Learning Agent State Online with Recurrent Generate-and-Test

by

Amir Samani

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

The concept of state is fundamental to a reinforcement learning agent. The state is the input to the agent's action-selection policy, value functions, and environmental model. A reinforcement learning agent interacts with the environment by performing actions and receiving observations, resulting in the agent's data stream of experience. In many cases, the observations only provide partial information about the state, and the agent needs to learn the state directly based on the data stream of experience. We refer to the state directly learned from the data stream of experience as the *agent state*. Existing methods based on gradient descent, including Real-Time Recurrent Learning (RTRL) and Backpropagation Through Time (BPTT), can learn the agent state. However, these methods are computationally expensive, making them unsuitable for online learning of the agent state.

In this thesis, we propose computationally efficient methods based on the generate-and-test approach to learn the agent state. We study the effectiveness of our generate-and-test methods for learning the agent state on two partially observable multi-step prediction problems—the *trace conditioning problem* and the *trace patterning problem*. The trace conditioning problem focuses on the agent's ability to remember a cue presented in the past to predict a signal in the future. The trace patterning problem is an extension of the trace conditioning problem in which a non-linear combination of observation signals triggers the arrival of a temporally distant signal. In the trace patterning problem, the agent must learn non-linear configurations of observation signals to predict

the signal of interest accurately. Our experiments show that the proposed generate-and-test methods can learn the agent state online and make accurate predictions in both problems mentioned above.

Preface

No parts of this thesis have been published.

To my wife Tina and our little kitty Millo

for all the joy they bring everyday

The road to wisdom? - Well, it's plain and simple to express: Err and err and err again but less and less and less.

– Piet Hein

Acknowledgements

During my study at the RLAI lab, I have been mentored by several people, and I am forever grateful for their support. First and foremost, I wish to express my gratitude to my supervisor Rich Sutton. Rich is an exceptional scientist and inspires me in every meeting we have. Rich patiently taught me how to ask the right questions and design simple yet meaningful experiments. He taught me not to get excited by what I don't understand and work on important and challenging research problems. I feel tremendously lucky to be one of his students. I would also like to thank Rupam Mahmood and Joseph Modayil for agreeing to be on my committee and providing insightful feedback. Rupam helped me to better understand how to design representation search methods which are the foundation of my thesis. Joseph provided great directions when I started my research on the predictive representation of state. I thank Adam White for the discussions we had about finding the right research problems and designing careful experiments. I would also like to thank Martha Steenstrup for the great course about scientific writing and all the soccer games we played—shout out to CS Crackers.

I am grateful to Khurram for his assistance with the step-size adaptation method used in this thesis. Also, I thank Banafsheh for the discussions about animal learning problems. I would also like to thank Shibhansh and Parash for the many great discussion we had about the generate-and-test method. Finally, I would like to thank all the incredible people at the agent state meeting, RLAI, and AMII for providing a wonderful environment for research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Online continual learning involves learning from an unending data stream of experience without reusing past data points. An agent interacting with the world receives feedback in the form of observations of the environment dynamics or outcomes of the actions taken by the agent. The agent then uses the feedback to adapt its predictions and behaviour accordingly. The world is often much larger than the agent and inherently non-stationary. *Online-learning* agents *track* the best solution and thus perform better than agents that learn a fixed sub-optimal solution (Sutton et al., 2007).

A reinforcement learning agent interacts with the environment by taking actions and receiving observations, making up the agent's data stream of experience. The underlying state of the environment is often hidden from the agent, and the agent only receives observations providing partial information about the environment state. The notion of state is central to reinforcement learning, and the agent must learn the state based on the data stream of experience. The data stream of experience contains everything that the agent could know about the environment. The natural world is complicated and vast, and intelligent individuals only receive partial information about the world through their sensory inputs. For instance, objects can be distant and not visible. Despite that, classical conditioning experiments on animals demonstrate that animals

can make accurate long-term predictions, suggesting that they make internal representations of their data stream of experience.

The state is crucial for a reinforcement learning agent. The state is the input to the action-selection policy and value functions. For a model-based reinforcement learning agent, the state is both the input and output of the environmental model. Traditionally, domain experts designed the state based on their knowledge of the environment and what they thought was helpful for the agent to use as the state. However, part of the reinforcement learning strength is that it can learn directly from the data stream of experience in real-time with minimal expert interventions. Consequently, we seek online algorithms to learn state using the data stream of experience. We refer to the state directly learned from the data stream of experience as the agent state to distinguish it from the environment state.

Deep learning methods based on gradient descent are often used to learn the agent state. Most notably, truncated-BPTT and RTRL are two learning algorithms for training recurrent neural networks (RNNs) to represent the agent state (Williams and Zipser, 1989; Williams and Peng, 1990). The computational and memory complexity of these algorithms is not ideal for online learning of the agent state. In contrast, reinforcement learning agents can learn a large body of value functions in an online fashion with linear complexity (Sutton et al., 2011), and we would like the learning algorithm for the agent state to exhibit these properties.

Mahmood and Sutton (2013) propose the *generate-and-test* approach as an efficient and effective method for learning features online with linear computational complexity. The generate-and-test approach is based on the idea of *representation search* in which good representations are searched for by generating candidate features and assessing them through testing. A generate-and-test algorithm consists of two processes, the *generator* and the *tester*. The genera-

tor is responsible for feature generation. The tester evaluates the features and decides which features to eliminate. The computational and memory resources are limited and valuable, and by removing the least useful features, the tester opens the capacity for new features to be generated. Mahmood and Sutton (2013) offer a generate-and-test algorithm for training a feed-forward neural network and show the effectiveness of the algorithm on a synthetic supervised learning problem. In order to extend the generate-and-test idea for learning the agent state, we need to propose novel generators and testers which could be applied to a recursive neural network and sequential data.

Classical conditioning experiments on animals show that they make associations between temporally distant events, allowing them to make accurate multi-step predictions. Pavlov and Anrep (1927) show that after several pairings, an unconditioned stimulus (US) that is naturally appealing to the animal, such as food, gets associated with a conditioned stimulus (CS) such as bell tone. The animal has a natural response to the US, referred to as the unconditioned response (UR). For instance, the dog would start salivating in the presence of the food as form a natural response. Interestingly, after several trials of presenting the bell tone followed by the arrival of the food, the animal would start salivating after hearing the bell tone in anticipation of the food. These experiments suggest that animals make an internal state representing their experiences, enabling them to make multi-step predictions.

Inspired by the classical conditioning experiments, Rafiee et al. (2020) introduce partially observable online multi-step prediction problems. In our study, we focus on the trace conditioning problem and the trace patterning problem. In the trace conditioning problem, the agent needs to remember a CS to predict a temporally distant US—similar to a dog that remembers the bell tone to predict the arrival of the food. In the trace patterning problem, a particular combination of CSs results in the arrival of the temporally distant

US. For instance, the dog would receive the food only if the bell tone was present and the light was absent. The agent can no longer rely on the individual CSs to make accurate predictions and needs to remember a non-linear configuration of the CSs to predict the arrival of the US accurately.

This thesis contributes two generators and accompanying testers for learning the agent state online with linear complexity. We present the architecture for representing the agent state. As our first contribution, we propose the *deep trace generator* to learn *deep trace features* that keep memories of other features and observations signals. Deep trace features enable the agent to remember events from the past to make temporally distant associations. We show the effectiveness of the deep trace generator on the trace conditioning problem. Our second contribution, the *imprinting generator*, makes *imprinting features* that represent non-linear configurations of observation signals. We show that learning imprinting features and deep trace features enable the agent to simultaneously represent non-linear configurations of observation signals and remember them to make accurate multi-step predictions in the trace patterning problem.

The rest of the thesis is organized as follows. In Chapter 2, we cover the necessary background for the rest of the thesis. We discuss temporal-difference learning for online learning of multi-step predictions, the architecture of the agent state, and details of the generate-and-test approach. In Chapter 3, we present the trace conditioning problem and the trace patterning problem that we use in our experiments. In Chapter 4, we introduce the deep trace generator and present our experiments on the trace conditioning problem. In Chapter 5, we introduce the imprinting generator. We show the effectiveness of learning imprinting and deep trace features on the trace patterning problem. In Chapter 6, we discuss related work to the thesis. In Chapter 7, we offer our concluding remarks and directions for future studies.

Chapter 2

Background

This chapter introduces the background knowledge for the following chapters. Section 2.1 defines the multi-step prediction problem and shows how to learn multi-step predictions online using temporal-difference (TD) learning. Section 2.2 describes the agent state architecture and shows how to agent state can be used for learning multi-step predictions. Section 2.3 explains the generate-and-test approach to representation search and discusses some of the challenges we need to address to extend the generate-and-test approach for learning the agent state.

2.1 Learning Multi-Step Predictions Online

The ability to make predictions about signals in the environment can be considered knowledge, often referred to as *predictive knowledge*. The agent can acquire predictive knowledge through interaction with the environment (Sutton et al., 2011). We describe the interaction between the agent and the environment as an *uncontrolled dynamical system*. At time step t, the agent receives the observation $\mathbf{o}_t \in \mathbb{R}^m$, which includes the *cumulant* C_t, the signal of interest, and computes the agent state $\mathbf{s}_t \in \mathbb{R}^n$. We discuss how the agent state is computed and learned based on the observation in the rest of this thesis, but for now, we assume that the agent can compute the agent state

and use it to make predictions. Using the agent state $\mathbf{s}_t$, the agent makes predictions about the future value of the cumulant denoted by $Y_t \in \mathbb{R}$. These predictions are often referred to as *nexting* predictions since they anticipate what happens next. Nexting predictions can be formulated as discounted sum of future cumulants, also known as the expected *return* (Modayil et al., 2014). At time step t, the return $G_t \in \mathbb{R}$ is computed as follows:

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k C_{t+k+1} \tag{2.1}$$

the *discount factor* $\gamma \in [0, 1]$ determines the horizon of prediction, which is how the future cumulants are weighted. The return is used to measure the performance of the agent based on the Squared Return Error (SRE), which is computed as $(Y_t - G_t)^2$.

Temporal-difference (TD) learning is a well-known approach to learning multi-step predictions with *bootstrapping*—updating the estimates based on other estimates (Sutton and Barto, 2018). To update the estimate Y_t with semi-gradient TD(λ), we parameterize the estimate Y_t with the weight vector $\mathbf{w}_t \in \mathbb{R}^n$. At each time step the weight vector $\mathbf{w}_t$ is updated as follows:

$$\delta_t \doteq C_{t+1} + \gamma Y_{t+1} - Y_t \tag{2.2}$$

$$\mathbf{z}_t \doteq \gamma\lambda \mathbf{z}_{t-1} + \nabla_{\mathbf{w}} Y_t \tag{2.3}$$

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t \mathbf{z}_t \tag{2.4}$$

where $\delta_t \in \mathbb{R}$ is the *TD error*, $\mathbf{z}_t \in \mathbb{R}^n$ is the *eligibility trace*, $\lambda \in [0, 1]$ is the decay of the eligibility trace, and $\alpha \in \mathbb{R}$ is the *step-size*. In the case of linear parameterization of the estimate Y_t, the estimate can be computed as $Y_t \doteq \mathbf{w}_t^T \mathbf{s}_t$ and the gradient of the estimate with respect to the weight vector is computed as $\nabla_{\mathbf{w}} Y_t = \mathbf{s}_t$. Note that the estimate Y_{t+1} in Equation 2.2 is computed based on $\mathbf{w}_t$ instead of $\mathbf{w}_{t+1}$.

The performance of the TD learning methods rely on a carefully selected

step-size. Instead of using a single fixed step-size for all the features, we can adapt a vector of step-sizes. Incremental Delta-Bar-Delta (IDBD) algorithm learns individual step-sizes for each feature in the supervised learning setting (Sutton, 1992). Temporal-difference IDBD methods—also known as TIDBD—generalize IDBD for the TD learning setting (Thill, 2015; Kearney et al., 2018). Javed (2021) suggests using the implementation of Thill (2015) when learning with eligibility traces ($\lambda > 0$). Each feature s_t^i has a corresponding step-size α_t^i that is used by the TIDBD(λ) to update the weight w_t^i. At time step t, each step-size α_t^i is updated as follows:

$$\beta_t^i \doteq \beta_{t-1}^i + \theta \delta_t z_{t-1}^i h_{t-1}^i \tag{2.5}$$

$$\alpha_t^i \doteq e^{\beta_t^i} \tag{2.6}$$

$$h_t^i \doteq h_{t-1}^i [1 - \alpha_t^i s_t^i z_t^i]^+ + \alpha_t^i \delta_t z_t^i \tag{2.7}$$

In which the β^i is the parameter we use to change the step-size α^i, θ is the meta step-size, h^i is a decaying trace of the current updates, and the operator $[x]^+$ is x if $x > 0$ and is 0 if $x \leq 0$. The intuition behind TIDBD(λ) is that if the current weight update is positively correlated with previous weight updates, it is more efficient to make larger updates in that direction, and we should increase its corresponding step-size. If the current weight update is negatively correlated with the previous weight updates, we should decrease its corresponding step-size.

2.2 Agent State Architecture

In the previous section, we assume that the agent can compute the agent state based on the observation provided by the environment. In this section we discuss the architecture for representing the agent state based on the observations given by the environment. Observations provide partial information about the environment state, and the agent should learn the agent state based

on the data stream of experience. The data stream of experience consists of actions taken by the agent and observations received from the environment. The action at time step t is denoted by $\mathbf{a}_t \in \mathbb{R}^d$ and the observation is denoted by $\mathbf{o}_t \in \mathbb{R}^m$. The data stream of experience is the sequence

$$\mathbf{a}_0, \mathbf{o}_1, \mathbf{a}_1, \mathbf{o}_2, \mathbf{a}_2, \mathbf{o}_3, ...$$

going on forever for the life of the agent. *History* at time step t contains all the actions and observations up to time t and is denoted by $\mathbf{h}_t$:

$$\mathbf{h}_t \doteq [\mathbf{a}_0, \mathbf{o}_1, \mathbf{a}_1, \mathbf{o}_2, \mathbf{a}_2, \mathbf{o}_3, ..., \mathbf{a}_{t-1}, \mathbf{o}_t]$$

where [.] is the concatenation operator. Directly using the history as the agent state is not ideal as the history grows with time, and we need different parameters for different lengths of history—no shared parameters. The general idea is that the state should be a compact summary of the history useful for predicting or controlling future experiences. Let us denote the agent state at time step t by $\mathbf{s}_t \in \mathbb{R}^n$. Approximating the agent state using the whole history is not computationally feasible. We prefer the agent state to be computed incrementally based on the previous agent state $\mathbf{s}_{t-1}$ and the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$. We refer to this update function as the *state-update* function and it is denoted by u:

$$\mathbf{s}_t \doteq u(\mathbf{s}_{t-1}, \mathbf{o}_t, \mathbf{a}_{t-1}). \tag{2.8}$$

The abstract view of the agent state is demonstrated in Figure 2.1. The state-update function u maps the previous agent state $\mathbf{s}_{t-1}$ and the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$ to the current time step agent state $\mathbf{s}_t$. The agent may use the agent state to learn value functions, policy, and environment model. In the case of learning a multi-step prediction, the agent learn a weight vector from the current time step agent state $\mathbf{s}_t$ and the most recent observation

$\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$. Note that the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$ are used both in the construction of the current time step agent state $\mathbf{s}_t$ and computation of the final prediction.

