Sequence Compression Speeds Up Credit Assignment in Reinforcement Learning

Aditya A. Ramesh [1] **Kenny Young** [2] **Louis Kirsch** [1] **Jürgen Schmidhuber** [1 3]

Abstract

Temporal credit assignment in reinforcement learning is challenging due to delayed and stochastic outcomes. Monte Carlo targets can bridge long delays between action and consequence but lead to high-variance targets due to stochasticity. Temporal difference (TD) learning uses bootstrapping to overcome variance but introduces a bias that can only be corrected through many iterations. TD(λ) provides a mechanism to navigate this bias-variance tradeoff smoothly. Appropriately selecting λ can significantly improve performance. Here, we propose *Chunked-TD*, which uses predicted probabilities of transitions from a model for computing λ-return targets. Unlike other model-based solutions to credit assignment, Chunked-TD is less vulnerable to model inaccuracies. Our approach is motivated by the principle of history compression and 'chunks' trajectories for conventional TD learning. Chunking with learned world models compresses near-deterministic regions of the environment-policy interaction to speed up credit assignment while still bootstrapping when necessary. We propose algorithms that can be implemented online and show that they solve some problems much faster than conventional TD(λ).

1. Introduction

An intelligent agent should recognize which of its actions contributed to success or failure (Minsky, 1961). Assigning credit or blame can be particularly difficult when there are long delays between a critical action and its consequences. Stochastic events between the action and its eventual consequence further complicate credit assignment. For example, consider the scenario where taking the umbrella in the morning helped the agent stay dry in the evening rain.

Reinforcement learning (RL) agents can bridge long delays by using Monte Carlo (MC) returns to evaluate—and subsequently improve—their behavior. However, using MC returns comes at the cost of high variance when the environment or the agent's behavior is stochastic. Temporal-difference (TD) approaches are designed to counter the variance of MC returns by constructing targets using predicted outcomes in future states (bootstrapping). TD lowers variance at the cost of introducing bias. TD(λ) approaches provide a way to smoothly interpolate between one-step bootstrapping (TD(0)) and MC targets by selecting λ. Unfortunately, correcting the bias of TD approaches can be very slow due to long delays between action and consequence (Arjona-Medina et al., 2019). TD with underparameterized function approximation introduces additional bias, even asymptotically (Sutton & Barto, 2018). Navigating this bias-variance trade-off is critical to achieving accurate and sample-efficient credit assignment (Watkins, 1989; Kearns & Singh, 2000). Previous research has attempted to automate the selection of λ to improve the performance of TD algorithms (Sutton & Singh, 1994; White & White, 2016; Xu et al., 2018).

An alternative to the previously discussed model-free approaches is to build a model of the world and use it for credit assignment. A world model can make associations between cause and effect, which could be used to guide policy learning. A prominent approach in this direction is to learn a differentiable model of the world and use gradients for credit assignment (Werbos, 1987; Munro, 1987; Schmidhuber, 1990; Heess et al., 2015). Such approaches have successfully been applied to continuous control problems with smooth dynamics (Deisenroth & Rasmussen, 2011; Amos et al., 2021). Unfortunately, this idea is not easily applicable in environments with discrete actions and is quite sensitive to inaccuracies in the learned model (Hafner et al., 2021; Meulemans et al., 2023).

World models can also be used to simulate experience to train the agent's policy or value function (Schmidhuber, 1990; Sutton, 1990; Ha & Schmidhuber, 2018; Hafner et al., 2021). Imagination with world models (or experience re-

[1]The Swiss AI Lab IDSIA, USI & SUPSI [2]University of Alberta and the Alberta Machine Intelligence Institute [3]AI Initiative, King Abdullah University of Science and Technology. Correspondence to: Aditya A. Ramesh <aditya@idsia.ch>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

play) can propagate credit without requiring the agent to re-experience the same transitions in the true environment. This can speed up credit assignment in terms of interactions needed with the true environment. However, imagination-based approaches can drastically fail when the learned world model is inaccurate (Talvitie, 2017; Abbas et al., 2020).

In this paper, we propose an approach that uses (possibly inaccurate) learned models to dynamically chunk trajectories of experience for TD learning. We introduce *Chunked Temporal Difference (Chunked-TD)* algorithms that compute adaptive λ-returns using model predictions for on-policy value learning. Our approach is inspired by the principle of history compression (Schmidhuber, 1991; 1992), which uses predictive coding to remove redundant information from a sequence to shorten credit assignment paths. Chunking in RL can shorten credit assignment paths in deterministic and predictable regions of the environment-policy interaction and facilitate faster learning while also bootstrapping to handle noise where necessary. Our proposed algorithms can be implemented online using eligibility traces (Sutton, 1984). Importantly, since the model is used only to decide when to bootstrap, our approach is robust to arbitrary levels of model error and simply degrades toward TD(0) as model errors become extreme.

Our experiments show that Chunked-TD can assign credit much faster than conventional TD(λ) in the considered Markov decision processes with tabular value functions. We also present a variant of Chunked-TD that can use factored rewards to solve a typical hard credit assignment problem. [§]

Our approach is closely related to a previously proposed variable TD(λ) strategy (Sutton & Singh, 1994) which is motivated from the perspective of reducing the bias of TD approaches. Similar to our approach, it uses transition probabilities from tabular models to decay traces. As a separate contribution, we revisit their proposed algorithms and show that their approach can be viewed as using 'backward' model predictions for constructing λ-returns.

2. Preliminaries

2.1. RL notation and definitions

Consider an agent interacting with a Markov Decision Process (MDP) (Puterman, 1990) with finite state, action, and reward spaces ($\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$) and a discount factor $\gamma \in [0, 1]$. The discount factor can be one in episodic tasks. The agent interacts with the environment at discrete time steps through actions sampled from a policy $A_t \sim \pi(.|S_t)$, where the policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ maps states to a distribution over actions. In response to the agent's action, the environment produces a new state and reward according to transition dynamics $S_{t+1} \sim T(.|S_t, A_t)$ and $R_{t+1} \sim R(.|S_t, A_t)$.

Interaction between policy and MDP generates a sequence

$$R_0, S_0, A_0, R_1, S_1, A_1 \ldots S_{t-1}, A_{t-1}, R_t, S_t, A_t \ldots,$$

where $R_0 = 0$ (for convenience and uniformity). Let $X_t = (R_t, S_t)$ be the 'concatenation' of reward and state. We will refer to X_t as the percept at time step t. It represents the response of the environment as a single object. In general, we could use $X_t = f(R_t, S_t)$ with some suitable function for state abstraction.

The above interaction can then be re-expressed as

$$X_0, A_0, X_1, A_1 \ldots X_{t-1}, A_{t-1}, X_t \ldots.$$

The agent's goal is to maximize the expected discounted return, $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1}\right]$, where the expectation accounts for randomness in the environment and the agent's policy.

RL algorithms often learn estimates of state or state-action values as an intermediate step for improvement. The value of a state $V_\pi(s)$ is defined as the expected sum of discounted rewards when starting from s and acting according to π, $V_\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s\right]$. Similarly, action-value functions can be defined as $Q_\pi(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | S_0 = s, A_0 = a\right]$.

Targets for value learning can be obtained via Monte-Carlo estimates or bootstrapped targets through temporal difference. Interpolating smoothly between MC and one-step bootstrapped targets is possible through the use of λ-returns (Watkins, 1989; Sutton & Barto, 2018). The λ-return (G_t^λ) can be recursively defined as

$$G_t^\lambda = R_{t+1} + \gamma\lambda G_{t+1}^\lambda + (1 - \lambda)\gamma\hat{V}(S_{t+1}), \qquad (1)$$

where $\hat{V}(S_{t+1})$ is the current estimate of the value of S_{t+1}. Using $\lambda = 1$ amounts to the MC target and $\lambda = 0$ uses one-step bootstrapped TD targets.

One can use λ-return targets to update value functions in the following manner,

$$\hat{V}(S_t) \leftarrow \hat{V}(S_t) + \alpha\left(G_t^\lambda - \hat{V}(S_t)\right), \qquad (2)$$

where α is the learning rate. Similar updates can be used for learning $\hat{Q}(S_t, A_t)$.

Updates with λ-returns can be done in an offline manner upon completion of episodes. This is because computing the λ-return according to Equation 1 requires knowing the entire sequence of future states and rewards. However, it is possible to approximate updates from offline λ-returns in an online, incremental way through TD(λ), which uses eligibility traces (Sutton, 1984).

[§]Code is available at https://github.com/Aditya-Ramesh-10/chunktd.

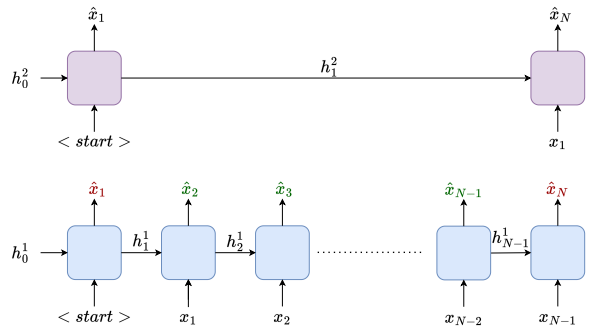

Figure 1. Two-level Chunker-RNNs can shorten credit assignment paths. Only the tokens associated with incorrect predictions (denoted with red) at the lower level are passed on to the higher level RNN (with the start token). The hidden state of the RNN at level l and index i is denoted as h_i^l.

2.2. Chunking and the principle of history compression

The principle of history compression (Schmidhuber, 1991; 1992) suggests removing redundant information from a sequence to shorten credit assignment paths. The aforementioned paper introduces a multi-level hierarchy of recurrent neural networks (RNNs) for sequence modeling or predictive coding, i.e., predicting the next input token. The central idea is that incorrectly predicted next inputs are identified based on a threshold on the error and passed to a higher level RNN, which can relate its inputs with shorter credit assignment paths.

We illustrate the idea through the example presented by Schmidhuber (1992) where there are two distinct sequences $< start >, a, i_1, i_2, \ldots i_{N-1}, 0$ and $< start >, b, i_1, i_2, \ldots i_{N-1}, 1$. Depending on whether the sequence starts with an a or b, the final token is 0 or 1. A standard RNN predicting the next input token finds it challenging to model this sequence due to the long delay between the critical input (a or b) and the classification label (0 or 1). However, the architecture with two RNNs, as presented in Figure 1, easily solves this problem.

At every time step t, each RNN tries to predict its next input token (x_{t+1}). Since the sub-sequence $i_1, i_2 \ldots i_{N-1}$ is highly predictable from local context, the lower level RNN easily learns to predict these tokens. The 'unpredictable' and compressed sequence passed to the higher level would be $< start >, a, 0$ or $< start >, b, 1$, which removes the delay in credit assignment.

In this paper, we take inspiration from the idea of history compression. While we do not use a hierarchical policy or value function, we employ the idea of 'compressing' sequences through next token prediction with a learned model. This can help address some of the difficulties arising from long gaps between crucial action and rewarding consequences in RL. In our case, the sequences are trajectories or episodes of interaction in reinforcement learning settings. As we explain in the next section, we use the compressed sequence to construct targets for value estimation.

3. Method

First, in Section 3.1, we present an approach that uses n-step targets for learning value functions based on a dynamic (and stochastic) choice of n obtained through sequence compression/chunking.

Next, in Section 3.2, we reformulate the stochastic chunking procedure as an exact, expected update, which is expressed as a λ-return. We show that our update can also be used to implement online algorithms through eligibility traces.

