27,000	Number of timesteps in evaluation episodes. Corresponds to 30 minutes of gameplay.[3]
evaluation phase length	125,000	Length of periodic evaluation phases in terms of number of timesteps.
evaluation frequency	250,000	Frequency (in terms of training timesteps) of evaluation phase.

Double DQN is trained so that

$$Q(s,a;\boldsymbol{\theta}) \approx \mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t R_{t+1} | S_0 = s, A_0 = a, A_{t\geq 1} \sim \mathfrak{g}_{Q(\cdot|\boldsymbol{\theta})}\right].$$

During the periodic evaluations, we compute overestimation using the completed episodes. The agents use ϵ-greedy exploration with $\epsilon = 0.001$, which is near-greedy, in order to obtain a near-unbiased sample of a discounted return of the greedy policy. This evaluation phase produces k completed episodes (discarding incomplete episodes) $\tau_1, ..., \tau_k$ of lengths $T_1, ..., T_k$ respectively. For example, $\tau_1 = \{s_0, a_0, r_1, s_1, a_1, \ldots, a_{T_1-1}, r_{T_1}, s_{T_1}\}$. We then compute the average predicted state-action values across these state-action pairs across all the completed trajectories:

$$\hat{Q} = \frac{1}{\sum_{i=1}^{k} T_i}\sum_{i=1}^{k}\sum_{(s,a)\in\tau_i} Q(s,a;\boldsymbol{\theta}).$$

For an episode τ_i, the discounted return for a state-action pair (s_t, a_t) in the trajectory is $\text{Return}(\tau_i, s_t, a_t) = \sum_{r_j\in\tau_i; j\geq t+1}\gamma^{j-t-1}r_j$. Then the average return across all state-action pairs in

the set of completed episodes is:

$$\hat{G} = \frac{1}{\sum_{i=1}^{k} T_i} \sum_{i=1}^{k} \sum_{(s,a)\in\tau_i} \text{Return}(\tau_i, s, a).$$

$\hat{G}$ is a near-unbiased estimate of the expected discounted return under a greedy policy with respect to the Q-values. We then compute the overestimation: $\hat{Q} - \hat{G}$. For DDQL, we use the near-greedy policy with respect to the average of the two Q-functions, and $\hat{Q}$ is formed from the average Q-value:

$$\hat{Q} = \frac{1}{2\sum_{i=1}^{k} T_i} \sum_{i=1}^{k} \sum_{(s,a)\in\tau_i} \big(Q(s,a;\boldsymbol{\theta}_1) + Q(s,a;\boldsymbol{\theta}_2)\big).$$

Agents trained on Atari environments typically employ reward clipping, where rewards are clipped to the range $[-1, 1]$. Since agents are trained to predict return estimates under this reward, we also ensure that this clipping is applied when computing returns. This clipping, combined with discounting, often causes discounted returns to be much smaller than the raw, unclipped scores.

Another point to note is that evaluation episodes are truncated after 30 minutes of play (or 27k timesteps) as is typical in ALE evaluations. In these instances, to compute the return, we bootstrap the final Q-value of the non-terminal state at which the episode is truncated. This can indeed have an impact on results, but this occurs infrequently. Moreover, in the instances in which this does occur, this bootstrapped value is discounted to less than 0.005 for over 98% of state-action pairs in that truncated episode.

We counted the incidences of truncations across all algorithms, seeds, and environments. This totals well over 1k training runs, each with 200 evaluation phases of multiple episodes. We found that in 37 out of the 57 environments, truncation was never once exercised during evaluation across all algorithms and seeds. Upon inspecting the 20 environments for which truncation was exercised at least once, we found that in four of these environments truncation was exercised only in 1-2 episodes across all evaluation phases, seeds, and algorithms. Moreover, in some environments, like Montezuma's Revenge, where our agents are unable to achieve rewards but avoid ending the game for 30 minutes of gameplay, truncations occur but have no impact on the overestimation.

The use of a near-greedy policy instead of a greedy policy introduces some bias. However, policies with moderate stochasticity, even when sticky actions are used, generally improve performance. To understand why, let us consider an example. After a life loss in the game of Breakout, the agent must take a specific action ("Fire" action) to launch the ball and continue to receive rewards. If the agent selects any other action deterministically, even sticky action repeats cannot cause the agent to fire the ball.

D Target bootstrap decoupling and double estimation

In this Appendix, we provide a broader discussion, largely appealing to prior literature, on the various ways in which target bootstrap decoupling and double estimation with reciprocal bootstrapping can be integrated into learning algorithms.

D.1 Target bootstrap decoupling

In algorithms where only a single Q-network and target network are available, such as DQN and Double DQN, we can at most achieve target bootstrap decoupling. To implement target bootstrap decoupling, one network must be used as Q_{sel} and the other as Q_{eval}. This admits two possible choices, which are Double DQN and inverse Double DQN [36].[4]

Table 5: Single network target bootstrap decoupling.

Algorithm	$\boldsymbol{Q}_{\text{eval}}$	$\boldsymbol{Q}_{\text{sel}}$
Double DQN	$\boldsymbol{\theta}^-$	$\boldsymbol{\theta}$
Inverse Double DQN	$\boldsymbol{\theta}$	$\boldsymbol{\theta}^-$

Double DQN sets $\boldsymbol{\theta}_{\text{eval}} = \boldsymbol{\theta}^-$ and $\boldsymbol{\theta}_{\text{sel}} = \boldsymbol{\theta}$. Inverse Double DQN inverts this by setting $\boldsymbol{\theta}_{\text{eval}} = \boldsymbol{\theta}$ and $\boldsymbol{\theta}_{\text{sel}} = \boldsymbol{\theta}^-$. Though inverse Double DQN implements target bootstrap decoupling, it has been

[4]We are introducing this terminology here. This algorithm is called *inverse Double Q-learning* by van Hasselt et al. [36].

investigated by van Hasselt et al. [36] and has been shown to be unstable relative to both DQN and Double DQN. This makes Double DQN the clear choice as an algorithm that both performs well and reduces overestimation. These two variants are highlighted in Table 5.

D.2 Double estimation

DDQL algorithms implement double estimation with reciprocal bootstrapping. In particular, we now train two separate Q-functions with parameters $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$, which may have corresponding target networks $\boldsymbol{\theta}_1^-$ and $\boldsymbol{\theta}_2^-$. In this setting too, we have multiple options for computing bootstrap targets. If $\boldsymbol{\theta}_1$ or $\boldsymbol{\theta}_1^-$ is selecting the action in the target, then either $\boldsymbol{\theta}_2$ or $\boldsymbol{\theta}_2^-$ is used to evaluate the selected action, and vice versa. Suppose $\boldsymbol{\theta}_1$ is being updated. We are again presented with four main algorithm variants, which are summarized in Table 6.

