DEEP REINFORCEMENT LEARNING AND THE TALE OF TWO TEMPORAL DIFFERENCE ERRORS

Anonymous authors
Paper under double-blind review

ABSTRACT

The temporal difference (TD) error was first formalized in Sutton (1988), where it was first characterized as the difference between temporally successive predictions, and later, in that same work, formulated as the difference between a bootstrapped target and a prediction. Since then, these two interpretations of the TD error have been used interchangeably, with the latter eventually being adopted as the critic loss in deep reinforcement learning (RL) architectures. In this work, we show that these two interpretations of the TD error are not always equivalent. In particular, we show that increasingly nonlinear deep RL architectures can cause these interpretations of the TD error to yield increasingly different numerical values. Then, building on this insight, we show how choosing one interpretation of the TD error over the other can affect the performance of deep RL algorithms that utilize the TD error to compute quantities beyond the value function, such as with deep differential (i.e., average-reward) RL methods. All in all, our results show that the default interpretation of the TD error as the difference between a bootstrapped target and a prediction does not always hold in deep RL settings.

1 INTRODUCTION

One of the defining aspects of reinforcement learning (RL) is that of the temporal difference (TD) error. The concept of the TD error was first formalized in Sutton (1988), where it was first characterized as the difference between temporally successive predictions, and later, in that same work, formulated as the difference between a bootstrapped target and a prediction. Since then, these two seemingly equivalent interpretations of the TD error have been used interchangeably in the literature, where the latter interpretation is what often comes to mind when one thinks of the TD error.

When the first modern deep RL methods were proposed (e.g. Mnih et al. (2015)), they followed prior work (e.g. Tesauro (1992)) in adopting the interpretation of the TD error as the difference between a bootstrapped target and a prediction as the critic (i.e., value network) loss. This TD-based loss has become synonymous with the TD error in deep RL settings, and has been adopted in modern value-based (e.g. Mnih et al. (2015)) and actor-critic (e.g. Haarnoja et al. (2018)) deep RL architectures. Indeed, if one were asked to compute the TD error, δ_t, at a given time step for a deep RL method, most would do so as the B-sized batch average difference between bootstrapped targets and estimates. For example, in a DQN-style algorithm (we will formalize our notation in Section 2):

$$\delta_t \approx \frac{1}{B}\sum_{j=1}^{B}\left[R_{j,t+1} + \gamma \max_{a_*} Q(S_{j,t+1}, a_*) - Q(S_{j,t}, A_{j,t})\right]. \tag{1}$$

But how reliable is this estimate of the TD error? And how does it compare to the original interpretation of the TD error as the difference between temporally successive predictions? In other words, is the output of Equation (1) always an accurate and appropriate representation of the TD error?

In this work, we explore such questions and arrive at a surprising finding: increasingly nonlinear deep RL architectures can cause the two interpretations of the TD error to yield increasingly different numerical values. We then leverage this insight to show how choosing one interpretation of the TD error over the other can significantly affect the performance of deep RL algorithms that utilize the TD error to compute quantities beyond the value function, such as with deep differential (i.e., average-reward) RL methods. All in all, our results show that the default interpretation of the TD error as the difference between a bootstrapped target and a prediction does not always hold in deep RL settings.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

2 PRELIMINARIES

The agent-environment interaction that occurs in reinforcement learning is often modeled as a Markov decision process (MDP) (Puterman, 1994). More formally, a finite discounted MDP is the tuple $\mathcal{M} \doteq \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma \rangle$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{R} \subset \mathbb{R}$ is a finite set of rewards, $p : \mathcal{S} \times \mathcal{A} \times \mathcal{R} \times \mathcal{S} \to [0, 1]$ is a probabilistic transition function that describes the dynamics of the environment, and $\gamma \in [0, 1]$ is the discount factor. At each discrete time step, $t = 0, 1, 2, \ldots$, an agent chooses an action, $A_t \in \mathcal{A}$, based on its current state, $S_t \in \mathcal{S}$, and receives a reward, $R_{t+1} \in \mathcal{R}$, while transitioning to a (potentially) new state, S_{t+1}, such that $p(s', r \mid s, a) = \mathbb{P}(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a)$. In a discounted MDP, an agent aims to find a policy, $\pi : \mathcal{S} \to \mathcal{A}$, that optimizes the expected discounted sum of rewards. The performance of a policy can be characterized via a *value function*, such as the state-action value function, Q_π:

$$Q_\pi(s, a) \doteq \mathbb{E}_\pi \left[\sum_{t=0}^{T} \gamma^t R_{t+1} \mid S_0 = s, A_0 = a, A_{1:T} \sim \pi \right], \tag{2}$$

where T is the length of the time horizon of the MDP. In this work, we limit our discussion to *stationary Markov* policies, which are time-independent policies that satisfy the Markov property.

In the context of RL, the state-action value function (2) can be optimized via Q-learning (Watkins & Dayan, 1992). A tabular version of the algorithm is shown below:

$$\delta_t = R_{t+1} + \gamma \max_{a_* \in \mathcal{A}} Q_t(S_{t+1}, a_*) - Q_t(S_t, A_t) \tag{3a}$$

$$Q_{t+1}(S_t, A_t) = Q_t(S_t, A_t) + \alpha_t \delta_t \tag{3b}$$

$$Q_{t+1}(s, a) = Q_t(s, a), \quad \forall (s, a) \neq (S_t, A_t)\,, \tag{3c}$$

where $Q_t : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a table of state-action value function estimates, δ_t is the TD error, and α_t is the value function step size at time t. In deep RL settings, the value function is approximated with a neural network, $Q_{\boldsymbol{w}}(s, a)$, where $\boldsymbol{w}$ denotes the parameters of the network. The agent learns the parameters of this *critic* network by minimizing the *mean squared Bellman error* via gradient updates as follows (Mnih et al., 2015):

$$\mathcal{L}_{\text{critic}}(\boldsymbol{w}_t) \doteq \frac{1}{B} \sum_{j=1}^{B} \frac{1}{2} \left(R_{j,t+1} + \gamma \max_{a_* \in \mathcal{A}} Q_{\boldsymbol{w}_t}(S_{j,t+1}, a_*) - Q_{\boldsymbol{w}_t}(S_{j,t}, A_{j,t}) \right)^2 \tag{4a}$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha_t \nabla_{\boldsymbol{w}_t} \mathcal{L}_{\text{critic}}\,, \tag{4b}$$

where $\{(S_{j,t}, A_{j,t}, R_{j,t+1}, S_{j,t+1})\}_{j=1}^{B}$ denotes a B-sized batch of agent-environment interactions sampled from a replay buffer, and $\nabla_{\boldsymbol{w}_t} \mathcal{L}_{\text{critic}}$ denotes the gradient of the loss (4a) with respect to the network parameters. This kind of value function parameterization is widely used in both value-based (e.g. Mnih et al. (2015)) and actor-critic (e.g. Haarnoja et al. (2018)) deep RL algorithms.

3 THE TALE OF TWO TEMPORAL DIFFERENCE ERRORS

In this section, we perform a formal exploration and characterization of the differences between the two interpretations of the TD error. We first perform this characterization from an analytical perspective, followed by a characterization from an empirical perspective. Altogether, we will show that increasingly nonlinear deep RL architectures can cause the two interpretations of the TD error to yield increasingly different numerical values.

We begin our characterization by formalizing the two interpretations of the TD error. We begin with the common interpretation of the TD error as the difference between a bootstrapped target and a prediction. We will refer to this interpretation as the *explicit* TD error, δ^e. For example, in the tabular Q-learning algorithm (3), the explicit TD error at time t is defined as follows:

$$\delta_t^e = R_{t+1} + \gamma \max_{a_*} Q_t(S_{t+1}, a_*) - Q_t(S_t, A_t), \tag{5}$$

where, in this case (i.e., for Q-learning), $R_{t+1} + \gamma \max_{a_*} Q_t(S_{t+1}, a_*)$ denotes the bootstrapped target, and $Q_t(S_t, A_t)$ denotes the prediction at time t. Analogous explicit TD errors can be derived for other tabular RL algorithms using different targets and estimates. For simplicity, we will

use Q-learning as a concrete example throughout this work, however, we emphasize that our definition of the explicit TD error encompasses all TD errors defined based on the difference between a bootstrapped target and a prediction. This is formalized as Definition 3.1 below:

Definition 3.1 (Explicit TD Error)**.** *An explicit TD error, δ^e, is any TD error defined based on the difference between a bootstrapped target and a prediction.*

We now proceed with the interpretation of the TD error as the difference between temporally successive predictions. We will refer to this interpretation as the *implicit* TD error, δ^i. For example, in the tabular Q-learning algorithm (3), the implicit TD error at time t is defined as follows:

$$\delta_t^i = \frac{1}{\alpha_t}\left(Q_{t+1}(S_t, A_t) - Q_t(S_t, A_t)\right). \tag{6}$$

Another way to think about the implicit TD error (6) is by taking Equation (3b) and "solving" for the TD error. Again, we will use Q-learning as a concrete example throughout this work, however, our definition of the implicit TD error encompasses all TD errors defined based on the difference between temporally successive predictions. This is formalized as Definition 3.2 below:

Definition 3.2 (Implicit TD Error)**.** *An implicit TD error, δ^i, is any TD error defined based on the difference between temporally successive predictions.*

Next, having formally defined the explicit and implicit TD errors, we now begin our characterization of the differences between the two. As a starting point, let us consider tabular algorithms. In particular, when we look at tabular RL algorithms, such as Q-learning (3), it is easily seen that $\delta_t^e = \delta_t^i$. This is formalized as Proposition 3.3 below:

Proposition 3.3. *In tabular settings, both interpretations of the TD error are always equivalent. That is, $\delta_t^e = \delta_t^i \; \forall t \in \mathbb{N}$.*

Proof. This follows directly from Proposition 3.4, which will be discussed later in Section 3.1. □

Indeed, in tabular settings, the two interpretations of the TD error are always equivalent. However, we will now show that this is not always the case in function approximation settings, particularly with increasingly nonlinear deep RL architectures. To this end, let us begin by extending our definitions of the tabular Q-learning TD errors (5) and (6) into function approximation settings. In particular, in such RL settings, the explicit and implicit TD errors can be defined as follows:

$$\delta_t^e = \frac{1}{B}\sum_{j=1}^{B}\left[R_{j,t+1} + \gamma \max_{a_*} Q_{\boldsymbol{w}_t}(S_{j,t+1}, a_*) - Q_{\boldsymbol{w}_t}(S_{j,t}, A_{j,t})\right] \text{ and} \tag{7}$$

$$\delta_t^i = \frac{1}{B}\sum_{j=1}^{B}\left[\frac{1}{\alpha_t}\left(Q_{\boldsymbol{w}_{t+1}}(S_{j,t}, A_{j,t}) - Q_{\boldsymbol{w}_t}(S_{j,t}, A_{j,t})\right)\right], \tag{8}$$

where we interpret these TD errors as the B-sized batch average difference between relevant terms.