3.1. n-step chunked targets

Long delays between an action and its rewarding consequence make credit assignment in RL difficult (Arjona-Medina et al., 2019). As discussed in Section 2.2, removing redundant information from a sequence can shorten credit assignment paths for sequence modeling (Schmidhuber, 1992). With a similar motivation, we aim to use a learned (generative) model of the environment to shorten credit assignment paths in RL when possible. We use the predicted probability of the next input under the learned model to chunk an episode of policy-environment interaction. The chunked episode will provide us with the states from which we bootstrap.

In our first variant, which applies to bootstrapping from state values, we use $\hat{P}^\pi(X_{t+1}|X_t)$, the probability of the next percept under the current policy. The following factorization can be used in MDPs with Markov policies,

$$\hat{P}^\pi(X_{t+1}|X_t) = \sum_{a \in \mathcal{A}} \hat{P}(X_{t+1}|X_t, a)\pi(a|X_t), \quad (3)$$

where $\hat{P}(X_{t+1}|X_t, a)$ is the estimated transition probability given a particular action. Note that while policies are typically defined as a function of states, they can also be defined on percepts directly by constraining policy functions to ignore the reward component of the percept.

Consider the following chunking strategy that applies to learning $\hat{V}(s)$ in episodic MDPs: (1) We always keep the first state s_0 (2) For $t \geq 1$ we drop state s_t with probability $\hat{P}^\pi(x_{t+1}|x_t)$ (3) Whenever we drop a percept, we sum the intermediate rewards such that the compressed and original episode have equivalent returns (4) We always keep the terminal/ultimate state s_T.

The compressed episode is used to construct targets for the states encountered in the entire episode. Consider the exam-

ple presented in Figure 2, where an episode of interaction produces percepts $x_0, x_1, \ldots x_T$. All transitions are deterministic and accurately modeled, apart from one transition $x_k \rightarrow x_{k+1}$ which has a low probability. In this situation, our chunking procedure may return the compressed state sequence of x_o, x_k, x_T, which are used for bootstrapping. All states before x_k using the value of $\hat{V}(s_k)$ to bootstrap. All states thereafter use Monte-Carlo targets (bootstrapping from the terminal state $\hat{V}(s_T)$ which has a value of zero).

Thus, we can use a learned model to chunk and obtain n-step bootstrapped backups with a stochastic n. The key feature of this approach is that if the environment-policy interaction is quite deterministic and our model is accurate, then it is like Monte Carlo learning. If there is high stochasticity, *or if our model is inaccurate*, our approach falls back to bootstrapping, with TD(0) being the extreme case. A similar approach can be used for bootstrapping from Q-value functions, where we chunk the episode using $\hat{P}^\pi(X_{t+1}, A_{t+1}|X_t, A_t)$ (see Appendix B).

3.2. Chunked-TD(λ)

The sampling-based chunking of the previous section can be reformulated as an expected update without the variance of sampling indicator variables to drop or keep states.

We recall the recursive definition of the λ-return (G_t^λ) with a time-varying λ_{t+1},

$$G_t^\lambda = R_{t+1} + \gamma\lambda_{t+1}G_{t+1}^\lambda + (1-\lambda_{t+1})\gamma V(S_{t+1}). \quad (4)$$

As motivated in Section 3.1, we bootstrap based on predictions from a forward model. Concretely, we keep S_{t+1} in the compressed sequence with probability $1 - \hat{P}^\pi(x_{t+2}|x_{t+1})$. Keeping it in the compressed sequence is equivalent to using it for bootstrapping, i.e., using $\lambda_{t+1} = 0$ in Equation 4.

It can be seen that using $\lambda_{t+1} = \hat{P}^\pi(x_{t+2}|x_{t+1})$ is equivalent to the expected update from the chunking procedure described in the previous section.

Eligibility-trace based online algorithm Since the target for S_t (G_t^λ) depends on S_{t+1} and S_{t+2}, it may not be immediately obvious that updates based on such a λ-return can be implemented online. We show that online updates are indeed possible. Algorithm 1 presents an incremental and online algorithm that approximates updates based on the λ-return described in Equation 4 with $\lambda_{t+1} = \hat{P}^\pi(x_{t+2}|x_{t+1})$. The equivalence is exact at the end of episodes in acyclic MDPs with tabular value functions.

Proposition 1. *Let M be an acyclic episodic Markov decision process with state space $\mathcal{S}$ and let $\hat{V} : \mathcal{S} \rightarrow \mathbb{R}$ be the estimated tabular value function. Let the sequence $x_0, a_0, x_1, a_1 \ldots x_T$ correspond to an episode of interaction. Then updates from offline-λ returns from Equations 2 and 4 with $\lambda_{t+1} = \hat{P}^\pi(x_{t+2}|x_{t+1})$ match the total updates made by Algorithm 1 at the end of the episode.*

Algorithm 1 Chunked-TD state value evaluation

Ensure: Eligibility $e(s) = 0\ \forall s \in \mathcal{S}$ at the beginning of each episode
  for each step of the episode (X_t, A_t, X_{t+1}) **do**
    Optionally train dynamics model
    for each state s **do**
      $e(s) \leftarrow \gamma\hat{P}^\pi(X_{t+1}|X_t)e(s)$
    end for
    $e(S_t) \leftarrow e(S_t) + 1$
    $\delta_t \leftarrow R_{t+1} + \gamma\hat{V}(S_{t+1}) - \hat{V}(S_t)$
    for each state s **do**
      $\hat{V}(s) \leftarrow \hat{V}(s) + \alpha\delta_t e(s)$
    end for
  end for

Proof. Proof sketch only. We know that a state in the sequence, s_t, is encountered exactly once at step t since M is acyclic. Without loss of generality, we consider a single state s_t.

Consider the update term u_t added to $\hat{V}(s_t)$ by the offline-λ return of Equation 2,

$$u_t = \alpha(G_t^\lambda - \hat{V}(s_t)).$$

Unrolling the recursive definition from Equation 4, we get

$$u_t = \alpha(r_{t+1} + \gamma\lambda_{t+1}G_{t+1}^\lambda + (1-\lambda_{t+1})\gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)).$$

Add and subtract $\gamma\lambda_{t+1}\hat{V}(S_{t+1})$,

$$u_t = \alpha(r_{t+1} + (1-\lambda_{t+1})\gamma\hat{V}(s_{t+1}) + \gamma\lambda_{t+1}\hat{V}(s_{t+1}) - \hat{V}(s_t) + \gamma\lambda_{t+1}G_{t+1}^\lambda - \gamma\lambda_{t+1}\hat{V}(s_{t+1})).$$

Using $\delta_t = r_{t+1} + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)$, and unrolling,

$$\begin{aligned} u_t &= \alpha(\delta_t + \gamma\lambda_{t+1}(G_{t+1}^\lambda - \hat{V}(s_{t+1})) \\ &= \alpha(\delta_t + \gamma\lambda_{t+1}(\delta_{t+1} + \gamma\lambda_{t+2}(G_{t+2}^\lambda - \hat{V}(s_{t+2})))). \end{aligned}$$

Unrolling on, and noting $\lambda_{t+1} = \hat{P}^\pi(x_{t+2}|x_{t+1})$,

$$u_t = \alpha \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k \prod_{i=t+1}^{k} \hat{P}^\pi(x_{i+1}|x_i). \quad (5)$$

Now, let us consider the updates made by Algorithm 1. Only TD errors encountered from s_t onward contribute towards updating $\hat{V}(s_t)$. The first transition from s_t adds $\alpha\delta_t$ to $\hat{V}(s_t)$ (as $e(s_t)$ is 1 when s_t is first encountered). The next update adds $\alpha\gamma\delta_{t+1}\hat{P}^\pi(x_{t+2}|x_{t+1})$, the one after adds $\alpha\gamma^2\delta_{t+2}\hat{P}^\pi(x_{t+3}|x_{t+2})\hat{P}^\pi(x_{t+2}|x_{t+1})$. Summing all of them gives the same update as Equation 5. □

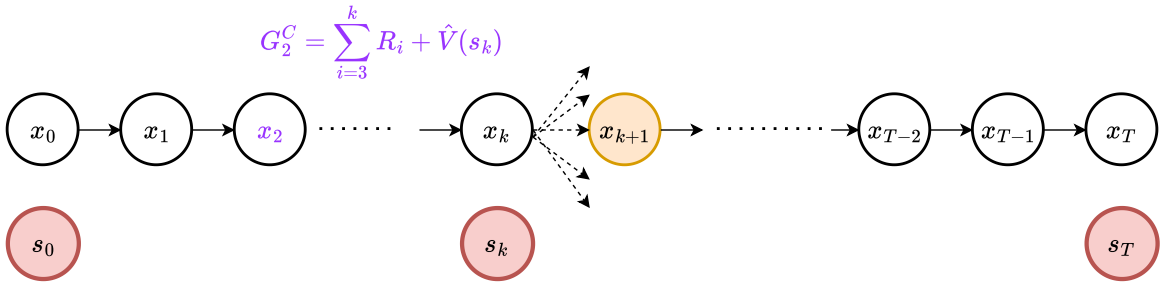

Figure 2. Sample-based chunking (see Section 3.1) for state value estimation. All next states (and rewards) are deterministic apart from the outcome at x_k, where $P_\pi(x_{k+1}|x_k)$ is low. The states in red are a sample from our algorithm, i.e., chunking based on model probabilities that provide targets for bootstrapping. The chunked target for s_2 (with $\gamma = 1$, denoted as G_2^C) is shown as an example.

So far, we have presented Chunked-TD for state value estimation. Chunked algorithms can similarly be defined for action-value functions, which can be used to implement SARSA-like algorithms (Rummery & Niranjan, 1994; Van Seijen et al., 2009) for evaluation and control. We present these variants in Algorithms 2 and 3 (see Appendix B).

Backward model An interesting alternative is to use a backward model for chunking or choosing λ_{t+1}. There one would have to model $\hat{P}^\pi(X_t, A_t|X_{t+1}, A_{t+1})$ or $\hat{P}^\pi(X_t|X_{t+1})$. Intuitively, this will bootstrap more from states that may be reached by many different paths, which in turn means their value is likely to be updated many times between each visit to a particular transition. Conversely, if a state is only ever visited from a single predecessor, bootstrapping is unlikely to be helpful as the value of that state and its predecessor will always be updated together, even with MC updates. In Appendix C, we show that, with careful analysis, a previously proposed algorithm by Sutton & Singh (1994) can be interpreted as using a λ-return by chunking with a backward model. [†] A crucial disadvantage of backward models is that they are policy-dependent (Chelu et al., 2020), making them harder to learn, especially in control, i.e., learning the optimal policy.

4. Experiments

This section presents an empirical comparison of Chunked-TD and conventional TD(λ), i.e., with a constant scalar λ, for learning tabular action-value functions. We conduct experiments in MDPs with fully-observable states to avoid conflating memory and credit assignment (Osband et al., 2019; Ni et al., 2023).

[†]While the trace-based TDC algorithm in Sutton & Singh (1994) uses a forward model to cut traces, with adjustments to the learning rate the corresponding λ-return (or forward view) amounts to using a backward model. Please see Appendix C for further details.

