Automatic Bootstrapping on a Vast Predictive Network of Reinforcement Learning Sub-agents

Dylan R. Ashley [1]

Abstract

We experiment with using the λ-greedy algorithm to simplify the task of tuning the trace-decay parameter on a vast network of temporal difference learning sub-agents, a problem that currently limits the utility of such networks. We find that λ-greedy can achieve good performance on our network. We also extend λ-greedy by using a recent, robust method of estimating the variance of the return. We find that this new variant has more stability in the λ values selected from timestep to timestep but at the cost of decreased performance.

1. Introduction

Tuning the hyperparameters of each temporal difference learning sub-agent independently when working with a vast network of such sub-agents is unfeasible. Because of this, groups of learning sub-agents are usually tuned together. However, tuning learning sub-agents in groups naturally results in less desirable learning. An alternative approach to this strategy is to adapt the hyperparameters of the learning agents on the fly. While there is an extensive body of work adapting the step-size parameter on the fly (Zeiler, 2012; Kearney et al., 2017) there is not yet much work on adapting the other key hyperparameter used in temporal difference methods, the trace-decay parameter (White and White, 2016). Therefore we focus our efforts on evaluating the effectiveness of one such method, the λ-greedy algorithm (White and White, 2016) when it is used to learn a horde of general value functions (Sutton et al., 2011).

We apply the λ-greedy algorithm (White and White, 2016) to the task of learning a horde of general value functions (Sutton et al., 2011) defined over the sensorimotor data stream of a robot. We additionally extend the λ-greedy algorithm to make use of a recent, more robust variance estimation technique which estimates the variance of the return directly rather than combining estimates of the first and second moments (Sherstan et al., 2018). We find that both versions of the λ-greedy algorithms have similar performance to GTD(λ) (Sutton et al., 2009) with the original version of the λ-greedy algorithm performing the best overall. We find that the modified version of the λ-greedy algorithm seems to adjust λ in a smoother manner compared to the original λ-greedy algorithm at the cost of worse performance.

The remainder of this report expands on these results with an explanation of the robot used to generate the sensorimotor data stream in Section 3. Afterward, in Section 4, we describe the abstract interpretation of the sensorimotor data stream on which we define our value functions. Then, in Section 5, we elaborate on the specific value functions that compose of horde. Following that, in Section 6, we describe the results of our experiment. Finally we conclude in Section 7.

2. Algorithms

We compare the GTD(λ) algorithm by (Sutton et al., 2009) and the λ-greedy from (White and White, 2016). The λ-greedy algorithm uses an estimate of the variance of the return to select a value for λ at each timestep. Recently a new method for estimating the variance of the return has emerged that has been shown to be more robust in some cases (Sherstan et al., 2018). We include a version of λ-greedy that uses this new estimator in our analysis. From here on we refer to the original λ-greedy algorithm as λ-greedy and the modified version as direct λ-greedy. For both, we only consider the case where they are used with GTD(λ). The pseudocode for direct λ-greedy appears in Algorithm 1.

3. Robot

We describe how we assemble and power the robot in two parts. In Section 3.1 we describe how we build the robot's body and wire it. Then in Section 3.2 we describe the procedure for connecting the robot to a computer that we must follow each time this is done.

[1]University of Alberta, Edmonton, Alberta, Canada. Correspondence to: Dylan R. Ashley <dashley@ualberta.ca>.

