Metatrace Actor-Critic: Online Step-Size Tuning by Meta-Gradient Descent for Reinforcement Learning Control

Kenny Young\textsuperscript{1,2}, Baoxiang Wang\textsuperscript{1} and Matthew E. Taylor\textsuperscript{1}
\textsuperscript{1}Borealis AI, Edmonton, Alberta, Canada
\textsuperscript{2}University of Alberta, Edmonton, Alberta, Canada
kjyoung@ualberta.ca, \{baoxiang.wang, matthew.taylor\}@BorealisAI.com

Abstract

Reinforcement learning (RL) has had many successes, but significant hyperparameter tuning is commonly required to achieve good performance. Furthermore, when nonlinear function approximation is used, non-stationarity in the state representation can lead to learning instability. A variety of techniques exist to combat this — notably experience replay or the use of parallel actors. These techniques stabilize learning by making the RL problem more similar to the supervised setting. However, they come at the cost of moving away from the RL problem as it is typically formulated, that is, a single agent learning online without maintaining a large database of training examples. To address these issues, we propose Metatrace, a meta-gradient descent based algorithm to tune the step-size online. Metatrace leverages the structure of eligibility traces, and works for both tuning a scalar step-size and a respective step-size for each parameter. We empirically evaluate Metatrace for actor-critic on the Arcade Learning Environment. Results show Metatrace can speed up learning, and improve performance in non-stationary settings.

1 Introduction

In the supervised learning setting, there are a variety of optimization methods that build on stochastic gradient descent (SGD) for tuning neural network parameters. Applying ordinary SGD is difficult due to the sensitivity to the step-size parameter, which controls the magnitude of each update, as well as potential for wide variation in the gradient magnitude across different dimensions. Methods like RMSProp [Tieleman and Hinton, 2012] and ADAM [Kingma and Ba, 2014] aim to accelerate learning and improve stability by approximately accounting for higher order gradients of a fixed loss surface. While these methods are effective in the supervised case, they do not translate straightforwardly to online reinforcement learning (RL), where targets incorporate future estimates and subsequent observations are correlated.

The eligibility trace method is a key component used in RL to control the degree of bootstrapping from future estimates. However, this further complicates the analogy with supervised learning, as individual updates do not correspond to a gradient descent step toward any target on their own. Instead, the error signal is broken up into a series of updates such that only the sum of updates over time optimizes it. The appropriate analogy to the previously mentioned SGD techniques from supervised learning is not clear in this setting.

To apply these SGD techniques in the RL setting, a common strategy is to make the RL problem as close to the supervised learning problem as possible. Techniques that help to achieve this include multiple actors [Mnih et al., 2016], large experience replay buffers [Mnih et al., 2015], and separate online and target networks [Van Hasselt et al., 2016]. These all help smooth gradient noise and mitigate non-stationarity. However, they do so at the cost of losing the advantages of online updating. These advantages include lower computation and memory use, learning immediately from new experience, and allowing a single agent to learn from its own trajectories.

We propose Metatrace, a novel set of algorithms that tune the step-size parameter online for the actor-critic algorithm. Metatrace applies meta-gradient descent, propagating gradients through the optimization algorithm itself, to derive step-size tuning algorithms specifically for the online RL control problem. These algorithms build on the incremental delta-bar-delta (IDBD) approach [Sutton, 1992] while solving an open issue of previous work. To be exact, we use an eligibility trace-like mechanism to allow the meta-objective to account for time-dependent weights in an online manner. We apply this insight to define algorithms for tuning a scalar step-size, as well as a vector of step-sizes. Finally, we propose a mixed version that leverages the benefits of both, and integrate the algorithm with entropy regularization and normalization of the metaupdate to improve performance in practice.