We are now ready to pose the central question that we seek to answer in this work:

When are δ_t^e (7) and δ_t^i (8) equivalent?

3.1 The Two TD Errors Are Not Equivalent Even in Linear Function Approximation Settings

In Proposition 3.3, we showed that the two interpretations of the TD error are always equivalent in tabular settings. In this section, we will show that even in the simplest kind of function approximation settings, this equivalence begins to break down.

To this end, let us begin by considering the explicit and implicit TD errors in the linear function approximation setting (with a batch size of 1):

$$\delta_t^e = R_{t+1} + \gamma \max_{a_*} Q_{\boldsymbol{w}_t}(S_{t+1}, a_*) - Q_{\boldsymbol{w}_t}(S_t, A_t) \tag{9a}$$

$$= R_{t+1} + \gamma \max_{a_*} \boldsymbol{w}_t^T \boldsymbol{x}(S_{t+1}, a_*) - \boldsymbol{w}_t^T \boldsymbol{x}(S_t, A_t), \tag{9b}$$

and

$$\delta_t^i = \frac{1}{\alpha_t}\left(Q_{\boldsymbol{w}_{t+1}}(S_t, A_t) - Q_{\boldsymbol{w}_t}(S_t, A_t)\right) \tag{10a}$$

$$= \frac{1}{\alpha_t}\left(\boldsymbol{w}_{t+1}^T \boldsymbol{x}(S_t, A_t) - \boldsymbol{w}_t^T \boldsymbol{x}(S_t, A_t)\right), \tag{10b}$$

where $\boldsymbol{x}(S_t, A_t)$ denotes a feature vector with the same dimensions as the network parameter vector, $\boldsymbol{w}$, for the state-action pair (S_t, A_t). Next, as per Equation (4b), we can write the standard *semi-gradient* update for $\boldsymbol{w}_{t+1}$ as follows (Sutton, 1988; Sutton & Barto, 2018):

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha_t \nabla_{\boldsymbol{w}_t} \frac{1}{2}\left(R_{t+1} + \gamma \max_{a_* \in \mathcal{A}} Q_{\boldsymbol{w}_t}(S_{t+1}, a_*) - Q_{\boldsymbol{w}_t}(S_t, A_t)\right)^2 \tag{11a}$$

$$\approx \boldsymbol{w}_t + \alpha_t \delta_t^e \boldsymbol{x}(S_t, A_t). \tag{11b}$$

Finally, we can combine Equations (9), (10), and (11b) as follows:

$$\delta_t^i = \frac{1}{\alpha_t}\left(\boldsymbol{w}_{t+1}^T \boldsymbol{x}(S_t, A_t) - \boldsymbol{w}_t^T \boldsymbol{x}(S_t, A_t)\right) \tag{12a}$$

$$= \frac{1}{\alpha_t}\left((\boldsymbol{w}_t + \alpha_t \delta_t^e \boldsymbol{x}(S_t, A_t))^T \boldsymbol{x}(S_t, A_t) - \boldsymbol{w}_t^T \boldsymbol{x}(S_t, A_t)\right) \tag{12b}$$

$$= \delta_t^e \boldsymbol{x}(S_t, A_t)^T \boldsymbol{x}(S_t, A_t). \tag{12c}$$

Let us examine the implications of Equation (12c). We can see that the two TD errors are only equal to each other in the linear function approximation case if $\boldsymbol{x}(S_t, A_t)^T \boldsymbol{x}(S_t, A_t) \doteq ||\boldsymbol{x}(S_t, A_t)||^2 = 1$, or if $\delta_t^e = \delta_t^i = 0$. We note that this result validates Proposition 3.3, given that it is well-known that tabular RL methods can be viewed as a special case of linear function approximation with a feature vector that satisfies $||\boldsymbol{x}(S_t, A_t)||^2 = 1$.

As such, we have shown that even with linear function approximation, the implicit and explicit TD errors are not always equal to each other. This is formalized as Proposition 3.4 below:

Proposition 3.4. *In the linear function approximation setting, given an update batch size of 1, the explicit and implicit TD errors are only equal to each other at time t if the feature vector satisfies $||\boldsymbol{x}(S_t, A_t)||^2 = 1$, or if $\delta_t^e = \delta_t^i = 0$.*

3.2 Nonlinearity Can Cause Further Divergence Between the Two TD Errors

In the previous section, we saw that even in the linear function approximation case the two TD errors are not always equal to each other. In this section, we will show that adding even just a single nonlinearity into the function approximator complicates the picture even further. In particular, let us consider the effect of adding a single nonlinear activation function to the linear function approximator described in Section 3.1 (while still maintaining a batch size of 1), such that $Q_{\boldsymbol{w}_t}(S_t, A_t) = \sigma(\boldsymbol{w}_t^T \boldsymbol{x}(S_t, A_t))$, where σ denotes the nonlinear activation function. Applying a similar process to the one shown in Equation (12) yields:

$$\delta_t^i = \frac{1}{\alpha_t}\left[\sigma(\boldsymbol{w}_{t+1}^T \boldsymbol{x}(S_t, A_t)) - \sigma(\boldsymbol{w}_t^T \boldsymbol{x}(S_t, A_t))\right] \tag{13a}$$

$$= \frac{1}{\alpha_t}\left[\sigma\left((\boldsymbol{w}_t + \alpha_t \delta_t^e \nabla_{\boldsymbol{w}_t} \sigma(\boldsymbol{w}_t^T \boldsymbol{x}(S_t, A_t)))^T \boldsymbol{x}(S_t, A_t)\right) - \sigma(\boldsymbol{w}_t^T \boldsymbol{x}(S_t, A_t))\right]. \tag{13b}$$

As per Equation (13b), we can see that even adding a single nonlinear activation function greatly limits our ability to compare the explicit and implicit TD errors from an analytical perspective. That is, while it may be *possible* that for some choice of feature vector and (nonlinear) activation function that the expression on the right-hand side of Equation (13b) evaluates to δ_t^e, we can no longer provide such guarantees in the general case from an analytical perspective. Moreover, it is not hard to imagine how adding more layers and nonlinearities compounds this analytical divergence, ultimately making an exact (analytical) equivalence between the two TD errors increasingly unlikely.

3.3 Batch Updates Can Also Induce Divergence Between the Two TD Errors

In this section, we explore, from an analytical perspective, the effect of batch updates as it pertains to the equivalence of the two TD errors. Given that we showed in Section 3.2 how nonlinearity limits our ability to make an analytical comparison between the two TD errors, we will examine the effect of batch updates in the linear function approximation case. In particular, for a batch update of size B, we can apply a similar process to the one described in Equation (12), which yields:

$$\delta_t^i \doteq \frac{1}{B}\sum_{j=1}^{B}\delta_{j,t}^i = \frac{1}{B}\sum_{j=1}^{B}\frac{1}{\alpha_t}\left(\boldsymbol{w}_{t+1}^T\boldsymbol{x}(S_{j,t},A_{j,t}) - \boldsymbol{w}_t^T\boldsymbol{x}(S_{j,t},A_{j,t})\right) \tag{14a}$$

$$= \frac{1}{B}\sum_{j=1}^{B}\frac{1}{\alpha_t}\left[\left(\boldsymbol{w}_t + \alpha_t\frac{1}{B}\sum_{k=1}^{B}\delta_{k,t}^e\boldsymbol{x}(S_{k,t},A_{k,t})\right)^T\boldsymbol{x}(S_{j,t},A_{j,t}) - \boldsymbol{w}_t^T\boldsymbol{x}(S_{j,t},A_{j,t})\right] \tag{14b}$$

$$= \frac{1}{B}\sum_{k=1}^{B}\delta_{k,t}^e\frac{1}{B}\sum_{j=1}^{B}\boldsymbol{x}(S_{k,t},A_{k,t})^T\boldsymbol{x}(S_{j,t},A_{j,t}). \tag{14c}$$

In this case, we would consider the explicit and implicit TD errors equivalent if their batch-averaged values are the same. From Equation (14c), we can see that this occurs if:

$$\frac{1}{B}\sum_{j=1}^{B}\boldsymbol{x}(S_{k,t},A_{k,t})^T\boldsymbol{x}(S_{j,t},A_{j,t}) = 1,\ \forall k = 1,2,\ldots,B. \tag{15}$$

In words, this condition requires that for every sample k in the batch, the average inner product between its feature vector and all feature vectors in the batch (including itself) equals one. This condition is considerably more restrictive than the single-sample case, where we only required $||\boldsymbol{x}(S_t,A_t)||^2 = 1$. Consequently, this shows that even in the linear function approximation case, batch updates introduce an additional source of divergence between the explicit and implicit TD errors. This argument is formalized as Proposition 3.5 below:

Proposition 3.5. *In the linear function approximation setting with batch updates, the explicit and implicit TD errors are only equal to each other at time t if Eqn. (15) is satisfied, or if $\delta_t^e = \delta_t^i = 0$.*

3.4 An Empirical Comparison Between the Two TD Errors

In the previous sections, we showed, from an analytical perspective, how the equivalence between the explicit and implicit TD errors can break down in deep RL settings. In this section, we complement these analytical results by performing an empirical characterization of this divergence.