1. $\boldsymbol{\theta}_{\text{eval}} = \boldsymbol{\theta}_2^-$ and $\boldsymbol{\theta}_{\text{sel}} = \boldsymbol{\theta}_1^-$. We denote this DDQL_{DQN} because it uses target networks for both action-selection and action-evaluation as is done in DQN. This strategy ensures a stationary target for the interval between target network updates.
2. $\boldsymbol{\theta}_{\text{eval}} = \boldsymbol{\theta}_2^-$ and $\boldsymbol{\theta}_{\text{sel}} = \boldsymbol{\theta}_1$. We denote this $\text{DDQL}_{\text{Double DQN}}$ because it selects an action for computing the target using a Q-network and evaluates it with a target network, as is done in Double DQN.
3. $\boldsymbol{\theta}_{\text{eval}} = \boldsymbol{\theta}_2$ and $\boldsymbol{\theta}_{\text{sel}} = \boldsymbol{\theta}_1^-$. We denote this $\text{DDQL}_{\text{Inverse}}$. In some sense, this is the analog of inverse Double DQN in the setting where we train two networks. Action-selection is performed with a target network, and action-evaluation is performed with a Q-network.
4. $\boldsymbol{\theta}_{\text{eval}} = \boldsymbol{\theta}_2$ and $\boldsymbol{\theta}_{\text{sel}} = \boldsymbol{\theta}_1$. We denote this $\text{DDQL}_{\text{No target}}$. In this variant, only the Q-networks are used to both select and evaluate actions. Target networks are not used.

While all of these algorithms implement double estimation, our study is primarily on the first algorithm, DDQL_{DQN}. DDQL_{DQN} is compared to $\text{DDQL}_{\text{Double DQN}}$ in Appendix E.3. When computing the bootstrap target, $\text{DDQL}_{\text{Inverse}}$ has stationary action-selection but nonstationary action-evaluation. Moreover, since $\boldsymbol{\theta}_2$ will also be updated with $\boldsymbol{\theta}_1$ as its action-evaluator, the targets will be highly nonstationary with both Q-functions changing. Furthermore, given that van Hasselt et al. [36] found Inverse Double DQN to be unstable, we did not study $\text{DDQL}_{\text{Inverse}}$.

Table 6: Options for target bootstrap decoupling with reciprocal bootstrapping in DDQL.

Algorithm	$\boldsymbol{\theta}_{\text{eval}}$	$\boldsymbol{\theta}_{\text{sel}}$
DDQL_{DQN}	$\boldsymbol{\theta}_2^-$	$\boldsymbol{\theta}_1^-$
$\text{DDQL}_{\text{Double DQN}}$	$\boldsymbol{\theta}_2^-$	$\boldsymbol{\theta}_1$
$\text{DDQL}_{\text{Inverse}}$	$\boldsymbol{\theta}_2$	$\boldsymbol{\theta}_1^-$
$\text{DDQL}_{\text{No target}}$	$\boldsymbol{\theta}_2$	$\boldsymbol{\theta}_1$

Additionally, we do not study $\text{DDQL}_{\text{No target}}$, which does not leverage any of the stationarity afforded by target networks. It may seem that a separate Q-network can replace the role of a target network by providing a secondary estimate. However, this secondary estimate is nonstationary, as this other Q-network is itself being updated frequently, leading to instability. Our preliminary results found this to perform poorly, which is consistent with findings that show that entirely forgoing the use of target networks in DQN is generally worse [22].

E Expanded discussion and results

In this appendix, we discuss the use of simultaneous updates, identical initialization, and alternative strategies to compute bootstrap targets. For the latter two, we also provide results.

E.1 Simultaneous updates

In the original Double Q-learning algorithm, at every timestep, the experience transition is used to update one of the two Q-functions. The updates are not done simultaneously, and in expectation, a Q-function is updated once every two timesteps. When we train Q-networks through experience replay, we do not typically use the online experiences to train. Rather, we treat the replay buffer as a pseudo-offline dataset from which we sample minibatches for parameter updates. This permits us to sample distinct minibatches and perform simultaneous updates. Moreover, simultaneous updates intuitively should benefit DDQL by avoiding the conflicting gradients that can arise in successive updates that optimize different losses. Furthermore, optimizing both losses simultaneously can enrich the learned representation by offering a mutual auxiliary task effect.

E.2 The importance of identical initialization

In tabular TD learning, we typically initialize the action-value lookup tables to zeros. When training Q-networks, however, we randomly initialize the weights of our Q-networks. When training two Q-networks (or heads) through DDQL, we are then presented with a choice of whether to initialize both Q-functions with the same random weights or with different random weights. We opt for the former.

Our choice of identically initializing the two Q-functions is sensible for two reasons. First, in tabular Double Q-learning, Q-functions are typically initialized identically, and the distinction between the two Q-functions initially emerges from using different experience transitions. Second, in Double Q-learning, we eventually want the two Q-functions to converge. Since both Q-functions are trained through reciprocal bootstrapping, training two interdependent Q-functions that start from different initializations may make optimization more challenging. Anecdotally, we did find that at lower target network update intervals, DDQL had fewer divergent or non-learning runs when identically initialized.

Identical initialization is not absolutely essential, but it can have a strong effect on DN-DDQL in some environments, at smaller target network update intervals. Figure 7 depicts three environments in which a non-identical initialization performs worse than an identical initialization. We run DN-DDQL with DQN's original shorter target network update interval of 2,500 gradient updates as opposed to 7,500 gradient updates. We denote this variant "short target". We compare it to DN-DDQL agents that share this shorter target network update interval and are initialized non-identically. We denote these latter agents "short target, ablate init". All other hyperparameters and training configurations are otherwise unchanged. We can see a visible difference in performance between the variants, where the only difference between the curves emerges from whether or not the networks are initialized identically. For many environments, the initialization is immaterial, especially at larger target network update intervals. Nonetheless, these results are interesting even if only from the perspective of better understanding learning dynamics.

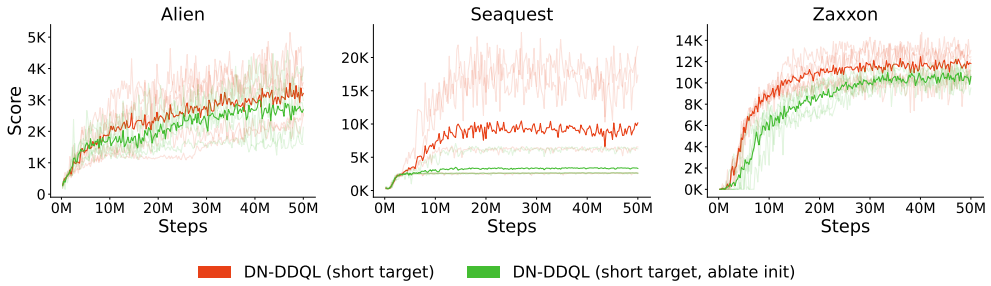


Figure 7: A comparison of DN-DDQL with a short target network update interval to DN-DDQL with a short target network update interval and non-identical initialization. Identical initialization can help performance.

E.3 Fixed DQN-style targets versus Double DQN-style targets

When training two Q-networks, each with its own target network, through reciprocal bootstrapping, we have several choices, as depicted in Table 6 in Appendix D.2. Appendix D.2 discussed why $\text{DDQL}_{\text{No target}}$ and $\text{DDQL}_{\text{Inverse}}$ are discarded as choices. Our primary candidate algorithms are DDQL_{DQN} and $\text{DDQL}_{\text{Double DQN}}$. We use the former in this paper.

Our choice can be motivated by the implicit goals underlying the development of DQN. DQN essentially formulates the learning problem as a sequence of relatively stationary supervised regression tasks. The replay buffer serves as a stationary dataset from which minibatches are sampled and used to minimize some regression loss over a fixed set of targets induced by the target network. Every target network refresh begins a new regression task, as the targets change. For the duration that the target network is held fixed, the targets are stationary and the only source of nonstationarity is the changing replay buffer, which is only a mild form of nonstationarity.