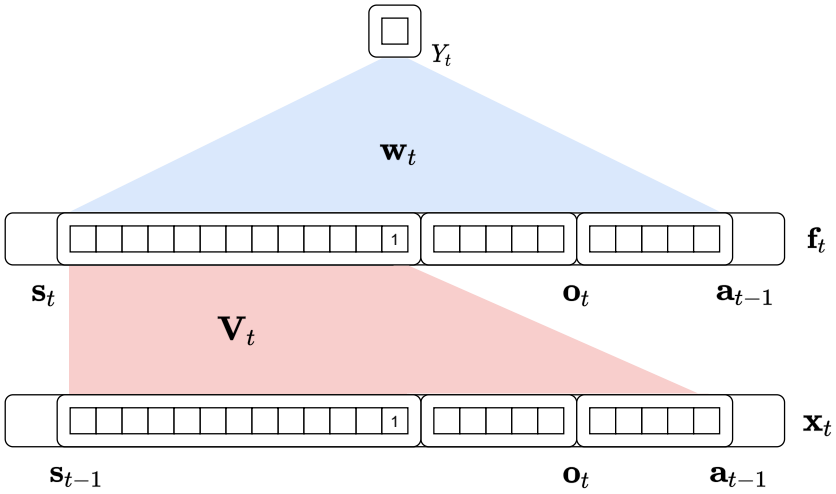

Figure 2.1: At time step t the agent state $\mathbf{s}_t$ is computed using the previous state $\mathbf{s}_{t-1}$ and the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$. The agent learns the outgoing weights $\mathbf{w}_t$ to learn a prediction of interest. There may also be an always-on bias bit in the agent state. To learn a multi-step prediction, the agent may update the weights $\mathbf{w}_t$ with semi-gradient TD(λ).

Now that we know how the agent state is constructed and used to learn a multi-step prediction, let us describe the process in details. At time step t, the previous agent state $\mathbf{s}_{t-1}$ and the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$ are used to construct the current time step agent state $\mathbf{s}_t$. For notational convenience, we define the input as $\mathbf{x}_t = [\mathbf{s}_{t-1}, \mathbf{o}_t, \mathbf{a}_{t-1}] \in \mathbb{R}^{m+n+d}$, which is used to compute the current time step agent state $\mathbf{s}_t$. The current time step input $\mathbf{x}_t$ is mapped to the current time step agent state $\mathbf{s}_t$ using the weight matrix $\mathbf{V}_t$. At time step t, feature s_t^i is connected to x_t^j with the weight of $v_t^{i,j}$. The feature s_t^i is computed as follows:

$$s_t^i = \sum_{j=0}^{m+n+d} v_t^{i,j} x_t^j.$$

Note that we refer to the elements of the weight matrix $\mathbf{V}_t$ as $v_t^{i,j}$, similar to how we refer to the features of the agent state $\mathbf{s}_t$ as s_t^i. Let us denote

the prediction made by the agent at time step t by $Y_t \in \mathbb{R}$. Prediction Y_t is computed based on the current time step agent state $\mathbf{s}_t$ and the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$. For notational convenience, we define $\mathbf{f}_t = [\mathbf{s}_t, \mathbf{o}_t, \mathbf{a}_{t-1}] \in \mathbb{R}^{m+n+d}$, which is the agent state augmented with the most recent observation and action that is used to compute the final prediction. The augmentation provides direct connections from the observation and action signals to the final prediction. The final prediction Y_t is computed as follows:

$$Y_t = \sum_{k=0}^{m+n+d} w_t^k f_t^k.$$

Learning the state-update function u corresponds to learning the weight matrix $\mathbf{V}$. Deep learning methods based on gradient descent, such as RTRL and BPTT, learn the state-update function, but they are computationally expensive. Learning the weight vector $\mathbf{w}$ with semi-gradient TD(λ) is computationally efficient, and it does not require storing past data points, agent states, or weights. We seek learning algorithms for the state-update function to be online and linear in computational complexity and memory. Mahmood and Sutton (2013) propose a generate-and-test algorithm to learn features online in a supervised learning setting. In the next section, we explain the idea behind the generate-and-test approach and present some of the challenges when applying the generate-and-test approach to learn the agent state.

2.3 Generate-and-Test Algorithm

Mahmood and Sutton (2013) study the problem of learning representation in a fully online setting from an endless stream of data. In such a case, the computational complexity of learning the representation should not increase with time. Also, the cost of learning the representation should not exceed the cost of performing on the task. Mahmood and Sutton (2013) apply a search approach to learning the representation with the mentioned computational

constraints. The proposed search method generates the features that compose the representation, and their utility on the task is tested. Then, some of the least useful features are eliminated and replaced with newly generated features. This approach to searching for the representation is referred to as the generate-and-test approach.

The generate-and-test approach divides the problem of feature finding into two sub-problems. The first problem is how to make new features, which is addressed by the *generator*. The second problem is how to evaluate the utility of the features, which concerns the *tester*. To better understand how the generator and the tester are used for representation search, let us discuss the details of the generator and the tester used in Mahmood and Sutton (2013).

Mahmood and Sutton (2013) implement representation search on a synthetic feature finding problem. At every time step t, the binary input vector $\mathbf{x}_t$ and the scalar target y_t are presented, and the goal is to predict the target based on the input vector. The generate-and-test approach searches for features and offers them to the base system. The base system then learns the weights that map the features to the prediction $\hat{y}_t$. The generator makes a new feature i by randomly setting the weights $V^{i,j}$ from the input vector with either -1 or $+1$. The tester uses the magnitude of the learned weights to choose which features to eliminate. Figure 2.2 shows the structure of the system.

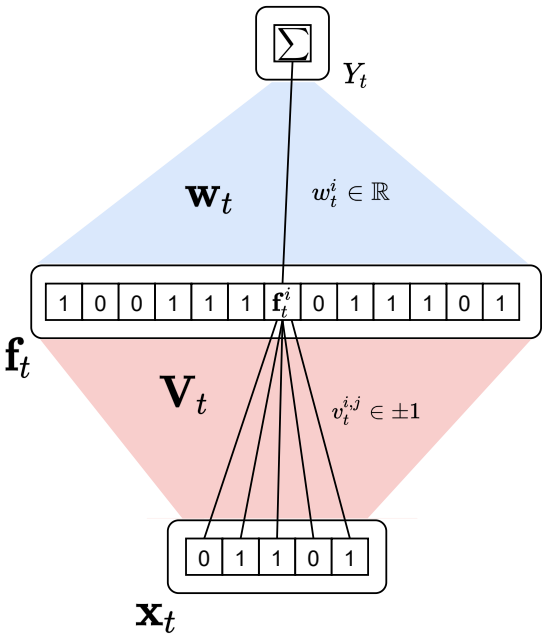

Figure 2.2: The generate-and-test algorithm learns the weight matrix $\mathbf{V}$ by searching through the feature space (Mahmood and Sutton, 2013). The base system updates the feature's outgoing weights using the Least Mean Squares algorithm to learn the prediction. The tester uses the magnitude of the outgoing weights to determine which features are least useful. Intuitively, features with smaller weight magnitude contribute less to final prediction and are better candidates for elimination.

Mahmood and Sutton (2013) show the generate-and-test algorithm is computationally inexpensive, yet it is effective in learning useful features, enabling the base system to make accurate predictions. However, we need different generators and testers to learn the agent state. The agent state uses the previous time step agent state in its computation (see Figure 3). The generator needs to be careful of which connections to make. By densely connecting the features in the agent state with all features in the previous agent state, we can make the tester's task unnecessarily difficult.

In Mahmood and Sutton (2013), each feature could be tested independently of the other features. Intuitively, a feature with a small magnitude of the outgoing weight is not much responsible for the prediction, and removing it, would not significantly change the predictions. On the contrary, in our case,

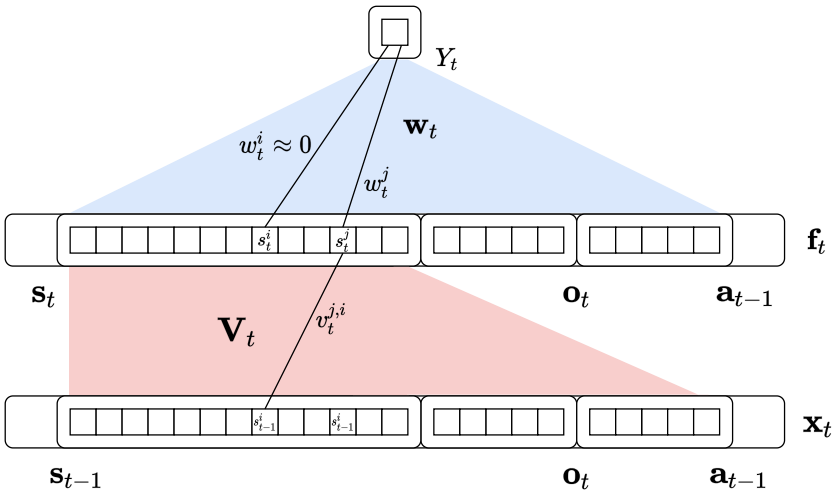

Figure 2.3: The tester needs to consider the indirect effect of removing features. Features that seem useless based on the magnitude of their outgoing weight may be an integral part of another feature that is directly useful to make the prediction. Feature s^i is the input to feature s^j and is useful despite having a near-zero outgoing weight. Removing feature s^i would impair the valuable feature s^j.

a feature can have a near-zero outgoing weight and be input to an essential feature with a large outgoing weight. Figure 2.3 illustrates this situation. The tester should take the connections from $\mathbf{s}_{t-1}$ to $\mathbf{s}_t$ into account. The dense connections from the previous agent state to the current agent state can be problematic to the point that it becomes challenging to remove a feature. Therefore, the generator should be cognizant of this issue not to make the testing process overcomplicated. Similarly, The tester should consider the effect of removing features on other dependent features. In Chapters 4 and 5, we introduce methods for generating and testing features to learn the agent state.

Chapter 3

Animal Learning Problems

This chapter introduces two online multi-step prediction problems used in the rest of this thesis. The first problem, trace conditioning, focuses on the agent's ability to remember a cue and represent it to make predictions about a temporally distant signal. The second problem, trace patterning, requires the agent to make features representing the non-linear configuration of stimuli that help the agent predict a signal that is only activated if a particular pattern of stimuli is presented.

3.1 Learning to Remember

One aspect of partial observability that the agent state needs to address is remembering events that happened in the past and representing them in a useful way for future predictions. Due to limited resources, we can not expect the agent to remember everything that happened in the past. A more reasonable expectation is that the agent can only remember a compact summary of the experiences useful for future predictions. Classical conditioning experiments have shown that animals can make long-term predictions based on cues presented in the past, which suggests that the animals form representations that summarize their experiences.

In classical conditioning experiments, two stimuli with no prior association

in nature are presented to an animal in a particular arrangement over several trials. Each trial starts with the conditioned stimulus (CS), followed by the unconditioned stimulus (US). The animal produces a natural response to the US, referred to as the unconditioned response (UR). After enough trials, the animal would produce a conditioned response (CR) when presented with the CS. For instance, a dog would salivate when receiving food as a natural response. In this case, the food is the US and salivating is the UR. We can present a tone as the CS to the dog before the arrival of the food. There is no natural association between the tone and the food; however, presenting them in this particular order makes the dog associate them and start salivating after hearing the tone—the salivation is also the conditioned response. Note that the salivation starts in anticipation of the food and before the food arrives. There might be a gap between the offset of the CS and the onset of the US when there is no relevant stimulus available, and experiments show that the animal still can accurately predict the arrival of the US. This gap is referred to as the *trace interval*, and if an agent wants to predict the arrival of the US, the agent needs to make features that remember the CS to fill the trace interval gap.

Inspired by these animal experiments, Rafiee et al. (2020) introduce the trace conditioning problem that enables us to study how the agent can learn features that help it remember and represent relevant events from the past. In the trace conditioning problem, the agent needs to predict the arrival of the US based on a temporally distant CS—with a trace interval gap. The CS and the US are not the only stimuli in the problem, and there are additional *distractor stimuli* that provide no information about the CS or the US. The agent should ignore the distractor stimuli to conserve computational resources. Using the trace conditioning problem, we can study two questions. First, since there are multiple stimuli, which stimuli should the agent remember? Second,

how to represent the stimuli to enable the agent to make accurate multi-step predictions?

Figure 3.1 shows an example of the trace conditioning problem. The CS is presented at time step 1 and lasts for 4 time steps—the onset of the CS is at time step 0, and the offset of the CS is at time step 4. The US arrives at time step 15 and lasts for 2 time steps. The time from the onset of the CS to the onset of the US is referred to as the *inter-stimulus interval* (ISI), and in the example, it is 15 time steps. The goal is to predict the arrival of the US based on observing the CS. The challenge is that during the time from the offset of the CS and the onset of the US—the *trace interval*—there are no relevant stimuli available. The agent needs to remember the CS that happened many times before to predict the US accurately.

Prior studies show that the predictions made by animals match the discounted return (Wagner, 1978; Dickinson, 1980), and we can learn these predictions online using TD methods (Sutton and Barto, 1990). Ludvig et al. (2012) confirm that the TD-model of classical conditioning can make accurate predictions compatible with the data observed from animal experiments if the agent state provides features that represent the gap. Chapter 2 discusses how the agent could learn a multi-step prediction problem using TD(λ). We can predict the arrival of the US by learning a multi-step prediction in which the cumulant is set to be the US. The challenge is how to learn the agent state.

Perhaps the simplest form of representing the agent state is to use the *presence representation*. The presence representation has a single binary feature for each stimulus that indicates whether that stimulus is active or not. Despite the simplicity of presence representation, the TD-model of classical conditioning can replicate several of the timing phenomena observed in animal experiments. The presence representation is unable to represent the gap; thus, using presence representation, the agent can not accurately predict the

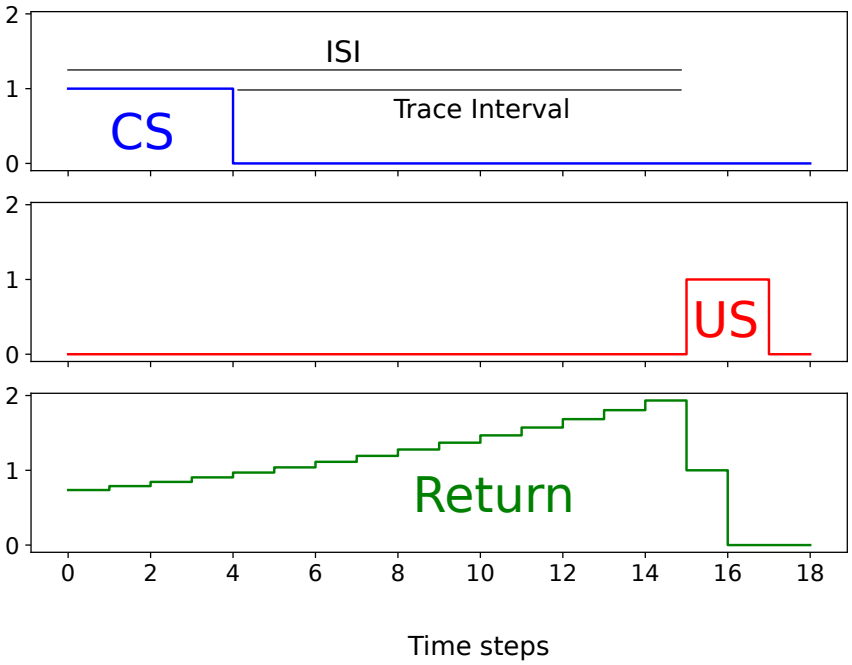

Figure 3.1: The trial starts with the CS being active for 4 time steps. The US arrives after 15 time steps and lasts for 2 time steps. The time from the onset of the CS to the onset of the US is called the inter-stimulus interval (ISI). The gap between the offset of the CS and the onset of the US is called the trace interval. During the trace interval, there is no immediate and relevant observation available to the agent to predict the arrival of the US. The agent needs to remember the CS and represent it in a useful way for predicting the US. Studies show that the prediction made by the animal matches the discounted return, and an agent needs to learn features for the trace interval gap to make similar predictions.

arrival of the US, especially when the trace interval is prolonged. Figure 3.2 shows an example of the trace conditioning problem and the prediction made using the presence representation.