We evaluate Metatrace for actor-critic with eligibility traces (AC(\lambda)). Our experiments include a classic mountain car problem to demonstrate the efficiency of the algorithms and a drifting version of mountain car to explore the performance of meta-gradient descent with a non-stationary state representation. Finally, we evaluate Metatrace for training a deep neural network online in the original training-set games of the Arcade Learning Environment (ALE) [Bellemare et al., 2013]. Without the use of either multiple actors or experience replay, our method learns faster and achieves better performance than an optimal fixed step-size in most of the games.
2 Related Work

Meta-learning has a long history dating back to Schmidhuber’s thesis [Schmidhuber, 1987], where a genetic programming algorithm was applied to evolve better genetic programming algorithms. Our work is closely related to IDDBD [Sutton, 1992] and autostep [Mahmood et al., 2012], which are meta-gradient descent algorithms for step-size tuning in the supervised learning case. Also closely related are SIDS and NOSID [Dabney, 2014], which analogize meta-gradient descent procedures for the SARSA(\(\lambda\)) method [Sutton and Barto, 2018]. Our approach differs primarily by explicitly accounting for time-varying weights in the optimization objective for the step-size. We also extend the approach to AC(\(\lambda\)) and to vector-valued step-sizes, as well as a mixed version which combines scalar and vector step-sizes.

Meta-gradient RL [Xu et al., 2018] applies meta-gradient descent to optimize \(\gamma\) and \(\lambda\) in AC(\(\lambda\)). They use multiple parallel agents, deviating from the online RL case we focus on.

It is worth noting that previous works such as TIDDBD and its extension AutoTIDDBD [Kearney et al., 2019] tune vector step-sizes online. That work focuses on TD(\(\lambda\)) for prediction, and explores both vector and scalar step-sizes. They demonstrate that both scalar and vector AutoTIDDBD outperform ordinary TD(\(\lambda\)), while vector AutoTIDDBD outperforms a variety of scalar step-size adaptation methods. Aside from focusing on control rather than prediction, our methods differ from TIDDBD in the meta-objective optimized by the step-size tuning: they use one step TD error, while we use a more general multi-step objective.

Our neural network experiments draw inspiration from the work of Elfwing et al. [Elfwing et al., 2018], to our best knowledge the only prior work to apply online RL with eligibility traces to train a deep neural network. In their case, the neural network is used as a function approximator in SARSA(\(\lambda\)). They do not explore step-size tuning.

3 Background

We consider the RL problem, where a learning agent interacting with an environment strives to maximize a reward signal. The problem is formalized as a Markov decision process \((\mathcal{S}, \mathcal{A}, p, r, \gamma)\). We consider stochastic environments and stochastic policies. At each time-step the agent observes the state \(S_t \in \mathcal{S}\) and selects an action \(A_t \in \mathcal{A}\). Based on \(S_t\) and \(A_t\), the next state \(S_{t+1}\) is generated by \(p(S_{t+1} | S_t, A_t)\). The agent additionally observes a reward \(R_{t+1}\), generated by \(r: \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}\). We focus on the control problem, where an agent learns, through interaction with the environment, a policy \(\pi\) that maximizes the expected return \(G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\), with discount factor \(\gamma \in [0, 1]\).

Action-value methods like Q-learning are often used for RL control [Mnih et al., 2015, Watkins and Dayan, 1992]. However, actor-critic (AC) methods, which more directly optimize the expected return, are becoming increasingly popular in Deep RL [Espeholt et al., 2018]. We will focus on AC, but our proposed method is general enough to be applied to other RL algorithms. AC methods learn a state value function for the current policy, as well as a policy which attempts to maximize that value function. In particular, we will derive Meta-trace for actor-critic with eligibility traces, AC(\(\lambda\)) [Degris et al., 2012, Schulman et al., 2015].

We define the objective of AC(\(\lambda\)) in terms of the lambda return, which is recursively defined as \(G_{w,t}^\lambda = R_{t+1} + \gamma \left((1-\lambda)V_w(S_{t+1}) + \lambda G_{w,t+1}^\lambda\right)\).