Double DQN forgoes stationary targets by using the actively changing Q-network to select actions for stationary evaluation. The more the Q-function's greedy actions change, which they often do [28], the more nonstationary the targets become. Double DQN finds a nice tradeoff where overestimation is reduced at the cost of some additional amount of nonstationarity in the selected actions in bootstrap targets, with stable action-evaluations.

When using two Q-networks, each with its own target network, we can avoid this tradeoff and can decouple both action-selection and action-evaluation in the target while retaining stationarity in both. Moreover, our results throughout this paper are consistent with the hypothesis that slowing down nonstationarity is essential for reciprocal training of Q-functions. As such, we use DDQL_{DQN}, where, when updating $\boldsymbol{\theta}_1$, we use $\boldsymbol{\theta}_{\text{sel}} = \boldsymbol{\theta}_1^-$ and $\boldsymbol{\theta}_{\text{eval}} = \boldsymbol{\theta}_2^-$. By doing so, we fix the target values for both Q-functions for the duration of the target network update interval.

Figure 8 compares DDQL to $\text{DDQL}_{\text{Double DQN}}$. The two algorithms generally perform on par with one another, though DDQL substantially outperforms $\text{DDQL}_{\text{Double DQN}}$ in PHOENIX and appears to do reliably better in DOUBLEDUNK. DDQL may be more reliable in BATTLEZONE, but it is difficult to draw any conclusions as performance is similar between the two algorithms if we exclude the single bad seed for $\text{DDQL}_{\text{Double DQN}}$.

Figure 9 compares DN-DDQL to $\text{DN-DDQL}_{\text{Double DQN}}$. Again, it seems the stationarity of DQN-style updates is helpful. In many environments, the performances are almost indistinguishable. However, in environments like ZAXXON, WIZARDOFWOR, ALIEN, and QBERT, DN-DDQL performs reliably better. In BATTLEZONE, DN-DDQL is clearly better, with $\text{DN-DDQL}_{\text{Double DQN}}$ seemingly being unable to learn. We conclude that for DN-DDQL, $\text{DN-DDQL}_{\text{DQN}}$ is more stable than $\text{DN-DDQL}_{\text{Double DQN}}$.

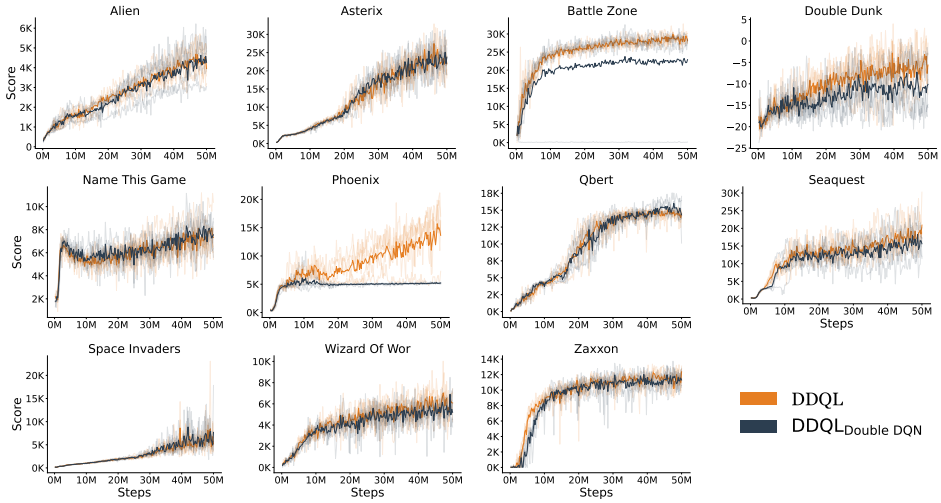


Figure 8: Performance of DDQL compared to $\text{DDQL}_{\text{Double DQN}}$. The algorithms perform at a similar level, with DDQL performing better in two environments.

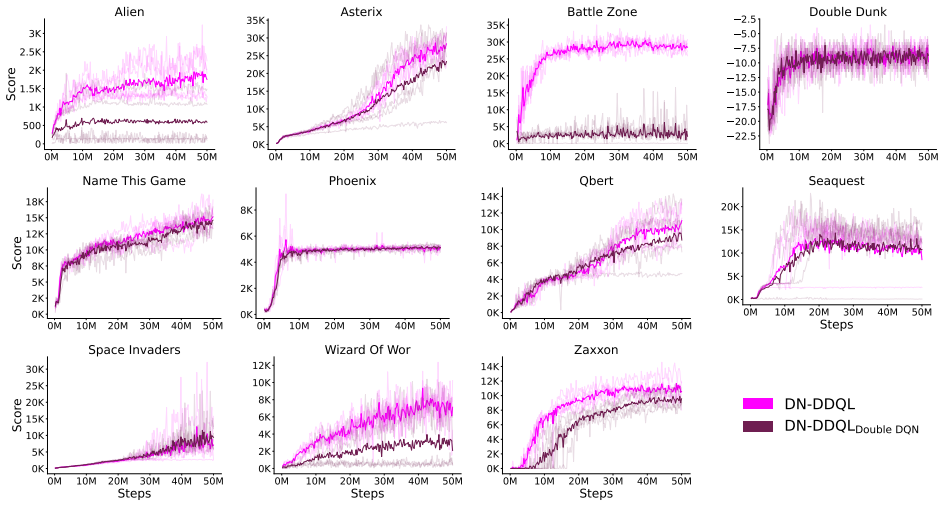


Figure 9: Performance of DN-DDQL compared to DN-DDQL$_{\text{Double DQN}}$. DN-DDQL performs more stably.

F Deep Double Q-learning: full results

This appendix does not include new results. Rather, it expands the results shown in the body regarding our ablations, overestimation, and performance of Double DQN, DDQL, DM-DDQL, and DN-DDQL.

F.1 Ablations

This subsection contains the full curves corresponding to the ablations in Section 5. Figure 10 shows the performance curves during training for DDQL's dataset partitioning experiments. Figure 11 shows the performance curves during training for DN-DDQL's dataset partitioning experiments. Figure 12 shows the overestimation curves during training for DDQL's dataset partitioning experiments. Figure 13 shows the performance curves during training for DDQL's replay ratio experiments. Figure 14 shows the performance curves during training for DN-DDQL's replay ratio experiments. Figure 15 shows the overestimation curves during training for DDQL's replay ratio experiments.

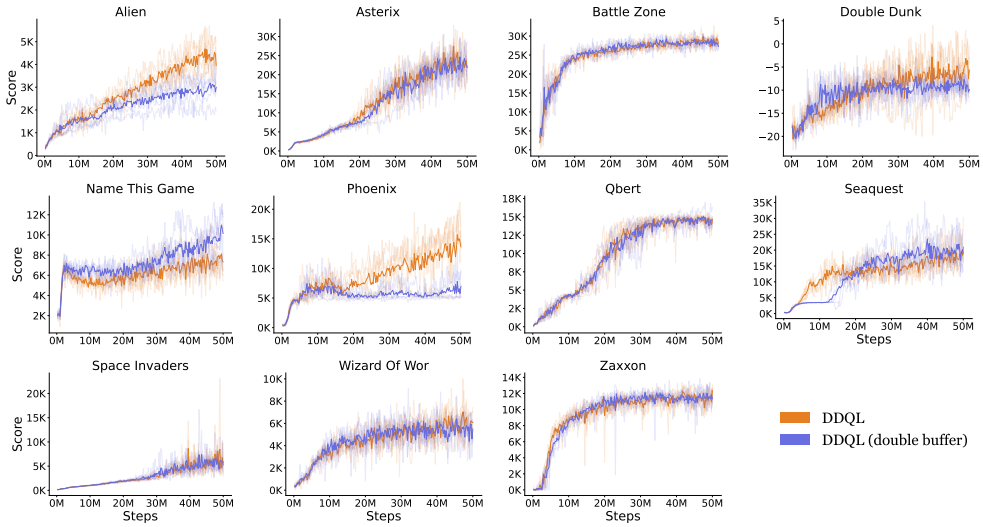


Figure 10: Performance of DDQL compared to DDQL (double buffer). The algorithms largely perform at a similar level, with a few exceptions.