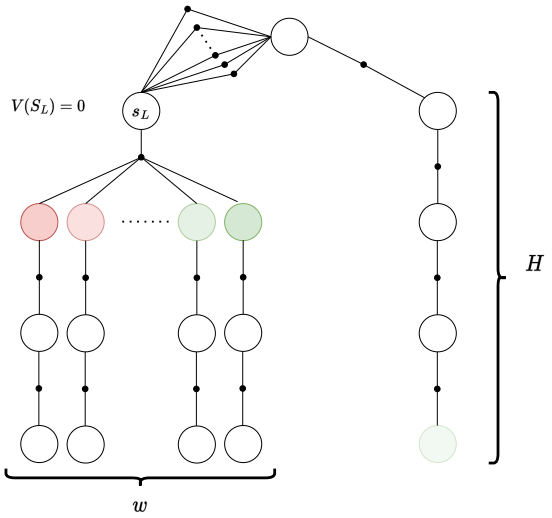

Figure 3. Chain-and-Split environment. States are denoted by large circles and actions by lines leading to the small black circles. Colored states have a reward upon entering that state.

We note that in our experiments, we set $X_t = S_t$ for Chunked-TD as all randomness related to the reward is absorbed in the next state in our considered environments.

4.1. Chain-and-Split

Chain-and-Split (see Figure 3) is an environment that requires adaptive values of λ for efficient policy evaluation. It is an episodic MDP in which only the first action influences transitions and rewards. One action leads to a 'delayed' deterministic reward, whereas the other actions lead to stochastic consequences.

Environment specification The agent has n actions $\{a_1, a_2, \ldots a_n\}$ available at the start/root state. After the first step, there is only one action available in every state. Taking a_1 at the start state leads to a linear chain of length H, which ultimately provides a deterministic reward of 0.01.

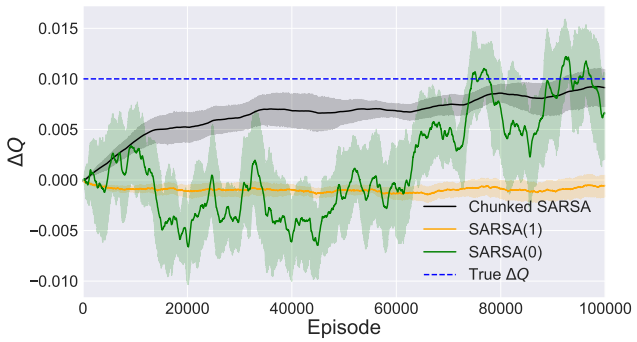

Figure 4. Results from the Chain-and-Split environment. The action-value gap (ΔQ) between optimal action a_1 and the maximum action value among the remaining actions. The true value of $\Delta Q = 0.01$. Shading indicates 95% bootstrapped confidence intervals over 10 independent trials.

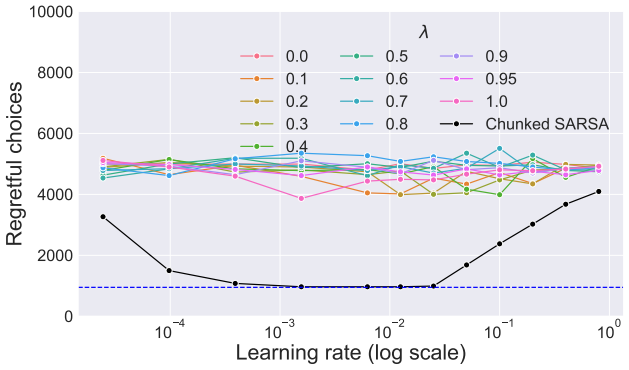

Figure 5. Results from the Accumulated-Charge environment. The average number of regretful choices over 10 runs for different algorithms with different learning rates for the accumulated-charge experiment. The dashed blue line represents the best possible expected regretful choices under the exploration scheme.

This is the optimal choice. All remaining actions lead to a common 'parent' state on the left (s_L), which has a zero value but is followed by noisy transitions/rewards. The state s_L branches to one of w states with equal probability. Each of the w states provides a deterministic reward between -1 and 1, with the average over all states being equal to zero. We set $H = 20$, $n = 10$, and $w = 101$ for our experiment.

Evaluation details We compare the performance of SARSA(0), SARSA(1)/MC, and Chunked SARSA (Algorithm 2). Tabular Q-values are learned from data collected by a policy that takes uniform random actions. We use a learned tabular model for computing λ_{t+1} for chunking. See Appendix A.1 for further details.

We compare algorithms based on the action-value gap with respect to the optimal action. Concretely, we consider the quantity $\Delta Q = Q(start, a_1) - \max_{a \in \{a_2, a_3 \ldots a_n\}} Q(start, a)$, which is positive only when the optimal action is assigned a higher value than all others. We consider multiple learning rates for each approach (see Appendix A.1).

Results We present results with the best learning rate for each approach. Figure 4 shows that Chunked SARSA quickly discovers that taking a_1 is the best choice and smoothly approaches the true value of ΔQ.

In contrast, MC/SARSA(1) fails because it does not bootstrap from the value of s_L. This results in learning entirely independent action values $\hat{Q}(start, a_i), i \in \{2, 3 \ldots 10\}$, without relying on the fact that all these actions actually lead to the same next state (s_L in Figure 3). This results in negative values of ΔQ, as the value of some action $\hat{Q}(start, a_i), i \in \{2, 3 \ldots 10\}$ is typically overestimated due to limited samples.

SARSA(0) bootstraps immediately, which is good for the left sub-tree as it correctly averages all the experience from sub-optimal actions. However, SARSA(0) is slower and less stable than Chunked SARSA. This is because SARSA(0) requires many experiences of taking a_1 to back up values along the linear chain on the right since the reward is provided only H steps after selecting a_1 in the start state.

Additional results with longer evaluation are reported in Appendix A.1.

4.2. Accumulated-Charge

In the previous section, we looked at evaluating actions under a uniform random policy. Here, we focus on control, i.e., learning optimal behavior. We consider a variant of the Choice environment studied by Arjona-Medina et al. (2019). This environment illustrates the benefit of our approach in environments where transitions are mostly deterministic, except at certain points, leading to highly compressible trajectories.

Environment specification The agent has two available actions in the initial state (a_1 and a_2) and only one available action (a_1) thereafter.

The state is represented as three components s^1, s^2, s^3. s^1 is a boolean indicating whether the first action was a_1. s^2 is a non-negative integer that indicates the total charge accumulated so far. s^3 is the time step.

During the episode, at k values of the time step, the agent stochastically accumulates 'charge'. The k time steps where a charge is accumulated are fixed for the interaction but are randomly chosen for each independent initialization (seed) of the environment. Further details are presented in Appendix A.2. In our selected setting, the episode lasts

$H = 200$ steps, and the agent encounters $k = 10$ accumulation points, where it accumulates charge sampled from a binomial variable with $n = H/k$ and $p = 0.5$.

A reward is provided at the final time step. The final reward depends on the amount of charge accumulated and on a bonus b that depends on whether the first action was a_1 or a_2. Choosing a_1 leads to an expected return of b, and choosing a_2 leads to an expected return of zero. We set $b = 0.1$ in our experiment. The precise reward function is described in Appendix A.2.

Evaluation details We compare the performance of Chunked SARSA and SARSA(λ) over multiple choices of λ and the learning rate. As the performance measure, we look at the number of regretful choices made by each algorithm. If the agent's choice at the first time step is not the optimal choice (i.e., a_1), it is labeled as regretful. We learn tabular Q-values. The agent takes uniform random actions in the first 1000 episodes and then acts ϵ-greedily for 9000 episodes, with $\epsilon = 0.1$.

We learn a neural network model to compute λ_{t+1} for Chunked SARSA. The neural network is trained to predict $\Delta s_t = s_{t+1} - s_t$ based on s_t and a_t. The neural network is trained on samples from a replay buffer. Details about the architecture and training are provided in Appendix A.2.

Results Figure 5 shows the number of regretful choices made by Chunked SARSA and SARSA(λ) for different values of λ and the learning rate. We see that Chunked SARSA performs best in this environment across several learning rates. These results are also presented with 95% bootstrapped confidence intervals in Figure 7 (Appendix A.2). We see that SARSA(λ) exhibits a lot of variance, achieving low regret on some trials and high regret on others. The reliable success of Chunked SARSA is due to the learned model's accurate prediction of transitions where charge does not accumulate. This bridges delays while still bootstrapping to a larger degree upon charge accumulation.

4.3. Key-to-Door with factored rewards

A limitation of our Chunked-TD approach is the presence of noisy components that don't alter the expected return. Since noise is inherently incompressible, our approach would not be able to solve some *hard* credit assignment problems previously considered in the literature (Meulemans et al., 2023; Ni et al., 2023).

However, if we can decompose or factor rewards in a certain way, we can apply our chunking idea on a per-component basis to perform well in such environments.

Environment specification The environment state comprises of $n_d + 4$ components. There are n_d Boolean distractors ($s_{d_1} \dots s_{d_{n_d}}$), which are sampled independently from a Bernoulli distribution with probability $p = 0.5$ at each time step. The state includes Boolean variables that indicate whether the agent has the key, or is in the door or treasure state ($s_{key}, s_{door}, s_{treasure}$). Another component indicates the time step (s_{time}). Further details about the state and reward representation are available in Appendix A.3.

The agent has two available actions at every step. They correspond to "pick key" and "unlock door". Using "pick key" in the start state turns the "key" boolean on. If the agent has the key, using "unlock door" at the door state (penultimate time step) will lead to the treasure. Other than this, the actions have no effect on the state. The environment is episodic with $H = 100$ steps and $n_d = 4$.

The environment returns a vector of rewards, one for each state component. If the treasure bit is on, it corresponds to a treasure reward of 0.01. Each distractor being on corresponds to a distracting reward of $0.01/(n_d)$. All other components are associated with zero rewards. The optimal policy is to pick up the key in the first step and use it to access the treasure in the last step.

Evaluation details When rewards are decomposed, we can learn independent tabular Q-values for each reward component, i.e., $n_d + 4$ Q-functions $\hat{Q}_1(s,a), \hat{Q}_2(s,a), \dots \hat{Q}_{n_d+4}(s,a)$. Each $Q_i(s,a)$ maps the whole state vector to the Q-value of a particular reward component. We act according to the global Q-value estimate, which is the sum of component-wise tabular Q-values (Russell & Zimdars, 2003), $\hat{Q}(s,a) = \sum_i^{n_d+4} Q_i(s,a)$. See Appendix B for additional details. We compare three approaches to learning Q-values.

The first corresponds to Expected-SARSA(λ), where the value of λ is a fixed scalar and remains the same for each $Q_i(s,a)$. The second is the chunked counterpart of Expected-SARSA (Algorithm 3), but uses the same value of λ_{t+1} for each $Q_i(s,a)$, where $\lambda_{t+1} = \hat{P}^\pi(x_{t+2}|x_{t+1})$ as is usual. We refer to this variant as *C-default*.

The third is Chunked Expected-SARSA, which uses different values of λ^i_{t+1} for each $Q_i(s,a)$. We refer to this algorithm as *C-factored*. Specifically, the bootstrapping factor of each component only depends on the predictability of that component, $\lambda^i_{t+1} = \hat{P}^\pi(x^i_{t+2}|x_{t+1})$. The exact algorithm is presented in Appendix B.

Actions are selected ϵ-greedily from the global Q-value. ϵ is annealed from 1 to 0.1 over the first 500 episodes. The agent interacts with the environment for 5000 episodes. We compare algorithms based on the number of episodes in which the treasure was not collected. We consider multiple learning rates for each approach (see Appendix A.3).