AC(\(\lambda\)) bootstraps future evaluations to a degree controlled by \(\lambda\). Here we define \(G_{w,t}^\lambda\), for a fixed weight vector \(w\); in Section 4, we will extend this to a time varying \(w_t\). Also, define the TD-error \(\delta_t = R_t + \gamma V_w(S_{t+1}) - V_w(S_t)\). Then we can expand \(G_{w,t}^\lambda\) as the current state value estimate plus the sum of future discounted \(\delta_t\) values as

\[
G_{w,t}^\lambda = V_w(S_t) + \sum_{k=t}^{\infty} (\gamma \lambda)^{k-t} \delta_k. \tag{1}
\]

TD(\(\lambda\)) can be understood as performing gradient descent along the mean squared error \((G_{w,t}^\lambda - V_w(S_t))^2\) between the value function \(V_w\) and the lambda return \(G_{w,t}^\lambda\). In the derivation of TD(\(\lambda\)), the target \(G_{w,t}^\lambda\) is taken as a constant despite its dependence on \(w\). For this reason, TD(\(\lambda\)) is often called a semi-gradient method. Intuitively, this reflects our desire to modify our current estimate to match our future estimates, not the other way around. AC(\(\lambda\)), combines this mean squared error objective with a policy improvement term, such that the combined objective represents a trade-off between the quality of our value estimates and the performance of our policy:

\[
J(\lambda) = \frac{1}{2} \left( \sum_{t=0}^{\infty} \frac{G_{w,t}^\lambda - V_w(S_t)^2}{2} \right) - \sum_{t=0}^{\infty} \log(\pi_w(A_t | S_t)) \left[ (G_{w,t}^\lambda - V_w(S_t)) \right]. \tag{2}
\]

As is in TD(\(\lambda\)), we aim to perform semi-gradient descent to minimize equation (2). Along with AC(\(\lambda\)), \(G_{w,t}^\lambda\), \(V_w(S_t)\) in the right sum is taken to be constant. Similar to previous arguments, the intuition is that we wish to improve our actor under the evaluation of our critic, instead of modifying our critic to make our actor’s performance look better. With this caveat in mind, by the policy gradient theorem [Sutton et al., 2000], the expectation of the gradient of the right term in equation (2) is approximately equal to the negated gradient of the expected return. This approximation is accurate to the extent that our advantage estimate \((G_{w,t}^\lambda - V_w(S_t))\) is accurate. Descending the gradient of the right half of \(J(\lambda)\) is then ascending the gradient of an estimate of expected return. Taking the semi-gradient of equation (2) yields

\[
\frac{\partial}{\partial w} J(\lambda) = -\sum_{t=0}^{\infty} \left( \frac{\partial V_w(S_t)}{\partial w} \frac{G_{w,t}^\lambda - V_w(S_t)}{2} \right) + \frac{1}{2} \log(\pi_w(A_t | S_t)) \frac{\partial G_{w,t}^\lambda}{\partial w} \left[ (G_{w,t}^\lambda - V_w(S_t)) \right].
\]

Now, define for compactness

\[
U_w(S_t) = V_w(S_t) + \frac{1}{2} \log(\pi_w(A_t | S_t))
\]

4186
so that, applying equation (1), we can rewrite
\[
\frac{\partial}{\partial w} J_{\lambda}(w) = - \sum_{t=0}^{\infty} \frac{\partial U_w(S_t)}{\partial w} \sum_{k=t}^{\infty} (\gamma \lambda)^{k-t} \delta_k
\]
and define the eligibility trace at time \( t \) as \( z_t = \sum_{k=0}^{t} (\gamma \lambda)^{t-k} \frac{\partial U_w(S_k)}{\partial w} \), such that
\[
\frac{\partial}{\partial w} J_{\lambda}(w) = - \sum_{t=0}^{\infty} \delta_t z_t. \tag{3}
\]
Offline AC(\( \lambda \)) makes updates in the direction of equation (3). Online AC(\( \lambda \)), analogous to online TD(\( \lambda \)), is an approximation to this offline version (exact in the limit of quasi-static weights) that updates weights after every time-step. Advantages of the online version include making immediate use of new information, and being applicable to continual learning\(^{1}\). Online AC(\( \lambda \)) is defined by the following set of equations:
\[
z_t = \gamma \lambda z_{t-1} + \frac{\partial U_w(S_t)}{\partial w_t}, \tag{4}
\]
where \( \alpha \) is the step-size parameter, controlling the magnitude of each update. In practice, performance can be highly sensitive to the value of \( \alpha \), hence tuning it online as we do in this work is a worthwhile goal.