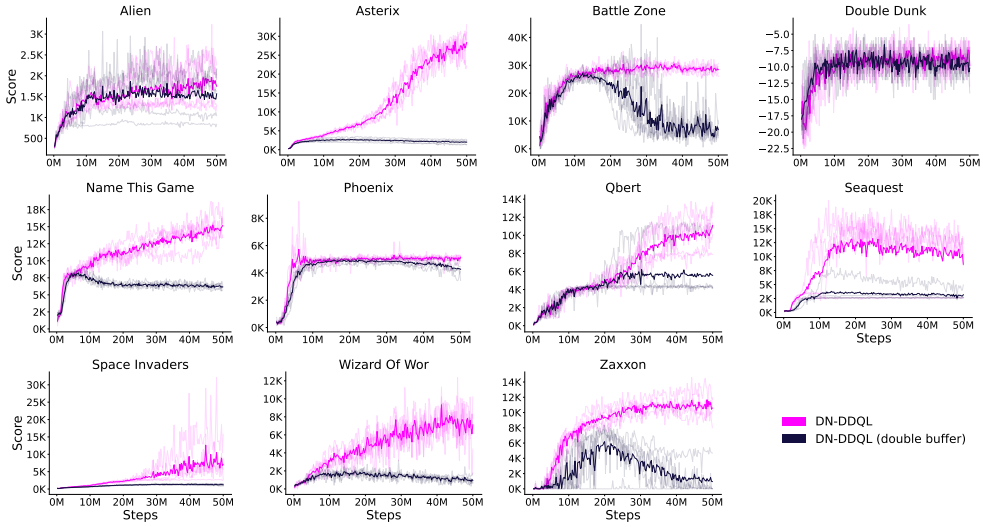


Figure 11: Performance of DN-DDQL compared to DN-DDQL (double buffer). DN-DDQL (double buffer) performs quite poorly.

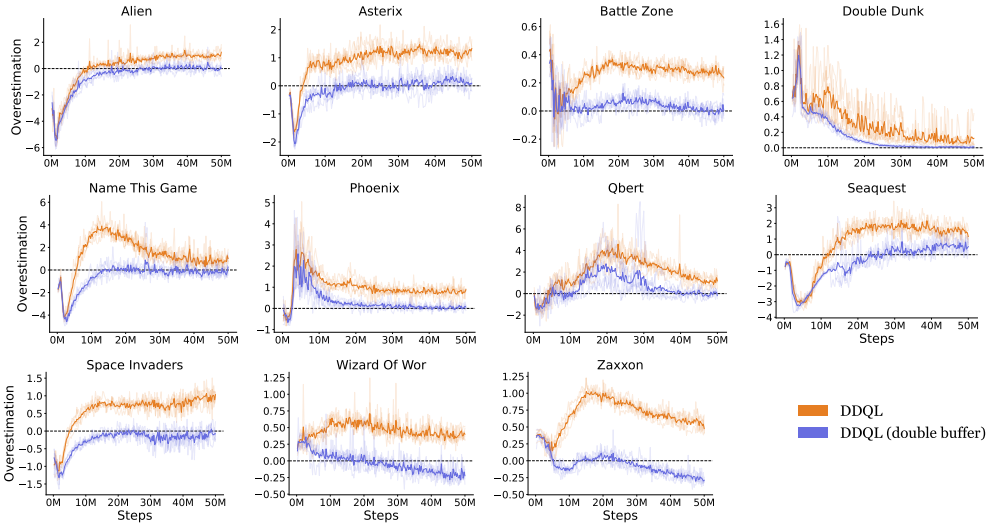


Figure 12: Overestimation of DDQL compared to DDQL (double buffer). The double buffer strategy reduces overestimation.

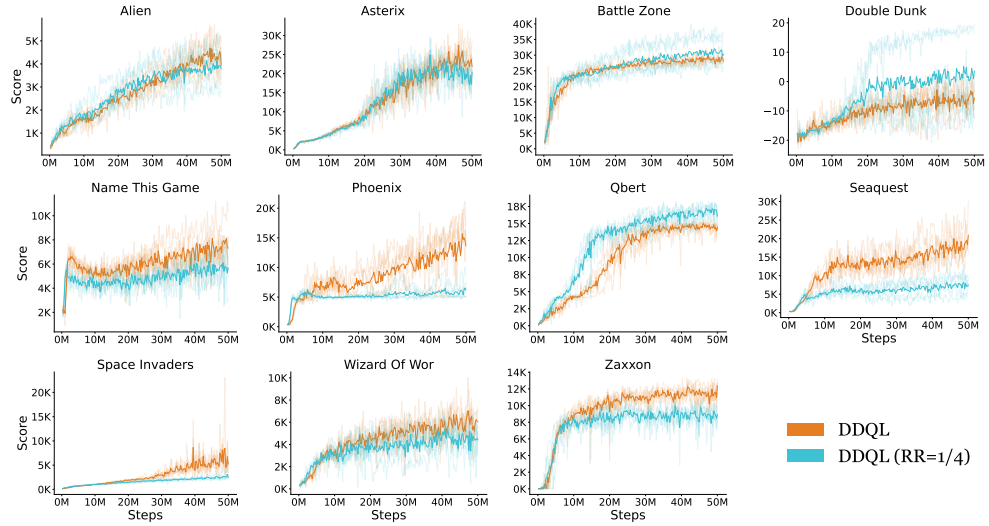


Figure 13: Performance of DDQL compared to DDQL(RR $= 1/4$). They perform similarly in some environments, but DDQL(RR $= 1/4$) is generally worse.

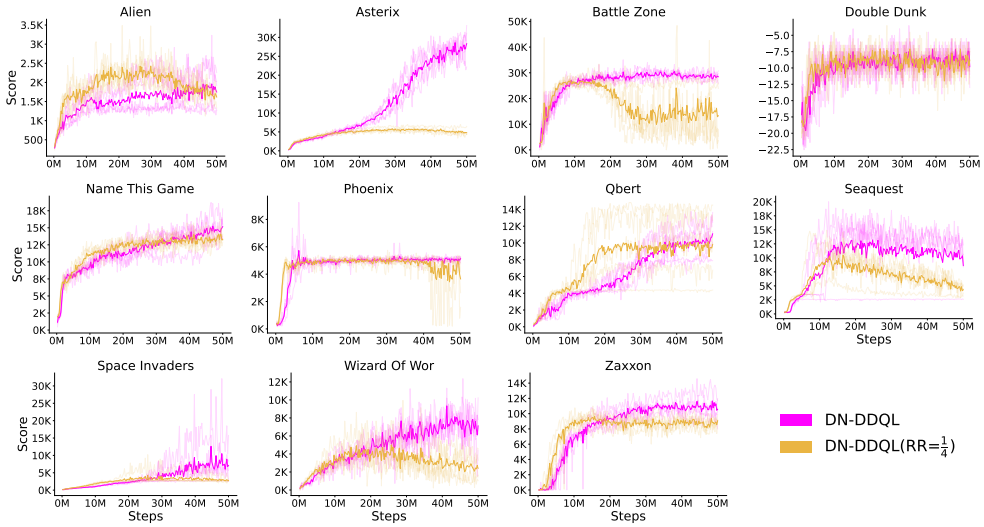


Figure 14: Performance of DN-DDQL compared to DN-DDQL(RR $= 1/4$). DN-DDQL(RR $= 1/4$) is generally worse.