To this end, we compared the explicit and implicit TD errors when applying Q-learning algorithms on two RL tasks. In the first task, we used a Q-learning algorithm on the well-known *inverted pendulum* environment. Then, in the second task, we used a DQN algorithm (Mnih et al., 2015) on the Atari *Breakout* environment (Bellemare et al., 2013). In both experiments, we compared the two TD errors when introducing increased nonlinearity into the function approximator. In particular, in the inverted pendulum experiment, we compared the two TD errors when using a linear function approximator (with no batch updates) vs. a function approximator with two hidden layers and batch updates. Similarly, in the Breakout experiment, we compared the two TD errors when using a value network with 0.4M parameters vs. a value network with 6.7M parameters (in both cases, we used batch updates, and the value network had three convolutional layers and two fully connected layers).

Figure 1 shows the results of this comparison. In particular, the figure shows the *TD error absolute difference* (i.e., $|\delta_t^e - \delta_t^i|$) as learning progresses in both tasks. As shown in Figure 1a), even in linear function approximation settings, there is still divergence between the explicit and implicit TD errors. We can also see that, in both experiments, this divergence is only intensified by the inclusion of more nonlinearity into the function approximator.

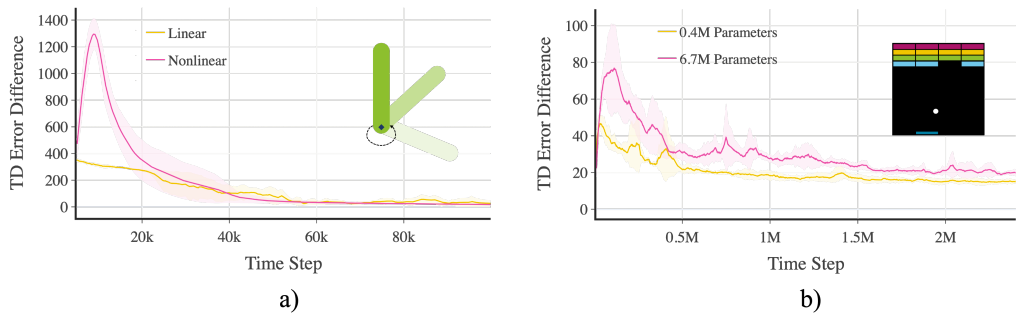

Figure 1: Rolling average of the TD error absolute difference (i.e., $|\delta_t^e - \delta_t^i|$) as learning progresses when using **a)** a Q-learning algorithm in the inverted pendulum environment, and **b)** a DQN algorithm in the Atari Breakout environment. A solid line denotes the mean TD error absolute difference, and a shaded region denotes a 95% confidence interval over 4 runs.

4 The Choice of TD Error Matters

In the previous section, we showed how increasingly nonlinear deep RL architectures can cause the two interpretations of the TD error to yield increasingly different numerical values. In this section, we show how choosing one interpretation of the TD error over the other can affect the performance of deep RL algorithms that utilize the TD error to compute quantities beyond the value function.

In particular, we will focus on evaluating the effect of using the explicit vs. implicit TD error in deep differential Q-learning algorithms (Wan et al., 2021), Q-learning algorithms with value-based reward centering (Naik et al., 2024), and A2C algorithms (Mnih et al., 2016). Each of these algorithms utilizes the TD error to compute quantities beyond the value function. In differential and reward centering algorithms, the TD error is used to compute a running estimate of the average of the one-step rewards. In A2C algorithms, the TD error is used as a proxy for the advantage function in the actor loss. We discuss the role of the TD error in each class of algorithm in more detail below:

4.1 Differential Algorithms and the TD Error

In differential RL algorithms (Wan et al., 2021), the agent aims to find a policy that optimizes the long-run (or limiting) average-reward, $\bar{r}$, which is defined as follows for a given policy, π:

$$\bar{r}_\pi(s) \doteq \lim_{n\to\infty} \frac{1}{n} \sum_{t=1}^{n} \mathbb{E}[R_t \mid S_0 = s, A_{0:t-1} \sim \pi]. \tag{16}$$

Typically, differential algorithms invoke *ergodicity-like* assumptions (e.g. see Wan et al. (2021)) so that the average-reward objective (16) remains well-defined and becomes independent of initial conditions (i.e., $\bar{r}_\pi(s) = \bar{r}_\pi$). To optimize the average-reward (16), differential methods optimize *differential value functions*, such as the state-action differential value function:

$$Q_\pi(s,a) \doteq \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} (R_{t+1} - \bar{r}_\pi) \mid S_0 = s, A_0 = a, A_{1:\infty} \sim \pi \right]. \tag{17}$$

Crucially, optimizing the differential value function (17) provides a means to indirectly optimize the average-reward objective (16), such that optimizing the differential value function indirectly optimizes the average-reward objective (16).

Importantly, to optimize a differential value function, differential algorithms need to maintain a running estimate of the average-reward. From an algorithmic perspective, this means that the algorithms start with initial estimates (or guesses) for the value function *and* average-reward, then update these estimates over time, until they have simultaneously learned and/or optimized these two objectives. There are different ways through which the average-reward estimate can be updated. One way is by using empirical reward samples from the environment (e.g. Adamczyk et al. (2025)). Another such

way is via the TD error. For instance, a tabular Differential Q-learning algorithm (Wan et al., 2021), which updates the average-reward estimate with the TD error, is shown below:

$$\delta_t = R_{t+1} - \bar{R}_t + \max_{a_*} Q_t(S_{t+1}, a_*) - Q_t(S_t, A_t) \tag{18a}$$

$$Q_{t+1}(S_t, A_t) = Q_t(S_t, A_t) + \alpha_t \delta_t \tag{18b}$$

$$Q_{t+1}(s, a) = Q_t(s, a), \quad \forall (s, a) \neq (S_t, A_t) \tag{18c}$$

$$\bar{R}_{t+1} = \bar{R}_t + \eta \alpha_t \delta_t \,, \tag{18d}$$

where $\bar{R}_t$ denotes the average-reward estimate and $\eta > 0$ denotes a step size parameter.

As per Proposition 3.3, the choice of TD error (δ_t^e vs. δ_t^i) by which to update the average-reward estimate (i.e., in Equation (18d)) does not matter in the tabular case. However, we will now show that in deep RL settings, the choice of explicit vs. implicit TD error for the average-reward update can have a significant effect. In particular, consider Figure 2, which shows the average-reward estimate when using a deep differential Q-learning algorithm with explicit vs. implicit TD error average-reward updates (see Algorithms 1 and 4 for the full pseudocode) in the Atari Breakout environment. As shown in the figure, using the explicit TD error to update the average-reward estimate can cause the estimate to diverge. Conversely, using the implicit TD error mitigates this instability.

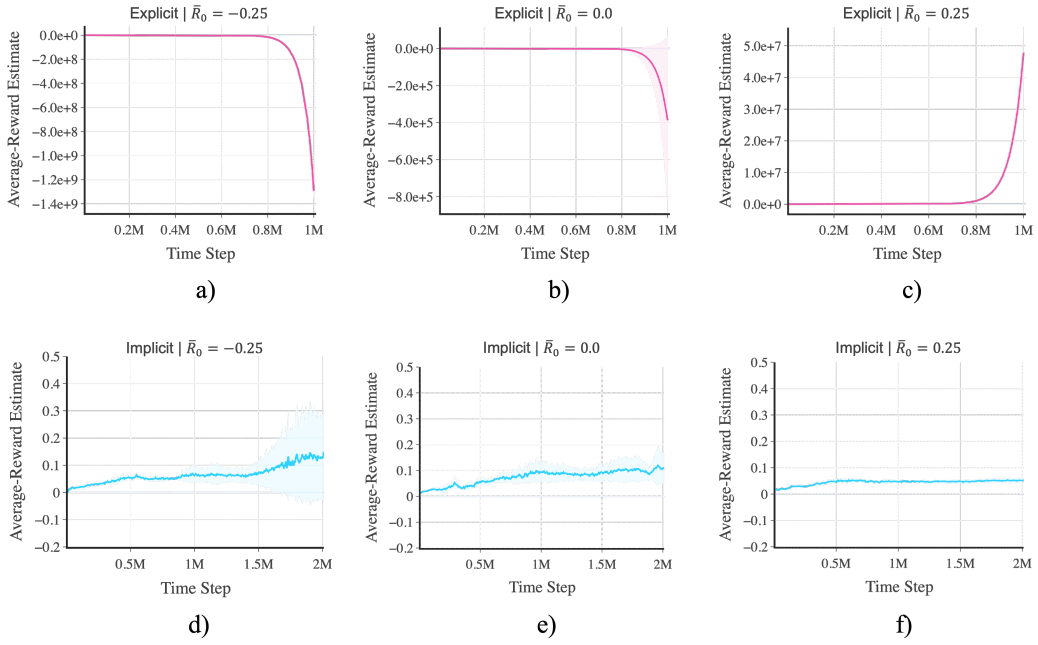

Figure 2: Rolling average-reward estimates when using a deep differential Q-learning algorithm in the Breakout environment. Figures **a)**, **b)**, and **c)** utilize an *explicit* TD error average-reward update with an initial guess of -0.25, 0.0, and 0.25, respectively. Figures **d)**, **e)**, and **f)** utilize an *implicit* TD error update with an initial guess of -0.25, 0.0, and 0.25, respectively. A solid line denotes the mean average-reward estimate, and a shaded region denotes a 95% confidence interval over 4 runs.