Algorithm 1 Policy evaluation using direct λ-greedy

observe $\mathbf{x}_0$
$\mathbf{w} \leftarrow \mathbf{0}$
$\mathbf{h} \leftarrow \mathbf{0}$
$\mathbf{z} \leftarrow \mathbf{x}_0$
// initialize λ-greedy
$\mathbf{w}^{err} \leftarrow G_{max} \times \mathbf{1}$
$\mathbf{w}^{var} \leftarrow \mathbf{1}$
$\mathbf{z}^{err} \leftarrow \mathbf{x}_0$
$\mathbf{z}^{var} \leftarrow \mathbf{x}_0$
repeat
  take action a_t, observe $\mathbf{x}_{t+1}$ and r_{t+1}
  $\rho_t = \pi(s_t, a_t)/\mu(s_t, a_t)$
  // update $\mathbf{w}^{err}$
  $g^{err} \leftarrow \mathbf{x}_{t+1}^\top \mathbf{w}^{err}$
  $\delta^{err} \leftarrow r_{t+1} + \gamma_{t+1} g^{err} - \mathbf{x}_t^\top \mathbf{w}^{err}$
  $\mathbf{z}^{err} \leftarrow \rho_t \mathbf{z}^{err}$
  $\mathbf{w}^{err} \leftarrow \mathbf{w}^{err} + \alpha_{t+1} \delta^{err} \mathbf{z}^{err}$
  $\mathbf{z}^{err} \leftarrow \gamma_{t+1} \mathbf{z}^{err} + \mathbf{x}_{t+1}$
  // update $\mathbf{w}^{var}$
  $r^{var} \leftarrow (\rho_t \delta^{err} - (\rho_t - 1)\mathbf{x}_t^\top \mathbf{w}^{err})^2$
  $\gamma^{var} \leftarrow (\rho_t \gamma_{t+1})^2$
  $\delta^{var} \leftarrow r^{var} + \gamma^{var} \mathbf{x}_{t+1}^\top \mathbf{w}^{var} - \mathbf{x}_t^\top \mathbf{w}^{var}$
  $\mathbf{w}^{var} \leftarrow \mathbf{w}^{var} + \alpha_{t+1} \delta^{var} \mathbf{z}^{var}$
  $\mathbf{z}^{var} \leftarrow \gamma^{var} \mathbf{z}^{var} + \mathbf{x}_{t+1}$
  // compute λ estimate
  errsq $\leftarrow (g^{err} - \mathbf{x}_{t+1}^\top \mathbf{w})^2$
  varg $\leftarrow \max(0, \mathbf{x}_{t+1}^\top \mathbf{w}^{var})$
  $\lambda \leftarrow$ errsq$/$(varg + errsq)
  // update $\mathbf{w}$
  $\delta \leftarrow r_{t+1} + \gamma_{t+1} \mathbf{x}_{t+1}^\top \mathbf{w} - \mathbf{x}_t^\top \mathbf{w}$
  $\mathbf{z} \leftarrow \rho_t \mathbf{z}$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha_{t+1}(\delta \mathbf{z} - \gamma_{t+1}(1 - \lambda)\mathbf{x}_{t+1})(\mathbf{z}^\top \mathbf{h})$
  $\mathbf{h} \leftarrow \mathbf{h} + \alpha_{t+1} \eta_{t+1} (\delta \mathbf{z} - (\mathbf{h}^\top \mathbf{x}_t)\mathbf{x}_t)$
  $\mathbf{z} \leftarrow \gamma_{t+1} \lambda \mathbf{z} + \mathbf{x}_{t+1}$
until done

3.1. Assembling the Robot

We begin by wiring the servo motors. We use two ROBOTIS AX-12 servos, a 2.1mm Barrel Jack to terminal from DFRobot (model number RB-Dfr-182), and three ROBOTIS 3-Pin Signal-Power-Ground 200mm cables. We remove the signal cable from one of the 3-Pin cables and the power cable from another one. To wire the servos together, we first connect a power and ground cable to the adapter for the power supply. We connect this to a servo motor. Next, we connect a ground and data cable to the other servo motor to create a communication cable. Finally, we connect a power, ground, and data cable between the motors. When complete, the robot looks like the robot in Figure 1. Once the servo motors are wired together, we build the body of the robot.

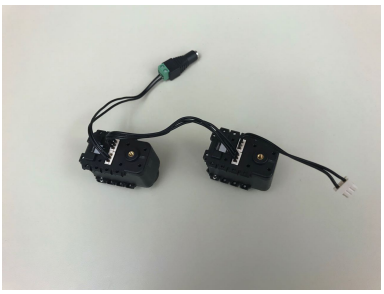

Figure 1. Wired Servo Motors

When building the body of the robot, we use four ROBOTIS OF-12SH frames, two ROBOTIS OF-12H frames, two ROBOTIS BPF-WA/BU Sets, two ROBOTIS Bolt PHS M3*10, sixteen ROBOTIS Bolt PHS M2*6, and eight ROBOTIS N1 NUT M2. We begin by connecting the two servos using the OF-12H frames as shown in Figure 2. Note how one servo motor is inverted and how the wires are fed through space in between the bases of the servo motors.

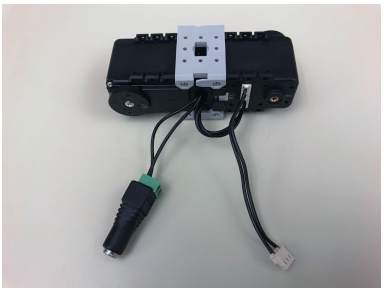

Figure 2. Connected Servo Motors

Now that the servo motors are connected, we attach a weight to the back of the robot. We select a weight of about 100 grams. After attaching the weight, the robot appears similar to the robot shown in Figure 3. Now that the weight is connected to the robot we construct and attach the legs of the robot.

Figure 3. Connected Servo Motors with Weight

We build the legs by screwing together the OF-12SH frames into two "H" frames as shown in Figure 4. We place

some electrical tape on two tips of each of the H frames as shown in Figure 4. This tape provides friction to help the robot move. After placing the electrical tape onto the H frames, we connect one to each servo as shown in Figure 5.

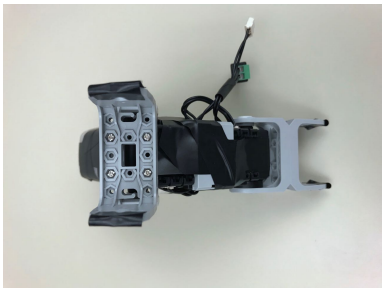

Figure 4. H Frames on Fully Constructed Robot

3.2. Powering the Robot

To power the robot and connect it to a computer we use a ROBOTIS USB to dynamixel adapter and a RoHS switching power supply (model number FY1203000). We begin by connecting the USB to dynamixel adapter to a computer as shown in Figure 6. Afterward, we connect the power supply adapter to the power supply as shown in Figure 7. We ensure the power supply is connected to an outlet. Finally, we connect the communication cable to the USB to dynamixel adapter as shown in Figure 8.