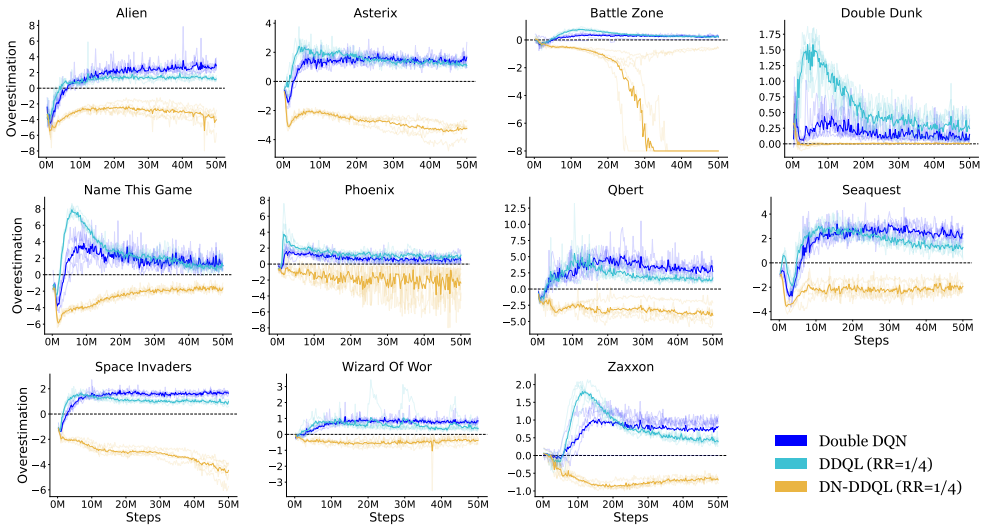


Figure 15: Overestimation of DDQL(RR $= 1/4$), DN-DDQL(RR $= 1/4$), and Double DQN. Overestimation is clipped at -8 due to divergence in BATTLEZONE. The DDQL variants continue to reduce overestimation even with double the replay ratio.

F.2 Ablation: Double DQN with a lower replay ratio

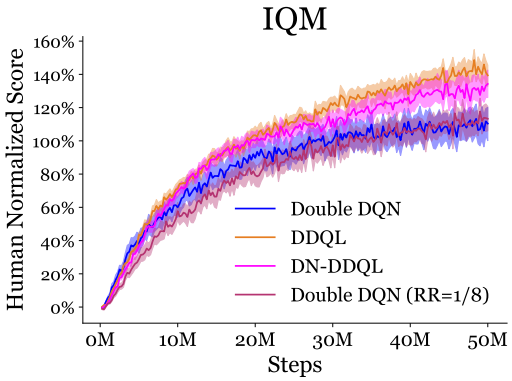
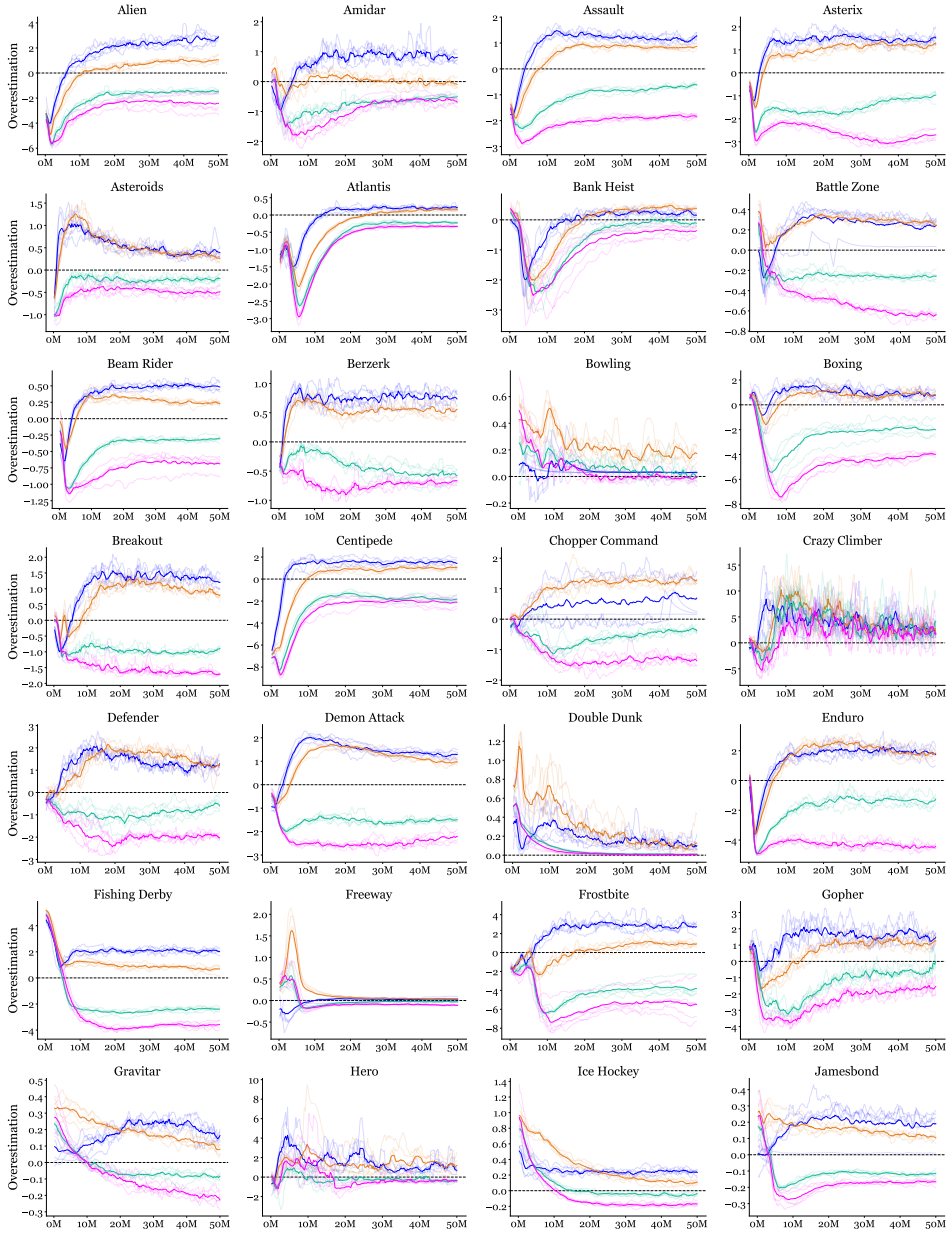


Figure 16: Interquartile mean of the HNS throughout training on 57 Atari environments. The shaded region represents a 95% stratified bootstrap confidence interval. Double DQN(RR $= 1/8$) is run for three seeds per game and the other algorithms are run for five seeds per game.