In order to fill the gap with features, Ludvig et al. (2012) introduce the *microstimulus* (MS) representation which is generated by coarse-coding a decaying memory trace of a stimulus. These MS features can extend beyond the offset of the stimuli and provide features that remember the cue, helping the agent learn accurate predictions despite not having access to immediate and

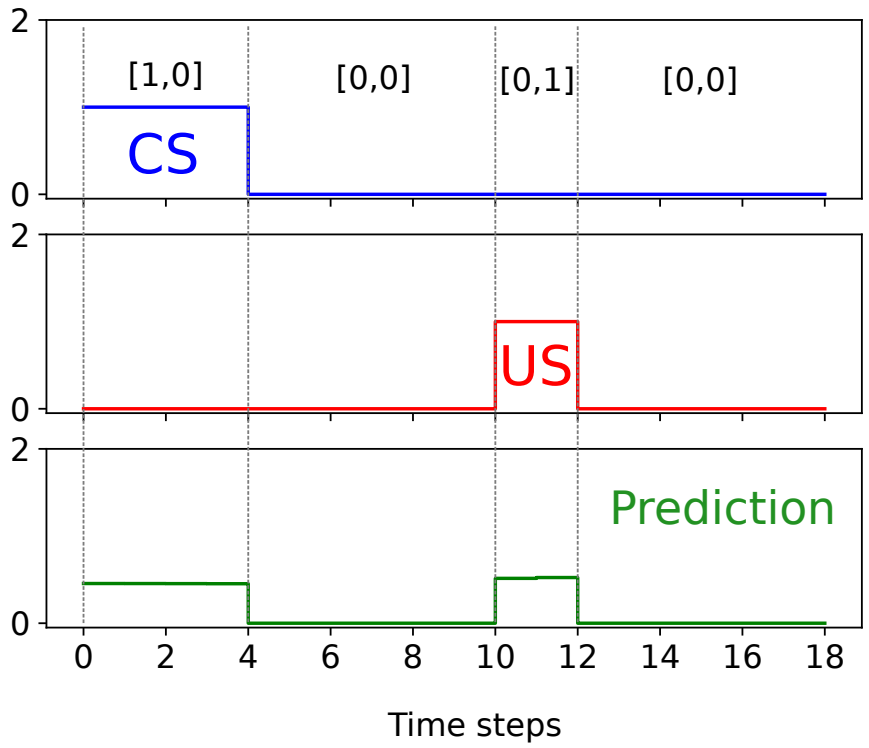

Figure 3.2: The presence representation uses the available observations as the agent state. In the trace conditioning problem, the agent can only make associations when the CS or the US is present. However, the agent can not make accurate predictions for the trace interval since neither stimuli are present. The feature vector is shown for the duration of the trial. Each stimulus has one bit in the feature vector indicating its presence. The agent state has no active features during the trace interval and can not make accurate predictions.

relevant stimuli. Figure 3.3 shows the activity of the MS features during the trial that essentially fills the trace interval gap.

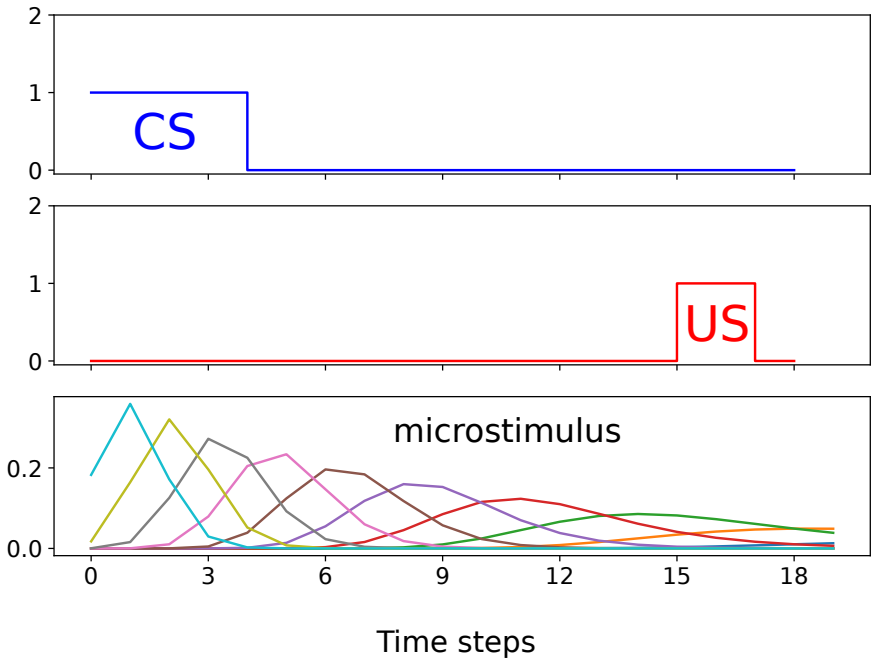

Figure 3.3: The MS features are a set of fixed designed features based on coarse coding of a fading memory trace of a stimulus. The activities of 10 MS features are shown based on a fading memory trace of the CS. The MS features can fill the trace interval gap and help the agent make predictions of the US. However, MS features are designed and thus not learnt, making them unideal to use in the agent state.

Although MS features can remember the CS and fill the trace interval gap, MS representation has significant shortcomings. In order to get MS representation to perform well, the experiment designer needs to set specific parameters for each experiment—different lengths of ISI and arrangement of stimuli may need drastically different parameters. Besides, the designer needs to choose the stimuli to make MS features. Although we can make MS features for all stimuli, it would be unnecessarily computationally demanding when there are several distractors stimuli. Distractors provide no relevant information for the prediction, and by making MS features of the distractors, we are wasting valu-

able resources that otherwise could help the agent make accurate predictions.

Hand designing the features is not practical, and we seek learning algorithms that can learn the agent state by making features of the relevant stimuli and ignoring the distractors. Using the trace conditioning problem, we can solely focus on the problem of remembering the relevant cue and how to represent it to enable the agent to make an accurate prediction. In following chapters, we develop methods to fill the trace interval gap online and learn multi-step predictions using the learned features.

3.2 Learning Non-Linear Configurations

The trace conditioning problem is a special case in which the agent only needs to remember a single CS to predict the arrival of the US. Although there are multiple distractors, none of them provide any valuable information. The agent can successfully predict the arrival of the US by ignoring the distractors and focusing on the CS. The trace patterning problem is a more general case in which patterns in the observation signals trigger the arrival of the US (Rafiee et al., 2020).

In the trace patterning problem, the US arrives only if a particular pattern of stimuli is present—referred to as the *activation pattern*. For example, the dog would receive food only in the presence of tone and the absence of light. In order to make accurate predictions, the agent needs to make features that represent the non-linear configuration of the stimuli. Similar to the trace conditioning problem, distractor stimuli make it harder for the agent to determine which stimuli to focus on to make predictions.

Generally, there can be multiple CSs, and only a particular ordering and pattern of active and inactive CSs results in the arrival of the US. To simplify the problem, we assume that all the CSs and distractors happen simultaneously, and for a particular pattern of active and inactive CSs, the US would

arrive after ISI. The challenge here is to discover the relevant stimuli in the presence of the distractors. For instance, imagine an experiment with two CSs, a light and a tone, and one distractor, an air puff. Figure 3.4 shows all the possible configuration of the CSs and the distractor. The US only occurs if the tone is present and the light is absent—regardless of the status of the distractor. Trace patterning imposes a challenging problem on the agent. The US is still temporally distant, and the agent needs to remember the cues from the past. However, even by representing the individual stimulus with MS features, the agent cannot make accurate predictions.

Finding non-linear relationships of stimuli resembles the XOR problem that artificial neural networks are recognized for solving. However, when the target for the prediction is temporally distant, gradient descent-based methods such as BPTT and RTRL require enormous computational and memory resources (see Chapter 6). In Chapter 5, we develop methods for generating features that represent the non-linear combination of the stimuli that enables the agent to make accurate multi-step predictions.

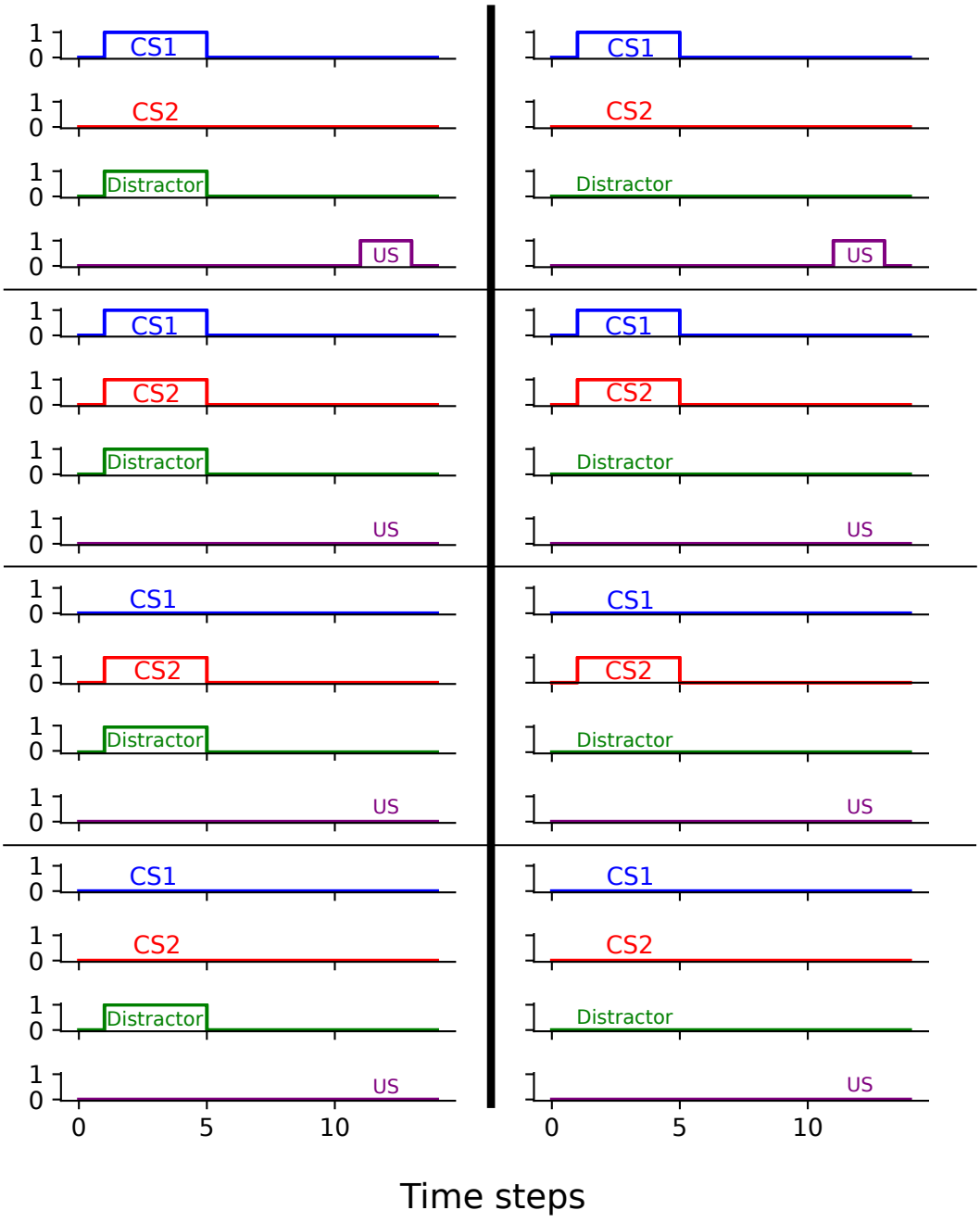

Figure 3.4: In the trace patterning problem, a particular combination of active and inactive CSs triggers the arrival of the US. For instance, the dog would receive food if the tone is present and the light is absent. There may be distractor stimuli, providing no information about the CSs or the US. In this example, there are 2 CSs and 1 distractor. The US would arrive if the CS1 is present and CS is absent—regardless of the distractor stimulus. The agent needs to learn features representing the particular configuration of CS1 and CS2 and remember them to predict the US accurately.

Chapter 4

Deep Trace Generator

In this chapter, we present the first contribution of this thesis, the *deep trace generator*. Chapter 3 introduces the trace conditioning problem, which requires the agent to remember the CS to predict the arrival of the US while ignoring the distractor stimuli. The deep trace generator produces features that remember the CS and fill the trace interval gap enabling the agent to make accurate predictions about the US. In Section 4.1, we introduce the deep trace generate-and-test algorithm. In Section 4.2, we describe the details of experiments on the trace conditioning problem. In Section 4.3, we report results of applying the deep trace generate-and-test algorithm on the trace conditioning problem. Finally, in Section 4.4, we show the robustness of the deep trace generate-and-test algorithm by varying the hyper-parameters.

4.1 Deep Trace Features

In a partially observable setting, the relevant information needed for current predictions may be presented to the agent in the past. The agent needs to remember and represent relevant information in a useful way for future predictions. We can study this problem in isolation using the trace conditioning problem. In Chapter 2, we introduce the agent state architecture and briefly discuss the generate-and-test approach to searching for features. We intro-

duce the deep trace generator to address the challenge of remembering events for future predictions. The deep trace generator produces features that *trace* observation signals or other features. We call these features *deep trace features*.

The deep trace feature s^i traces x^j—either an observation signal or another feature—by connecting itself from the previous time step with the weight $\psi \in (0, 1)$ and x^j with the weight $1 - \psi \in (0, 1)$. At time step t, the deep trace feature s^i_t is computed as follows:

$$s^i_t = \psi s^i_{t-1} + (1 - \psi)x^j_t \tag{4.1}$$

in this update, ψ is the *decay rate*, and x^j is the *source* of deep trace feature s^i. Decay rate determines how quickly the deep trace feature responds to the source and how quickly it fades away. Deep trace features can retain a memory of past observation signals or features. Figure 4.1 depicts an abstraction of the deep trace feature s^i. The source of the deep trace feature s^i is o^j, and the decay rate is 0.9. Figure 4.2 shows an example where o^j becomes active at time step 3 for one time step. The deep trace feature s^i instantly responds to the change in the observation o^j by growing to 0.1—computed based on Equation 4.1. While o^j only lasts for one time step, the deep trace feature s^i remembers that o^j was active, and due to the fading nature of the deep trace feature, it also remembers how long has passed since its activation.