Indeed, the choice of TD error has a drastic effect on TD-based average-reward updates in deep RL settings. But why? In Appendix A, we perform a theoretical analysis on this topic, and find that there is theoretical justification for why updating the average-reward estimate with the explicit TD error can cause divergence, as well as why using the implicit TD error mitigates this instability.

Importantly, the above results motivate the use of implicit TD error-based average-reward updates in deep differential algorithms. To the best of our knowledge, our work is the first to provide theoretical justification for the instability caused by explicit TD error-based average-reward updates, as well as the first to propose using implicit TD error-based average-reward updates to mitigate this issue (along with theoretical justification; see Appendix A). Consequently, our work is only the second to provide a deep differential Q-learning algorithm with stable TD error-based average-reward updates.

The first, to the best of our knowledge, was the baseline used in Rojas & Lee (2026), which used the smallest-magnitude explicit TD error in a batch, rather than the batch-average explicit TD error. In Section 4.4, we compare the performance of our proposed differential Q-learning algorithm with implicit TD error updates to that of the smallest-magnitude baseline used in Rojas & Lee (2026).

4.2 Reward Centering Algorithms and the TD Error

In reward centering algorithms (Naik et al., 2024; De Asis et al., 2026), the agent aims to find a policy that optimizes the discounted value function (i.e., Equation (2)), such that all the one-step rewards are "centered" by subtracting their average. For example, in the context of Q-learning, we have that:

$$Q_\pi(s,a) \doteq \mathbb{E}_\pi\left[\sum_{t=0}^{T} \gamma^t (R_{t+1} - \bar{r}_\pi) \mid S_0 = s, A_0 = a, A_{1:T} \sim \pi\right]. \tag{19}$$

Like the differential algorithms discussed in Section 4.1, the average-reward, $\bar{r}_\pi$, needs to be estimated in reward centering algorithms. In *value-based reward centering*, the estimate is updated via the TD error. Crucially, one can show, using an identical process to the one described in Section 4.1 (and Appendix A) for differential algorithms, that value-based reward centering algorithms can be unstable when utilizing batch-averaged explicit TD error average-reward updates in increasingly deep RL settings, and that utilizing batch-averaged implicit TD error updates mitigates this instability. In Section 4.4, we compare the empirical performance when using explicit vs. implicit batch-averaged TD error updates in reward centering algorithms.

4.3 A2C Algorithms and the TD Error

In Advantage Actor-Critic (A2C) algorithms (Mnih et al., 2016), the agent aims to optimize a parameterized policy (the actor) alongside a parameterized state-value function (the critic, $V_{\boldsymbol{w}_t}(S_t)$). The critic is trained to approximate the value function, while the actor is trained using an estimate of the *advantage function*, which measures, at a given state, how much better taking a specific action is relative to the average performance under the current policy at that state. In particular, A2C algorithms use the TD error as a proxy for the advantage. That is, the actor is trained by minimizing a loss function in which the TD error acts as an estimate of the advantage, such that:

$$\mathcal{L}_{\text{actor}}(\boldsymbol{u}_t) \doteq -\log \pi_{\boldsymbol{u}_t}(A_t \mid S_t)\, \delta_t\,, \tag{20}$$

where $\pi_{\boldsymbol{u}_t}$ denotes the parameterized policy and $\boldsymbol{u}_t$ denotes the policy parameters at time t.

A2C algorithms that utilize the explicit TD error (in this case, in the actor loss) have been shown to yield good performance in deep RL settings (Mnih et al., 2016), and do not suffer from the same instability issues as the deep differential and (value-based) reward centering algorithms described in Sections 4.1 and 4.2, respectively. As such, we will focus our exploration of A2C algorithms in terms of comparing the empirical performance when using the explicit vs. implicit TD errors as estimates of the advantage. This empirical comparison is done in Section 4.4.

4.4 The Empirical Implications of Using the Explicit vs. Implicit TD Errors

In this section, we perform an empirical evaluation of how using the explicit TD error vs. the implicit TD error affects the performance of deep differential, reward centering, and A2C algorithms. In particular, for each class of algorithm, we compared how the algorithm performs in Atari or MuJoCo (Todorov et al., 2012) environments when using the explicit vs. implicit TD error to compute quantities beyond the value function (i.e., the average-reward or advantage estimates). We tested the differential algorithms in the Atari *Breakout* environment, the reward centering algorithms in the Atari *Pong* environment, and the A2C algorithms in the MuJoCo *Ant* environment.

Figure 3 shows the results of this comparison. In particular, Figure 3a) shows that for differential algorithms, using the implicit TD error to update the average-reward estimate yields superior performance in the early stages of learning, precisely when the divergence between the two TD errors is largest (see Figure 1). We can also see (in Figure 3a)) that using the implicit TD error to update the average-reward estimate results in more stable performance than when using the smallest-magnitude explicit TD error average-reward update from Rojas & Lee (2026). In Figure 3b), we can see that, for

value-based reward centering algorithms, using the implicit TD error to update the average-reward estimate yields superior performance. We can also see that, in this case, using the explicit TD error to update the average-reward estimate results in stable performance. This is consistent with the findings from Naik et al. (2024), which suggest that, in some cases, using the explicit TD error to update the average-reward estimate can be a viable option given sufficient step size tuning. Finally, Figure 3c) shows that for A2C algorithms, using the implicit TD error as the advantage estimate yields superior and more stable performance than when using the explicit TD error as the advantage estimate. The full set of experiment details and results can be found in Appendix C.

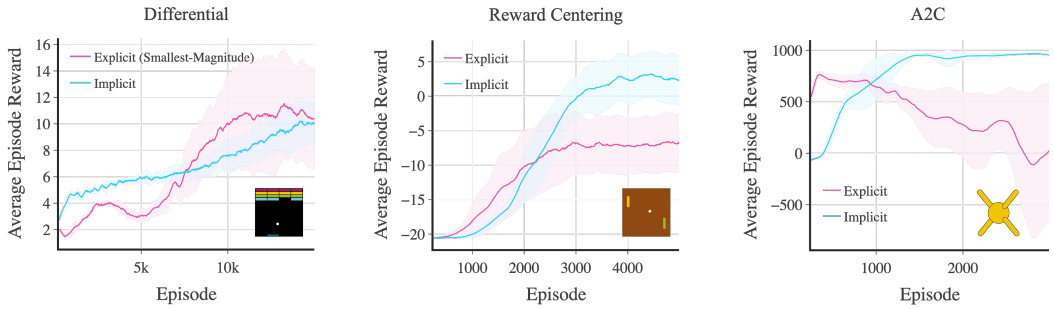

Figure 3: Rolling total reward per episode when using: **a)** a differential Q-learning algorithm in the Breakout environment, **b)** a Q-learning algorithm with value-based reward centering in the Pong environment, and **c)** an A2C algorithm in the Ant environment. Each plot shows the performance of each algorithm when using the explicit vs. implicit TD error. A solid line denotes the mean total reward per episode, and a shaded region denotes a 95% confidence interval over 8 runs.

5 Discussion, Limitations, and Future Work

In this work, we performed a formal exploration and characterization of the differences between two seemingly equivalent interpretations of the TD error: the TD error as the difference between temporally successive predictions (the *implicit* TD error), and the TD error as the difference between a bootstrapped target and a prediction (the *explicit* TD error). We showed that although these two interpretations are equivalent in tabular settings, increasingly nonlinear deep RL architectures can cause these two interpretations of the TD error to yield increasingly different numerical values. We then leveraged this insight to show how choosing one interpretation over the other can affect the stability and overall performance of deep RL algorithms that utilize the TD error to compute quantities beyond the value function, including differential (i.e., average-reward), reward centering, and actor-critic algorithms. Altogether, our results show that the default interpretation of the TD error as the difference between a bootstrapped target and a prediction does not always hold in deep RL settings. In other words, *the choice of implicit vs. explicit TD error matters in deep RL settings*.

In terms of limitations and future work, we note that our analytical investigation was limited to showing the existence of a difference between the two TD errors in increasingly nonlinear deep RL settings. Although we formally quantified this difference in linear function approximation settings, quantifying (or bounding) this difference in deep RL settings remains an open research question. In this regard, the type of nonlinear activations used, the number and type of layers used, the size of batch updates, and the type of optimizer used (e.g. stochastic gradient descent vs. Adam (Kingma & Ba, 2015)) are all possible sources of divergence between the two TD errors in deep RL settings.

More broadly, this work opens the door to several interesting research directions that could be pursued in future work. Namely, now that we have formalized the notion of the implicit TD error in deep RL settings, can we design deep RL algorithms that directly optimize this objective rather than, or alongside, the explicit TD error? Are there other ways in which the implicit TD error can be incorporated into deep RL algorithms to facilitate learning? Indeed, the implications of the implicit TD error in deep RL settings present a compelling direction for future research, one that suggests that the story of the TD error as it pertains to RL has only just begun.

AI USE STATEMENT

In this work, we used generative AI tools to improve readability of the paper by asking the AI tools to look for typos and grammatical errors in the writing, as well as inconsistencies in the notation. We have *not* used generative AI tools to generate synthetic datasets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, or interpret results. We have reviewed all AI-assisted work, and take responsibility for the content presented in this paper, including text, claims, and/or artifacts produced with the aid of generative AI.

REFERENCES

Jacob Adamczyk, Volodymyr Makarenko, Stas Tiomkin, and Rahul V Kulkarni. Average-reward soft actor-critic. *Reinforcement Learning Journal*, 2025.

Marc G Bellemare, Yavar Naddaf, J Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.*, pp. 253–279, 2013.

Kris De Asis, Mohamed Elsayed, and Jiamin He. Extending differential temporal difference methods for episodic problems. *Reinf. Learn. J.*, 2026.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning*, 2018.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

Abhishek Naik, Yi Wan, Manan Tomar, and Richard S Sutton. Reward centering. *Reinforcement Learning Journal*, 4:1995–2016, 2024.

Martin L Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.

Juan Sebastian Rojas and Chi-Guhn Lee. A differential perspective on distributional reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2026.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, 3:944, 1988.

Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction, 2nd edition*. MIT Press, November 2018.

Gerald Tesauro. Practical issues in temporal difference learning. *Mach. Learn.*, 8(3-4):257–277, May 1992.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, October 2012.

Yi Wan, Abhishek Naik, and Richard S Sutton. Learning and planning in average-reward markov decision processes. In *Proceedings of the 38th International Conference on Machine Learning*, 2021.

Christopher J Watkins and Peter Dayan. Q-learning. *Mach. Learn.*, 8:279–292, 1992.

Shangtong Zhang, Yi Wan, Richard S Sutton, and Shimon Whiteson. Average-reward off-policy policy evaluation with function approximation. In *Proceedings of the 38th International Conference on Machine Learning*, 2021.

A AN ANALYSIS OF EXPLICIT AND IMPLICIT TD ERROR-BASED AVERAGE-REWARD UPDATES IN DEEP DIFFERENTIAL RL ALGORITHMS

In this appendix, we perform a theoretical exploration and analysis of the results observed in Figure 2 for deep differential RL algorithms that utilize TD-based average-reward updates. In particular, we explore why updating the average-reward with the explicit TD error causes the observed instability, and why updating the average-reward with the implicit TD error mitigates this instability:

Indeed, the choice of TD error has a drastic effect on TD-based average-reward updates in deep RL settings. But why? A close examination of the convergence proof for the *tabular* Differential Q-learning algorithm (Wan et al., 2021) reveals some insight. In particular, the convergence proof utilized a TD error representation that could be interpreted as an "n-step variant" of the implicit TD error (see Equation B.12 of Wan et al. (2021)). That is, the average-reward estimate, $\bar{R}_t$, was shown to be a reliable estimate of the actual average-reward, $\bar{r}_\pi$, when updated with the TD error, but the specific interpretation of the TD error used to make that claim was that of the implicit TD error.

Accordingly, if the explicit TD error is used to update the average-reward estimate in deep RL settings, the divergence between the two TD errors degrades the quality of the average-reward estimate, and introduces instability and nonstationarity into the explicit TD error target. More formally, let $\epsilon_t \doteq \delta^e_t - \delta^i_t$ denote the difference between the two TD errors at time t, such that if at time t the average-reward estimate is updated with the explicit TD error, we have that:

$$\begin{aligned}
\bar{R}_{t+1} &= \bar{R}_t + \eta\alpha_t\delta^e_t && \text{(A.1a)}\\
&= \bar{R}_t + \eta\alpha_t(\delta^i_t + \epsilon_t) && \text{(A.1b)}\\
&= \bar{R}_t + \eta\alpha_t\delta^i_t + \eta\alpha_t\epsilon_t && \text{(A.1c)}\\
&\doteq \bar{R}^i_{t+1} + \eta\alpha_t\epsilon_t, && \text{(A.1d)}
\end{aligned}$$

where the $\bar{R}^i_{t+1} \doteq \bar{R}_t + \eta\alpha_t\delta^i_t$ term constitutes the average-reward update that would have been made if the estimate was updated via the implicit TD error. Hence, if there is large divergence between the two TD errors, as we have seen is the case in increasingly deep RL settings, then the otherwise-reliable estimate of the average-reward is degraded by this $\eta\alpha_t\epsilon_t$ term. Moreover, since the average-reward estimate is used to compute the explicit TD error target, this divergence introduces additional instability and nonstationarity into the agent's learning. That is, we have that:

$$\begin{aligned}
\delta^e_t &\doteq \frac{1}{B}\sum_{j=1}^{B}\left[R_{j,t+1} - \bar{R}_t + \max_{a_*} Q_{\boldsymbol{w}_t}(S_{j,t+1}, a_*) - Q_{\boldsymbol{w}_t}(S_{j,t}, A_{j,t})\right] && \text{(A.2a)}\\
&= \frac{1}{B}\sum_{j=1}^{B}\left[R_{j,t+1} - \bar{R}^i_t - \eta\alpha_{t-1}\epsilon_{t-1} + \max_{a_*} Q_{\boldsymbol{w}_t}(S_{j,t+1}, a_*) - Q_{\boldsymbol{w}_t}(S_{j,t}, A_{j,t})\right], && \text{(A.2b)}
\end{aligned}$$

such that the $\eta\alpha_{t-1}\epsilon_{t-1}$ term creates a source of instability and nonstationarity when learning not just the average-reward, but also the value function. Critically, if this degraded explicit TD error is then used to update the estimate of the average-reward in subsequent time steps, then the error in the average-reward estimate will compound, such that after n updates:

$$\bar{R}_n = \bar{R}_0 + \eta\sum_{t=0}^{n-1}\alpha_t\delta^i_t + \eta\sum_{t=0}^{n-1}\alpha_t\epsilon_t. \tag{A.3}$$

This compounded error term can hence be viewed as one of the sources of instability in the average-reward estimates displayed in Figure 2 for deep differential algorithms that utilize the explicit TD error to update the average-reward estimate. To the best of our knowledge, this result constitutes the first theoretically driven explanation for the instability of deep differential algorithms with TD error-based average-reward updates in control (i.e., optimization) settings (for the prediction setting, there exists a prior work, Zhang et al. (2021), which performed a theoretical analysis that was based on the explicit TD error interpretation). We emphasize that this may only be one of many possible sources of instability in deep differential algorithms, and that further theoretical work may be required to explore other sources of instability.

B Deep RL Algorithms with Implicit TD Error Updates

In this appendix, we provide the pseudocode for our proposed deep RL algorithms with implicit TD error updates. As is common practice in deep RL settings, we adopt the smooth L1 loss, $L^\lambda(x)$, (Mnih et al., 2015) as the critic loss in our algorithms:

$$L^\lambda(x) = \begin{cases} \frac{1}{2\lambda}x^2, & \text{if } |x| \leq \lambda \\ |x| - \frac{1}{2}\lambda, & \text{if } |x| > \lambda. \end{cases} \tag{B.1}$$

Algorithm 1 Differential DQN with Implicit TD Error Average-Reward Updates

Input: a differentiable state-action value function parameterization: $\hat{q}(s, a, \boldsymbol{w})$ (with target network $\hat{q}_{\boldsymbol{T}}(s, a, \boldsymbol{w}_{\boldsymbol{T}})$), the policy π to be used (e.g., ε-greedy)
Algorithm parameters: step size parameters $\{\alpha, \eta\}$, loss parameter λ
Initialize state-action value weights $\boldsymbol{w}, \boldsymbol{w}_{\boldsymbol{T}} \in \mathbb{R}^d$ arbitrarily (e.g. to $\mathbf{0}$)
Initialize $\bar{R}$ arbitrarily (e.g. to zero)
Obtain initial S
while still time to train **do**
  $A \leftarrow$ action given by π for S
  Take action A, observe R, S'
  Store (S, A, R, S') in replay buffer
  if time to update estimates **then**
    Sample a minibatch of B transitions from replay buffer: $\{(S_b, A_b, R_b, S'_b)\}_{b=1}^B$
    For each b-th transition: $\delta_b^e = R_b - \bar{R} + \max_a \hat{q}_{\boldsymbol{T}}(S'_b, a, \boldsymbol{w}_{\boldsymbol{T}}) - \hat{q}(S_b, A_b, \boldsymbol{w})$
    $\ell = \frac{1}{B}\sum_{b=1}^B L^\lambda(\delta_b^e)$ (See Equation (B.1))
    $\boldsymbol{w'} = \boldsymbol{w} - \alpha\frac{\partial \ell}{\partial \boldsymbol{w}}$
    For each b-th transition: $\delta_b^i = \frac{1}{\alpha}\left(\hat{q}(S_b, A_b, \boldsymbol{w'}) - \hat{q}(S_b, A_b, \boldsymbol{w})\right)$
    $\bar{R} = \bar{R} + \eta\alpha\frac{1}{B}\sum_{b=1}^B \delta_b^i$
    $\boldsymbol{w} = \boldsymbol{w'}$
    Update $\boldsymbol{w}_{\boldsymbol{T}}$ as needed (e.g. using Polyak averaging)
  end if
  $S = S'$
end while
return $\boldsymbol{w}$

Algorithm 2 Value-Based Reward Centering DQN with Implicit TD Error Average-Reward Updates

Input: a differentiable state-action value function parameterization: $\hat{q}(s, a, \boldsymbol{w})$ (with target network $\hat{q}_{\boldsymbol{T}}(s, a, \boldsymbol{w}_{\boldsymbol{T}})$), the policy π to be used (e.g., ε-greedy)
Algorithm parameters: discount factor γ, step size parameters $\{\alpha, \eta\}$, loss parameter λ
Initialize state-action value weights $\boldsymbol{w}, \boldsymbol{w}_{\boldsymbol{T}} \in \mathbb{R}^d$ arbitrarily (e.g. to $\mathbf{0}$)
Initialize $\bar{R}$ arbitrarily (e.g. to zero)
Obtain initial S
while still time to train **do**
  $A \leftarrow$ action given by π for S
  Take action A, observe R, S'
  Store (S, A, R, S') in replay buffer
  if time to update estimates **then**
    Sample a minibatch of B transitions from replay buffer: $\{(S_b, A_b, R_b, S'_b)\}_{b=1}^{B}$
    For each b-th transition: $\delta_b^e = R_b - \bar{R} + \gamma \max_a \hat{q}_{\boldsymbol{T}}(S'_b, a, \boldsymbol{w}_{\boldsymbol{T}}) - \hat{q}(S_b, A_b, \boldsymbol{w})$
    $\ell = \frac{1}{B} \sum_{b=1}^{B} L^{\lambda}(\delta_b^e)$ (See Equation (B.1))
    $\boldsymbol{w'} = \boldsymbol{w} - \alpha \frac{\partial \ell}{\partial \boldsymbol{w}}$
    For each b-th transition: $\delta_b^i = \frac{1}{\alpha} \left(\hat{q}(S_b, A_b, \boldsymbol{w'}) - \hat{q}(S_b, A_b, \boldsymbol{w})\right)$
    $\bar{R} = \bar{R} + \eta\alpha \frac{1}{B} \sum_{b=1}^{B} \delta_b^i$
    $\boldsymbol{w} = \boldsymbol{w'}$
    Update $\boldsymbol{w}_{\boldsymbol{T}}$ as needed (e.g. using Polyak averaging)
  end if
  $S = S'$
end while
return $\boldsymbol{w}$