4 Algorithm

We present three variations of Metatrace for control using AC(\( \lambda \)): scalar (single \( \alpha \) for all model weights), vector (one \( \alpha \) per model weight), and finally a “mixed” version that attempts to leverage the benefits of both. Additionally, we discuss two practical improvements over the basic algorithm: normalization that helps to mitigate parameter sensitivity across problems and avoid divergence; and entropy regularization, which is commonly employed in AC to avoid premature convergence [Mnih et al., 2016].

4.1 Scalar Metatrace for AC(\( \lambda \))

For convenience, we define our step-size as \( \alpha = e^\beta \), similar to IDBD [Sutton, 1992]. We desire to optimize \( \alpha \) to allow our weights to efficiently track the non-stationary AC(\( \lambda \)) objective. To account for non-stationarity in the representation, we define the objective with time-dependent weights
\[
J_{\lambda}^\beta(w_0...w_\infty) = \frac{1}{2} \left( \sum_{t=0}^{\infty} [G_t^\lambda - V_{w_t}(S_t)]^2 \right)
- \sum_{t=0}^{\infty} \log(\pi_{w_t}(A_t|S_t)) \left[ G_t^\lambda - V_{w_t}(S_t) \right]. \tag{5}
\]

Our derivation is similar to Section 4.3.1 of Dabney’s thesis [Dabney, 2014], but explicitly accounts for the time-dependency of \( w_t \). Regarding each \( w_t \) as a function of \( \beta \), differentiating equation (5) under the same semi-gradient treatment used in AC(\( \lambda \)) yields
\[
\frac{\partial}{\partial \beta} J_{\lambda}(w_0...w_\infty) = - \sum_{t=0}^{\infty} \frac{\partial U_w(S_t)}{\partial \beta} \left[ G_t^\lambda - V_{w_t}(S_t) \right]
- \sum_{t=0}^{\infty} \delta_t \sum_{k=0}^{t} (\gamma \lambda)^{t-k} \frac{\partial U_w(S_k)}{\partial w_t} \frac{\partial w_t}{\partial \beta}. \tag{6}
\]
To handle the above objective, we define an auxiliary eligibility trace as
\[
z_{\beta,t} = \sum_{k=0}^{t} (\gamma \lambda)^{t-k} \left[ \frac{\partial U_w(S_k)}{\partial w_t}, \frac{\partial w_t}{\partial \beta} \right]. \tag{7}
\]
As \( z \) itself is a sum of first-order derivatives with respect to \( w \), we approximate \( \frac{\partial z_t}{\partial w} = 0 \). Since \( \log(\pi_{w_t}(A_t|S_t)) \) necessarily involves some non-linearity, this is a first-order approximation even in the linear case. Furthermore, in the control case, the weights affect action selection. Action selection, in turn, affects expected weight updates. Hence, there are additional higher order effects of modifying \( \beta \) on the expected weight updates. As is in Dabney’s thesis [Dabney, 2014], we do not account for this effect. We leave the question open of how to account for this interaction in the online setting, and to what extent it would make a difference in the algorithm. With the above reasoning, let \( h(t) \) denote \( \partial w_t / \partial \beta \), we update \( h(t) \) as
\[
h(t+1) = \frac{\partial}{\partial \beta} [w_t + \alpha h_t z_t]
\approx h(t) + \alpha h_t z_t + \alpha z_t \frac{\partial h_t}{\partial \beta}
= h(t) + \alpha z_t \left[ \delta_t + \left( \frac{\partial \delta_t}{\partial \beta}, \frac{\partial w_t}{\partial \beta} \right) \right]
= h(t) + \alpha z_t \delta_t + \left( \frac{\partial \delta_t}{\partial \beta}, h(t) \right). \tag{8}
\]
Following equation (6)-(8), our Scalar Metatrace for AC(\( \lambda \)) is described by
\[
z_{\beta} = \gamma \lambda z_{\beta} + \left[ \frac{\partial U_w(S_t)}{\partial w_t}, h \right],
\beta = \beta + \mu z_{\beta} \delta_t,
\]
h = \text{h} + e^\beta z \left[ \delta_t + \left( \frac{\partial \delta_t}{\partial \beta}, h \right) \right].
\]
Where \( \mu \) is the meta-step-size parameter, controlling the update magnitude for the step-size parameter itself. The update