Table 1. Number of episodes in which the treasure is not collected (out of 5000). Mean and standard deviation over 10 independent trials.

λ	0	0.1	0.5	0.9	1	C-default	C-factored
Episodes	1881 ± 1950	2277 ± 2068	2273 ± 2076	2688 ± 2119	2261 ± 582	2285 ± 2061	$\mathbf{669 \pm 17}$

Results Table 1 presents results for the best choice of learning rate for each approach. Results with all considered learning rates are presented in Figure 8. Expected-SARSA(λ), with $\lambda \in \{0, 0.1, 0.5, 0.9, 1\}$, does not reliably learn to collect the treasure. Similarly, the default version of Chunked Expected-SARSA (C-default) performs poorly.

Since rewards are suitably decomposed, we could use the factored version of Chunked Expected-SARSA (C-factored). C-factored bootstraps separately based on the predictability of each component of the state. For e.g., notice that the key, door, treasure, and time steps are perfectly predictable from the previous state. C-factored behaves like an MC update with respect to the value of only these factors. This allows the agent to better estimate Q-values quickly and successfully collect the treasure in most episodes.

5. Related Work

The algorithms introduced in this paper are closely related to the ones proposed by Sutton & Singh (1994). Their algorithms are motivated from the perspective of eliminating the bias of temporal difference algorithms, which is different from the compression-based motivation (Schmidhuber, 1992) we present here. Similar to our approach, their Corrected-TD and TD(n/n) algorithms use estimated forward transition probabilities from tabular models to decay traces. They also use state visitation counts to adapt the learning rate. As we show in Appendix C, their algorithm amounts to using a λ-return based on a 'backward' dynamics model's predictions. In contrast, our λ-return is defined through predictions of a forward model. Forward models can be easier to learn as they are not policy-dependent and can use replay buffers for training (Chelu et al., 2020).

Unlike our model-based approach to select λ, other works have studied model-free RL algorithms with adaptive or state-dependent values of λ. White & White (2016) discuss several factors that could guide the choice of λ in TD algorithms: bias-variance control, confidence in value estimates, etc. They design an objective that optimizes the immediate bias-variance trade-off to generate targets as a mixture of TD(0) and MC. Riquelme et al. (2019) propose an approach to policy evaluation that switches between TD(0) and MC based on confidence estimates of the value at that state. Watkins (1989) proposes the Q(λ) algorithm, which cuts the trace whenever an exploratory action is taken. Xu et al. (2018) use meta-gradients to adjust λ.

A differentiable world model can be used to directly obtain gradients for policy improvement (Werbos, 1987; Munro, 1987; Schmidhuber, 1990). In deterministic environments with continuous actions, we can update a policy by coupling it with a differentiable world model. This idea has been extended to handle stochastic environments/policies and to use actual trajectories (Heess et al., 2015). However, these approaches are not easily applicable to environments with discrete actions and are quite sensitive to inaccuracies in the learned model (Hafner et al., 2021; Meulemans et al., 2023). Our chunking-based approach presents a new way of using world models with actual trajectories for credit assignment in environments with discrete actions.

Learned models can also be used for imagination-based planning (Schmidhuber, 1990; Sutton, 1990; Ha & Schmidhuber, 2018; Hafner et al., 2021). Experience replay can be seen as a non-parametric model and be used to drive a similar effect (Van Hasselt et al., 2019). Using backward-model-based imagination can also speed up credit assignment by assigning credit to all state-actions that could have led to the current outcome (Moore & Atkeson, 1993; Goyal et al., 2018; Pitis, 2018; van Hasselt et al., 2021). The benefits of using imagined experience are orthogonal to our proposed approach which focuses on assigning credit based on real experience.

Alternatives to standard forward dynamics models can be used for assigning credit with real experience. The hindsight credit assignment (HCA) family of approaches learns a temporally extended inverse dynamics model to ascertain the influence of actions (Harutyunyan et al., 2019; Alipov et al., 2021; Meulemans et al., 2023). Similar to backward models, temporally extended inverse dynamics models are policy-dependent and can be challenging to learn.

Several works identify the need to move beyond conventional TD(λ) to tackle challenging temporal credit assignment problems; see the recent survey by Pignatelli et al. (2023). Finally, Arumugam et al. (2021) present an information-theoretic formulation of the credit assignment problem that may present interesting connections to our compression-based approach.

6. Limitations

A crucial limitation of our approach is that learning a model of the world can be challenging and expensive. Furthermore, our approach requires a 'generative' model that can estimate the probability of the next state, given the current state and action. Nevertheless, our approach could be particularly use-

ful when pre-trained generative models are already available or are being learned for imagination-based planning.

In this paper, we focus on tabular value functions to avoid conflating the generalization effects of function approximation with credit assignment while developing an initial understanding of Chunked-TD. Further work is required to extend Chunked-TD to continuous states and actions where probability mass functions are unavailable. One possibility is to chunk with discretized tokens (Janner et al., 2021) or with discrete latent representations (e.g., Van Den Oord et al., 2017). Another option is to apply a transformation on the error in the model's prediction to obtain λ_t. Systematically extending our approach to such settings requires careful consideration of additional design choices (and associated hyperparameters) and is deferred to future work.

Our approach relies on a suitable reward decomposition to solve hard credit assignment problems like the key-to-door (Section 4.3). Learning such decompositions is challenging in general, but some works aim to tackle similar problems through the use of successor features (Barreto et al., 2017), reward features (Meulemans et al., 2023) and general value functions (Van Seijen et al., 2017).

7. Conclusion

In this paper, we introduce Chunked Temporal Difference, an approach that uses predictions from a learned model to construct adaptive λ-returns for value learning in on-policy settings. Our work presents a new perspective on how learned world models can be used for credit assignment. Building upon the idea of history compression, we use model predictions to reduce trajectory descriptions, thus shortening credit assignment paths.

We propose algorithms that can be implemented online and show that they solve some problems much faster than conventional TD(λ). Our approach introduces a promising way to use a learned model that is less vulnerable to model inaccuracies. Since our approach only uses the learned model to chunk trajectories, it can also be useful when models are only accurate in some regions of the state space.

The clearest direction for future work is to extend this idea to higher dimensional state spaces and continuous actions. Other exciting next steps are to generalize our approach to the off-policy setting and to further explore different methods of chunking trajectories, e.g., with policy-conditioned backward models or even in model-free ways.

Acknowledgements

We would like to thank Kazuki Irie for helpful discussions. This research was supported by an ERC Advanced Grant (742870) and a Swiss National Science Foundation grant (200021_192356). Computational resources for this work were provided by the Swiss National Supercomputing Centre (CSCS project s1205).

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

References

Abbas, Z., Sokota, S., Talvitie, E., and White, M. Selective dyna-style planning under limited model capacity. In *International Conference on Machine Learning*, pp. 1–10. PMLR, 2020.

Alipov, V., Simmons-Edler, R., Putintsev, N., Kalinin, P., and Vetrov, D. Towards practical credit assignment for deep reinforcement learning. *arXiv preprint arXiv:2106.04499*, 2021.

Amos, B., Stanton, S., Yarats, D., and Wilson, A. G. On the model-based stochastic value gradient for continuous reinforcement learning. In *Learning for Dynamics and Control*, pp. 6–20. PMLR, 2021.

Arjona-Medina, J. A., Gillhofer, M., Widrich, M., Unterthiner, T., Brandstetter, J., and Hochreiter, S. Rudder: Return decomposition for delayed rewards. *Advances in Neural Information Processing Systems*, 32, 2019.

Arumugam, D., Henderson, P., and Bacon, P.-L. An information-theoretic perspective on credit assignment in reinforcement learning. *arXiv preprint arXiv:2103.06224*, 2021.

Barreto, A., Dabney, W., Munos, R., Hunt, J. J., Schaul, T., van Hasselt, H. P., and Silver, D. Successor features for transfer in reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

Chelu, V., Precup, D., and van Hasselt, H. P. Forethought and hindsight in credit assignment. *Advances in Neural Information Processing Systems*, 33:2270–2281, 2020.

Deisenroth, M. and Rasmussen, C. E. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pp. 465–472, 2011.

Goyal, A., Brakel, P., Fedus, W., Singhal, S., Lillicrap, T., Levine, S., Larochelle, H., and Bengio, Y. Recall traces: Backtracking models for efficient reinforcement learning. In *International Conference on Learning Representations*, 2018.

Ha, D. and Schmidhuber, J. World models. *Preprint arXiv:1803.10122 (variant at NeurIPS 2018)*, 2018.

Hafner, D., Lillicrap, T. P., Norouzi, M., and Ba, J. Mastering atari with discrete world models. In *9th International Conference on Learning Representations, ICLR*, 2021.

Harutyunyan, A., Dabney, W., Mesnard, T., Gheshlaghi Azar, M., Piot, B., Heess, N., van Hasselt, H. P., Wayne, G., Singh, S., Precup, D., et al. Hindsight credit assignment. *Advances in neural information processing systems*, 32, 2019.

Heess, N., Wayne, G., Silver, D., Lillicrap, T., Erez, T., and Tassa, Y. Learning continuous control policies by stochastic value gradients. *Advances in neural information processing systems*, 28, 2015.

Janner, M., Li, Q., and Levine, S. Offline reinforcement learning as one big sequence modeling problem. *Advances in neural information processing systems*, 34: 1273–1286, 2021.

Kearns, M. J. and Singh, S. Bias-variance error bounds for temporal difference updates. In *COLT*, pp. 142–147, 2000.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations*, 2015.

Meulemans, A., Schug, S., Kobayashi, S., Daw, N., and Wayne, G. Would i have gotten that reward? long-term credit assignment by counterfactual contribution analysis. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Minsky, M. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

Moore, A. W. and Atkeson, C. G. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine learning*, 13:103–130, 1993.

Munro, P. W. A dual back-propagation scheme for scalar reinforcement learning. *Proceedings of the Ninth Annual Conference of the Cognitive Science Society, Seattle, WA*, pp. 165–176, 1987.

Ni, T., Ma, M., Eysenbach, B., and Bacon, P.-L. When do transformers shine in rl? decoupling memory from credit assignment. *arXiv preprint arXiv:2307.03864*, 2023.

Osband, I., Doron, Y., Hessel, M., Aslanides, J., Sezener, E., Saraiva, A., McKinney, K., Lattimore, T., Szepesvari, C., Singh, S., et al. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568*, 2019.

Pignatelli, E., Ferret, J., Geist, M., Mesnard, T., van Hasselt, H., and Toni, L. A survey of temporal credit assignment in deep reinforcement learning. *arXiv preprint arXiv:2312.01072*, 2023.

Pitis, S. Source traces for temporal difference learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

Puterman, M. L. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

Riquelme, C., Penedones, H., Vincent, D., Maennel, H., Gelly, S., Mann, T. A., Barreto, A., and Neu, G. Adaptive temporal-difference learning for policy evaluation with per-state uncertainty estimates. *Advances in Neural Information Processing Systems*, 32, 2019.

Rummery, G. A. and Niranjan, M. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

Russell, S. J. and Zimdars, A. Q-decomposition for reinforcement learning agents. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 656–663, 2003.

Schmidhuber, J. Making the world differentiable: On using fully recurrent self-supervised neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical Report FKI-126-90, http://people.idsia.ch/~juergen/FKI-126-90_(revised)bw_ocr.pdf, Tech. Univ. Munich, 1990.