Algorithm 3 A2C with Implicit TD Error Advantage Estimate

Input: a differentiable state-value function parameterization $\hat{v}(s, \boldsymbol{w})$, a differentiable policy parameterization $\pi(a \mid s, \boldsymbol{u})$
Algorithm parameters: discount factor γ, step size parameters $\{\alpha, \eta\}$
Initialize state-value weights $\boldsymbol{w} \in \mathbb{R}^d$ and policy weights $\boldsymbol{u} \in \mathbb{R}^{d'}$ (e.g. to $\mathbf{0}$)
Obtain initial S
while still time to train **do**
  $A \sim \pi(\cdot \mid S, \boldsymbol{u})$
  Take action A, observe R, S'
  $\delta^e = R + \gamma \hat{v}(S', \boldsymbol{w}) - \hat{v}(S, \boldsymbol{w})$
  $\boldsymbol{w'} = \boldsymbol{w} + \alpha \delta^e \nabla \hat{v}(S, \boldsymbol{w})$
  $\delta^i = \frac{1}{\alpha} \left(\hat{v}(S, \boldsymbol{w'}) - \hat{v}(S, \boldsymbol{w})\right)$
  $\boldsymbol{u} = \boldsymbol{u} + \eta\alpha \delta^i \nabla \ln \pi(A \mid S, \boldsymbol{u})$
  $\boldsymbol{w} = \boldsymbol{w'}$
  $S = S'$
end while
return $\boldsymbol{w}$ and $\boldsymbol{u}$

C EXPERIMENT DETAILS

This appendix contains details regarding the experiments performed as part of this work.

We ran three groups of experiments. In the first group of experiments (Section C.1), we aimed to show, from an empirical perspective, how the explicit and implicit TD errors can yield increasingly different numerical values in increasingly nonlinear deep RL settings. In the second group of experiments (Section C.2), we aimed to show, from an empirical perspective, how the choice of explicit vs. implicit TD error for the average-reward update (i.e., Equation (18d)) in deep differential (i.e., average-reward) algorithms can have a significant effect on the stability of the average-reward estimate. In the third group of experiments (Section C.3), we aimed to show, from an empirical perspective, how using the explicit TD error vs. the implicit TD error affects the performance of deep differential, reward centering (Naik et al., 2024), and A2C (Mnih et al., 2016) algorithms.

C.1 TD ERROR DIFFERENCE EXPERIMENTS

In this section, we discuss the TD error difference experiments. Through these experiments, we aimed to show, from an empirical perspective, how the explicit and implicit TD errors can yield increasingly different numerical values in increasingly nonlinear deep RL settings.

To this end, we compared the explicit and implicit TD errors when applying Q-learning algorithms on two RL tasks. In the first task, we used a Q-learning algorithm on the well-known *inverted pendulum* environment, where an agent learns how to optimally balance an inverted pendulum. Then, in the second task, we used a DQN algorithm (Mnih et al., 2015) on the Atari *Breakout* environment (Bellemare et al., 2013). Both environments are displayed in Figure C.1. In both experiments, we compared the two TD errors when introducing increased nonlinearity into the function approximator.

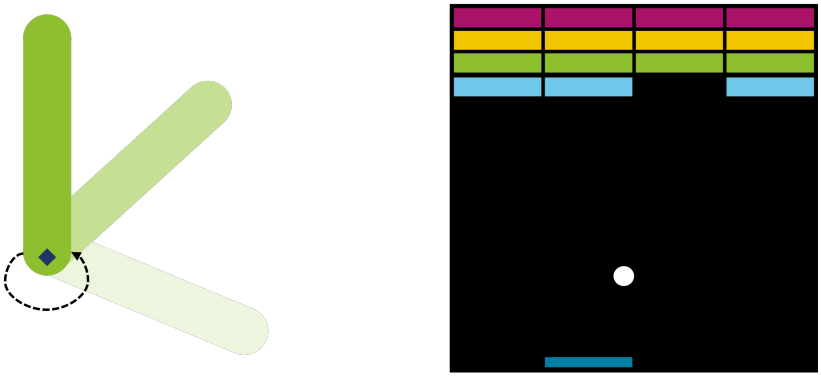

Figure C.1: An illustration of the **a)** inverted pendulum, and **b)** Atari Breakout environments.

In the inverted pendulum experiment, we compared the two TD errors when using a linear function approximator (with no batch updates) vs. a function approximator with two hidden layers and batch updates. For the linear function approximator, we used tile coding (Sutton & Barto, 2018) with 32 tilings, each with 8 X 8 tiles. For the nonlinear function approximator, we used the following network with *hidden_dim* = 32:

```
import torch

class ValueNetwork(torch.nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim):
        super(ValueNetwork, self).__init__()
        self.linear_1 = torch.nn.Linear(state_dim, hidden_dim)
        self.linear_2 = torch.nn.Linear(hidden_dim, hidden_dim)
```

```
        self.linear_3 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = torch.nn.functional.relu(self.linear_1(x))
        x = torch.nn.functional.relu(self.linear_2(x))
        return self.linear_3(x)
```

For this nonlinear function approximator, we used a replay buffer with a minimum size of 1e2, a maximum size of 1e5, and a sample batch size of 32. We used a target network, and updated it using Polyak averaging with a fixed step size of 0.005. For both approximators, we used stochastic gradient descent updates on the standard mean squared Bellman error (i.e., Equation (4a)), a value function step size of 2e-4, a discount factor of 0.99, and an ε-greedy policy with $\varepsilon = 0.1$.

In the Breakout experiment, we compared the two TD errors when using the following "small" and "large" networks, both of which consist of three convolutional layers and two fully connected layers:

"Small" network:

```
import torch

class ValueNetwork(torch.nn.Module):
    def __init__(self, state_dim, action_dim):
        super(ValueNetwork, self).__init__()
        self.conv1 = torch.nn.Conv2d(state_dim, 16, kernel_size=8, stride=4)
        self.conv2 = torch.nn.Conv2d(16, 32, kernel_size=4, stride=2)
        self.conv3 = torch.nn.Conv2d(32, 32, kernel_size=3, stride=1)
        self.fc1 = torch.nn.Linear(32 * 7 * 7, 256)
        self.fc2 = torch.nn.Linear(256, action_dim)

    def forward(self, x):
        x = torch.nn.functional.relu(self.conv1(x))
        x = torch.nn.functional.relu(self.conv2(x))
        x = torch.nn.functional.relu(self.conv3(x))
        x = x.view(x.size(0), -1)
        x = torch.nn.functional.relu(self.fc1(x))
        return self.fc2(x)
```

"Large" network:

```
import torch

class ValueNetwork(torch.nn.Module):
    def __init__(self, state_dim, action_dim):
        super(ValueNetwork, self).__init__()
        self.conv1 = torch.nn.Conv2d(state_dim, 64, kernel_size=8, stride=4)
        self.conv2 = torch.nn.Conv2d(64, 128, kernel_size=4, stride=2)
        self.conv3 = torch.nn.Conv2d(128, 128, kernel_size=3, stride=1)
        self.fc1 = torch.nn.Linear(128 * 7 * 7, 1024)
        self.fc2 = torch.nn.Linear(1024, action_dim)

    def forward(self, x):
        x = torch.nn.functional.relu(self.conv1(x))
        x = torch.nn.functional.relu(self.conv2(x))
        x = torch.nn.functional.relu(self.conv3(x))
        x = x.view(x.size(0), -1)
        x = torch.nn.functional.relu(self.fc1(x))
        return self.fc2(x)
```

These networks vary by the number of learnable parameters, such that the "small" network consists of 0.4M parameters, and the "large" network consists of 6.7M parameters. The calculation of the number of parameters is shown for both networks in Tables 1 and 2, respectively:

Layer	Input Shape	Output Shape	Calculation Formula	Parameters
conv1	$4 \times 84 \times 84$	$16 \times 20 \times 20$	$(16 \times 4 \times 8 \times 8) + 16$	4,112
conv2	$16 \times 20 \times 20$	$32 \times 9 \times 9$	$(32 \times 16 \times 4 \times 4) + 32$	8,224
conv3	$32 \times 9 \times 9$	$32 \times 7 \times 7$	$(32 \times 32 \times 3 \times 3) + 32$	9,248
fc1	1,568	256	$(256 \times 1,568) + 256$	401,664
fc2	256	4	$(4 \times 256) + 4$	1,028
Total				**424,276**

Table 1: Network Parameter Calculation for "Small" Network

Layer	Input Shape	Output Shape	Calculation Formula	Parameters
conv1	$4 \times 84 \times 84$	$64 \times 20 \times 20$	$(64 \times 4 \times 8 \times 8) + 64$	16,448
conv2	$64 \times 20 \times 20$	$128 \times 9 \times 9$	$(128 \times 64 \times 4 \times 4) + 128$	131,200
conv3	$128 \times 9 \times 9$	$128 \times 7 \times 7$	$(128 \times 128 \times 3 \times 3) + 128$	147,584
fc1	6,272	1,024	$(1,024 \times 6,272) + 1,024$	6,423,552
fc2	1,024	4	$(4 \times 1,024) + 4$	4,100
Total				**6,722,884**

Table 2: Network Parameter Calculation for "Large" Network

For both networks, we used the Adam optimizer (Kingma & Ba, 2015) with the default hyperparameters for the decay and stability (i.e., beta and epsilon) on the smooth L1 loss (Equation (B.1)) with a λ of 1.0. We used a replay buffer with a minimum size of 1e2, a maximum size of 1e5, and a sample batch size of 32. We used a target network, and updated it using Polyak averaging with a fixed step size of 0.005. We used a value function step size of 2e-4, a discount factor of 0.99, and an ε-greedy policy with $\varepsilon = 0.1$.