To understand whether the performance improvements of DDQL come from using a lower replay ratio, we ran Double DQN for three seeds on the 57 games with a replay ratio of $1/8$. Fig. 16 shows that Double DQN using a replay ratio of $1/8$ performs similarly to Double DQN. Importantly, both variants of Double DQN perform worse than DDQL, indicating that it is not solely DDQL's lower replay ratio that causes improved performance.

F.3 Atari-57 overestimation results

Figure 17 depicts the overestimation measured throughout training across all environments and seeds.


Alien
Amidar
Assault
Asterix
Asteroids
Atlantis
Bank Heist
Battle Zone
Beam Rider
Berzerk
Bowling
Boxing
Breakout
Centipede
Chopper Command
Crazy Climber
Defender
Demon Attack
Double Dunk
Enduro
Fishing Derby
Freeway
Frostbite
Gopher
Gravitar
Hero
Ice Hockey
Jamesbond
Overestimation
0M 10M 20M 30M 40M 50M

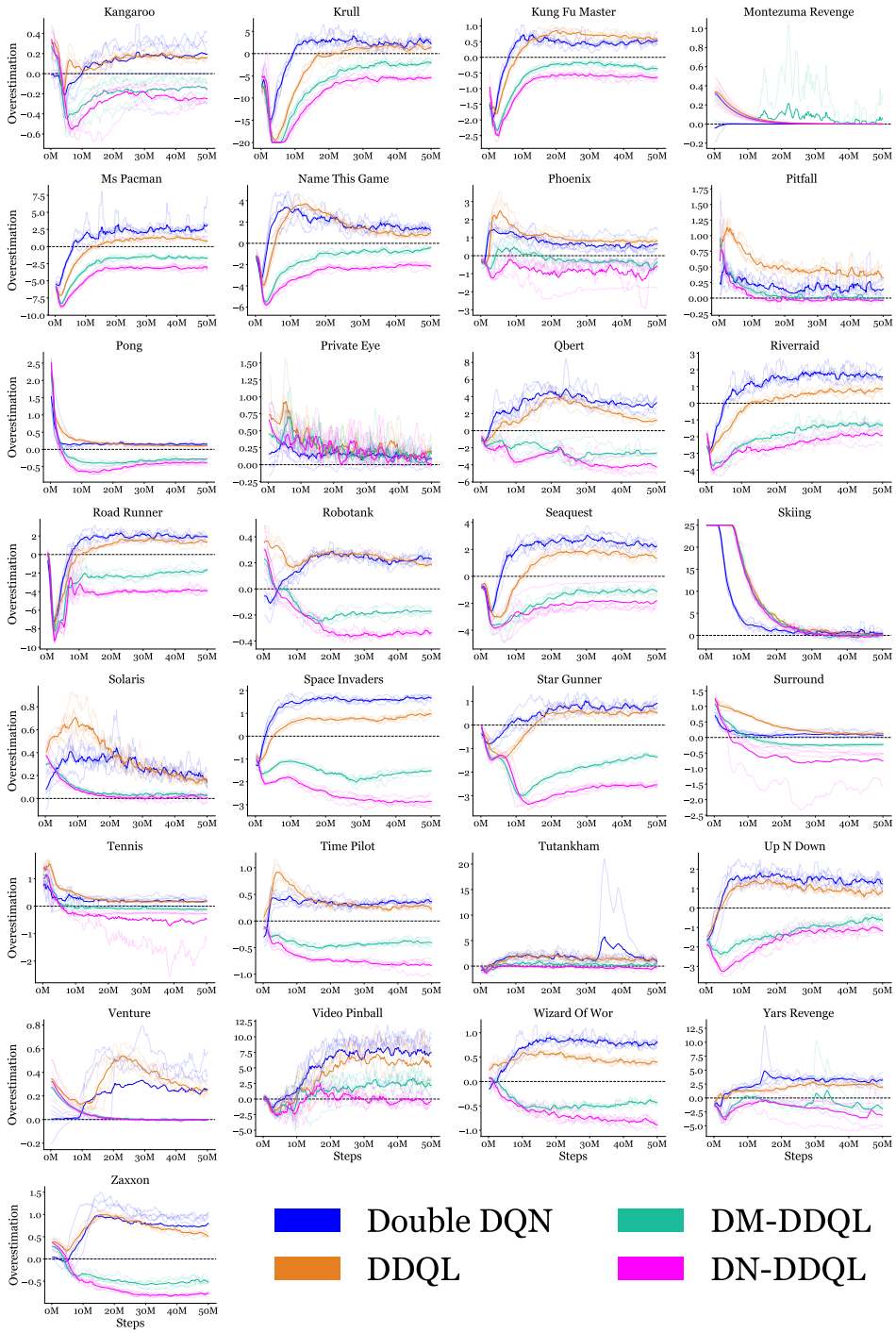


Figure 17: Overestimation over 50M timesteps for 57 Atari 2600 games. For readability, overestimation is bounded below at -20 and above at 25. Plots are smoothed with a moving average of five. The semitransparent curves are the individual seeds.