To detach the robot from the computer, we simply follow the above procedure in reverse order.

4. Domain

We now describe an abstract interpretation of the domain we define our general value functions on. To do so, we begin by defining a set of 4 states for each servo motor. We label these states $Parallel$, $MoreParallel$, $MorePerpendicular$, and $Perpendicular$. These four states represent four different positionings of the legs of the robot. Here $Parallel$ implies the leg is roughly parallel with the robot's body, $Perpendicular$ implies the leg is roughly perpendicular to the robot's body, and the other two states represent intermediate values. For each state we assign an encoder value as follows:

State	Encoder
$Parallel$	585
$MoreParallel$	653
$MorePerpendicular$	722
$Perpendicular$	790

We say that a leg is in a given state if the encoder position the leg is closer to the encoder position of that state than any of the other states. When moving a servo motor to a

Figure 5. Fully Constructed Robot

Figure 6. First Stage of Powering the Robot

Figure 7. Second Stage of Powering the Robot

Figure 8. Third Stage of Powering the Robot

state, we issue it a move command targetting the encoder value of that state. Because of error in the servo motors, it may not report the exact encoder value of the state after it has moved to it. By using these goalpost values rather than simply moving the servos a set distance, we minimize the interaction of this error with our learning. Now that we have elaborated on how the abstract states for each servo motor are defined we now describe how interaction with the robot occurs in the abstract interpretation of our domain.

At each timestep, for each servo motor, the acting agent has up to 3 possible actions. In the $Parallel$ state the agent can either keep the servo motor in the $Parallel$ state or move the servo motor forwards to the $MoreParallel$ state. In the $MoreParallel$ state the agent can either move the servo motor back to the $Parallel$ state, keep the servo motor in the $MoreParallel$ state, or move the servo motor forwards to the $MorePerpendicular$ state. In the $MorePerpendicular$ state the agent can either move the servo motor back to the $MoreParallel$ state, keep the servo motor in the $MorePerpendicular$ state, or move the servo motor forwards to the $Perpendicular$ state. In the $Perpendicular$ state the agent can either move the servo motor back to the $MorePerpendicular$ state or keep the servo motor in the $Perpendicular$ state.

With both servo motors having 4 states and up to 3 actions in each of these states, when both servo motors are considered together there are a total of 16 states and up to 9 actions per state in the abstract interpretation of our domain.

5. Value Functions

We implement twenty-two general value functions with a total of seven different cumulants, eight different termination functions, and five different policies. Many of these general value functions correspond to only one leg. For these cases, we define one such general value function for each leg. However, for simplicity, we do not discuss general value functions that only correspond to the second leg. The results and discussion of the results regarding value functions that correspond to only the first leg can be considered representative of the second leg.

5.1. On-policy

For our behavior policy, we use a random walk. Meaning that, for each leg and for each state as described in Section 4, the agent will select an action at random from the set of available action in that state. We define a total of fourteen on-policy general value functions. These value functions can be described as follows:

Leg 1 Parallel how many steps before the first leg is parallel to the body

Both Legs Parallel how many steps before both legs are parallel to the body

Leg 1 Perpendicular how many steps before the first leg is perpendicular to the body

Both Legs Perpendicular how many steps before both legs are perpendicular to the body

1-Step Leg 1 Distance what will be the absolute value of the distance traveled by the first leg within one step

8-Step Leg 1 Distance what will be the absolute value of the distance traveled by the first leg over the next eight timesteps

1-Step Leg 1 Position what will be the position of the first leg in one step

1-Step Leg 1 Load what will be the load on the first leg in one step

For the first four value functions, the cumulant is defined to be always one. All of them use a state-dependent termination function that is one while the leg(s) are not in the desired position and zero when they are. The following two value functions have a cumulant equal to the absolute value of the displacement of the leg in one step and a termination function that is, respectively, 0 and 0.875 for all states. The final two have cumulants equal to the position of the leg in the next step and the load of the leg in the next step. Both have a termination function that is zero for all states.

For all the value functions not involving the load, the true value function can be derived. As we wish to evaluate the performance of the various algorithms we compare, we derive the true value for these value functions. We additionally derive the maximum value of the return for each value function so that we can initialize the λ-greedy algorithm correctly. However, for the first four value functions above the true maximum return is infinite. For these value functions, we use 1500 as an estimate of the maximum value of the return. We find that a return of this magnitude is unlikely to be encountered but using such a value as initialization for the λ-greedy algorithm does not significantly hamper the ability of the algorithm to learn. Note that the maximum value of the load is defined by the specifications of the servos. For the servos, the maximum load that can be communicated is 511.