C.2 Average-Reward Estimate Experiments

In this section, we discuss the average-reward estimate experiments performed as part of this work. Through these experiments, we aimed to show, from an empirical perspective, how the choice of explicit vs. implicit TD error for the average-reward update in deep differential (i.e., average-reward) algorithms can have a significant effect on the stability of the average-reward estimate.

To this end, we compared a deep differential Q-learning algorithm that utilizes *implicit* TD error average-reward updates (Algorithm 1) against a deep differential Q-learning algorithm that utilizes *explicit* TD error average-reward updates (Algorithm 4).

For these experiments, we used the following network architecture:

```
import torch

class ValueNetwork(torch.nn.Module):
    def __init__(self, state_dim, action_dim):
        super(ValueNetwork, self).__init__()
        self.conv1 = torch.nn.Conv2d(state_dim, 32, kernel_size=8, stride=4)
        self.conv2 = torch.nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv3 = torch.nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.fc1 = torch.nn.Linear(64 * 7 * 7, 512)
        self.fc2 = torch.nn.Linear(512, action_dim)

    def forward(self, x):
        x = torch.nn.functional.relu(self.conv1(x))
        x = torch.nn.functional.relu(self.conv2(x))
        x = torch.nn.functional.relu(self.conv3(x))
        x = x.view(x.size(0), -1)
        x = torch.nn.functional.relu(self.fc1(x))
        return self.fc2(x)
```

For both algorithms, we used the Adam optimizer with the default hyperparameters for the decay and stability (beta and epsilon) on the smooth L1 loss (Equation (B.1)) with a λ of 1.0. We used a replay buffer with a minimum size of 1e2, a maximum size of 1e5, and a sample batch size of 32. We used a target network, and updated it using Polyak averaging with a fixed step size of 0.005. We used a value function step size, α, of 2e-5, an average-reward step size, $\eta\alpha$, with an η of 1.0, and an ε-greedy policy with $\varepsilon = 0.1$. We ran the experiment for different initial average-reward estimate guesses: $\bar{R}_0 \in \{$-0.25, 0.0, 0.25$\}$.

As shown in Figure 2, utilizing the explicit TD error to update the average-reward estimate can cause the average-reward estimate to diverge. Conversely, the figure shows that using the implicit TD error mitigates this instability. To complement these results, Figure C.2 shows the effect that this instability has on the overall performance of the algorithm. As shown in the figure, the instability caused by the explicit TD error average-reward updates results in significantly lower performance than when using implicit TD error average-reward updates.

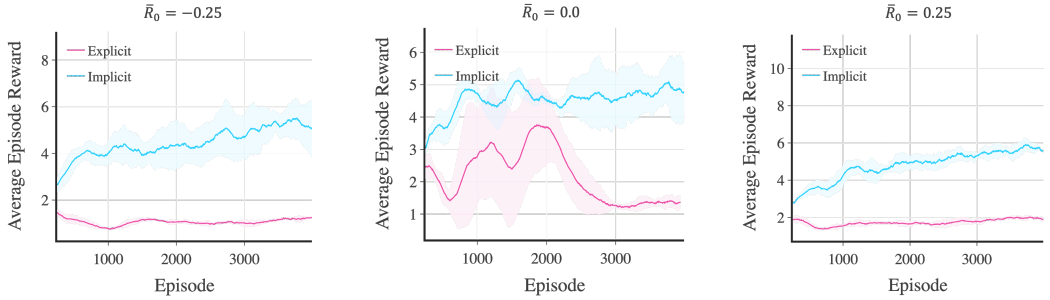

Figure C.2: Rolling total reward per episode when using a deep differential Q-learning algorithm in the Breakout environment with an initial average-reward guess of: **a)** -0.25, **b)** 0.0, and **c)** 0.25. Each plot shows the performance when using (batch-averaged) explicit vs. implicit TD error average-reward updates. A solid line denotes the mean total reward per episode, and a shaded region denotes a 95% confidence interval over 4 runs.

C.3 Performance Experiments

In this section, we discuss the performance experiments performed as part of this work. Through these experiments, we aimed to show, from an empirical perspective, how using the explicit TD error vs. the implicit TD error affects the performance of deep differential, reward centering (Naik et al., 2024), and A2C (Mnih et al., 2016) algorithms.



Figure C.3: Illustration of the **a)** Atari Breakout, **b)** Atari Pong, and **c)** MuJoCo Ant tasks.

In particular, for each class of algorithm, we compared how the algorithm performs in Atari (Bellemare et al., 2013) or MuJoCo (Todorov et al., 2012) environments when using the explicit vs. implicit TD error to compute quantities beyond the value function (i.e., the average-reward or advantage estimates). We tested the differential algorithms in the Atari *Breakout* environment, the reward centering algorithms in the Atari *Pong* environment, and the A2C algorithms in the MuJoCo *Ant* environment. These tasks are shown in Figure C.3.

Breakout Experiment:

In the Breakout experiment, we compared a deep differential Q-learning algorithm that utilizes implicit TD error average-reward updates (Algorithm 1) against the deep differential Q-learning algorithm from Rojas & Lee (2026) that utilizes smallest-magnitude explicit TD error average-reward updates (Algorithm 5).

For both algorithms, we used the Adam optimizer with the default hyperparameters for the decay and stability (beta and epsilon) on the smooth L1 loss (Equation (B.1)) with a λ of 1.0. We used a replay buffer with a minimum size of 1e2, a maximum size of 1e5, and a sample batch size of 32. We used a target network, and updated it using Polyak averaging with a fixed step size of 0.005. We used an ε-greedy policy with $\varepsilon = 0.1$. We used an initial average-reward estimate guess of 0.25. We used the value network shown in Section C.2.

For the differential algorithm with implicit TD error average-reward updates, we tested every combination of the value function step size, $\alpha \in \{$2e-6, 2e-5, 2e-4, 2e-3$\}$, with the average-reward step size, $\eta\alpha$, where $\eta \in \{$1e-2, 1e-1, 1.0, 10.0, 100.0$\}$, for a total of 20 unique combinations. Each combination was run 3 times using different random seeds for 1M steps each, and the results were averaged across the runs. A value function step size of 2e-5 and an average-reward η of 1.0 yielded the best results and were used to generate the results displayed in Figure 3a).

For the differential algorithm with smallest-magnitude explicit TD error average-reward updates, we tested every combination of the value function step size, $\alpha \in \{$2e-6, 2e-5, 2e-4, 2e-3$\}$, with the average-reward step size, $\eta\alpha$, where $\eta \in \{$1e-2, 1e-1, 1.0, 10.0, 100.0$\}$, for a total of 20 unique combinations. Each combination was run 3 times using different random seeds for 1M steps each, and the results were averaged across the runs. A value function step size of 2e-5 and an average-reward η of 10.0 yielded the best results and were used to generate the results in Figure 3a).

Pong Experiment:

In the Pong experiment, we compared a value-based reward centering algorithm that utilizes implicit TD error average-reward updates (Algorithm 2) against a value-based reward centering algorithm that utilizes explicit TD error average-reward updates (Algorithm 6).

For both algorithms, we used the Adam optimizer with the default hyperparameters for the decay and stability (beta and epsilon) on the smooth L1 loss (Equation (B.1)) with a λ of 1.0. We used a replay buffer with a minimum size of 1e2, a maximum size of 1e5, and a sample batch size of 32. We used a target network, and updated it using Polyak averaging with a fixed step size of 0.005. We used an ε-greedy policy with $\varepsilon = 0.1$. We used an initial average-reward estimate guess of 0.25. We used a discount factor of 0.99. We used the value network shown in Section C.2.

For the reward centering algorithm with implicit TD error average-reward updates, we tested every combination of the value function step size, $\alpha \in \{$2e-6, 2e-5, 2e-4, 2e-3$\}$, with the average-reward step size, $\eta\alpha$, where $\eta \in \{$1e-2, 1e-1, 1.0, 10.0, 100.0$\}$, for a total of 20 unique combinations. Each combination was run 3 times using different random seeds for 1M steps each, and the results were averaged across the runs. A value function step size of 2e-5 and an average-reward η of 1e-2 yielded the best results and were used to generate the results displayed in Figure 3b).

For the reward centering algorithm with explicit TD error average-reward updates, we tested every combination of the value function step size, $\alpha \in \{$2e-6, 2e-5, 2e-4, 2e-3$\}$, with the average-reward step size, $\eta\alpha$, where $\eta \in \{$1e-2, 1e-1, 1.0, 10.0, 100.0$\}$, for a total of 20 unique combinations. Each combination was run 3 times using different random seeds for 1M steps each, and the results were averaged across the runs. A value function step size of 2e-4 and an average-reward η of 1e-2 yielded the best results and were used to generate the results displayed in Figure 3b).

Ant Experiment:

In the Ant experiment, we compared an A2C algorithm that utilizes implicit TD error advantage estimates (Algorithm 3) against an A2C algorithm that utilizes explicit TD error advantage estimates (Algorithm 7).