Schmidhuber, J. Neural sequence chunkers. Technical Report FKI-148-91, Institut für Informatik, Technische Universität München, April 1991.

Schmidhuber, J. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992.

Sutton, R. S. *Temporal credit assignment in reinforcement learning*. University of Massachusetts Amherst, 1984.

Sutton, R. S. Integrated architectures for learning, planning and reacting based on dynamic programming. In *Machine Learning: Proceedings of the Seventh International Workshop*, 1990.

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

Sutton, R. S. and Singh, S. P. On step-size and bias in temporal-difference learning. In *Proceedings of the eighth*

Yale workshop on adaptive and learning systems, pp. 91–96. Citeseer, 1994.

Talvitie, E. Self-correcting models for model-based reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

Van Den Oord, A., Vinyals, O., et al. Neural discrete representation learning. *Advances in neural information processing systems*, 30, 2017.

van Hasselt, H., Madjiheurem, S., Hessel, M., Silver, D., Barreto, A., and Borsa, D. Expected eligibility traces. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 9997–10005, 2021.

Van Hasselt, H. P., Hessel, M., and Aslanides, J. When to use parametric models in reinforcement learning? *Advances in Neural Information Processing Systems*, 32, 2019.

Van Seijen, H., Van Hasselt, H., Whiteson, S., and Wiering, M. A theoretical and empirical analysis of expected sarsa. In *2009 ieee symposium on adaptive dynamic programming and reinforcement learning*, pp. 177–184. IEEE, 2009.

Van Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T., and Tsang, J. Hybrid reward architecture for reinforcement learning. *Advances in Neural Information Processing Systems*, 30, 2017.

Watkins, C. J. C. H. Learning from delayed rewards. 1989.

Werbos, P. J. Building and understanding adaptive systems: A statistical/numerical approach to factory automation and brain research. *IEEE Transactions on Systems, Man, and Cybernetics*, 17, 1987.

White, M. and White, A. A greedy approach to adapting the trace parameter for temporal difference learning. *arXiv preprint arXiv:1607.00446*, 2016.

Xu, Z., van Hasselt, H. P., and Silver, D. Meta-gradient reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

A. Experiments

A.1. Chain-and-Split

A.1.1. Environment details

The environment is depicted in Figure 3. The agent has n actions $\{a_1, a_2, \dots a_n\}$ available at the start/root state. After the first step, there is only one action available in every state. Taking a_1 at the starting state leads to a linear chain of length H, which ultimately provides a deterministic reward of 0.01. This is the optimal choice. All remaining actions lead to a common 'parent' state on the left (s_L), which has a value of zero but is followed immediately by a stochastic transition that leads to a range of different rewards. The state s_L branches to one of w states with equal probability. Each of the w states provides a deterministic reward between -1 and 1, with the average over all states being equal to zero. We set $H = 20$, $n = 10$, and $w = 101$ for our experiment. The discount factor $\gamma = 1$.

A.1.2. Implementation details

We learn tabular Q-value functions for SARSA(λ) and Chunked SARSA. The tabular Q-values are initialized at zero.

Chunked SARSA We use Algorithm 2. To model $\hat{P}^\pi(x_{t+1}, a_{t+1}|x_t, a_t)$ in this case, we simply maintain empirical counts of each possible transition $n(x, a, x')$ and state-action pair $n(x, a)$ and let $\hat{P}(x_{t+1}|x_t, a_t) = \frac{n(x_t, a_t, x_{t+1})}{n(x_t, a_t)}$. We decompose $\hat{P}^\pi(x_{t+1}, a_{t+1}|x_t, a_t) = \hat{P}(x_{t+1}|x_t, a_t)\pi(a_{t+1}|s_{t+1})$. Recall that x_t is the same as s_t in our experiments.

The tabular model is updated at every transition, and counts are incremented before they are used for computing $\hat{P}^\pi(x_{t+1}, a_{t+1}|x_t, a_t)$. This ensures that the counts are never zero for any observed transition.

Hyperparameter selection We consider multiple values of the learning rate α for SARSA(λ) with $\lambda \in \{0, 1.0\}$ and for Chunked SARSA. We consider $\alpha \in 0.1 \times \{2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}, 2^{-7}, 2^{-8}, 2^{-9}, 2^{-10} 2^{-11}, 2^{-12}, 2^{-13}\}$.

The selected choices are provided in Table 2. The selected learning rate for SARSA(0) and Chunked SARSA is the one with the least square error with the true value of ΔQ after 100000 episodes.

Since SARSA(1) is quite poor initially, we selected the learning rate with the least square error with ΔQ after 1000000 episodes instead of 100000 episodes. SARSA(1) fares poorly as it tends to have negative values of ΔQ due to overestimating the value of some action.

Table 2. Selecting the learning rate for Chain-and-Split experiments.

Algorithm	Selected α
SARSA(0)	0.1×2^{-5}
SARSA(1)	0.1×2^{-11}
Chunked SARSA	0.1×2^{-8}

Figure 4 in the main paper shows results for $\lambda \in \{0, 1.0\}$ and for Chunked SARSA with the selected learning rate.

A.1.3. Additional results

Here we show that SARSA(0) and SARSA(1)/MC eventually achieve positive values of ΔQ with a longer evaluation.

We use the same hyperparameters as those selected in Table 2 and evaluate for 1000000 episodes instead of the 100000 episode evaluation of the main paper. Note that the chosen learning rates may not be the best selection for SARSA(0) and Chunked SARSA with 1000000 episodes.

These results are presented in Figure 6.

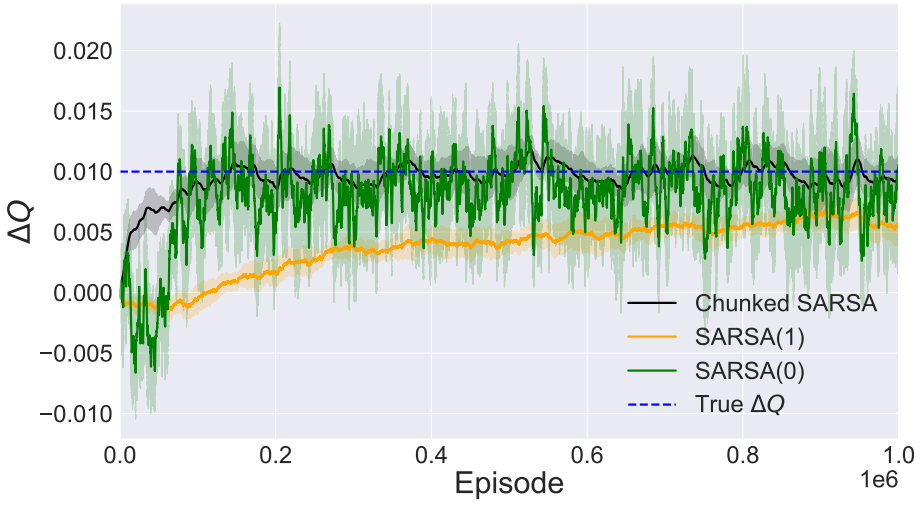

Figure 6. The action-value gap (ΔQ) between optimal action a_1 and the maximum action value among the remaining actions. The true value of $\Delta Q = 0.01$. Shading indicates 95% bootstrapped confidence intervals over 10 independent trials. These results present evaluation over 10 times the number of episodes as those presented in Figure 4

A.2. Accumulated-Charge

A.2.1. Environment details

The agent has two available actions in the initial state (a_1 and a_2) and only one available action (a_1) thereafter. The environment is episodic with H time steps. The discount factor $\gamma = 1$.

The state has three components $s = (s^1, s^2, s^3)$. s^1 is a boolean indicating whether the first action was a_1, s^2 is a non-negative integer that indicates the total charge accumulated so far, and s^3 is the time step.

During the episode, at k values of the time step, the agent stochastically accumulates 'charge'. The k time steps where a stochastic charge is accumulated are fixed for the interaction (all episodes) but are randomly chosen for each independent initialization (seed) of the environment. In our selected setting, the episode lasts 200 steps, and the agent encounters $k = 10$ accumulation points, where it accumulates charge sampled from a binomial variable with $n = H/k = 20$ and $p = 0.5$.

There are no rewards apart from those provided at the final time step. The final reward depends on the amount of charge accumulated and on a bonus b that depends on whether the first action was a_1 or a_2. Choosing a_1 leads to an expected return of b, and choosing a_2 leads to an expected return of zero. We set $b = 0.1$ in our experiment.

Reward definition The final reward (at H) is a sum of three terms $R_H = R^b + R^c + R^d$.

Let $c_0 = 0.5$ if $s^1_H = 1$ (i.e. the agent took a_1 at s_0). Otherwise, if $s^1_H = 0$ (i.e. the agent took a_2 at s_0), $c_0 = -0.5$.

We can now define the individual terms. The bonus term is provided if the agent had taken a_1 in s_0, $R^b = s^1_H b$. The next term R^c depends on the amount of charge accumulated on the trajectory, $R^c = s^2_H c_0$. The final component is $R^d = -c_0 pH$. Note that $\mathbb{E}[R^c + R^d] = 0$, since $\mathbb{E}[s^2_H] = pnk = pH$.

Randomization of charging time steps Evaluation in this environment may be considered as an evaluation over a distribution of MDPs. The choice of the environment seed fixes the location of the time steps where the charge accumulates. These steps remain fixed for all episodes in the environment instance.

A.2.2. Implementation details

We learn tabular Q-value functions for SARSA(λ) and Chunked SARSA. The tabular Q-values are initialized at zero.

Chunked SARSA We use Algorithm 2. We learn a neural network model to predict $\hat{P}^\pi(x_{t+1}, a_{t+1}|x_t, a_t)$. Recall that $x_t = s_t$ in our experiments.

We can decompose $\hat{P}^{\pi}(x_{t+1}, a_{t+1}|x_t, a_t) = \hat{P}(x_{t+1}|x_t, a_t)\pi(a_{t+1}|s_{t+1})$.

Since the state space is quite large, with 200 possible values of charge and time step, we learn a neural network model to predict $\Delta s_{t+1} = s_{t+1} - s_t$ as a categorical distribution. Using state-delta has the benefit of reducing the space of outputs under the model and is similar to models learned in video prediction. The probability of the whole next state (or state-delta) is modeled independently, i.e., taken as the product of the probability of each individual component (each component is a categorical variable in $n = H/k$, the maximum change possible in a state component in this environment). We use the probability of the actual delta and use $\hat{P}(\Delta s_{t+1}|s_t, a_t)$ as the model's predicted probability of the transition $\lambda_{t+1} = \hat{P}(\Delta s_{t+2}|s_{t+1}, a_{t+1})\pi(a_{t+1}|s_{t+1})$.

Our neural network has 3 hidden layers with 256 units each and hyperbolic tangent activation functions. The neural network is trained on a batch sampled from the replay buffer every k_{model} steps in the environment.

Hyperparameter selection The hyperparameters relating to the neural network model were selected as reasonable values based on preliminary experiments. These are presented in Table 3. We use the Adam optimizer for training the neural network (Kingma & Ba, 2015). All other hyperparameters match PyTorch defaults.

Table 3. NN model hyperparameters for accumulated charge experiment.

Hyperparameter	Value
Model trained every k_{model} steps	4
Batch size	128
Model learning rate (η)	0.0001
Weight decay for optimizer	1e-6
Replay buffer size	100000

We present results for SARSA(λ) with $\lambda \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 1\}$ and Chunked SARSA. We consider the following learning rates (α)for all approaches and report results for all settings, $\alpha \in 0.1 \times \{2^3, 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-6}, 2^{-8}, 2^{-10}, 2^{-12}\}$.