\textbf{Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19)}

4187
Algorithm 1 Normalized Scalar Metatrace for actor-critic

1: \( h \leftarrow 0, \beta \leftarrow \beta_0, v \leftarrow 0 \)
2: for each episode do
3: \( z_\beta \leftarrow 0, u \leftarrow 0 \)
4: while episode not complete do
5: \( z \leftarrow 0, u \leftarrow 0 \)
6: \( U_{w_t} \leftarrow V_{w_t} + \frac{1}{2} \log(\pi_{w_t}(A_t|S_t)) \)
7: \( \Delta_\beta \leftarrow \gamma \lambda z_\beta + \left\langle \frac{\partial U_{w_t}}{\partial w_t}, h \right\rangle \)
8: \( \Delta_\beta \leftarrow \gamma \lambda z_\beta + \psi \frac{\partial U_{w_t}(S_t)}{\partial w_t} \)
9: \( v \leftarrow \max(\{\Delta_\beta \}, v + \mu(\Delta_\beta - v) \) \)
10: \( \beta \leftarrow \beta + \mu(\Delta_\beta - v) \)
11: \( u \leftarrow \max(e^\beta \left\langle \frac{\partial U_{w_t}}{\partial w_t}, u + (1 - \gamma) \lambda \right\rangle (e^\beta \left\langle \frac{\partial U_{w_t}}{\partial w_t}, w \right\rangle - u) \)
12: \( M \leftarrow \text{max}(u, 1) \)
13: \( h \leftarrow h + e^\beta (z \delta_t + \left\langle \frac{\partial U_{w_t}}{\partial w_t}, h \right\rangle) + \psi \frac{\partial U_{w_t}(S_t)}{\partial w_t} \)
14: \( \text{output} \alpha = e^\beta \)
15: end while
16: end for

vectorized in this section, with elements

\[
\begin{align*}
z_{\beta, i, t} &= \sum_{k=0}^{t} (\gamma \lambda)^{t-k} \left\langle \frac{\partial U_{w_t}(S_t)}{\partial w_{i,k}}, \frac{\partial w_{i,k}}{\partial \beta_i} \right\rangle, \\
\end{align*}
\]

such that

\[
\frac{\partial}{\partial \beta_i} F_{\lambda}(w_0 \ldots w_\infty) \approx -\sum_{t=0}^{\infty} \delta_t z_{\beta, i, t}.
\]

To compute \( h(t) \left\langle \frac{\partial w_{i,t}}{\partial \beta_i}, h \right\rangle \) we use the immediate generalization of the scalar case and the approximation \( \frac{\partial w_{i,t}}{\partial \beta_j} = 0 \), which yields

\[
\begin{align*}
h_i(t+1) &= h_i(t) + \alpha_i z_{i,t} \left[ \delta_i + \left\langle \frac{\partial \delta_t}{\partial w_{i,t}}, \frac{\partial w_{i,t}}{\partial \beta_i} \right\rangle \right] \\
\approx h_i(t) + \alpha_i z_{i,t} \left[ \delta_i + \frac{\partial \delta_t}{\partial w_{i,t}} \frac{\partial w_{i,t}}{\partial \beta_i} \right] \\
&= h_i(t) + \alpha_i z_{i,t} \left[ \delta_i + \frac{\partial \delta_t}{\partial w_{i,t}} h_i(t) \right].
\end{align*}
\]

Therefore, Vector Metatrace for \( AC(\lambda) \) is described by

\[
\begin{align*}
z_\beta &\leftarrow \gamma \lambda z_\beta + \left\langle \frac{\partial U_{w_t}(S_t)}{\partial w_t}, h \right\rangle, \\
\beta &\leftarrow \beta + \mu z_\beta \delta_t, \\
h &\leftarrow h + e^\beta \left\langle \frac{\partial \delta_t}{\partial w_t}, h \right\rangle,
\end{align*}
\]

where \( \left\langle \frac{\partial \delta_t}{\partial w_t}, h \right\rangle \) denotes the Hadamard product. The full algorithm is detailed in Algorithm 2.