For both algorithms, we used the Adam optimizer with the default hyperparameters for the decay and stability (beta and epsilon). We used a parameterized value function with the smooth L1 loss (Equation (B.1) with a λ of 1.0). We used a parameterized softmax policy. We used a replay buffer with a minimum size of 1e2, a maximum size of 1e5, and a sample batch size of 32. We used a target network, and updated it using Polyak averaging with a fixed step size of 0.005. We used a discount factor of 0.99. We used the following critic (i.e., value) and actor (i.e., policy) networks with *hidden_dim* = 256:

Critic network:

```
import torch

class ValueNetwork(torch.nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim):
        super(ValueNetwork, self).__init__()
        self.linear1 = torch.nn.Linear(state_dim, hidden_dim)
        self.linear2 = torch.nn.Linear(hidden_dim, hidden_dim)
        self.linear3 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = torch.nn.functional.relu(self.linear1(x))
        x = torch.nn.functional.relu(self.linear2(x))
        return self.linear3(x)
```

Actor network:

```
import torch

class PolicyNetwork(torch.nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim, max_action,
                 log_std_min=-20, log_std_max=2):
        super(PolicyNetwork, self).__init__()

        self.log_std_min = log_std_min
        self.log_std_max = log_std_max

        self.linear1 = torch.nn.Linear(state_dim, hidden_dim)
        self.linear2 = torch.nn.Linear(hidden_dim, hidden_dim)

        self.mean = torch.nn.Linear(hidden_dim, action_dim)
        self.log_std = torch.nn.Linear(hidden_dim, action_dim)

        self.max_action = max_action

    def forward(self, state):
        a = torch.nn.functional.relu(self.linear1(state))
        a = torch.nn.functional.relu(self.linear2(a))
        mean = self.mean(a)
        log_std = self.log_std(a)
        log_std = torch.clamp(log_std, self.log_std_min, self.log_std_max)
        return mean, log_std.exp()

    def sample_action(self, state):
        mean, std = self.forward(state)
        normal = torch.distributions.Normal(mean, std)
```

```
        # re-parameterization trick
        x_t = normal.rsample()
        action = torch.tanh(x_t)

        return action * self.max_action
```

For the A2C algorithm with implicit TD error advantage estimates, we tested every combination of the value function step size, $\alpha \in \{$2e-6, 2e-5, 2e-4, 2e-3$\}$, with the policy step size, $\eta\alpha$, where $\eta \in \{$1e-2, 1e-1, 1.0, 10.0, 100.0$\}$, for a total of 20 unique combinations. Each combination was run 3 times using different random seeds for 1M steps each, and the results were averaged across the runs. A value function step size of 2e-6 and a policy η of 1.0 yielded the best results and were used to generate the results displayed in Figure 3c).

For the A2C algorithm with explicit TD error advantage estimates, we tested every combination of the value function step size, $\alpha \in \{$2e-6, 2e-5, 2e-4, 2e-3$\}$, with the policy step size, $\eta\alpha$, where $\eta \in \{$1e-2, 1e-1, 1.0, 10.0, 100.0$\}$, for a total of 20 unique combinations. Each combination was run 3 times using different random seeds for 1M steps each, and the results were averaged across the runs. A value function step size of 2e-4 and a policy η of 1e-1 yielded the best results and were used to generate the results displayed in Figure 3c).

C.4 BASELINES

Below is the pseudocode for the baseline algorithms used for comparison in our experiments:

Algorithm 4 Differential DQN

Input: a differentiable state-action value function parameterization: $\hat{q}(s, a, \boldsymbol{w})$ (with target network $\hat{q}_{\boldsymbol{T}}(s, a, \boldsymbol{w}_{\boldsymbol{T}})$), the policy π to be used (e.g., ε-greedy)
Algorithm parameters: step size parameters $\{\alpha, \eta\}$, loss parameter λ
Initialize state-action value weights $\boldsymbol{w}, \boldsymbol{w}_{\boldsymbol{T}} \in \mathbb{R}^d$ arbitrarily (e.g. to $\mathbf{0}$)
Initialize $\bar{R}$ arbitrarily (e.g. to zero)
Obtain initial S
while still time to train **do**
  $A \leftarrow$ action given by π for S
  Take action A, observe R, S'
  Store (S, A, R, S') in replay buffer
  if time to update estimates **then**
    Sample a minibatch of B transitions from replay buffer: $\{(S_b, A_b, R_b, S'_b)\}_{b=1}^{B}$
    For each b-th transition: $\delta_b^e = R_b - \bar{R} + \max_a \hat{q}_{\boldsymbol{T}}(S'_b, a, \boldsymbol{w}_{\boldsymbol{T}}) - \hat{q}(S_b, A_b, \boldsymbol{w})$
    $\ell = \frac{1}{B}\sum_{b=1}^{B} L^{\lambda}(\delta_b^e)$ (See Equation (B.1))
    $\boldsymbol{w} = \boldsymbol{w} - \alpha \frac{\partial \ell}{\partial \boldsymbol{w}}$
    $\bar{R} = \bar{R} + \eta\alpha \frac{1}{B}\sum_{b=1}^{B} \delta_b^e$
    Update $\boldsymbol{w}_{\boldsymbol{T}}$ as needed (e.g. using Polyak averaging)
  end if
  $S = S'$
end while
return $\boldsymbol{w}$

Algorithm 5 Differential DQN (Smallest-Magnitude Baseline from Rojas & Lee (2026))

Input: a differentiable state-action value function parameterization: $\hat{q}(s, a, \boldsymbol{w})$ (with target network $\hat{q}_{\boldsymbol{T}}(s, a, \boldsymbol{w}_{\boldsymbol{T}})$), the policy π to be used (e.g., ε-greedy)
Algorithm parameters: step size parameters $\{\alpha, \eta\}$, loss parameter λ
Initialize state-action value weights $\boldsymbol{w}, \boldsymbol{w}_{\boldsymbol{T}} \in \mathbb{R}^d$ arbitrarily (e.g. to $\mathbf{0}$)
Initialize $\bar{R}$ arbitrarily (e.g. to zero)
Obtain initial S
while still time to train **do**
  $A \leftarrow$ action given by π for S
  Take action A, observe R, S'
  Store (S, A, R, S') in replay buffer
  if time to update estimates **then**
    Sample a minibatch of B transitions from replay buffer: $\{(S_b, A_b, R_b, S'_b)\}_{b=1}^{B}$
    For each b-th transition: $\delta_b^e = R_b - \bar{R} + \max_a \hat{q}_{\boldsymbol{T}}(S'_b, a, \boldsymbol{w}_{\boldsymbol{T}}) - \hat{q}(S_b, A_b, \boldsymbol{w})$
    $\ell = \frac{1}{B}\sum_{b=1}^{B} L^{\lambda}(\delta_b^e)$ (See Equation (B.1))
    $\boldsymbol{w} = \boldsymbol{w} - \alpha \frac{\partial \ell}{\partial \boldsymbol{w}}$
    Use the smallest-magnitude explicit TD error for the average-reward update:
      $y = \arg\min_{b \in \{1,\ldots,B\}} |\delta_b^e|$ (breaking ties arbitrarily)
      $\bar{R} = \bar{R} + \eta\alpha\delta_y^e$
    Update $\boldsymbol{w}_{\boldsymbol{T}}$ as needed (e.g. using Polyak averaging)
  end if
  $S = S'$
end while
return $\boldsymbol{w}$

Algorithm 6 Value-Based Reward Centering DQN (Naik et al., 2024)

Input: a differentiable state-action value function parameterization: $\hat{q}(s, a, \boldsymbol{w})$ (with target network $\hat{q}_{\boldsymbol{T}}(s, a, \boldsymbol{w}_{\boldsymbol{T}})$), the policy π to be used (e.g., ε-greedy)
Algorithm parameters: discount factor γ, step size parameters $\{\alpha, \eta\}$, loss parameter λ
Initialize state-action value weights $\boldsymbol{w}, \boldsymbol{w}_{\boldsymbol{T}} \in \mathbb{R}^d$ arbitrarily (e.g. to $\mathbf{0}$)
Initialize $\bar{R}$ arbitrarily (e.g. to zero)
Obtain initial S
while still time to train **do**
  $A \leftarrow$ action given by π for S
  Take action A, observe R, S'
  Store (S, A, R, S') in replay buffer
  if time to update estimates **then**
    Sample a minibatch of B transitions from replay buffer: $\{(S_b, A_b, R_b, S'_b)\}_{b=1}^{B}$
    For each b-th transition: $\delta_b^e = R_b - \bar{R} + \gamma \max_a \hat{q}_{\boldsymbol{T}}(S'_b, a, \boldsymbol{w}_{\boldsymbol{T}}) - \hat{q}(S_b, A_b, \boldsymbol{w})$
    $\ell = \frac{1}{B} \sum_{b=1}^{B} L^{\lambda}(\delta_b^e)$ (See Equation (B.1))
    $\boldsymbol{w} = \boldsymbol{w} - \alpha \frac{\partial \ell}{\partial \boldsymbol{w}}$
    $\bar{R} = \bar{R} + \eta\alpha \frac{1}{B} \sum_{b=1}^{B} \delta_b^e$
    Update $\boldsymbol{w}_{\boldsymbol{T}}$ as needed (e.g. using Polyak averaging)
  end if
  $S = S'$
end while
return $\boldsymbol{w}$

Algorithm 7 A2C (Mnih et al., 2016)

Input: a differentiable state-value function parameterization $\hat{v}(s, \boldsymbol{w})$, a differentiable policy parameterization $\pi(a \mid s, \boldsymbol{u})$
Algorithm parameters: discount factor γ, step size parameters $\{\alpha, \eta\}$
Initialize state-value weights $\boldsymbol{w} \in \mathbb{R}^d$ and policy weights $\boldsymbol{u} \in \mathbb{R}^{d'}$ (e.g. to $\mathbf{0}$)
Obtain initial S
while still time to train **do**
  $A \sim \pi(\cdot \mid S, \boldsymbol{u})$
  Take action A, observe R, S'
  $\delta^e = R + \gamma \hat{v}(S', \boldsymbol{w}) - \hat{v}(S, \boldsymbol{w})$
  $\boldsymbol{w} = \boldsymbol{w} + \alpha \delta^e \nabla \hat{v}(S, \boldsymbol{w})$
  $\boldsymbol{u} = \boldsymbol{u} + \eta\alpha \delta^e \nabla \ln \pi(A \mid S, \boldsymbol{u})$
  $S = S'$
end while
return $\boldsymbol{w}$ and $\boldsymbol{u}$