A.2.3. Additional results

In the main text, we presented the average number of regretful choices for each approach and learning rate. Here, we present the same results with 95% bootstrapped confidence intervals (see Figure 7). The confidence intervals are quite wide for TD(λ) approaches as they prefer a_1 or a_2 at random (across seeds) after the initial exploratory phase. They fail to reliably prefer a_1, resulting in a high average number of regretful choices with high variance.

A.3. Key-to-Door

A.3.1. Environment details

The state comprises of n_d + 4 components, $s = (s_{key}, s_{door}, s_{d_1} \dots s_{d_{n_d}} s_{treasure}, s_{time})$. There are n_d Boolean distractors ($s_{d_1} \dots s_{d_{n_d}}$), which are sampled independently from a Bernoulli distribution with probability $p = 0.5$. One boolean indicates whether the agent has the key (s_{key}). Two other components are Booleans that indicate whether the agent is in front of a door (s_{door}), and if the current state has the treasure ($s_{treasure}$). The last component indicates the time step (s_{time}). Distractors are absent (with a zero value) in the start state and states with treasure or door bits on.

The agent has two available actions at every step. They correspond to "pick key" and "unlock door". Using "pick key" in the start state turns the "key" boolean on. If the agent has the key, using "unlock door" at the penultimate state will lead to the treasure. Other than this the actions have no effect on the state. The environment is episodic with H steps. The discount factor $\gamma = 1$.

The environment returns a vector of rewards, one for each state component. If the treasure bit is on, it corresponds to a treasure reward of 0.01. Each distractor being on corresponds to a distracting reward of $0.01/(n_d)$. All other components, the key, door, and time step, are associated with zero rewards. Concretely, the reward vector is a deterministic linear function of the next state, $\vec{r}_{t+1} = s_{t+1} \odot (0, 0, \frac{0.01}{n_d} \dots \frac{0.01}{n_d}, 0.01, 0)$, where $\odot$ represents the element-wise product. The overall scalar reward can be obtained by summing over the reward vector.

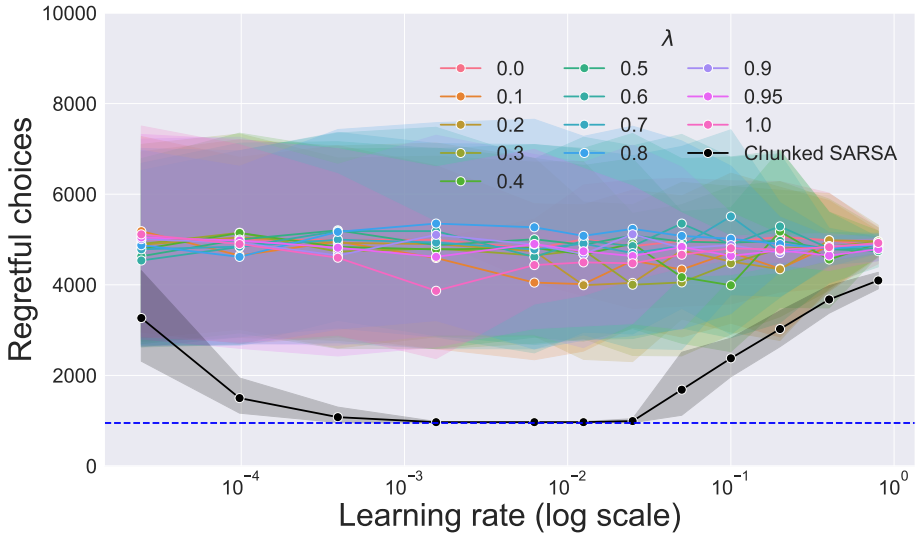

Figure 7. Results from the accumulated-charge environment. The average number of regretful choices over 10 runs for different algorithms with different learning rates for the accumulated-charge experiment. The shading represents 95% bootstrapped confidence intervals. The dashed blue line represents the best possible number of expected regretful choices under the exploration scheme.

In our experiment, we set $H = 100$ and $n_d = 4$.

A.3.2. Implementation details

We use variants of Chunked Expected-SARSA in these experiments (Algorithm 4). Further, since the reward provided by the environment is a vector of reward components, we learn a separate Q-value for each component. See Appendix B.3 for details.

We learn tabular Q-value functions for each reward component. All values are initialized as zero.

C-factored The C-factored algorithm refers to Algorithm 4. We decompose $\hat{P}^{\pi}(x^i_{t+1}, |x_t) = \sum_a \hat{P}(x^i_{t+1}|x_t, a)\pi(a|s_t)$. Recall that $x_t = s_t$ in our experiments.

Hyperparameter selection We learn a neural network model that predicts the next state components given the current state and action ($\hat{P}(s_{t+1}|s_t, a)$). All state components apart from the time step are Booleans, which we model as independent Bernoulli variables. We predict the time step as a categorical variable with H categories.

Our neural network has 2 hidden layers with 128 units each and hyperbolic tangent activation functions. The neural network is trained on a batch sampled from the replay buffer every k_{model} steps in the environment.

The hyperparameters relating to the neural network model were selected as reasonable values based on preliminary experiments. These are presented in Table 4. We use the Adam optimizer for training the neural network (Kingma & Ba, 2015). All other hyperparameters match PyTorch defaults.

Table 4. NN model hyperparameters for key-to-door

Hyperparameter	Value
Model trained every k_{model} steps	1
Batch size	64
Model learning rate (η)	0.0002
Replay buffer size	10000

We conducted a sweep over learning rates for all approaches. We show results for the best learning rate in terms of the minimum number of episodes in which the treasure was not collected (averaged over 10 seeds). The considered candidates and the selected values are provided in Table 5.

Table 5. Selecting the learning rate for Key-to-Door experiments.

Algorithm	Candidates	Selected α
Expected-SARSA(0)	$0.1 \times \{2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}, 2^{-7}, 2^{-8}, 2^{-9}\}$	0.1×2^1
Expected-SARSA(0.1)	$0.1 \times \{2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}, 2^{-7}, 2^{-8}, 2^{-9}\}$	0.1×2^2
Expected-SARSA(0.5)	$0.1 \times \{2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}, 2^{-7}, 2^{-8}, 2^{-9}\}$	0.1×2^0
Expected-SARSA(0.9)	$0.1 \times \{2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}, 2^{-7}, 2^{-8}, 2^{-9}\}$	0.1×2^0
Expected-SARSA(1)	$0.1 \times \{2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}, 2^{-7}, 2^{-8}, 2^{-9}\}$	0.1×2^{-3}
C-default	$0.1 \times \{2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}, 2^{-7}, 2^{-8}, 2^{-9}\}$	0.1×2^0
C-factored	$0.1 \times \{2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3}\}$	0.1×2^{-1}

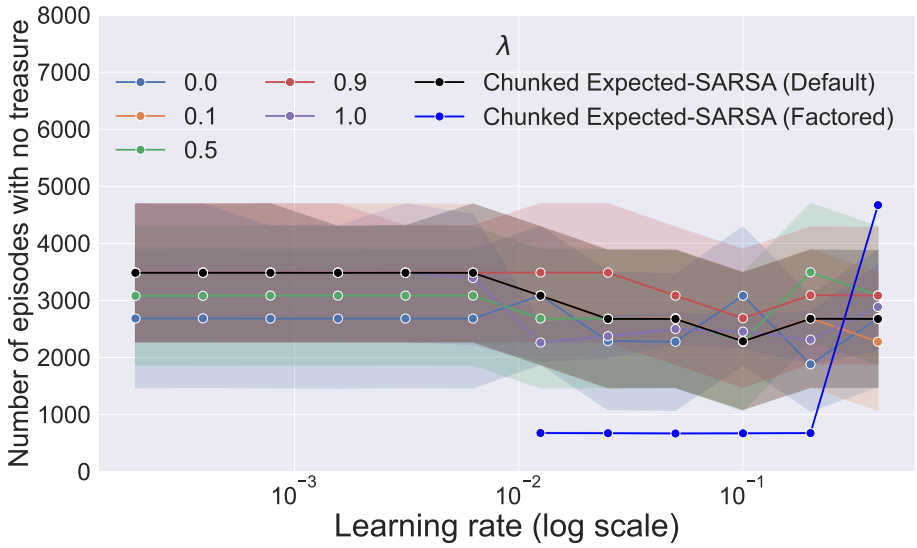

Figure 8. The solid line represents the average number of episodes in which the treasure was not collected. Shading represents 95% bootstrapped confidence intervals.

A.3.3. Additional Results

We present results for all considered learning rates (from Table 5) in Figure 8. We see that C-factored successfully collects the treasure in most episodes across multiple values of the learning rate.

B. Variants of our algorithm

In this section, we present and discuss variants of our chunking-based algorithms.

B.1. n-step Chunked SARSA

Section 3.1 presents a chunked n-step method for bootstrapping from state value functions. Here, we also present an n-step SARSA-like chunking procedure.

In this variant, which applies to bootstrapping from Q-values, we use $\hat{P}^\pi(X_{t+1}A_{t+1}|X_t, A_t)$, the probability of the next percept-action under the current policy. The following factorization can be used in MDPs with Markov policies,

$$\hat{P}^\pi(X_{t+1}, A_{t+1}|X_t, A_t) = \hat{P}(X_{t+1}|X_t, A_t)\pi(A_{t+1}|X_{t+1}), \tag{6}$$

where $\hat{P}(X_{t+1}|X_t, A_t)$ is the estimated transition probability.

Consider the following chunking strategy that applies to learning $\hat{Q}(s, a)$ in episodic MDPs: (1) We always keep the first state-action (s_0, a_0) (2) For $t \geq 1$ we drop state-action (s_t, a_t) with probability $\hat{P}^\pi(x_{t+1}, a_{t+1}|x_t, a_t)$ (3) Whenever we drop a percept-action, we sum the intermediate rewards such that the compressed and original episode have equivalent returns (4) We always keep the terminal state s_T.

The compressed episode of state-action pairs (and the final state) is used to construct targets for the state-action pairs encountered in the entire episode.

B.2. Chunked SARSA and Chunked Expected-SARSA

In this section, we present online algorithms for Chunked SARSA and Chunked Expected-SARSA.

Chunked SARSA Chunked SARSA is presented in Algorithm 2. The forward view of the algorithm would correspond to using $\lambda_{t+1} = \hat{P}^{\pi}(x_{t+2}, a_{t+2}|x_{t+1}, a_{t+1})$ in Equation 4. We used the Chunked SARSA algorithm for our experiments in Sections 4.1 and 4.2.

Chunked Expected-SARSA Chunked Expected-SARSA is presented in Algorithm 3. The forward view of the algorithm would correspond to using $\lambda_{t+1} = \hat{P}^{\pi}(x_{t+2}|x_{t+1})$. As is the case with Expected-SARSA, the value of the next state is computed as a weighted average of Q-values under the policy. We use Chunked Expected-SARSA for our experiment in Section 4.3.

An advantage of the Chunked Expected-SARSA over Chunked SARSA is in situations where multiple actions lead to the same next state. Notice that Chunked SARSA would cut the trace when an action of low probability is taken, even if the next state is the same for all actions ($\lambda_t = \hat{P}(X_{t+1}|X_t, A_t)\pi(A_{t+1}|X_{t+1})$). This would not happen with Chunked Expected-SARSA, as it averages over the consequence of all actions, i.e., $\lambda_t = \sum_a \hat{P}(X_{t+1}|X_t, a)\pi(a|X_t)$.