We now describe three variations based on Scalar Meta-
trace and Vector Metatrace that can be helpful in practice.

**Mixed Metatrace**

We explore a mixed algorithm where a vector correction to the step-size $\hat{\beta}$ is learned for each weight and added to a global value $\hat{\beta}$ which is learned collectively for all weights. The idea is that through $\hat{\beta}$ Mixed Metatrace can make use of all available data to learn a good base step-size, while $\hat{\beta}$ can be adjusted per parameter where it is useful. We omit the algorithm for this case for brevity: it is a straightforward combination of the scalar and vector algorithms.

**Entropy Regularization**

In practice, it is often helpful to add an entropy bonus, $-\psi \sum_{t=0}^{\infty} H_w(S_t)$, to the objective function to discourage premature convergence [Mnih et al., 2016]. We found it helpful to also include the entropy bonus in the meta-objective, which requires modification to the step-size tuning procedure. This modification is included in Algorithms 1 and 2.

**Normalization**

As discussed in autostep [Mahmood et al., 2012], even in the supervised learning case, the meta-gradient descent algorithms discussed so far can be unstable and sensitive to the parameter $\mu$. To combat this, we normalize the meta-update, and clip the step-size to avoid overshooting in a manner analogous to the procedure introduced in autostep.

# 5 Experiments and Results

## 5.1 Mountain Car

We begin by testing Scalar Metatrace on the mountain car domain from OpenAI Gym [Brockman et al., 2016]. A reward of $-1$ is given for each time-step until the goal is reached or 200 steps are taken. In this experiment, we use linear function approximation with tile-coding for state representation. The tiling size is $10 \times 10$ where 16 tilings create a feature vector of size 1600. The learning algorithm is based on $AC(\lambda)$ with $\gamma$ fixed to 0.99 and $\lambda$ fixed to 0.8 in all experiments. For mountain car, we use no entropy regularization.

Figure 1a shows the results for a variety of $\alpha$ values without step-size tuning. $\alpha$ values were chosen as powers of 2 which range from excessively low to excessively high (where the algorithm diverges). Figure 1b shows the results for Normalized Scalar Metatrace with different initial $\alpha$ values. In contrast to the untuned baseline, the learning curves of different $\alpha$s are very similar. Though the $\mu$ value has to be specified, similar results hold for a broad range of $\mu$ values below $\mu \geq 2^{-6}$, above which the algorithm becomes unstable.

## 5.2 Drifting Mountain Car

Motivated by the noisy, non-stationary experiment from IDBD [Sutton, 1992], we create drifting mountain car by adding noise and non-stationarity to the state representation. The environment is intended to provide a proxy for the issues inherent in representation learning such as using a neural network function approximator. We use the same tiling features as in the original mountain car experiment. However, at each time-step, each feature has a chance to randomly flip from indicating activation with 1 to $-1$ and vice-versa. We define a uniform flipping probability per time step across all features that we refer to as the drift rate, which is $6 \times 10^{-6}$ in our experiment. Additionally, we add 32 noisy features which are 1 or 0 with probability 0.5 for every time-step.

Due to the non-stationarity, an arbitrarily small $\alpha$ is not asymptotically optimal, as it is unable to track the changing features. Because of the noise, a scalar $\alpha$ will be suboptimal as well. Nonzero $\alpha$s for noisy features lead to weight updates when that feature is active in a particular state. This random fluctuation adds noise to the learning process. The noise can be avoided by annealing associated $\alpha$s to zero, which is what Vector and Mixed Metatrace are expected to achieve. We demonstrate the effectiveness in Figure 2, for Scalar, Vector, and Mixed Metatrace methods along with a baseline with no online tuning. The best $\mu$ value for each method was selected from $\{2^{-i} | i \in \{6, \ldots, 11\}\}$. We observe that Mixed Metatrace maintains the most stable performance in the presence of non-stationarity.