Keeping traces of the events helps the agent to remember cues for future predictions. Existing methods use traces as part of the representation. The traces that are part of the representation are often referred to as *stimulating traces*. Chapter 3 discusses how a trace from the onset of the CS is used to create MS features. Similarly, Rafiee et al. (2020) investigate tile-coded traces as part of the representation. These methods heavily inspire deep trace features, and perhaps a more proper name instead of the deep trace feature would be the *hierarchically generated microstimulus*. For simplicity, we use the

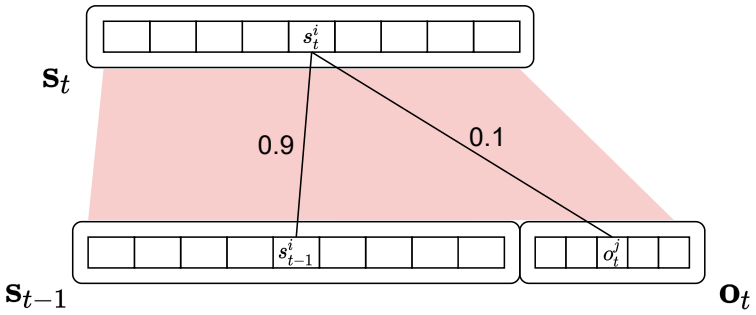

Figure 4.1: The deep trace feature s_t^i traces o_t^j by connecting itself from the previous time step–s_{t-1}^i–with the weight 0.9 and o^j with the weight 0.1. At every time step, the deep trace feature s_t^i is computed based on $s_t^i = 0.9s_{t-1}^i + 0.1o_t^j$. The deep trace feature s_t^i provides a fading memory of the observation o_t^j, which helps the agent remember previous values of o_t^j.

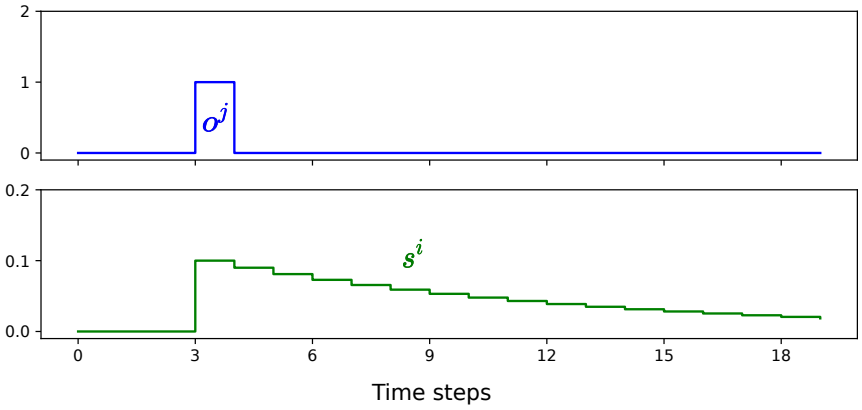

Figure 4.2: The observation signal o^j becomes active at time step 3 for one time step. The deep trace feature s^i traces o^j with a decay rate of 0.9. Although the observation signal o^j is only active at time step 3, the deep trace feature s^i remembers the activation of the observation signal o^j for many more time steps. The activity of the deep trace feature is shown as it slowly fades away with time.

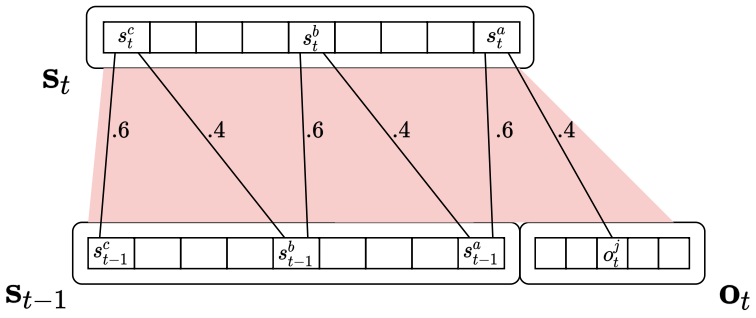

Figure 4.3: The deep trace features s^a, s^b, and s^c trace o^j, s^a, and s^b, respectively. All of decay rates are set to 0.6. The deep trace features provide direct and indirect traces of the observation signal o^j. Multiple direct or indirect deep traces features of an observation signal provide rich representation of the observation signal, helping the agent to summarize its history of interaction with the environment.

term deep traces feature to refer to hierarchically generated microstimulus.

Deep trace features and eligibility traces are different, despite the similarities in how they are calculated. Deep trace features are part of the representation, while eligibility traces are used by the update mechanisms of TD methods. In fact, deep trace features have eligibility traces when learning the outgoing weights with semi-gradient TD(λ).

Deep trace features can be the source for other deep trace features. Figure 4.3 shows three deep trace features and their sources. The deep trace features s^a, s^b, and s^c trace o^j, s^a, and s^b, respectively—all with the decay rate of 0.6. Figure 4.4 shows the values of the deep trace features following the activation of o^j at time step 3. The deep trace feature s^a is quicker to respond to the observation compared to the other deep trace features. It is also quickest to fade away. In essence, all three deep trace features trace the observation, either directly or indirectly. Multiple direct and indirect deep trace features provide a rich memory of the observation signals, enabling the agent to remember a summary of the interaction with the environment.

The deep trace generator needs to decide the source and the decay rate of the deep trace feature when generating a new deep trace feature. Perhaps the

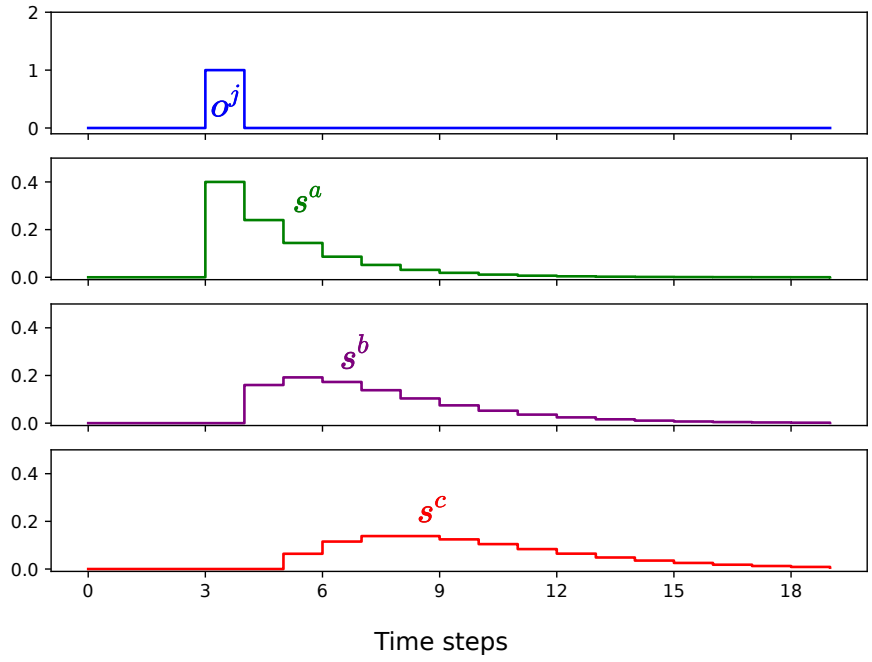

Figure 4.4: The deep trace features s^a, s^b, and s^c trace o^j, s^a, and s^b, respectively—all with the decay rate of 0.6. The observation signal o^j becomes active at time step 3 for one time step. The deep trace feature s^a traces o^j and immediately grows to 0.4—since the decay rate is set to 0.6. The deep trace features s^b and s^c are slower to grow compared to the deep trace feature s^a but are slower to fade away. The direct deep trace feature s^a grows on the same time step as the observation signal o^j is active since the deep trace feature s^a_t is connected to the same time step observation signal o^j_t. However, the deep trace feature s^b grows with a delay of one time step as the deep trace feature s^b_t is connected to the previous time step deep trace s^a_{t-1}. For the same reason, the deep trace feature s^c is delayed by one time step to grow compared to the deep trace feature s^b. Direct and indirect deep trace features of the observation signal o^j provide a rich representation of the observation signal.

most straightforward approach would be randomly choosing the source and the decay rate and letting the tester eliminate useless features and protect useful ones. We study two versions of the deep trace generator. One that gives equal probability to all features and observation signals when choosing the source of a new deep trace feature. The other version still chooses the source randomly; however, the probability of getting selected for each input x^i is based on the magnitude of the outgoing weight, which is computed as follows:

$$P(\text{selecting } x^i) = \frac{|w^i|}{\sum_{j=1}^{m+n} |w^j|}. \tag{4.2}$$

The intuition is that features or observation signals that are directly useful—since we use the outgoing weight magnitude—are better candidates to be the source for new deep trace features.

The tester decides which feature to eliminate based on the moving average of their weight magnitude. Intuitively, features with lower weight magnitude are less likely to be contributing to the final prediction. The tester is inspired by the tester proposed by Mahmood and Sutton (2013) with a few modifications. In Chapter 2, we discuss the challenges of removing features in the agent state architecture compared to the feed-forward setting. The tester refrains from removing features that are the source for other deep trace features. Thus, we ensure that useful features would not lose their source by protecting sources of the deep trace features. The tester also protects newly generated features by using the moving average of the weight magnitude as the metric to decide which features to delete. The moving average of the weight magnitude for the newly generated features is set to the median of all other features—similar to the *second tester* proposed by Mahmood and Sutton (2013). Finally, a top portion of the features with higher weight magnitude is protected from elimination. This guarantees that the tester would not remove top contributing features if no feature from the bottom portion is eligible for elimination—due

to being a source for other deep trace features. Fortunately, the connections among deep trace features are highly sparse, making the tester's job simpler. We illustrate the pseudo-code for our proposed generate-and-test method in Algorithm 1. Table 4.1 is the description of variables and hyper-parameters used in Algorithm 1.

Variable	Description
n_d	current number of deep trace features
c_d	maximum number of deep trace features (capacity)
g_d	maximum number of deep trace features to generate
r_d	maximum number of deep trace features to remove
p_d	protection ratio of deep trace features
μ	weight magnitude moving average decay rate

Table 4.1: Hyper-parameters and variables description for the deep trace generate-and-test algorithm.

4.2 Experiment Details

To confirm the effectiveness of deep trace generate-and-test, we experiment on the trace conditioning problem introduced in Chapter 3. The experiment is a series of trials. Each trial starts with the CS being active for 4 time steps followed by the US, which lasts for 2 time steps. The time from the onset of the CS to the onset of the US is called the *inter-stimulus interval* (ISI). We experiment with three different ISIs—10, 20, and 30. The maximum number of deep trace features—capacity of the agent state—for ISI=10 is 100, for ISI=20 is 200, and for ISI=30 is 300. The time from the onset of the US to the beginning of the next trial is referred to as the *inter-trial interval* (ITI). For each trial, the ITI is uniformly sampled from (80,120). There are ten distractor stimuli in our experiments that can happen during the trial and lasts for 4 time

Algorithm 1: Deep trace generate-and-test algorithm.

Initialize: Set the state-update function u with no initial features by setting weight matrix $\mathbf{V}$ to zero and set n_d to 0, and consider the agent state $\mathbf{s}_0 \in \mathbb{R}^n$ as zero

Initialize: Set weight vector $\mathbf{w} \in \mathbb{R}^{n+m}$ and eligibility trace vector $\mathbf{z} \in \mathbb{R}^{n+m}$ as zeros

Initialize: Set hyper-parameters α, θ, λ, c_d, g_d, r_d, μ, and p_d as desired

for *each observation* $\mathbf{o}_t \in \mathbb{R}^n$ *and* $US_t \in \mathbb{R}$ **do**
- Compute the current state: $\mathbf{s}_t = u(\mathbf{s}_{t-1}, \mathbf{o}_t)$
- Update the weight vector $\mathbf{w_t}$ using TD(λ) or TIDBD(λ)
- **if** $n_d < c_d$ **then**
  - **Deep trace generator:**
  - Generate $min(g_d, c_d - n_d)$ deep trace features
  - **for** *each generated feature* i **do**
    - Set $v^{i,i}$ to ψ by randomly selecting ψ from $(0, 1)$
    - Set $v^{i,j}$ to $1 - \psi$ by selecting the source j randomly
    - Set n_d to $n_d + 1$
- **if** $n_d = c_d$ **then**
  - **Deep trace tester:**
  - Partition the features based on the moving average of the weight magnitude and select the bottom $1 - p_d$ portion of the features
  - Set *num_deleted* to 0
  - **for** *each feature* i *from bottom* $1 - p_d$ *portion of the features* **do**
    - **if** *feature* i *is not a source for other features* **then**
      - Set the outgoing weight w^i to 0
      - Set the corresponding eligibility trace z^i to 0
      - Set $v^{i,j}$ for all $0 < j \leq m + n$ to zeros
      - Set *num_deleted* to *num_deleted* $+ 1$
      - Set n_d to $n_d - 1$
      - **if** *num_deleted* $= r_d$ **then**
        - Break out of the for loop

steps—including the ITI. The ten distractors provide no information about the CS or the US and occur randomly with the Poisson rate of $\frac{1}{10}, \frac{1}{20}, \frac{1}{30}, ..., \frac{1}{100}$, respectively. To measure the agent's performance, we used the Mean Squared Return Error (MSRE) over bins of 1000 time steps. At time step t, we can compute the Squared Return Error (SRE) using $(Y_t - G_t)^2$, in which G_t is the return (see Chapter 2). Table 4.2 is the list of hyper-parameters used in our experiments.

Hyper-parameter	**Value**
step-size α	0.01
meta step-size θ	0.01
eligibility trace decay rate λ	0.9
maximum number of deep traces c_d	100, 200, 300
maximum number of deep trace features to generate g_d	2
maximum number of deep trace features to remove r_d	2
weight magnitude moving average decay rate μ	0.99
deep trace feature protection ratio p_d	0.5

Table 4.2: Hyper-parameters for the deep trace generate-and-test experiments.

4.3 Results

Each experiment consists of 20000 trials which is more than 2 million time steps. We report the results of the performance of the deep trace generate-and-test algorithm in Figure 4.5. Generating deep trace features with random sources either with equal probability or biasing the probability towards features and stimuli with larger weight magnitudes can learn effectively. Since we do not present hyper-parameter studies to show any meaningful differences between these methods, we do not suggest that one method is better than the other. We also include the performance for a fixed representation agent, which uses a fixed set of deep trace features randomly generated at the beginning

of the experiment to show the effectiveness of the search process. Figure 4.6 illustrates the prediction made by the agent compared to the return. The agent predictions match the return, suggesting the agent's ability to effectively fill the trace interval gap with useful features.

Figure 4.7 shows the activity of the top 15 features—based on the weight magnitude—among 100 total features in a sample run with ISI=10. In this example, most of the top features activities occurred closer to the US. The prediction right before the US is at its highest, and top features are chosen based on their weight magnitude, resulting in the top features being highly active around the US.

The distractor stimuli provide no relevant information about the CS or the US. The agent should refrain from wasting valuable and limited computational resources on making features from them. Figure 4.8 shows the dependencies among the features after only 200 trials—the experiment with 100 total features and ISI=10. Most of the deep trace features either directly or indirectly trace distractors as distractors outnumber the relevant stimuli. Figure 4.9 shows the dependencies among features after 20000 trials. Even though the deep trace features were generated randomly, most of the deep trace features directly or indirectly trace the CS or the US through the generate-and-test process.