5.2. Off-policy

We define a total of eight off-policy (Sutton and Barto, 1998) general value functions. These value functions are described as follows:

Fastest Leg 1 Parallel under the minimizing policy how many steps before the first leg is parallel to the body

Fastest Leg 1 Perpendicular under the minimizing policy how many steps before the first leg is perpendicular to the body

1-Step Leg 1 Down Position if the robot moves the first leg down, what will be the position of the leg in the next step

1-Step Leg 1 Down Load if the robot moves the first leg down, what will be the load on the leg in the next step

The policies for the above value functions are self-explanatory from the description and looking at the abstraction in Section 4. In total, the aforementioned value functions define two target policies and two additional target policies when the equivalent value functions for the other leg are considered. Apart from the target policy, the definition of the above value functions are the same as their on-policy equivalents described in Section 5.1.

As with the on-policy case, we derive both the true value for each off-policy value function not involving the load, and we derive the maximum return for all off-policy value functions.

6. Results

We experiment with the ability of λ-greedy (LGGTD), direct λ-greedy (DLGGTD), and GTD to learn the horde described in Section 5. For all of our experiments we use a step size of $\alpha = 0.005$ (i.e., $\alpha = \frac{0.01}{\|x\|}$). We find that this provides a good balance between learning rate and smoothness of the learning curves. Notable it can learn most of the value functions adequately in 2500 timesteps while producing adequately smooth learning curves. We use a one-second timestep which is more than sufficient for the time it takes to update the learning sub-agents. Figure 9 shows the amount of time taken to update all of the sub-agents in one representative run. A single run of 2500 timesteps takes approximately forty-five minutes. Hence we only report results from ten runs. We find that ten runs adequately captures all our observations from earlier experiments. We use the word *Average* to describe an average value over the ten runs and omit it when we discuss a single run.

On each of our runs, we use an auxiliary step-size of $\eta = 0.01$ as this is both a value that frequently performs well (White, 2015), and because it produces results consistent with other values of η we tried. We use the same value of α and η for all of the algorithms we compare. For LGGTD and DLGGTD we use the same step size throughout the algorithm. For GTD we only consider the cases where λ is either one or zero as these were the values λ-greedy was previously compared against in the original paper by (White and White, 2016).

When we compare the algorithms, we frequently use a weighted average over value functions. In this cases, we weight the corresponding value for each value function by the inverse of its maximum return. As some of these values are infinite, we discuss in detail how we approximate these values in Section 5. Similarly, we frequently use a true error metric for value functions. As mentioned in Section 5, we are only able to calculate a true error metric for all value functions pertaining to the angle. Therefore we exclude any value functions pertaining to the load from these calculations.

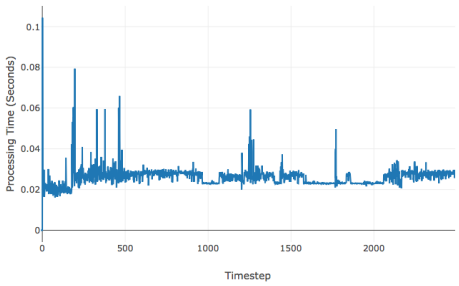

Figure 9. Time Taken to Update All Sub-agents Over One Run

We divide our analysis into a few sections. We begin by discussing the kind of signals coming from the robot in Section 6.1. Afterward, in Section 6.2, we discuss the overall performance of the algorithms. Then, in Section 6.3, we discuss important details regarding the performance of the algorithms on individual value functions. Finally, in Section 6.4, we discuss how the selected values for λ differ between LGGTD and DLGGTD.

6.1. Observed Behaviour

As the behavior policy is a random walk, there is considerable variability in the angle and load observed. Figure 10 shows the representative angle values observed in the first four hundred steps of one run, and Figure 11 shows the corresponding load values. Note that the angle readings clusters reasonably smoothly around the four encoder positions described in Section 4 while the load is a much noisier signal.

6.2. Overall Performance

We report the weighted average of the true error for each of the learners in Figure 12. Note that these values are an average of ten runs and most individual runs appear as slightly noisier variants of the result shown here. These results are consistent with earlier experiments.

The most notable takeaway from Figure 12 is that LGGTD performs similarly to GTD(1) but is more stable. The main reason for this is that GTD(1) encounters some issues with