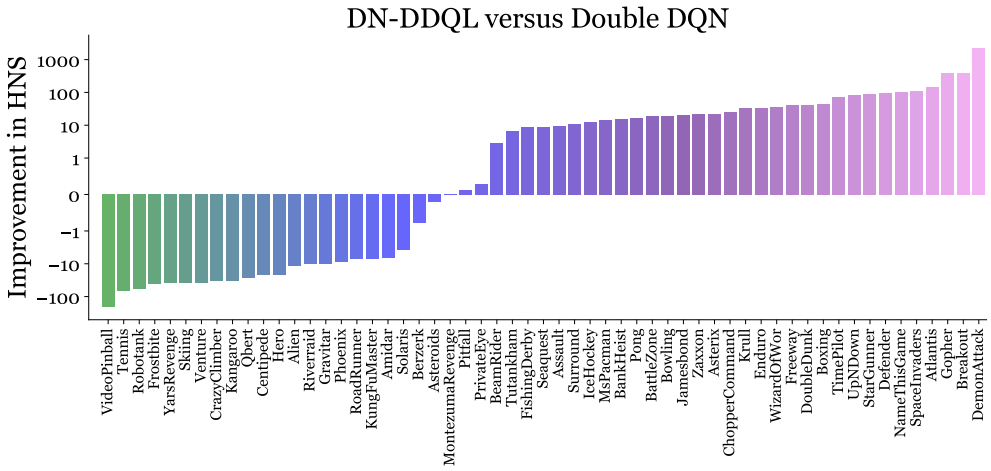
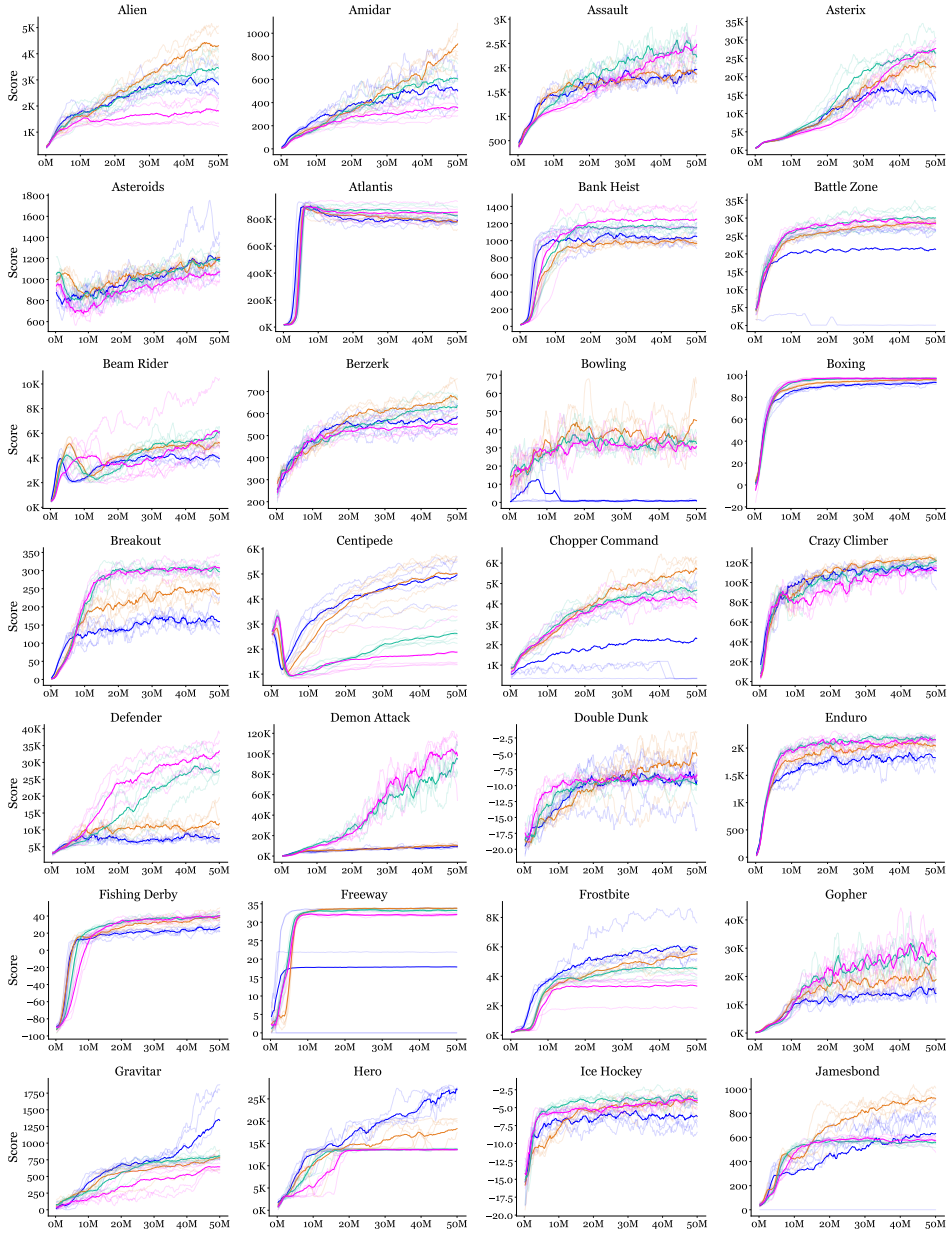


Figure 18: **Per-game performance improvements of DN-DDQL.** Per-game improvements in HNS of DN-DDQL over Double DQN in each of 57 games, calculated as the average area under the curve across 5 seeds. The y-axis is log-scale.

F.4 Atari-57 Performance

Figure 19 depicts the performance curves throughout training for Double DQN, DDQL, DM-DDQL, and DN-DDQL, across all games and seeds. These curves depict the raw game scores instead of the human-normalized scores. These curves depict the data used to produce Figure 3. Figure 18 depicts the per-game improvements of DN-DDQL over Double DQN.

Table 7 reports the mean score across the final three evaluations for each of our four algorithms in all 57 games for five seeds. Note that this differs from standard reporting protocols, which often use the best-performing checkpoint and re-evaluate it. Prior evaluation protocols should be expected to give better results than using the final evaluation [12]. Moreover, it should also be noted that we are using sticky actions, with the full action set, and game-over termination. As of this writing, most research in the sticky action setting still uses game-specific knowledge through the minimal action set, which should make the problem easier.


Alien
Amidar
Assault
Asterix
Asteroids
Atlantis
Bank Heist
Battle Zone
Beam Rider
Berzerk
Bowling
Boxing
Breakout
Centipede
Chopper Command
Crazy Climber
Defender
Demon Attack
Double Dunk
Enduro
Fishing Derby
Freeway
Frostbite
Gopher
Gravitar
Hero
Ice Hockey
Jamesbond
Score