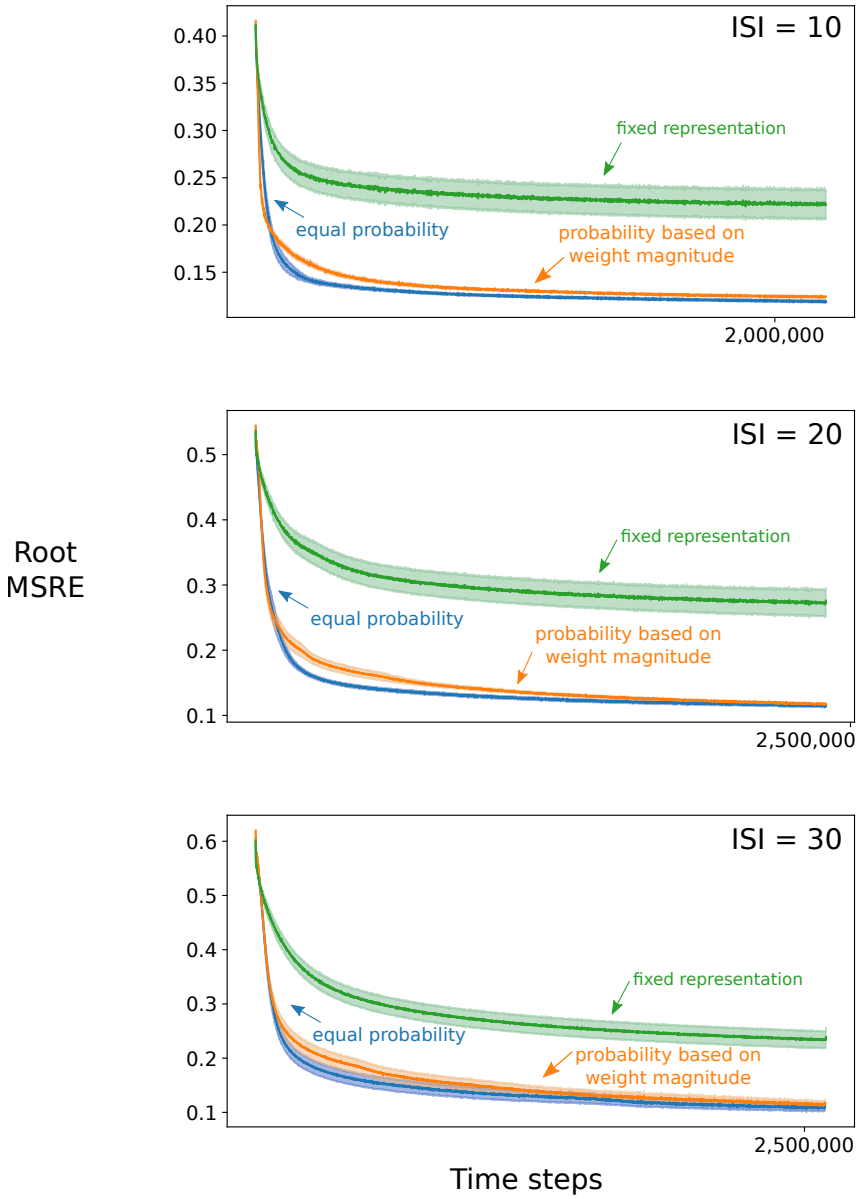

Figure 4.5: The performance of the deep trace generate-and-test algorithm on the trace conditioning problem over the course of 20000 trials. The performance is measured based on the root MSRE over bins of 1000 time steps and is averaged over 30 runs. The shaded area is the standard error. The subplots correspond to the three ISI settings—10,20, and 30. The blue lines represent the generator that gives equal probability to features and stimuli when choosing the source of deep trace features. The orange lines represent the generator that computes the probability of choosing features and stimuli based on their outgoing weight magnitude. Both approaches to selecting the source of deep trace features seem viable options, and comparing the two requires further studies. The green line represents a fixed set of deep trace features randomly generated at the beginning of the experiment and never changed.

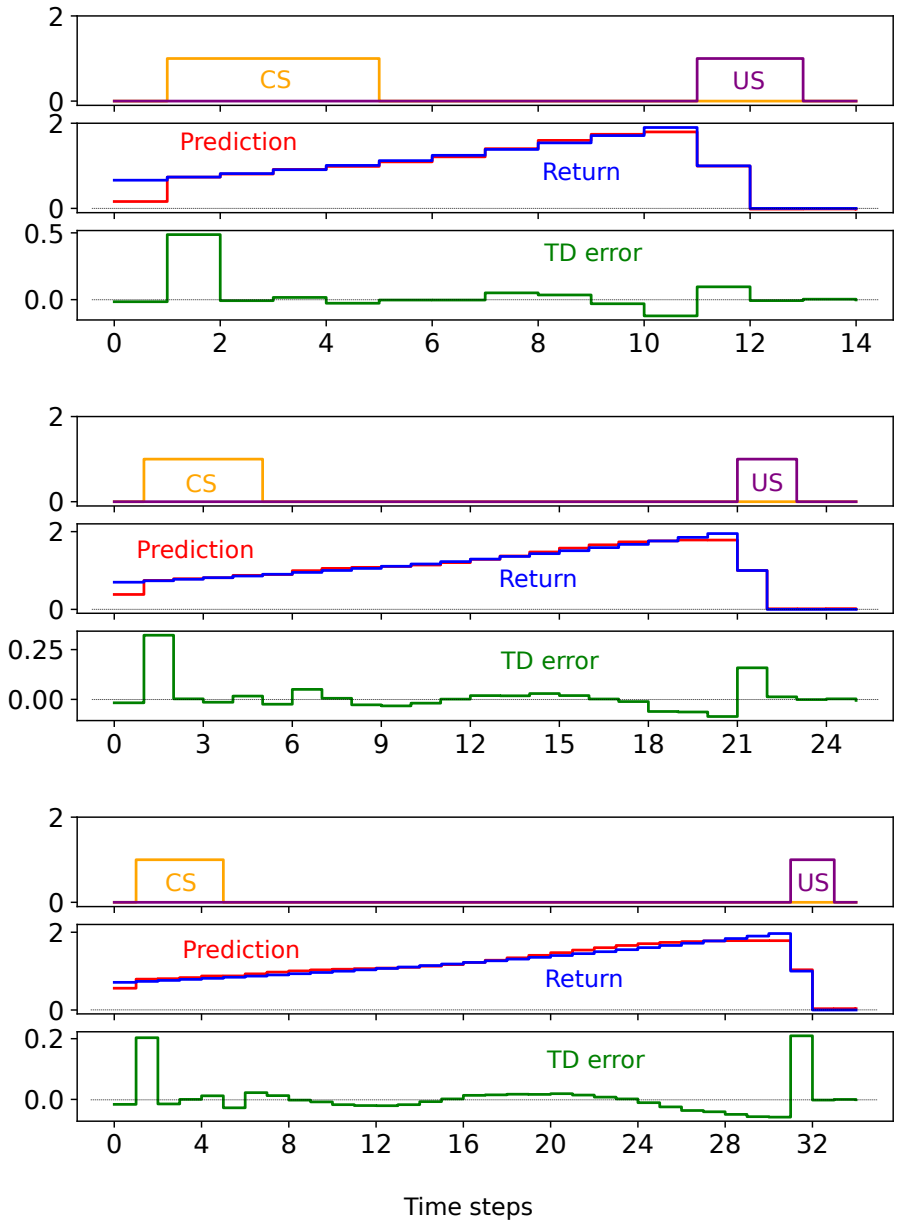

Figure 4.6: The predictions made by the agent match the return in all ISI settings—10, 20, and 30. The predictions shown are made at the final trial of the experiment using the features learned by the deep trace generate-and-test algorithm. The TD error is higher at the onset of the relevant stimuli—the CS and the US—compared to other times. Note that the prediction before the arrival of the CS is non-zero as the ITI is sampled uniformly from (80,120). The agent can predict the arrival of the US of the subsequent trial with a lower degree of precision.

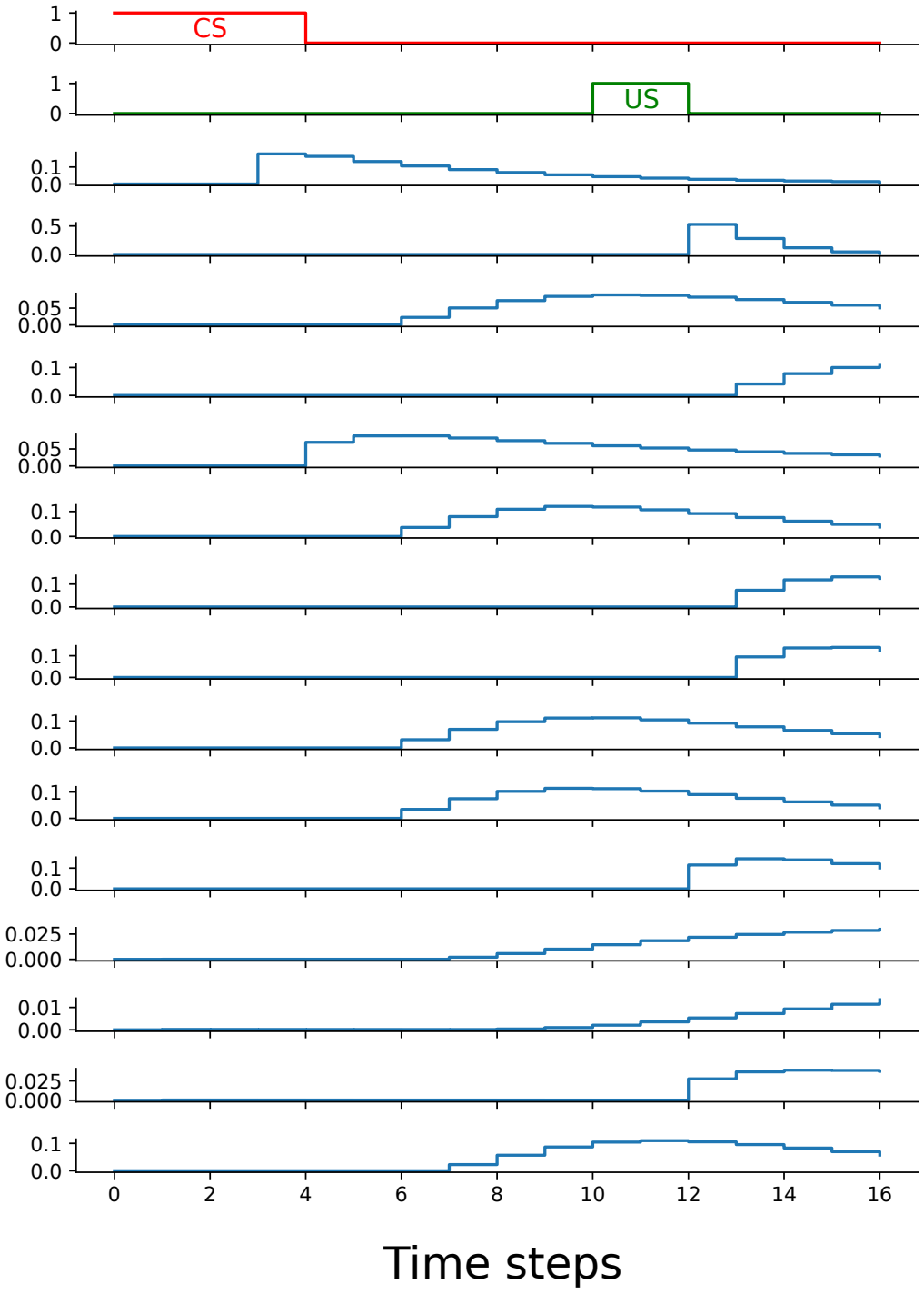

Figure 4.7: The activity of the top features throughout the final trial. The deep trace features learned by the deep trace generate-and-test algorithm make a rich representation of the trace interval gap enabling the agent to make accurate predictions.

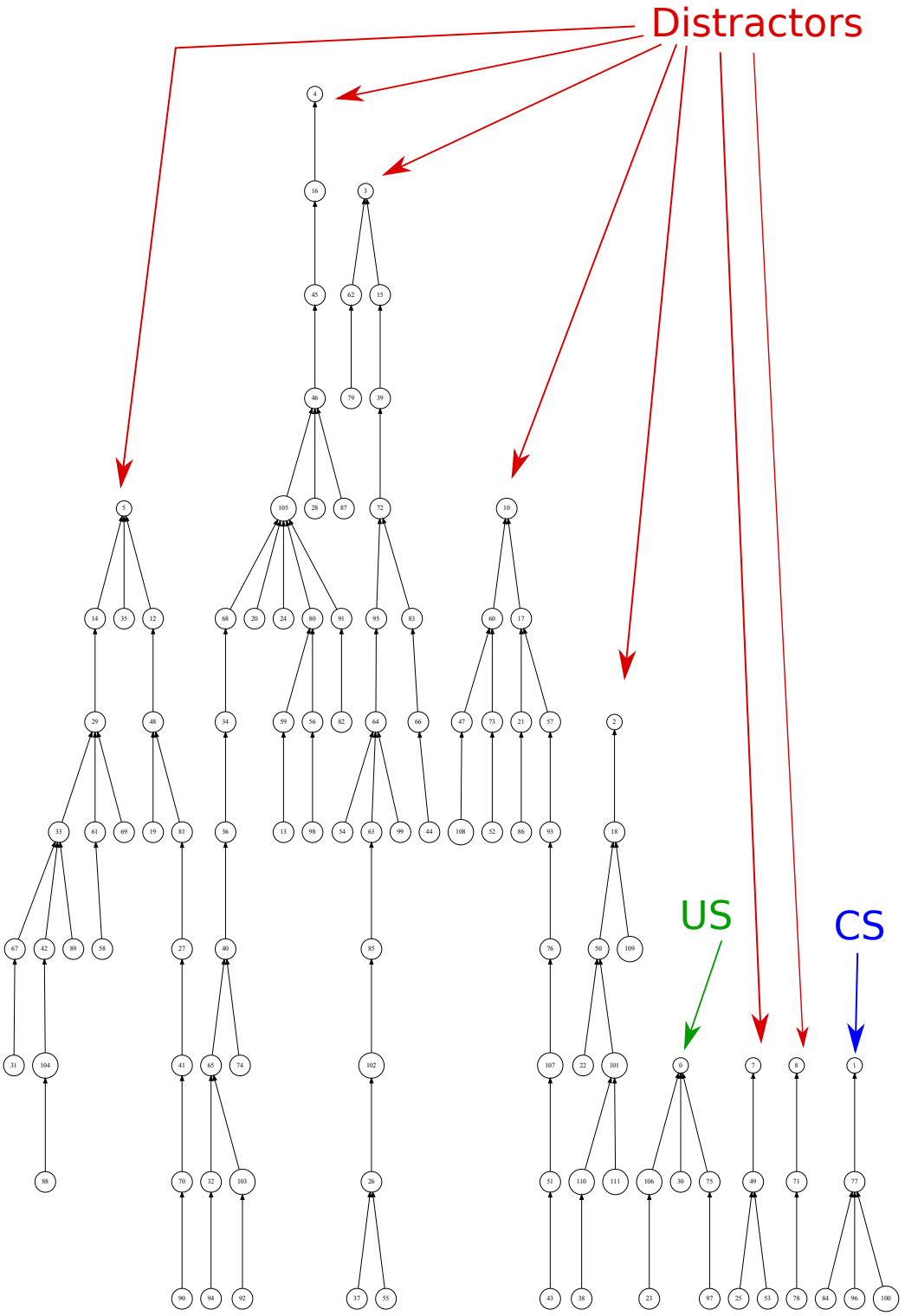

Figure 4.8: The dependency of the deep trace features early in the experiment—after 200 trials. The distractor stimuli outnumber the relevant stimuli, making them more likely to be selected as the source of the deep trace features. In this stage of the experiment, most of the deep trace features directly or indirectly trace the distractor stimuli.