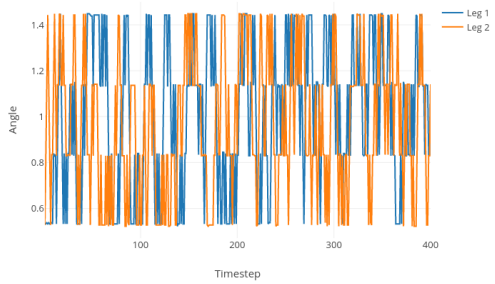

Figure 10. Observed Angle Values Over One Run

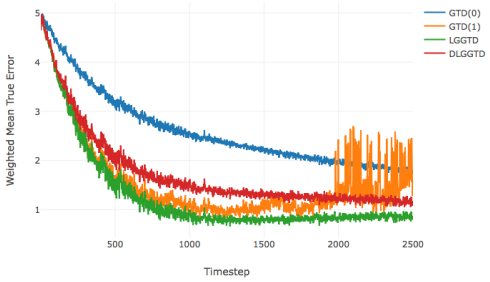

Figure 12. Average Weighted True Error

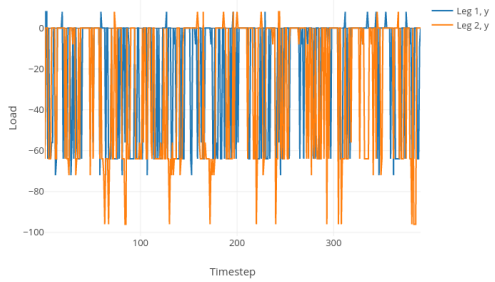

Figure 11. Observed Load Values Over One Run

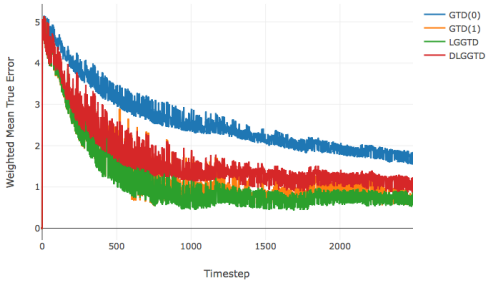

Figure 13. Weighted True Error Experienced in One Run

some instability when learning a few of the value functions. However, as shown in Figure 13, in runs where GTD(1) does not encounter these issues we observe that LGGTD typically matches or outperforms GTD(1). We acknowledge that this instability may be resolved by additional parameter tuning, but we believe the analysis of this instability in this context provides an interesting view into the stability of the algorithms.

Another interesting observation from Figure 12 is that DLGGTD performs considerably worse than LGGTD. As discussed later in Section 6.4 this appears to be due to the λ values selected by DLGGTD. However, as shown in Figure 14, when we observe the RUPEE values (White, 2015) experienced by the learners we note that DLGGTD typically appears below LGGTD. While we conjectured that this inconsistency with our expectations from Figure 12 is due to the performance on value functions relating to the load, this does not appear to be the case. Figure 15 shows the average RUPEE values experienced by LGGTD for value functions related to the load and Figure 16 shows the same values for DLGGTD. Note that both algorithms experience similar values, but LGGTD experiences these values with a higher variance. This observation is consistent with our hypothesis that DLGGTD is more stable than LGGTD but at the cost of lower performance.

6.3. Performance Over Individual Value Functions

Figures 17 through 20 show the predictions made by each algorithm for each of the value functions. Note that, as observed in Section 6.2, few substantial differences are observed between LGGTD and GTD(1), DLGGTD performs slightly worse than LGGTD, and GTD(0) performs substantially worse than all of the other algorithms. This difference in performance is most visible in the 8-Step Leg 1 Distance value function. We furthermore note that in Figure 18 the prediction for the Fastest Leg 1 Perpendicular value function appears to become unstable later in learning.

We highlight the apparent instability in the Fastest Leg 1 Perpendicular value function in Figure 21. We observed that this instability only occurs in some runs. We also observed that this is sometimes experienced by the other algorithms but is significantly rarer and, when it does occur, is typically much less severe. An example of this occurring in a run with LGGTD is shown in Figure 22. We further observed that, from earlier experiments not shown here, the only other value function that appears to exhibit this behavior is the Fastest Leg 1 Parallel. We argue that this apparent instability is because these value functions are off-policy and that the expected time before the random walk reaches a point where both legs are either parallel or perpendicular to the robot's body is considerable. We elaborate more on

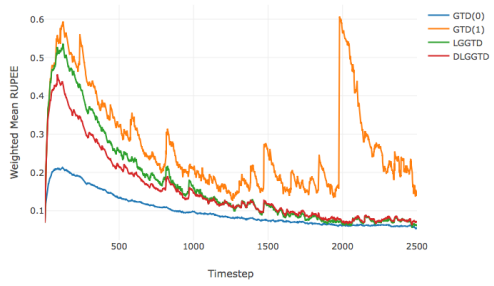

Figure 14. Average Weighted RUPEE

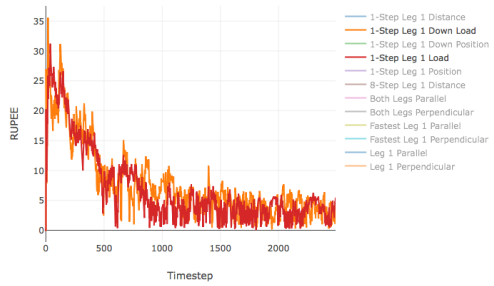

Figure 15. Average RUPEE Experienced by LGGTD for Value Functions Relating to the Load

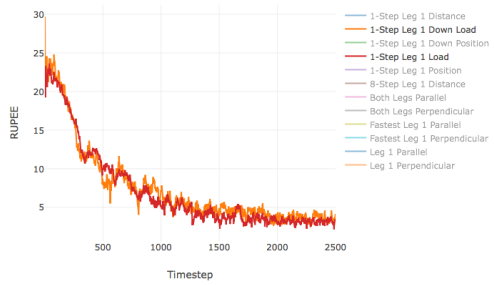

Figure 16. Average RUPEE Experienced by DLGGTD for Value Functions Relating to the Load

this instability in Section 6.4.