To better understand the algorithm, we illustrate the evolution of various elements of the log step-size $\beta$ under Mixed Metatrace, in Figure 3. $\beta$s refer to the log step-sizes of weights associated with the value function while policy $\beta$s refer to those associated with the policy. Noisy $\beta$s refer to the weights attached to noisy features, while informative $\beta$s refer to those associated with the tile-coding which are expected...
Figure 2: Return v.s. training episodes on drifting mountain car. Each curve shows the average of 20 random seeds, smoothed over 40 episodes.

Figure 3: Evolution of average values for various weights on drifting mountain car for initial $\alpha = 2^{-10}$, $\mu = 2^{-10}$. Each curve shows the average of 20 repeats, smoothed over 40 episodes.

to be useful. While all the step-sizes associated with noisy features are suppressed compared to the informative features, we observe a much stronger separation between the noisy and informative features for the value function than for the policy. One possible reason for this is that small errors in the value function have far more impact on optimizing the objective $J(\lambda)$ in mountain car than small imperfections in the policy. Learning a good policy requires fine-grained ability to distinguish the value of similar states, as individual actions will have a relatively minor impact on the car. On the other hand, errors in the value function in one state have a large negative impact on both the value learning of other states and the ability to learn a good policy.

5.3 Arcade Learning Environment

Finally, we test our algorithm with the 5 original training-set games of the ALE [Bellemare et al., 2013]. We use a convolutional neural network architecture similar to that used in DQN [Mnih et al., 2013], with SiLU and dSiLU activations to improve the performance in the online setting [Elfwing et al., 2018]. The policy and value network share all but the final output layer. We fix $\mu = 1 \times 10^{-3}$ and run all experiments up to $1.25 \times 10^7$ observed frames. We use sticky actions to add stochasticity to the environments [Machado et al., 2017].

Figure 4 shows the results of the ALE experiments.

Figure 4: Return v.s. learning steps for the ALE games. Each curve is the average of 5 random seeds, smoothed over 40 episodes. Solid lines correspond to Metatrace with meta-step-size parameter $\alpha = 1 \times 10^{-3}$ while dotted lines correspond to the fixed step-size baseline.

Metatrace significantly decreases the sensitivity to the initial choice of $\alpha$. In most cases, Metatrace also improves the performance while accelerating learning. In space invaders, each $\alpha$ value tested outperforms the best $\alpha$ without online tuning. In seaquest, two of the three do. In asterix, the final performance of all initial $\alpha$s with Metatrace was similar to the best-untuned $\alpha$ value, but the learning process is much faster. In beam rider, however, using no online tuning results in faster learning, especially for a well-chosen initial $\alpha$. This result is possibly due to the high $\alpha$ sensitivity and high sample complexity in the game. The impact of changes in $\alpha$ only occurs after a very long period, making the online tuning especially difficult. In future work, it would be interesting to explore how Metatrace could be made robust to this issue.

6 Conclusion

We propose Metatrace, a novel set of algorithms based on meta-gradient descent for reinforcement learning, which performs step-size tuning for $AC(\lambda)$. Our result extends the methods of IDBD [Sutton, 1992] and autostep [Mahmood et al., 2012] from supervised learning to reinforcement learning, which involves several technical challenges. To achieve this, we derive an eligibility trace-like mechanism to allow the meta-objective to account for time-dependent weights. We show that that Scalar Metatrace improves robustness to initial step-size choice, while Mixed Metatrace facilitates learning in an RL problem with non-stationary state representation. We also empirically test our approach for training a neural network online for several games in the Arcade Learning Environment, with nonlinear function approximation, where the state representation is inherently unstable. In most of the games, Metatrace allows a range of initial step-sizes to learn faster. Metatrace can also achieve better performance compared to the best-fixed choice of the step-size.
References