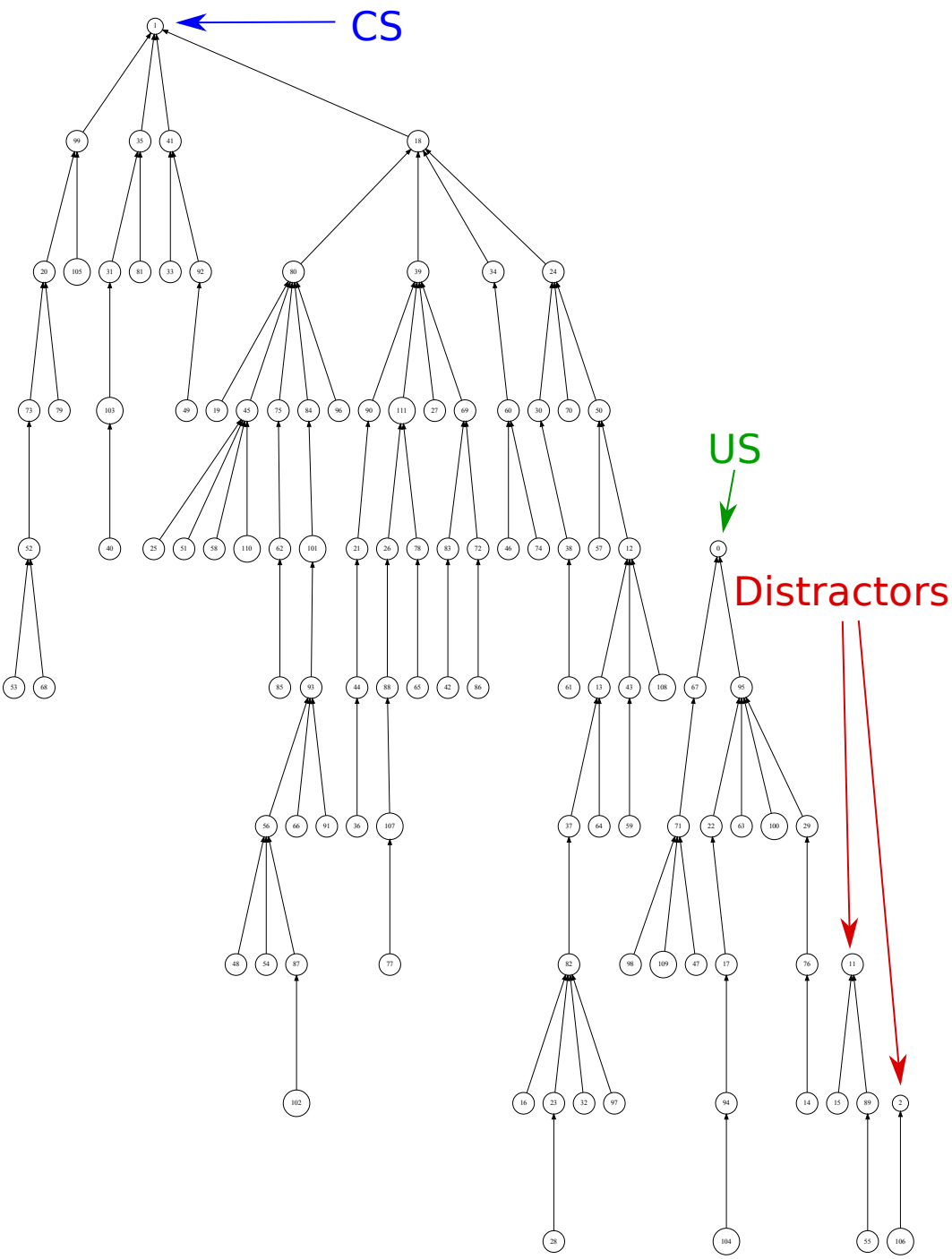

Figure 4.9: The dependency of the deep trace feature at the final trial—on the 20000th trial. Most of the deep trace features directly or indirectly trace the CS or the US—only a small fraction of the deep trace features trace the distractor stimuli.

4.4 Parameter Study

The deep trace generate-and-test algorithm works for a spectrum of step-sizes and meta step-sizes (see Table 4.3)—for ISI=10 and all other settings are identical to Table 4.2. We report the agent's performance based on MSRE over all time steps in Figure 4.10. The missing data points for TIDBD(λ) are due to instability in the learning process.

Hyper-parameter	Value
step-size α	0.001 0.005 0.01 0.05 0.1
meta step-size θ	0.01 0.025 0.05 0.1

Table 4.3: List of step-sizes and meta step-sizes used in the experiment.

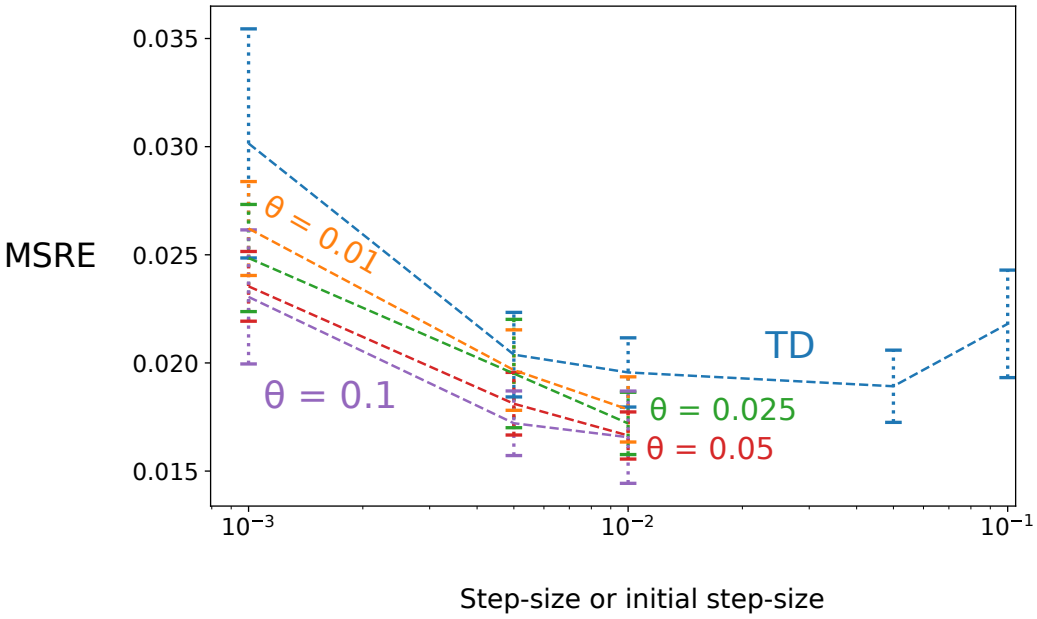

Figure 4.10: The agent's performance based on MSRE over all time steps averaged over 30 runs on the trace conditioning problem with ISI=10. The bars correspond to the standard error. Setting the step-sizes and meta step-sizes can be challenging and require trying various settings. Achieving the best possible performance is out of the scope of this work, and we only show that the deep trace generate-and-test algorithm is robust to the choice of step-size and meta step-size.

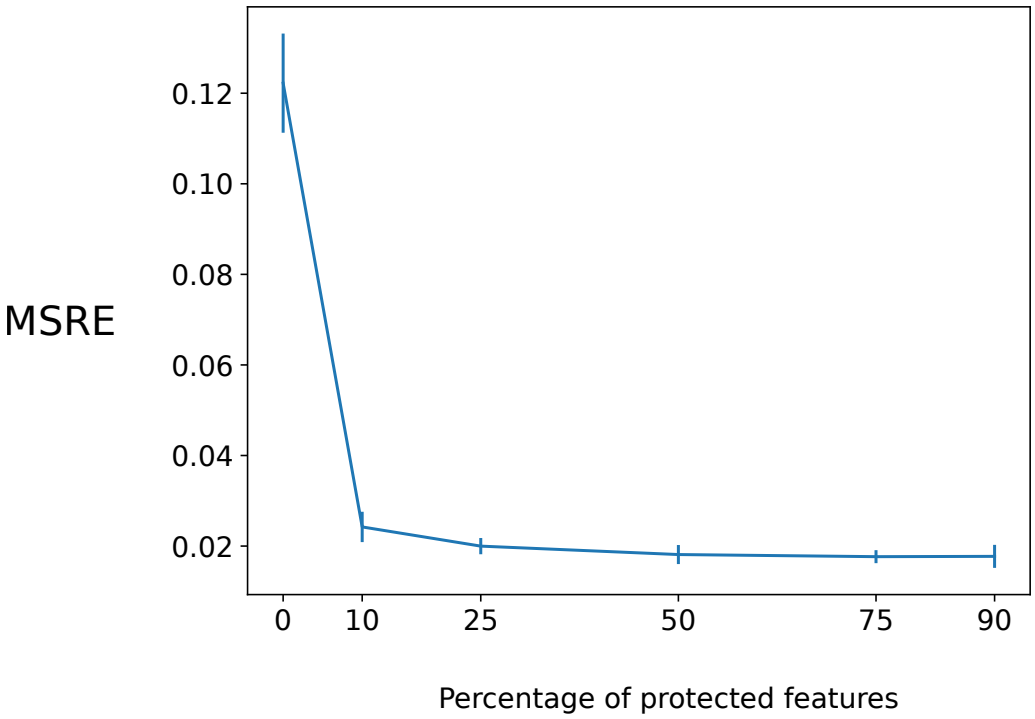

Figure 4.11: The agent's performance based on the MSRE over all time steps averaged over 30 runs on the trace conditioning problem with ISI=10. The bars correspond to the standard error. The x-axis corresponds to the percentage of features being protected by the tester. At every time step, the tester tries to eliminate a certain number of features to make space for the generator to make new features. The newly generated features are protected for deletion, forcing the agent to consider deleting older but possibly useful features from deletion. The results suggest that some features need to be protected; otherwise, the performance is drastically degraded.

Section 4.1 discusses that the newly generated features are protected by setting the moving average of their weight magnitude as the median of all other features. On the other hand, the tester is trying to remove a fixed number of features at every time step. Not protecting a portion of top features results in the tester removing some of the top contributing features. Thus, the tester protects a portion of the deep trace features from elimination. Figure 4.11 shows the agent's performance for a variety of protected features—for ISI=10 and all other settings are identical to Table 4.2. Without any feature protection, the agent's performance is drastically degraded.

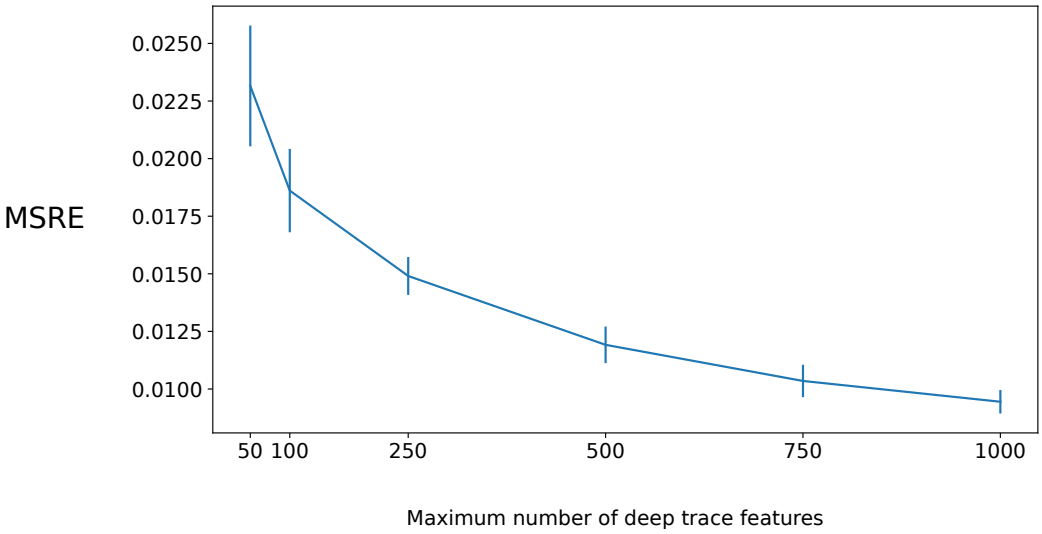

Figure 4.12: The agent's performance based on the MSRE over all time steps averaged over 30 runs on the trace conditioning problem with ISI=10. The bars correspond to the standard error. The x-axis corresponds to the maximum number of deep trace features. The agent's performance improves as we increase the maximum number of deep trace features. Providing more computational and memory resources improves the agent's performance as the deep trace generate-and-test algorithm provides more features and searches the feature space more quickly.

Computational resources are limited and valuable, and we keep the maximum number of deep trace features constant, as they can not grow indefinitely. However, we expect the deep trace generate-and-test algorithm to perform better given more computational and memory resources. Figure 4.12 shows the agent's performance with a various maximum number of deep trace features. The deep trace generate-and-test algorithm scales well with the maximum number of deep trace features.

The hyper-parameters g_d and r_d control the maximum number of deep trace features that we add to or remove from the agent state at every step, respectively. These numbers have been chosen arbitrarily as the other controlling factors are more dominant. Adding or removing a constant number of deep trace features at every step would eventually get controlled by the agent state's capacity or protection measures.

Chapter 5

Configuration Generator

In this chapter, we present our second contribution, the *imprinting* generate-and-test algorithm. In the trace conditioning problem, the agent only needs to remember a single CS. However, in the trace patterning problem, a specific configuration of multiple CSs triggers the arrival of the US. The imprinting generator produces imprinting features representing non-linear configurations in observation signals. The imprinting generator combined with the deep trace generator can make long-term associations between multiple CSs and the US.

In Section 5.1, we introduce the imprinting generate-and-test algorithm. In Section 5.2, we describe the details of our experiments on the trace patterning problem. Finally, in Section 5.3, we report the performance of the imprinting generate-and-test algorithm on the trace patterning problem.

5.1 Imprinting Features

In the trace conditioning problem, the agent can predict the US accurately by remembering the CS. The CS is a single observation signal in the observation vector $\mathbf{o}$, and the agent can remember the CS and make linear features from the CS to fill the trace interval gap using the deep trace generate-and-test algorithm. This chapter relaxes the assumption that a single stimulus is all the agent needs to remember. In the trace patterning problem, there are

multiple CSs, and only a particular combination of active and inactive CSs would result in the arrival of the US. This problem requires the agent to make features that simultaneously respond to a non-linear combination of stimuli and remember it for future predictions. The imprinting generator produces features that respond to a particular configuration of select stimuli. We refer to these features as imprinting features.

The imprinting feature s^i responds to a configuration in observation signals by connecting to select observation signals with either weight of +1 or -1. The weight +1 or -1 corresponds to whether the observation signal should be active or inactive in order for the imprinting feature to respond. Note that the imprinting feature s^i is not necessarily connected to all the observation signals. To make the imprinting feature respond to the particular pattern based on the selected weights, we use the Linear Threshold Unit (LTU) activation function (as in Sutton and Whitehead (1993)). The imprinting feature s^i is computed as follows:

$$s_t^i = \begin{cases} 1 & \sum_{j=1}^{m} v_t^{i,j} o_t^j > \sum_{j=1}^{m} v_t^{i,j} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

We choose the term imprinting since the generated feature recognizes a specific pattern in the observation signals, and when the agent reencounters the same pattern, the imprinting feature gets activated. In essence, the imprinting feature is a snapshot of a pattern that happened to the agent. We can make imprinting features by including other features as well. By imprinting on a selection of features and observation, the imprinting feature considers both the immediate observation signals and what has been summarized already in the agent state. In this thesis, we explicitly imprint only on the observation signals and leave imprinting on features for future studies.