B.3. Component-wise chunked-TD

The algorithm presented here follows from a combination of Chunked Expected-SARSA (Algorithm 3) and Q-decomposition (Russell & Zimdars, 2003).

We assume the state has d components, each associated with its own reward. Formally, at each time step we receive $\vec{R}_{t+1} = (R^1_{t+1}, R^2_{t+1}, \dots R^d_{t+1})$ and $S_{t+1} = (S^1_{t+1}, S^2_{t+1}, \dots S^d_{t+1})$. Our algorithm relies on the fact that each reward component is associated with a single state component, i.e., $R^i_{t+1} = f(S^i_{t+1})$.

The typical RL reward is the sum of all the component rewards $R_{t+1} = \sum_{i=1}^{d} R^i_{t+1}$, whose expected future sum we would like to maximize.

We learn d Q-value functions $\hat{Q}^1(s,a), \hat{Q}^2(s,a) \dots \hat{Q}^d(s,a)$. Each is learned by Chunked Expected-SARSA with its own trace ($e^i(s,a)$), which is based on the predictability of each individual state/percept component $\hat{P}^{\pi}(X^i_{t+1}|X_t)$. Since the reward is the sum of individual components, the Q-value also decomposes in the same way. Actions are taken according to the global/overall Q-value $\hat{Q}(s,a) = \sum_{i=1}^{d} \hat{Q}^i(s,a)$.

See Algorithm 4 for a complete description. We call this algorithm *C-factored* in our experiment in Section 4.3.

The algorithm that we call *C-default* would use the probability of the *entire* next percept $\hat{P}^{\pi}(X_{t+1}|X_t)$ to decay the trace.

It is important to note that the policy $\pi(.|s)$ is the global policy. So, in the ϵ-greedy case, the policy should be computed using the global Q-value.

Algorithm 2 Chunked SARSA

Ensure: Eligibility $e(s,a) = 0\ \forall (s,a) \in \mathcal{S} \times \mathcal{A}$ at the beginning of each episode

for each $(X_t, A_t, X_{t+1}, A_{t+1})$ of the episode **do**
  Optionally train dynamics model
  for each state-action pair (s,a) **do**
    $e(s,a) \leftarrow \gamma \hat{P}^{\pi}(X_{t+1}, A_{t+1} | X_t, A_t) e(s,a)$
  end for
  $e(S_t, A_t) \leftarrow e(S_t, A_t) + 1$
  $\delta_t \leftarrow R_{t+1} + \gamma \hat{Q}(S_{t+1}, A_{t+1}) - \hat{Q}(S_t, A_t)$
  for each state-action pair (s,a) **do**
    $\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha \delta_t e(s,a)$
  end for
end for

Algorithm 3 Chunked Expected-SARSA

Ensure: Eligibility $e(s,a) = 0\ \forall (s,a) \in \mathcal{S} \times \mathcal{A}$ at the beginning of each episode

for each (X_t, A_t, X_{t+1}) of the episode **do**
  Optionally train dynamics model
  for each state-action pair (s,a) **do**
    $e(s,a) \leftarrow \gamma \hat{P}^{\pi}(X_{t+1} | X_t) e(s,a)$
  end for
  $e(S_t, A_t) \leftarrow e(S_t, A_t) + 1$
  $\delta_t \leftarrow R_{t+1} + \gamma \hat{V}(S_{t+1}) - \hat{Q}(S_t, A_t)$, where $\hat{V}(S_{t+1}) = \sum_a \pi(a|S_{t+1}) \hat{Q}(S_{t+1}, a)$
  for each state-action pair (s,a) **do**
    $\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha \delta_t e(s,a)$
  end for
end for

Algorithm 4 Chunked Expected-SARSA with decomposed rewards

Ensure: Eligibility $e^i(s,a) = 0\ \forall (i,s,a) \in d \times \mathcal{S} \times \mathcal{A}$ at the beginning of each episode

for each (X_t, A_t, X_{t+1}) of the episode **do**
  Optionally train dynamics model
  for each component-state-action (i,s,a) **do**
    $e^i(s,a) \leftarrow \gamma \hat{P}^{\pi}(X^i_{t+1} | X_t) e^i(s,a)$
  end for
  for each component i **do**
    $e^i(S_t, A_t) \leftarrow e^i(S_t, A_t) + 1$
    $\delta^i_t \leftarrow R^i_{t+1} + \gamma \hat{V}^i(S_{t+1}) - \hat{Q}^i(S_t, A_t)$, where $\hat{V}^i(S_{t+1}) = \sum_a \pi(a|S_{t+1}) \hat{Q}^i(S_{t+1}, a)$
    for each state-action pair (s,a) **do**
      $\hat{Q}^i(s,a) \leftarrow \hat{Q}^i(s,a) + \alpha \delta^i_t e^i(s,a)$
    end for
  end for
end for

C. Algorithms from Sutton and Singh 1994

Here, we re-derive the approaches proposed by Sutton & Singh (1994) in order to compare them with the Chunked-TD approach proposed in this work. We modify the approaches slightly to include a reward at each step since Sutton & Singh (1994) assume the rewards only come at the end of the episode.

The basic goal of all the approaches highlighted by Sutton & Singh (1994) is to maintain

$$\hat{V}_{t'}(s) = \frac{1}{n_{t'}(s)} \left(\sum_{t<t':S_t=s} R_{t+1} + \hat{V}_{\tau(t,t')}(S_{t+1}) \right) \tag{7}$$

where $\hat{V}_{t'}(s)$ represents the value estimate for s after accounting for the transition observed at t', $n_{t'}(s)$ is the total number of times s has been visited, and $\tau(t, t')$ represents some time future time compared to t to be defined. The approaches differ only in $\tau(t, t')$, that is, which time step's value estimator is used to evaluate the successor states. Equation 7 is only enforced at the end of episodes for those states s which occur within each episode. The values of states which do not occur within an episode are left unchanged. Note that here, we use a shared time index across all episodes, so if an episode ends at time t, the next episode will begin at $t + 1$. We will assume acyclic MDPs as in Sutton & Singh (1994).

One can imagine we will generally get a better estimator by pushing $\tau(t, t')$ as far into the future as possible relative to t as this means the values used in the right-hand side of Equation 7 will be based on more recent information. However, doing so will also come at the price of algorithmic complexity.

C.1. Naive TD(0)

The simplest approach is just to use $\tau(t, t') = t + 1$ and thus $\hat{V}_{t+1}(S_{t+1})$, that is, the value estimate at the time S_{t+1} was entered. This can be achieved algorithmically with the following simple update on each transition:

$$n(S_t) \leftarrow n(S_t) + 1 \tag{8}$$

$$\hat{V}(S_t) \leftarrow \hat{V}(S_t) + \frac{1}{n(S_t)}(R_{t+1} + \hat{V}(S_{t+1}) - \hat{V}(S_t)), \tag{9}$$

which just computes an incremental average of the value estimates and rewards observed directly following each state S. This is easily seen to be $TD(0)$ with a learning rate of $\frac{1}{n(S_t)}$ and we can identify the target as $G_t^\lambda = R_{t+1} + \hat{V}(S_{t+1})$. This, however, suffers from only backing up reward information one step at a time.

C.2. TD(1/n)

A better approach might be to use the value estimates computed at the end of the episode in the update for each state, effectively equivalent to updating value functions in reverse order from the end of the episode to the start. We can write this as $\tau(t, t') = T(t)$ where $T(t)$ represents the termination of the episode which is in progress at time t and we define $\hat{V}(\perp) = \hat{V}_{t'}(\perp) = 0$ for the terminal state $\perp$ at all times t'.

To express this algorithmically, let's define $\tilde{V}_t$ to be the value we will set $\hat{V}_{T(t)}(S_t)$ to at the termination time $T(t)$. Note that this is different from the λ-return, toward which $\hat{V}(S_t)$ is normally updated only incrementally. With this definition, we can express the approach as follows:

$$n(S_t) \leftarrow n(S_t) + 1 \tag{10}$$

$$\tilde{V}_t \leftarrow \hat{V}(S_t) + \frac{1}{n(S_t)}(R_{t+1} + \tilde{V}_{t+1} - \hat{V}(S_t)). \tag{11}$$

Again, this is just performing an incremental average, but this time using the final value target $\tilde{V}_t$ for each state rather than its value at the time the state is visited. * We can identify the λ-return which acts as the update target for $\hat{V}(S_t)$ as

*Note that the assumption of acyclic MDPs avoids possible circular reference here if a particular state occurs multiple times in an episode.

Algorithm 5 TD$(1/n)$

Ensure: Trace $e(i) = 0\ \forall i$ at the beginning of each episode
1: **for** each step of the episode **do**
2: $n(S_t) \leftarrow n(S_t) + 1$
3: $e(S_t) \leftarrow 1$
4: $\delta_t \leftarrow R_{t+1} + \hat{V}(S_{t+1}) - \hat{V}(S_t)$
5: **for** each state s **do**
6: $\hat{V}(s) \leftarrow \hat{V}(s) + \frac{1}{n(S_t)}\delta_t e(s)$
7: $e(s) \leftarrow e(s)\frac{1}{n(S_t)}$
8: **end for**
9: **end for**

$G_t^\lambda = R_{t+1} + \tilde{V}_{t+1}$. We can then derive a recursive expression for G_t^λ as follows

$$\begin{aligned}
\tilde{V}_t &= \hat{V}(S_t) + \frac{1}{n(S_t)}(G_t^\lambda - \hat{V}(S_t)) \\
G_t^\lambda &= R_{t+1} + \tilde{V}_{t+1} \\
\implies G_t^\lambda &= R_{t+1} + \frac{1}{n(S_{t+1})}G_{t+1}^\lambda + \frac{n(S_{t+1}) - 1}{n(S_{t+1})}\hat{V}(S_{t+1}).
\end{aligned}$$

So we have in this case $\lambda_t = \frac{1}{n(S_{t+1})}$.

Equivalence between TD$(1/n)$ and corresponding λ-return We can confirm that the TD$(1/n)$ algorithm (Algorithm 5) achieves the update of Equation 11 in an incremental manner.

We consider the update term u_t from Equation 11 for state S_t,

$$u_t = \tilde{V}_t - \hat{V}(S_t) = \lambda_t(G_t^\lambda - \hat{V}(S_t)),$$

where $\lambda_t = 1/n(S_t)$. Substituting the recursive relation for G_t^λ, we get

$$u_t = \lambda_t(R_{t+1} + \lambda_{t+1}G_{t+1}^\lambda + (1 - \lambda_{t+1})\hat{V}(S_{t+1}) - \hat{V}(S_t)).$$

We can add and subtract $\lambda_{t+1}\hat{V}(S_{t+1})$ inside the bracket, to get the following,

$$\begin{aligned}
u_t &= \lambda_t(R_{t+1} + (1 - \lambda_{t+1})\hat{V}(S_{t+1}) + \lambda_{t+1}\hat{V}(S_{t+1}) - \hat{V}(S_t) + \lambda_{t+1}G_{t+1}^\lambda - \lambda_{t+1}\hat{V}(S_{t+1})), \\
u_t &= \lambda_t(\delta_t + \lambda_{t+1}(G_{t+1}^\lambda - \hat{V}(S_{t+1}))),
\end{aligned}$$

where $\delta_t = R_{t+1} + \hat{V}(S_{t+1}) - \hat{V}(S_t)$ is the standard TD-error.