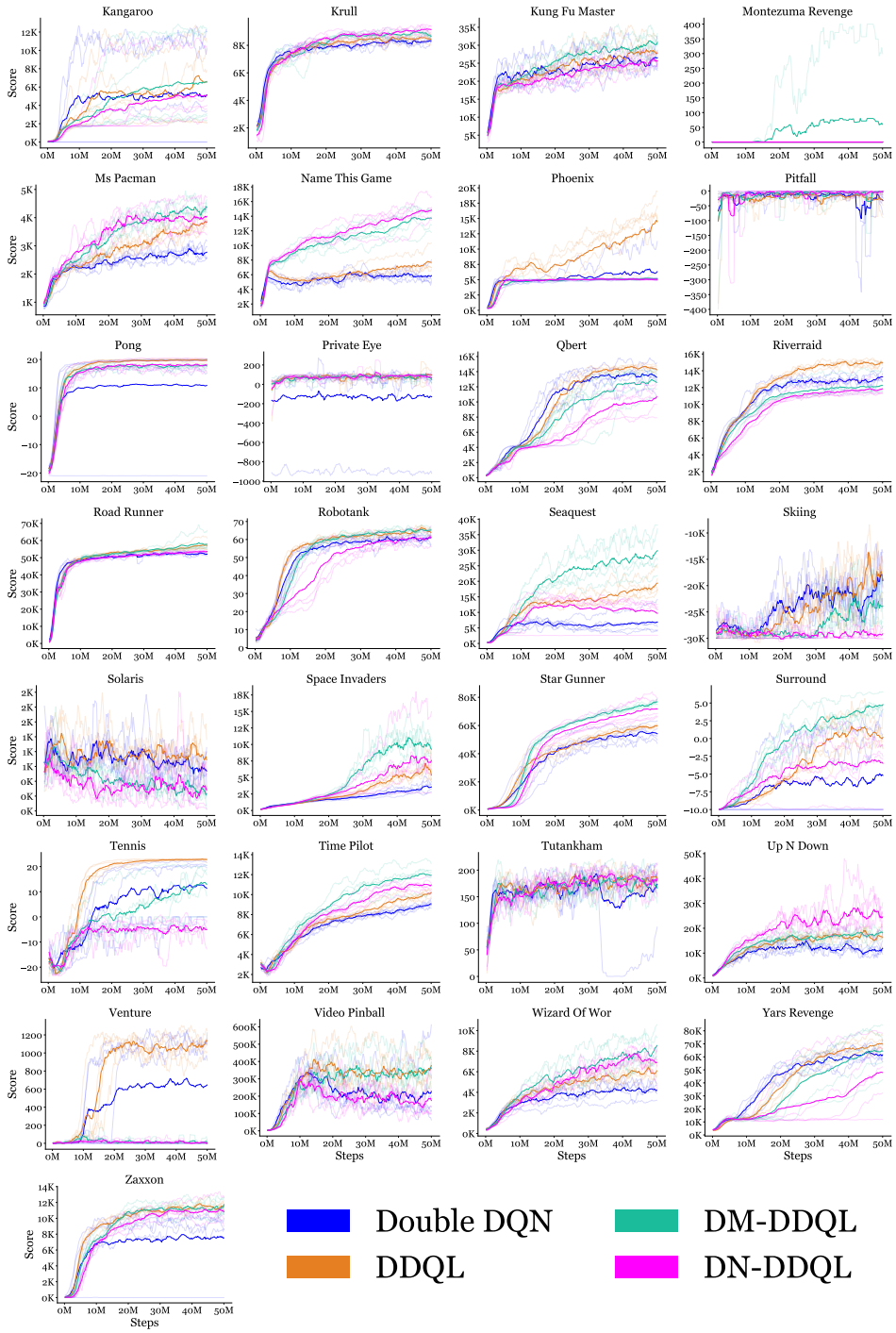


Figure 19: Scores across 50M timesteps across 57 Atari 2600 games. Plots are smoothed with a moving average of seven. The semitransparent curves are the individual seeds.

Environment	Double DQN	DDQL	DM-DDQL	DN-DDQL
ALIEN	2805.1	**4314.7**	3483.1	1811.0
AMIDAR	508.9	**922.1**	600.9	366.6
ASSAULT	1888.3	1870.4	2290.9	**2462.5**
ASTERIX	13371.6	22896.3	26851.5	**27554.4**
ASTEROIDS	1172.4	**1221.9**	1176.6	1071.7
ATLANTIS	788923.3	782803.3	827005.0	**852533.3**
BANKHEIST	1047.5	964.3	1148.2	**1242.7**
BATTLEZONE	21036.8	28890.6	**30055.1**	28458.9
BEAMRIDER	3988.6	5110.9	**6209.2**	6084.0
BERZERK	579.2	**664.4**	636.1	555.5
BOWLING	1.1	**46.1**	33.7	31.1
BOXING	93.6	95.8	97.2	**97.5**
BREAKOUT	163.4	234.2	292.8	**307.0**
CENTIPEDE	4942.2	**4993.0**	2623.3	1875.2
CHOPPERCOMMAND	2204.7	**5620.7**	4754.7	4152.3
CRAZYCLIMBER	110893.9	**122650.0**	122173.1	115661.2
DEFENDER	7330.3	11725.4	28707.9	**33471.0**
DEMONATTACK	9060.2	10496.3	91866.8	**99210.5**
DOUBLEDUNK	−9.5	**−5.6**	−10.1	−8.2
ENDURO	1820.8	2044.0	**2144.3**	2133.7
FISHINGDERBY	26.9	37.9	**40.3**	39.7
FREEWAY	17.9	**33.8**	33.2	32.1
FROSTBITE	**5889.5**	5520.6	4531.7	3341.6
GOPHER	14134.3	19253.2	25712.6	**26542.9**
GRAVITAR	**1326.1**	776.0	801.5	641.0
HERO	**27155.9**	18281.5	13710.1	13595.8
ICEHOCKEY	−6.1	−4.4	**−4.0**	**−4.0**
JAMESBOND	626.2	**934.6**	555.7	569.2
KANGAROO	5040.0	6560.8	**6722.1**	5106.6
KRULL	8324.5	8381.0	8580.0	**9190.5**
KUNGFUMASTER	26818.1	27777.2	**30671.6**	25966.0
MONTEZUMAREVENGE	0.0	0.0	**53.3**	0.0
MSPACMAN	2728.3	3834.2	**4304.7**	4215.8
NAMETHISGAME	5810.1	7618.7	13759.8	**14972.4**
PHOENIX	6245.6	**14372.1**	5120.8	5084.8
PITFALL	−37.8	−40.1	−4.8	**−1.7**
PONG	10.8	**19.8**	17.6	18.4
PRIVATEEYE	−137.1	**100.4**	81.6	100.0
QBERT	13104.1	**14269.0**	12373.4	10806.1
RIVERRAID	13160.2	**14953.2**	12309.0	11859.0
ROADRUNNER	52216.0	56757.9	**57172.6**	53529.3
ROBOTANK	61.5	63.8	**65.2**	61.0
SEAQUEST	6414.8	19617.2	**30658.2**	9692.6
SKIING	−19531.8	**−17661.6**	−24623.8	−29432.1
SOLARIS	579.7	**805.7**	222.0	306.4
SPACEINVADERS	3544.4	5609.2	**9559.0**	7593.0
STARGUNNER	53216.3	59591.5	**76406.0**	71549.3
SURROUND	−5.3	0.0	**4.7**	−3.6
TENNIS	11.1	**22.6**	13.3	−3.5
TIMEPILOT	9128.9	10314.7	**12015.9**	10932.3
TUTANKHAM	**184.3**	183.0	159.8	176.1
UPNDOWN	11864.2	15349.0	19151.8	**23914.9**
VENTURE	628.5	**1160.2**	6.1	0.0
VIDEOPINBALL	222658.8	360292.7	**364436.5**	185664.2
WIZARDOFWOR	4204.4	6142.2	**8681.3**	6671.7
YARSREVENGE	63051.5	**69460.9**	64182.2	48159.0
ZAXXON	7344.5	11636.4	**11697.9**	10771.2

Table 7: The mean evaluation score across the last three evaluations during training for different algorithms. The highest scores for an environment are bolded.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction claim that we adapt Double Q-learning to deep RL and that it outperforms Double DQN and reduces overestimation. Section 3 introduces this adaptation, DDQL, and Section 4 demonstrates DDQL's reduced overestimation and improved performance.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: In the discussion and conclusion section, we discuss how the paper's scope is limited to discrete actions and that it does not incorporate many recent deep RL advances.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The appendix provides granular details for reproducibility purposes for readers as well as citations to work we have closely replicated. We additionally released our code, which runs on entirely open source software, ensuring reproducibility.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We have provided a link to the code in the paper. We do not release the data.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Yes, our hyperparameters, optimizers, and games (our evaluation environments) are all clearly specified and we indicate where they come from. The appendices also provide detailed information on all hyperparameters, training details, and measurements.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: When appropriate (i.e., when we have sufficient runs), we use 95% stratified bootstrap confidence intervals, following standard practice in the literature [2]. When we do not, we do not provide a statistically inaccurate summary or misuse error bars; rather we provide in the appendix the individual learning curves of each run.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: In a footnote in the experiments section we provide a rough calculation of the compute costs to run the experiments in the paper.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the NeurIPS Code of Ethics and confirm that it conforms with the code. Our research has no test subjects, human or environment data, or deployment into real world systems. Our work is algorithmic in nature and our data is entirely artificial, being game data.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

Justification: This work examines fundamental deep RL algorithms on artificial environments (games) without direct societal impact. While the broader field of deep RL may have societal impact, our paper does not create specific societal impacts that are not broadly applicable to the field.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: Our data is performance data on games and is not sensitive. Hence, safeguards are not needed.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We use the PFRL library for our experiments. It is an open-source library with an associated paper, which we cite.

    Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We release our code, linked in the paper.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not have study participants nor crowdsourcing.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: Our core method development in this research does not involve LLMs as any important, original, or non-standard components.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.