Figure 5.1 shows an example of the imprinting feature s^i connected to the observation signal o^1 and o^2 with the weight of +1 and -1, respectively. Using

Equation 5.1, the imprinting feature s^i would be active if and only if o^1 is active and o^2 is inactive—regardless of other observation signals since the imprinting feature s^i is not connected to those observation signals. Figure 5.2 shows all the possible cases for o^1 and o^2 and the resulting US. Imprinting feature s^i captures the non-linear configuration necessary to predict the arrival of the US accurately.

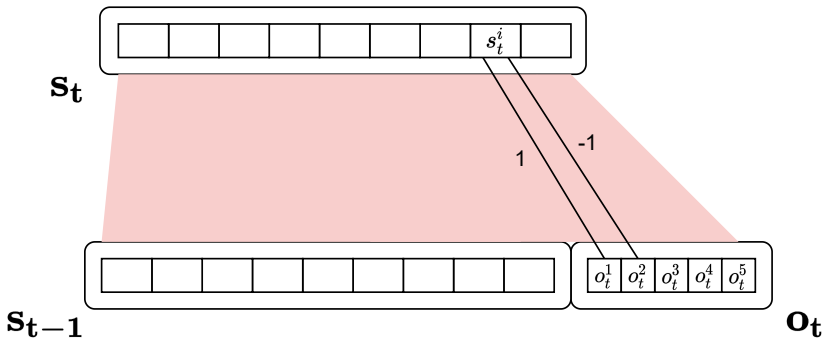

Figure 5.1: Imprinting features represent non-linear configurations of observation signals. The imprinting feature s^i is connected to observation o^1 with a weight of +1 and the observation o^2 with a weight of -1, resulting in the imprinting feature s^i becoming active if the observation o^1 is active and observation o^2 is inactive.

The challenge for the imprinting generator is when to make imprinting features and which observation signals to connect. In our implementation, the imprinting generator makes new features if there is a non-zero activity in the observation signals—not all observation signals are inactive. It is not easy to decide which observation signals should be selected for making connections. Randomly selecting the observation signals and their weights result in 3^m possible imprinting features—m observation signals. The imprinting generator uses the outgoing weight of the observation signals to the final prediction to decide which observation signal to include in the imprinting feature. The intuition is similar to the tester that utilizes the outgoing weights of features to measure their usefulness. The probability of including the observation o_t^i in the creation of new imprinting features at time step t is computed as follows:

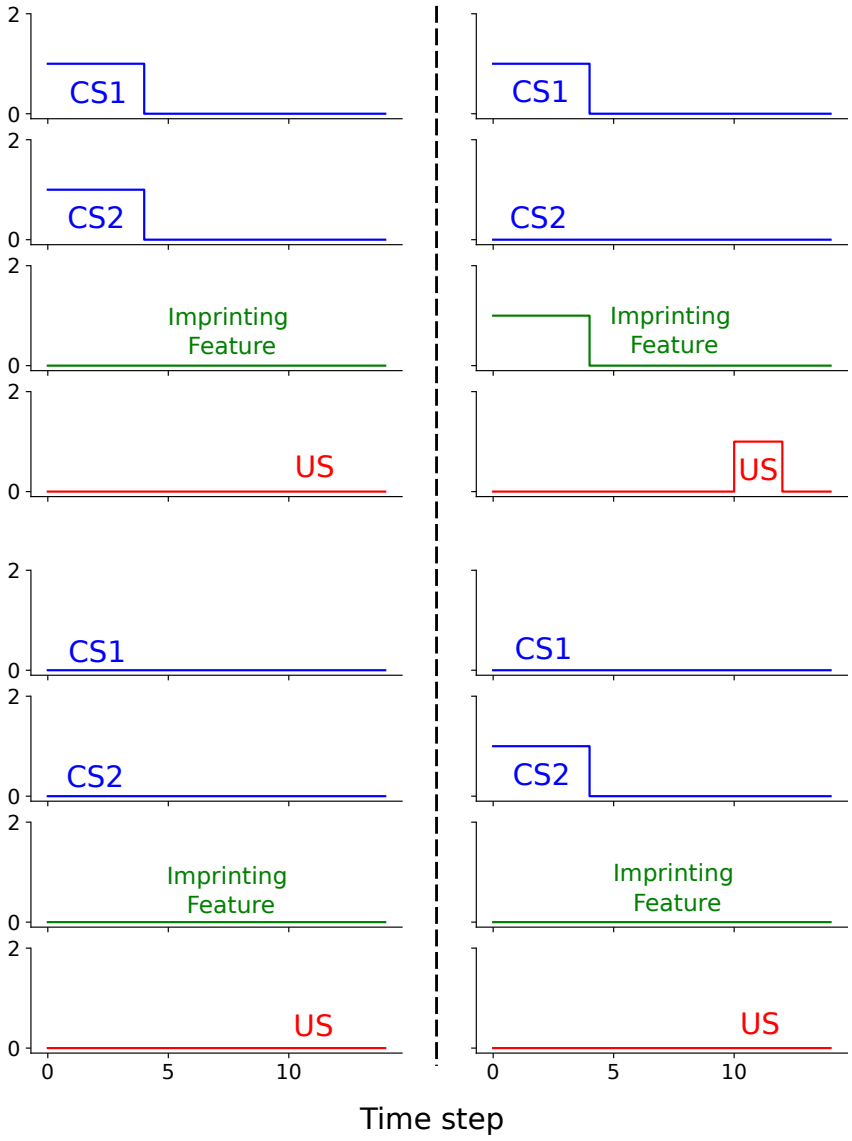

Figure 5.2: In the trace patterning problem, a particular configuration of active and inactive CSs results in the arrival of the US. In this example, the US only occurs when CS1 is active, and CS2 is inactive. By just remembering individual CSs, the agent may not predict the arrival of the US. The imprinting feature is learned to be activated when CS1 is active and CS2 is inactive. Using imprinting features, the agent can represent non-linear configurations of stimuli. In this example, the imprinting feature recognizes when CS 1 is active and CS2 is inactive. The agent can use the imprinting features to predict the US accurately.

$$\frac{|w_t^{i+n}|}{\sum_{j=n+1}^{m+n} |w_t^j|} \geq \frac{1}{m} + \epsilon \tag{5.2}$$

in which $\epsilon \sim \mathcal{N}(0, \frac{1}{m})$ is a small random number, giving observation signals with comparatively small weight a random chance to be selected. When an observation o^j is selected to participate in the imprinting feature s^i, the weight $V^{i,j}$ is set to 1 if the observation is active and -1 if the observation is inactive at the time step of the feature creation. Using the observation signals at the current time to choose the weights ensures that the imprinting feature would be active the next time the same pattern is present. The imprinting generator makes features representing the non-linear configuration of observation signals, and the deep trace generator can enable the agent to remember and represent these imprinting features to make temporally distant associations.

The tester is similar to the deep trace tester. However, the agent state's capacity is divided between deep trace and imprinting features, and the testing process applies to each feature type separately. Otherwise, the deep trace generator would use all of the capacity. Algorithm 2 describes the details of the imprinting generate-and-test method when combined with the deep trace generate-and-test algorithm to learn the agent state. Table 5.1 is the description of new variables and hyper-parameters used in Algorithm 2.

Variable	**Description**
n_p	current number of imprinting features
c_p	maximum number of imprinting features (capacity)
g_p	maximum number of imprinting features to generate
r_p	maximum number of imprinting features to remove
p_p	protection ratio of imprinting features

Table 5.1: Hyper-parameters and variables description for the imprinting generate-and-test algorithm.

Algorithm 2: Imprinting and Deep trace generate-and-test algorithm.

Initialize: Set the state-update function u with no initial features by setting weight matrix $\mathbf{V}$ to zero and set n_d and n_p to 0, and consider the agent state $\mathbf{s}_0 \in \mathbb{R}^n$ as zero

Initialize: Set weight vector $\mathbf{w} \in \mathbb{R}^{n+m}$ and eligibility trace vector $\mathbf{z} \in \mathbb{R}^{n+m}$ as zeros

Initialize: Set hyper-parameters α, θ, λ, c_d, g_d, r_d, p_d, c_p, g_p, r_p, p_p, and μ as desired

for *each observation* $\mathbf{o}_t \in \mathbb{R}^n$ *and* $US_t \in \mathbb{R}$ **do**
- **if** *there is non-zero activity in* $\mathbf{o}_t$ *and* $n_p < c_p$ **then**
  - **Imprinting generator:**
  - Generate $min(g_p, c_p - n_p)$ imprinting features
  - **for** *each generated feature* i **do**
    - Select the observation signals using Equation 5.2
    - Add the feature i if it is a new feature—no duplicates
    - Set n_p to $n_p + 1$
- Compute the current state: $\mathbf{s}_t = u(\mathbf{s}_{t-1}, \mathbf{o}_t)$
- Update the weight vector $\mathbf{w_t}$ using TD(λ) or TIDBD(λ)
- **if** $n_d < c_d$ **then**
  - **Deep trace generator** (details in Chapter 5)
- **if** $n_p = c_p$ **then**
  - **Imprinting tester:**
  - Partition the imprinting features based on the moving average of the weight magnitude and select the bottom $1 - p_p$ portion of the features
  - Set *num_deleted* to 0
  - **for** *each feature* i *from the bottom* $1 - p_p$ *portion of the imprinting features* **do**
    - **if** *feature* i *is not a source for other features* **then**
      - Set the outgoing weight w^i to 0
      - Set the corresponding eligibility trace z^i to 0
      - Set $v^{i,j}$ for all $0 < j \leq m + n$ to zeros
      - Set *num_deleted* to *num_deleted* $+ 1$
      - Set n_p to $n_p - 1$
      - **if** *num_deleted* $= r_p$ **then**
        - Break out of the for loop
- **if** $n_d = c_d$ **then**
  - **Deep trace tester** (details in Chapter 5)

5.2 Experiment Details

We show the effectiveness of the imprinting and deep trace generate-and-test algorithm on the trace patterning problem introduced in Chapter 3. In our setup of the trace patterning problem, there are 6 CSs and 10 distractors. The CSs and the distractors have a duration of 4 time steps and happen at the same time. A particular configuration of 3 active and 3 inactive CS triggers the arrival of US—the *activation pattern*. The CSs are activated in a way to make the activation pattern happen in half the trials. Each distractor occurs independently with a probability of 0.5—simultaneously with the CSs and other distractors. If the activation pattern happens in a trial, the US would happen in 10 time steps and remains active for 2 time steps—the ISI is set to 10. We use MSRE over bins of 1000 time steps to measure the agent's performance. Table 5.2 is the list of hyper-parameters used in our experiments.

Hyper-parameter	**Value**
step-size α	0.01
meta step-size θ	0.01
eligibility trace decay rate λ	0.9
discount factor γ	0.9
weight magnitude moving average decay rate μ	0.99
maximum number of deep traces c_d	200
maximum number of deep trace features to generate g_d	2
maximum number of deep trace features to remove r_d	2
maximum number of imprinting c_c	60
maximum number of imprinting features to generate g_c	2
maximum number of imprinting features to remove r_c	2
deep trace feature protection ratio p_d	0.5
imprinting feature protection ratio p_c	0.5

Table 5.2: Hyper-parameters for the imprinting generate-and-test experiments.

5.3 Results

Figure 5.3 shows the agent's performance based on root MSRE over bins of 1000 time steps. The agent usually learns the imprinting feature representing the activation pattern in the first few thousand trials. We compare the performance of the agent utilizing both imprinting and deep trace features with the agent that uses the total capacity of the network—260 features—making deep trace features. The deep trace features cannot represent configurations in the observation signals and fail to capture the activation pattern. Figure 5.4 shows the agent's predictions for both cases of activation pattern being present and absent after training for 20000 trials for the agent that generates imprinting features. Figure 5.5 shows the predictions made by the agent that only generates deep trace features. The imprinting generator produces imprinting features representing the activation pattern. The deep trace generator uses the imprinting features and produces deep trace features that fill the trace interval and make accurate predictions of the US.

Figure 5.6 shows an example of the activity of several features when the activation pattern is present. Figure 5.7 shows the activity of the same set of features when the activation pattern is absent. Most of the deep trace features trace the imprinting feature that represents the activation pattern or the US. There are deep trace features of other stimuli, but they are in the minority. Figure 5.8 shows the dependency of the deep trace features. A significant portion of deep trace features is learned to directly or indirectly trace the imprinting feature representing the activation pattern. Unlike the distractors, CSs provide useful and relevant information about the arrival of the US. However, using the imprinting feature representing the activation pattern to make deep trace features is more promising. Figure 5.9 shows the outgoing weight of the CSs and the learned imprinting feature representing the activation pat-

tern. As the imprinting feature gains weight, the CSs lose their weight since they are no longer necessary to make predictions about the US.

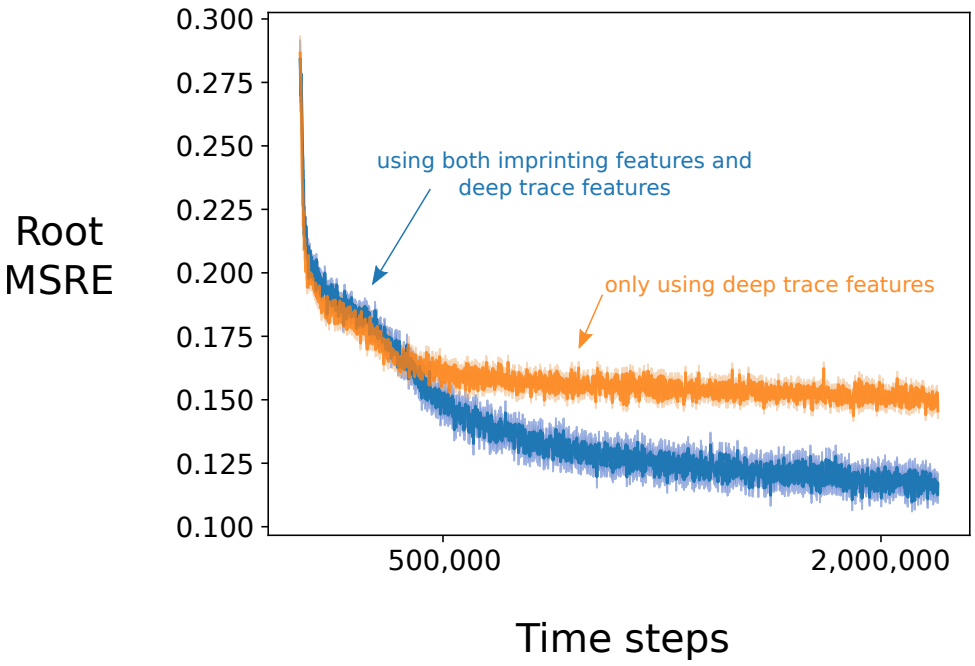

Figure 5.3: The performance of the imprinting and deep trace generate-and-test algorithm on the trace patterning problem throughout 20000 trials compared to only using deep trace generate-and-test algorithm. The performance is measured based on the root MSRE over bins of 1000 time steps and is averaged over 60 runs. The shaded area is the standard error. The agent usually learns the imprinting feature representing the activation pattern in the first few thousand trials—before 500000 time steps. The bump early in the learning curve is where the agent is still searching for the proper imprinting feature. Using only the deep trace features, the agent cannot learn configurations in the observation signals and cannot represent the activation pattern; thus, the agent performance is degraded compared to the agent that also utilizes the imprinting features. The deep trace features enable the agent to remember the imprinting features to predict the US accurately.