Unrolling further, we obtain that the value of S_t is updated as

$$u_t = \sum_{k=t}^{T-1} \delta_k \prod_{i=t}^{k} \lambda_i. \tag{12}$$

Now, we can verify that this is exactly the update TD$(1/n)$ achieves by the end of the episode.

Since the MDP is assumed to be acyclic, an encountered state is never seen again in that episode. Once a state is encountered, it is 'eligible' for all future TD-errors. For a transition at time step t, $S_t \to S_{t+1}$, the immediate update for S_t is $\lambda_t\delta_t$, since $e(S_t) = 1$ (see Line 6 of Algorithm 5). At the next time step $(t + 1)$, $e(S_t) = \lambda_t$, and the update for S_t would be $\lambda_t\lambda_{t+1}\delta_{t+1}$. Thus, the sum of updates for S_t by the end of the episode is $\sum_{k=t}^{T-1} \delta_k \prod_{i=t}^{k} \lambda_i$, the same as Equation 12.

C.3. TDC

We can take the approach of using future estimates even further by retroactively correcting the value estimates used in past episodes each time a particular transition is revisited. We can express this as $\tau(t, t') = T(t')$. This means rather than simply using value estimates at the end of the episode in which a particular successor state was visited, we retroactively correct past value estimates used in Equation 7 to be equal to those at the end of the current episode for each visited state s. It now becomes a little more involved to formulate incrementally. In particular, we will have to maintain some extra information.

Towards formulating an incremental update, define $n(s', s)$ as the number of times the transition (s', s) has been visited. Also, define $\hat{V}(s', s) = \hat{V}_{T(s',s)}(s')$ where $T(s', s)$ is the termination time of the most recent episode in which the transition from s to s' was visited. The reason for maintaining these quantities is essentially so we can subtract the older estimate $\hat{V}(s', s)$ from the estimator $\hat{V}(s)$ in order to replace it with an updated value each time the transition is visited. With these definitions in place, we are ready to define $\tilde{V}_t$ which, as in the previous section, represents the value we will set $\hat{V}_{T(t)}(S_t)$ to at the termination time $T(t)$.

$$n(S_{t+1}, S_t) \leftarrow n(S_{t+1}, S_t) + 1 \tag{13}$$

$$n(S_t) \leftarrow n(S_t) + 1 \tag{14}$$

$$\begin{aligned}\tilde{V}_t \leftarrow \frac{n(S_t) - 1}{n(S_t)}\hat{V}(S_t) - \frac{n(S_{t+1}, S_t) - 1}{n(S_t)}\hat{V}(S_{t+1}, S_t) \\ + \frac{n(S_{t+1}, S_t)}{n(S_t)}\tilde{V}_{t+1} + \frac{1}{n(S_t)}R_{t+1}.\end{aligned} \tag{15}$$

To see how these updates enforce Equation 7, assume that at time $t - 1$ Equation 7 holds for S_t. Then, at time t, we visit S_t and wish to make the correction for the new visit. Because we maintain $\hat{V}(s', s) = \hat{V}_{T(s',s)}(s')$, we can write

$$\hat{V}(S_t) = \bar{R} + \frac{1}{n(S_t) - 1}\left(\sum_{s' \neq S_t} n(s', S_t)\hat{V}(s', S_t) + (n(S_{t+1}, S_t) - 1)\hat{V}(S_{t+1}, S_t)\right), \tag{16}$$

at the time t for which we are computing $\tilde{V}_t$, where $\bar{R}$ is the average of past rewards following state S_t. Substituting this into the expression for $\tilde{V}_t$ we get:

$$\begin{aligned}\tilde{V}_t = \frac{1}{n(S_t)}\Big(\sum_{s' \neq S_t} (n(s', S_t) - 1)\hat{V}(s', S_t) + (n(S_{t+1}, S_t) - 1)\hat{V}(S_{t+1}, S_t) \\ - (n(S_{t+1}, S_t) - 1)\hat{V}(S_{t+1}, S_t) + n(S_{t+1}, S_t)\tilde{V}_{t+1} + (n(S_t) - 1)\bar{R} + R_{t+1}\Big).\end{aligned} \tag{17}$$

In effect, the update subtracts the outdated estimate from the sum, replaces it with the newer estimate, updates the incremental average for the reward, and corrects the denominator to account for the additional visit to S_t. Having defined $\tilde{V}_t$, we can now try to rewrite it in a form that looks more like a conventional TD update

$$\tilde{V}_t = \hat{V}(S_t) + \frac{1}{n(S_t)}(R_{t+1} + n(S_{t+1}, S_t)\tilde{V}_{t+1} - (n(S_{t+1}, S_t) - 1)\hat{V}(S_{t+1}, S_t) - \hat{V}(S_t)). \tag{18}$$

The analogy to the lambda return in this case is

$$G_t^\lambda = R_{t+1} + n(S_{t+1}, S_t)\tilde{V}_{t+1} - (n(S_{t+1}, S_t) - 1)\hat{V}(S_{t+1}, S_t). \tag{19}$$

Which we can once again put in a recursive form

$$G_t^\lambda = R_{t+1} + n(S_{t+1}, S_t)\tilde{V}_{t+1} - (n(S_{t+1}, S_t) - 1)\hat{V}(S_{t+1}, S_t) \tag{20}$$

$$\tilde{V}_t = \hat{V}(S_t) + \frac{1}{n(S_t)}(G_t^\lambda - \hat{V}(S_t)) \tag{21}$$

$$\begin{aligned}\implies G_t^\lambda = R_{t+1} + n(S_{t+1}, S_t)\frac{n(S_{t+1}) - 1}{n(S_{t+1})}\hat{V}(S_{t+1}) - \\ (n(S_{t+1}, S_t) - 1)\hat{V}(S_{t+1}, S_t) + \frac{n(S_{t+1}, S_t)}{n(S_{t+1})}G_{t+1}^\lambda\end{aligned} \tag{22}$$

Algorithm 6 TDC

Ensure: Trace $e(i) = 0\ \forall i$ at the beginning of each episode
for each step of the episode **do**
  $n(S_{t+1}, S_t) \leftarrow n(S_{t+1}, S_t) + 1$
  $n(S_t) \leftarrow n(S_t) + 1$
  $e(S_t) \leftarrow 1$
  $\delta'_t \leftarrow R_{t+1} + \hat{V}(S_{t+1}) + (n(S_{t+1}, S_t) - 1)[\hat{V}(S_{t+1}) - \hat{V}(S_{t+1}, S_t)] - \hat{V}(S_t)$
  for each state s **do**
    $\hat{V}(s) \leftarrow \hat{V}(s) + \frac{1}{n(S_t)}\delta'_t e(s)$
    $e(s) \leftarrow e(s)\frac{n(S_{t+1},S_t)}{n(S_t)}$
  end for
end for
for each non-terminal S_{t+1} in the episode **do**
  $\hat{V}(S_{t+1}, S_t) = \hat{V}(S_{t+1})$
end for

This looks like a λ-return with $\lambda = \frac{n(S_{t+1},S_t)}{n(S_{t+1})}$, except that instead of $(1-\lambda)\hat{V}(S_{t+1})$ we have a rather elaborate difference term. However, note that in the case where $\hat{V}(S_{t+1}, S_t) = \hat{V}(S_{t+1})$ this extra term reduces to simply $\left(1 - \frac{n(S_{t+1},S_t)}{n(S_{t+1})}\right)\hat{V}(S_{t+1})$. Thus, as long as $\hat{V}(S_{t+1}, S_t)$ is up to date, this is indeed just a normal λ-return with $\lambda = \frac{n(S_{t+1},S_t)}{n(S_{t+1})}$. Otherwise, it uses some extra machinery to subtract the outdated value estimate from the current target before adding the new one.

Note that $\frac{n(S_{t+1},S_t)}{n(S_{t+1})}$ is the transition probability under an empirical *backward* model. Hence the TDC algorithm essentially corresponds to setting lambda based on estimated backward transition probabilities. One could also write this in terms of forward model probabilities as $\frac{n(S_{t+1},S_t)}{n(S_t)}\frac{n(S_t)}{n(S_{t+1})}$ by multiplying by the visitation ratio. As we show in the next section, the online backward view of the algorithm can be implemented using a forward model. The difference arises from whether the learning rate for the update is set to be equal to the count for the state being updated itself or the future state at which the update actually occurs in the backward view. In Chunked-TD we use a constant learning rate, so the correspondence is not exact.

Equivalence between TDC and corresponding λ-return Proceeding similarly to our analysis for TD$(1/n)$ in the previous section, let us consider the update (u_t) made by our recursive lambda return from Equation 22,

$$u_t = \tilde{V}_t - \hat{V}(S_t) = \frac{1}{n(S_t)}(G_t^\lambda - \hat{V}(S_t)).$$

We use the same trick of expanding G_t^λ recursively, to get

$$u_t = \frac{1}{n(S_t)}(R_{t+1} + n(S_{t+1}, S_t)\left(1 - \frac{1}{n(S_{t+1})}\right)\hat{V}(S_{t+1}) - (n(S_{t+1}, S_t) - 1)\,\hat{V}(S_{t+1}, S_t) \\ + \lambda_{t+1}G_{t+1}^\lambda - \hat{V}(S_t)),$$

where $\lambda_{t+1} = \frac{n(S_{t+1},S_t)}{n(S_{t+1})}$.

The above can be re-expressed as

$$u_t = \frac{1}{n(S_t)}(R_{t+1} + \hat{V}(S_{t+1}) + (n(S_{t+1}, S_t) - 1)\left(\hat{V}(S_{t+1}) - \hat{V}(S_{t+1}, S_t)\right) \\ + \lambda_{t+1}G_{t+1}^\lambda - \lambda_{t+1}\hat{V}(S_{t+1}) - \hat{V}(S_t)).$$

Using the definition of the corrected-TD error (δ'_t) from Algorithm 6,

$$u_t = \frac{1}{n(S_t)}\left(\delta'_t + \lambda_{t+1}(G_{t+1}^\lambda - \hat{V}(S_{t+1}))\right)$$

Unrolling the recursion, we get

$$u_t = \frac{1}{n(S_t)} \sum_{k=t}^{T-1} \delta'_k \prod_{i=t+1}^{k} \lambda_i. \tag{23}$$

Now let us consider the sequence of updates for a state S_t under Algorithm 6.

The first term of the update u_t^1 is

$$u_t^1 = \frac{1}{n(S_t)} \cdot \delta'_t \cdot 1$$

These updates are written as update = learning rate · corrected-td-error · trace value of S_t.

Now, in the next update the learning rate is $1/n(S_{t+1})$, and the trace has been decayed by $n(S_{t+1}, S_t)/n(S_t)$, giving us

$$u_t^2 = \frac{1}{n(S_{t+1})} \cdot \delta'_{t+1} \cdot \frac{n(S_{t+1}, S_t)}{n(S_t)}$$

Similarly,

$$u_t^3 = \frac{1}{n(S_{t+2})} \cdot \delta'_{t+2} \cdot \frac{n(S_{t+1}, S_t)}{n(S_t)} \cdot \frac{n(S_{t+2}, S_{t+1})}{n(S_{t+1})}$$

Summing all the updates to and taking $1/n(S_t)$ common, we get $\frac{1}{n(S_t)} \sum_{k=t}^{T-1} \delta'_k \prod_{i=t+1}^{k} \lambda_i$, which matches the update from Equation 23.