6.4. λ Values Selected

Both LGGTD and DLGGTD adaptively select a value for λ at each timestep. Notably, both methods successfully rid the user of having to tune λ by hand. In Section 6.2 we observed that LGGTD performed similarly to GTD(1), DLGGTD performed slightly worse, and GTD(0) performed much worse. As all algorithms only differ by the value of λ used, we can conclude that the difference in performance is due to the values of λ used by each algorithm. As GTD(0) was significantly outperformed by GTD(1), and DLGGTD was somewhat outperformed by LGGTD, we can conjecture that DLGGTD must be selecting lower values of λ than LGGTD. This hypothesis is confirmed by Figure 23. Figure 23 shows the average values for λ selected by the two algorithms over all the value functions in the ten runs. Notably, the values selected by DLGGTD are generally both smaller and lower in variance than those selected by LGGTD. Figure 24 shows the mean values selected for λ in one run over all the value functions. Note the much higher variance exhibited by LGGTD when selecting values for λ.

The trend that DLGGTD selects lower values for λ than LGGTD carries over to when we consider the values of λ selected for individual value functions. Figures 25 and 26 show the values of λ selected by LGGTD for each of the value functions and Figures 27 and 28 shown the same for DLGGTD. We separate the values selected for the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular value functions with the rest to make the difference between the rest of the value functions more visible.

Ignoring the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular value functions, we note that the behavior of the values of λ selected by both algorithms can be subdivided into two parts each. For LGGTD there is first a phase where λ is usually one. This phase it is followed by a period where the values of λ selected become rapidly fluctuate with a general trend that these values decrease somewhat over time. This trend is consistent with Figure 23. For DLGGTD we note that there is a phase where the values of λ selected smoothly decay towards zero. This phase is followed by a period where the values selected for λ begin to fluctuate wildly in a similar fashion to the second phase we observed with LGGTD. This behavior is again consistent with Figure 23. Of crucial importance is that the first phase experienced by LGGTD seems to last for a much shorter period than the first phase experienced by DLGGTD. Furthermore, as shown by Figure 12, the end of the first phase experienced by DLGGTD seems to roughly coincide with the predictions beginning to stabilize close to their final value. Altogether this explains the difference

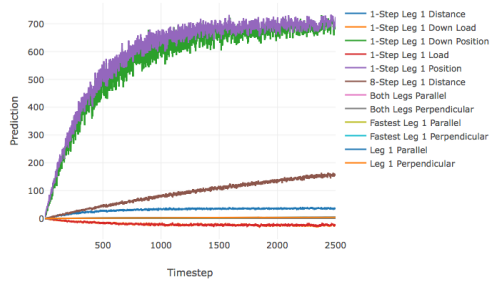

Figure 17. Average Estimates Made by GTD(0) for the Value of the Current State

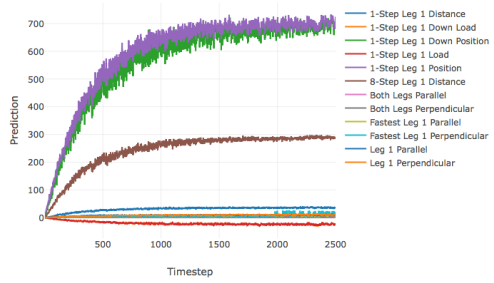

Figure 18. Average Estimates Made by GTD(1) for the Value of the Current State

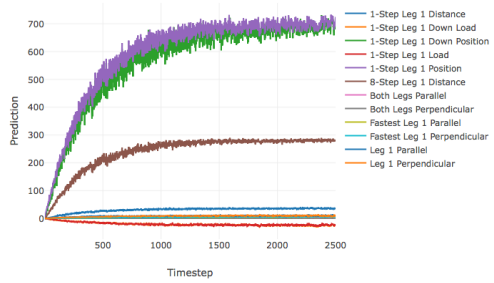

Figure 19. Average Estimates Made by LGGTD for the Value of the Current State

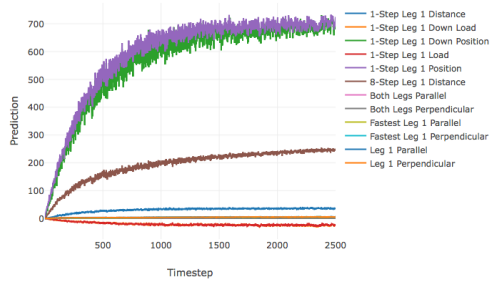

Figure 20. Average Estimates Made by DLGGTD for the Value of the Current State

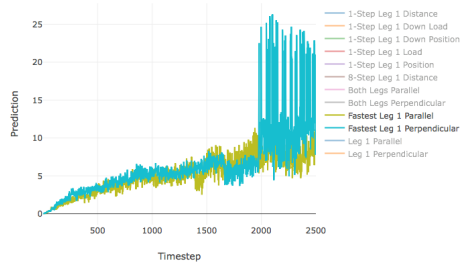

Figure 21. Average Estimates Made by GTD(1) for the Value of the Current State Under the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular Value Functions

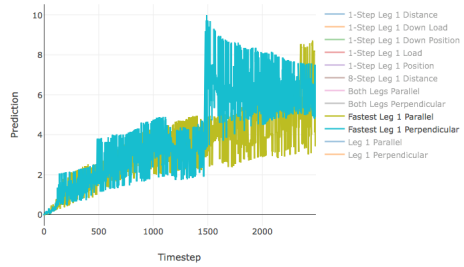

Figure 22. Estimates Made by LGGTD for the Value of the Current State Under the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular Value Functions in One Run

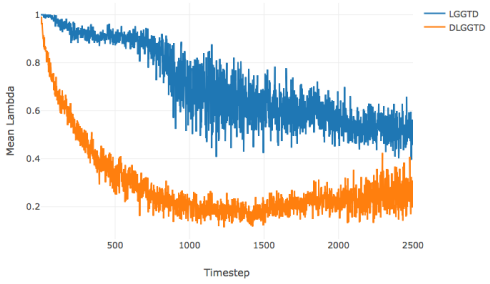

Figure 23. Average Values of λ Selected by LGGTD and DLGGTD

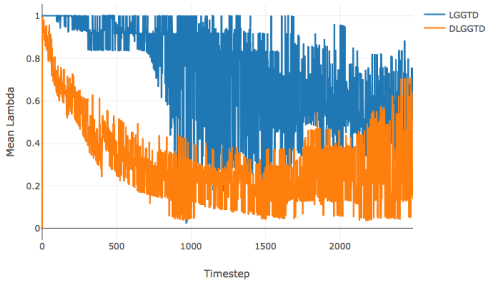

Figure 24. Values of λ Selected by LGGTD and DLGGTD in One Run

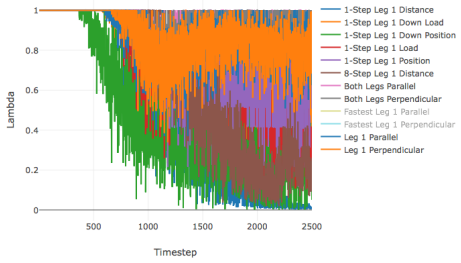

Figure 25. Average Values of λ Selected by LGGTD at Each Timestep

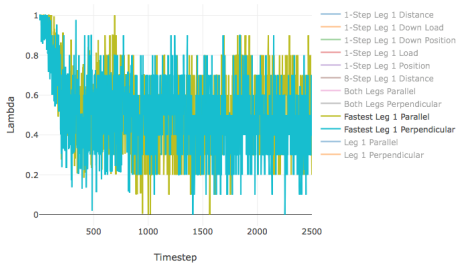

Figure 26. Average Values of λ Selected by LGGTD at Each Timestep for the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular Value Functions

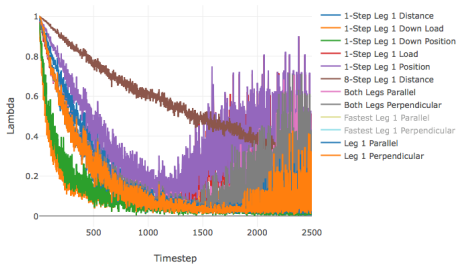

Figure 27. Average Values of λ Selected by DLGGTD at Each Timestep

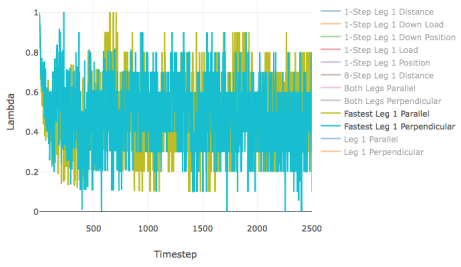

Figure 28. Average Values of λ Selected by DLGGTD at Each Timestep for the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular Value Functions

in performances observed between the algorithms in Section 6.2.

In Section 6.3 we noted that there was strange behaviour related to the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular value functions. Figures 26 and 28 show the values of λ selected by LGGTD and DLGGTD respectively for these value functions. Note that here little difference is observed between the two methods.

The similarity of LGGTD and DLGGTD when learning the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular value functions and the dissimilarity of them when learning the other value functions is further highlighted when we consider one run. Figures 29 and 30 show the values selected by LGGTD for λ with each of the value functions during a single run and Figures 31 and 32 shown the same for DLGGTD. Here we observe two interesting details. The first is that the values selected for λ by LGGTD tend to fluctuate between zero and one from one timestep to the next. This behavior is common over value functions. On the other hand, DLGGTD only exhibits this behavior, at least early on, with the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular value functions. However, it does adopt this behavior later on as well. This behavior is roughly bounded by the two phases of λ selection noted earlier.

7. Conclusion

We have applied the λ-greedy algorithm to the problem of adaptively setting trace-decay parameters in a horde architecture. We have shown that it can achieve good performance even when this predictive network is being learned over the sensorimotor data stream generated by a robot. We, therefore, conclude that the λ-greedy algorithm serves as a viable alternative to hand tuning these values in robotic domains.

In addition to experimenting the λ-greedy algorithm, we have also extended λ-greedy to use a direct estimate for the variance of the return. We have shown that this causes a smoother decay in the selected λ values at the cost of a decrease in performance.

8. Future Work

While the above experiments are performed on a robot, we note that our abstraction renders the domain the learning algorithms operate on somewhat tabular. Experimenting with these ideas on a more complicated system is essential to determine if the observations made here generalize. The need for a more complicated system also applies to the horde of value functions we use. The horde we use only consists of a few value functions making predictions about four signals. Previous incarnations of the horde architecture have made

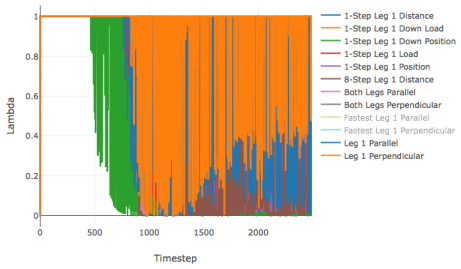

Figure 29. Values of λ Selected by LGGTD at Each Timestep in One Run

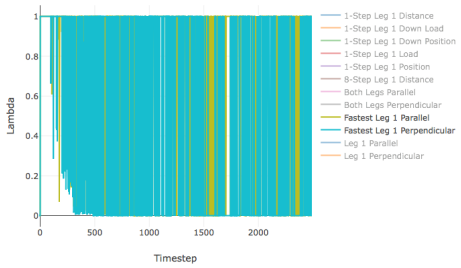

Figure 30. Values of λ Selected by LGGTD at Each Timestep for the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular Value Functions in One Run

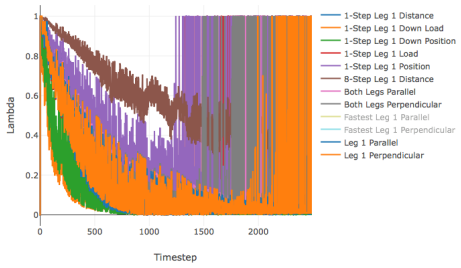

Figure 31. Values of λ Selected by DLGGTD at Each Timestep in One Run

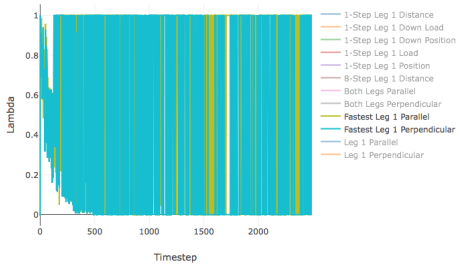

Figure 32. Values of λ Selected by DLGGTD at Each Timestep for the Fastest Leg 1 Parallel and Fastest Leg 1 Perpendicular Value Functions in One Run

vastly more predictions (Sutton et al., 2011). So one essential extension to the experiments in this paper would be extending the number and variety of value functions used in the horde.

In addition to the lack of diversity in the horde, we also note that we somewhat limit both the direct and indirect versions of the λ-greedy algorithm by not exploring more parameter settings. To be exact, we do not consider inconsistent step sizes between the λ-greedy specific portion of the learning and the rest of the learning. Furthermore, the λ-greedy algorithm implicitly uses a λ of one for the traces within the λ-greedy specific portion of the learning. This is necessary to produce an unbiased estimator (White and White, 2016). However, more work should be done to determine if using a biased estimator would produce different results.

In addition to a better exploration of parameter settings, we additionally note that, because of time constraints, we only consider the outcome of ten runs. If all of the above is implemented, it would be desirable to have many more runs to provide strong statistical significance.

References

Kearney, A., Veeriah, V., Travnik, J., Sutton, R. S., and Pilarski, P. M. (2017). TIDBD: Adapting temporal-difference step-sizes through stochastic meta-descent. University of Alberta.

Sherstan, C., Bennett, B., Young, K., Ashley, D. R., White, A., White, M., and Sutton, R. S. (2018). Directly estimating the variance of the λ-return using temporal-difference methods. *arXiv preprint arXiv:1801.08287*.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Sutton, R. S., Maei, H. R., Precup, D., Bhatnagar, S., Silver, D., Szepesvári, C., and Wiewiora, E. (2009). Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 993–1000. ACM.

Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., and Precup, D. (2011). Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768. International Foundation for Autonomous Agents and Multiagent Systems.

White, A. (2015). *Developing a predictive approach to knowledge*. PhD thesis, PhD thesis, University of Alberta.

White, M. and White, A. (2016). A greedy approach to adapting the trace parameter for temporal difference learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 557–565. International Foundation for Autonomous Agents and Multiagent Systems.

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.