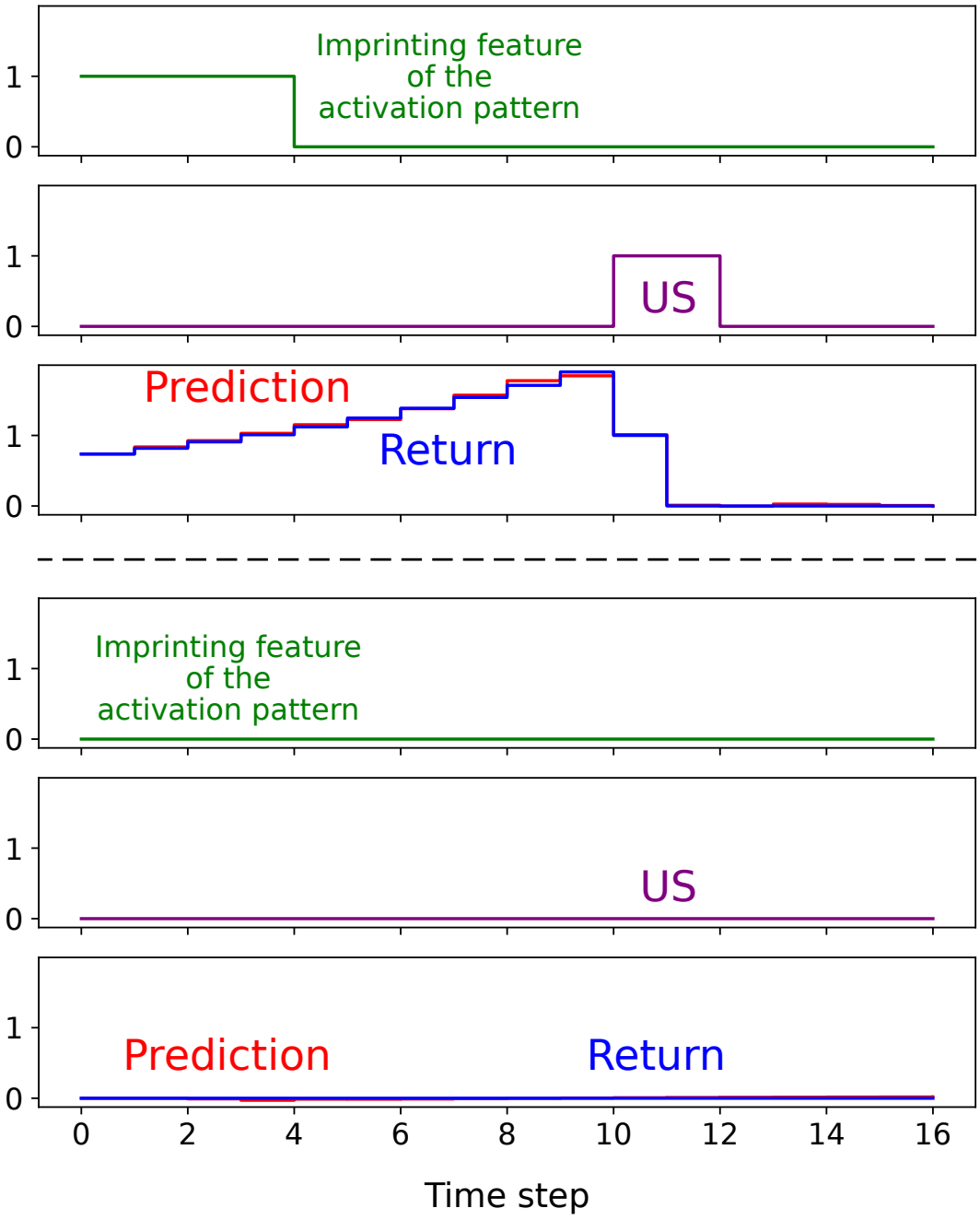

Figure 5.4: The predictions made by the agent match the return in both cases of the activation pattern being present and absent. The predictions shown are made at the final trial of the experiment using the features learned by the imprinting and deep trace generate-and-test algorithm.

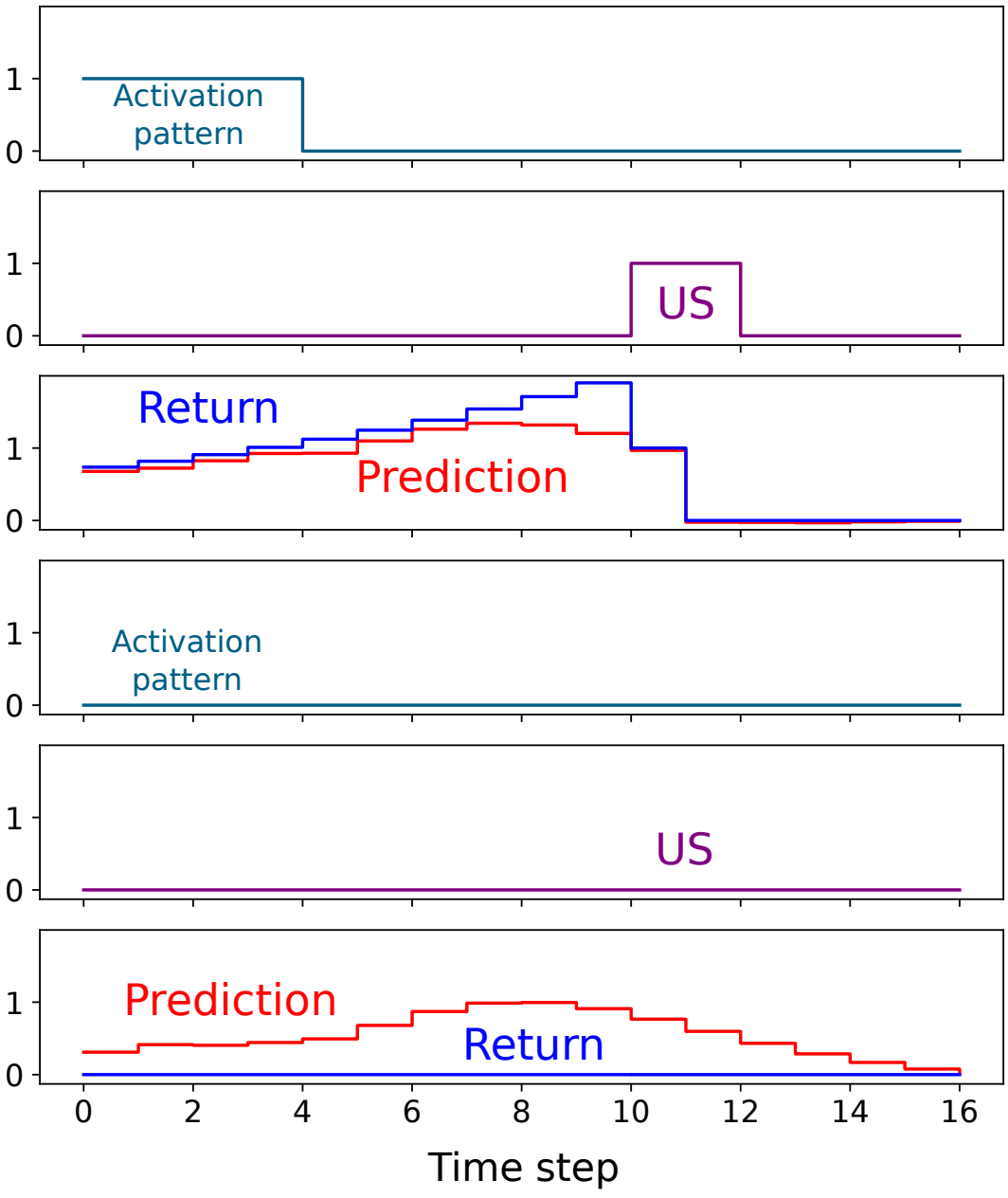

Figure 5.5: Using only the deep trace features, the agent cannot represent configurations in the observation signals and thus cannot represent the activation pattern and make accurate predictions. Note that the activation pattern here only demonstrates whether the activation pattern has occurred or not. There is no feature representing the activation pattern in the agent state.

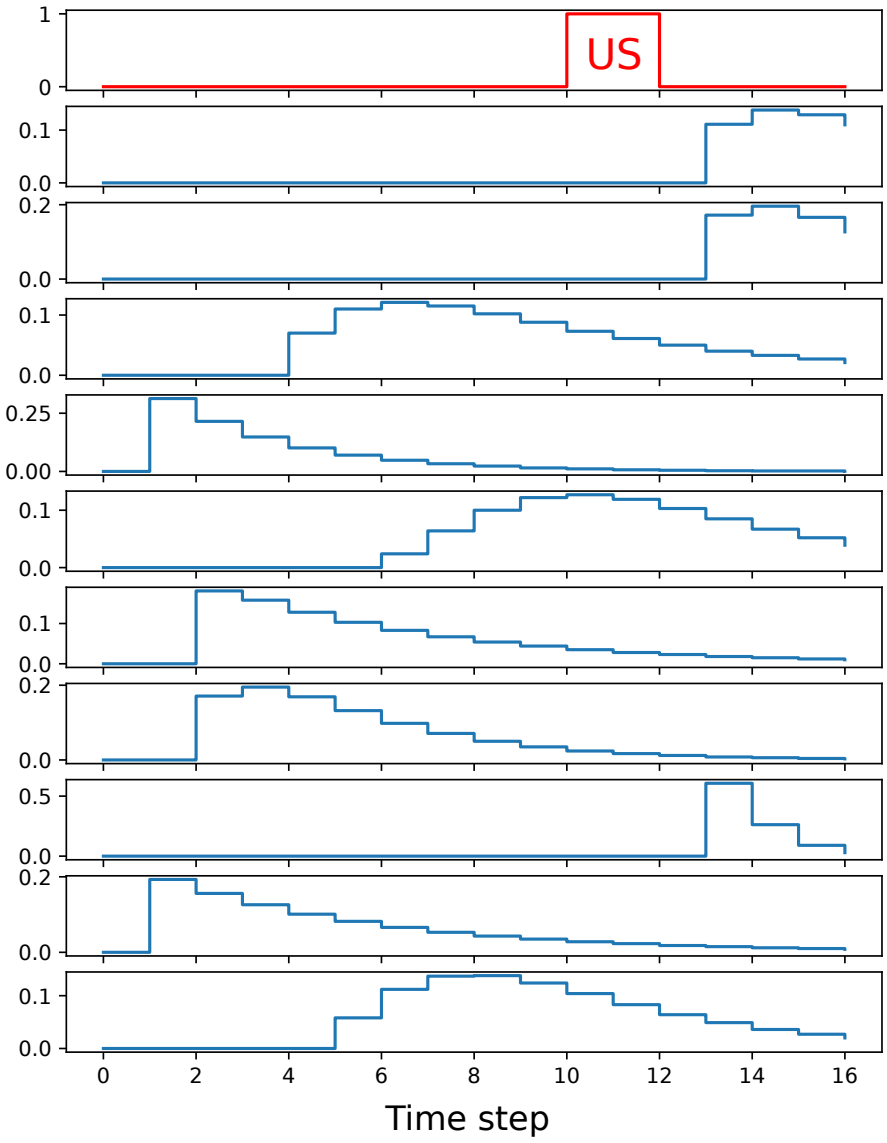

Figure 5.6: The activity of a number of deep traces features throughout a trial with the activation pattern present—after 20000 trials of training. The deep trace features make a rich representation of the trace interval gap enabling the agent to make accurate predictions. Most of the deep trace features directly or indirectly trace the imprinting feature representing the activation pattern.

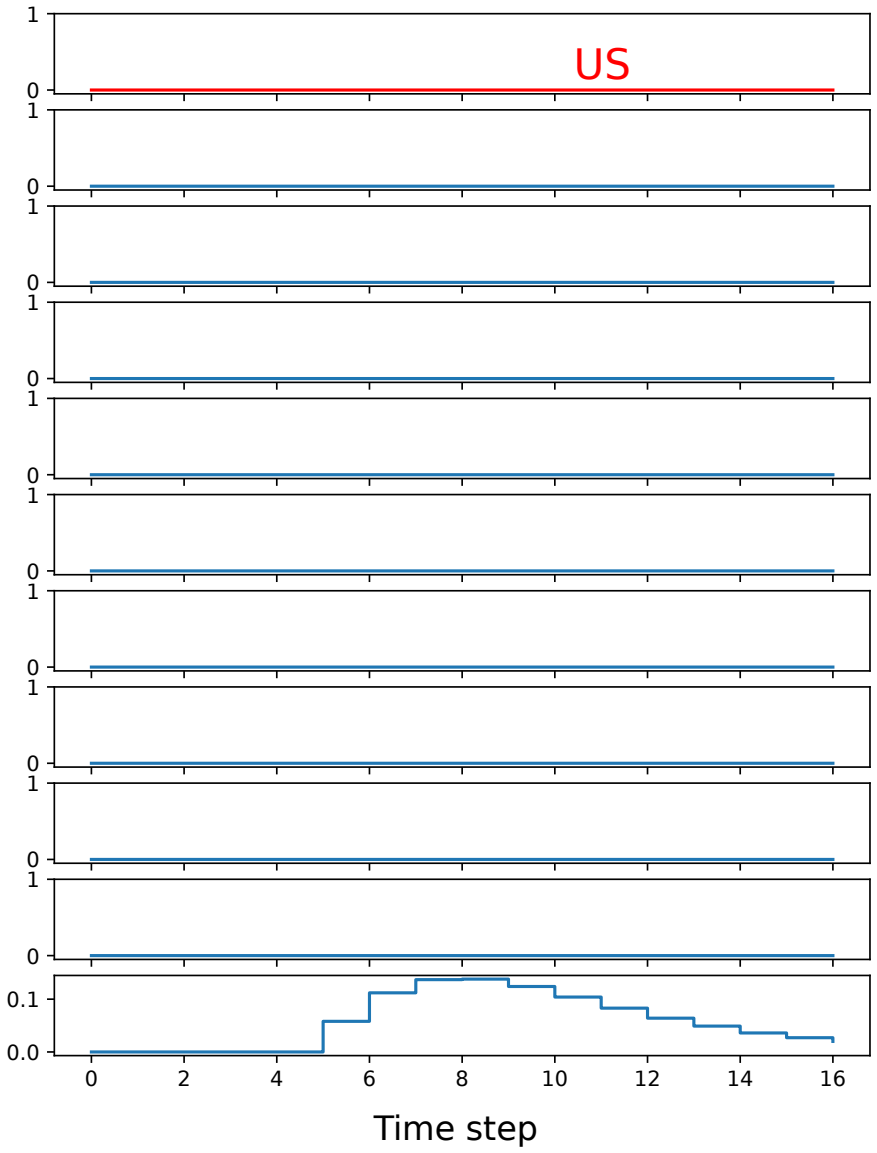

Figure 5.7: The activity of a number of deep traces features throughout a trial with the activation pattern absent—after 20000 trials of training. Since most deep trace features directly or indirectly trace the imprinting feature representing the activation pattern, there is not much activity when the activation pattern is not present. There are deep trace features of the CSs themselves that might be active during the trial even though the activation pattern is absent.

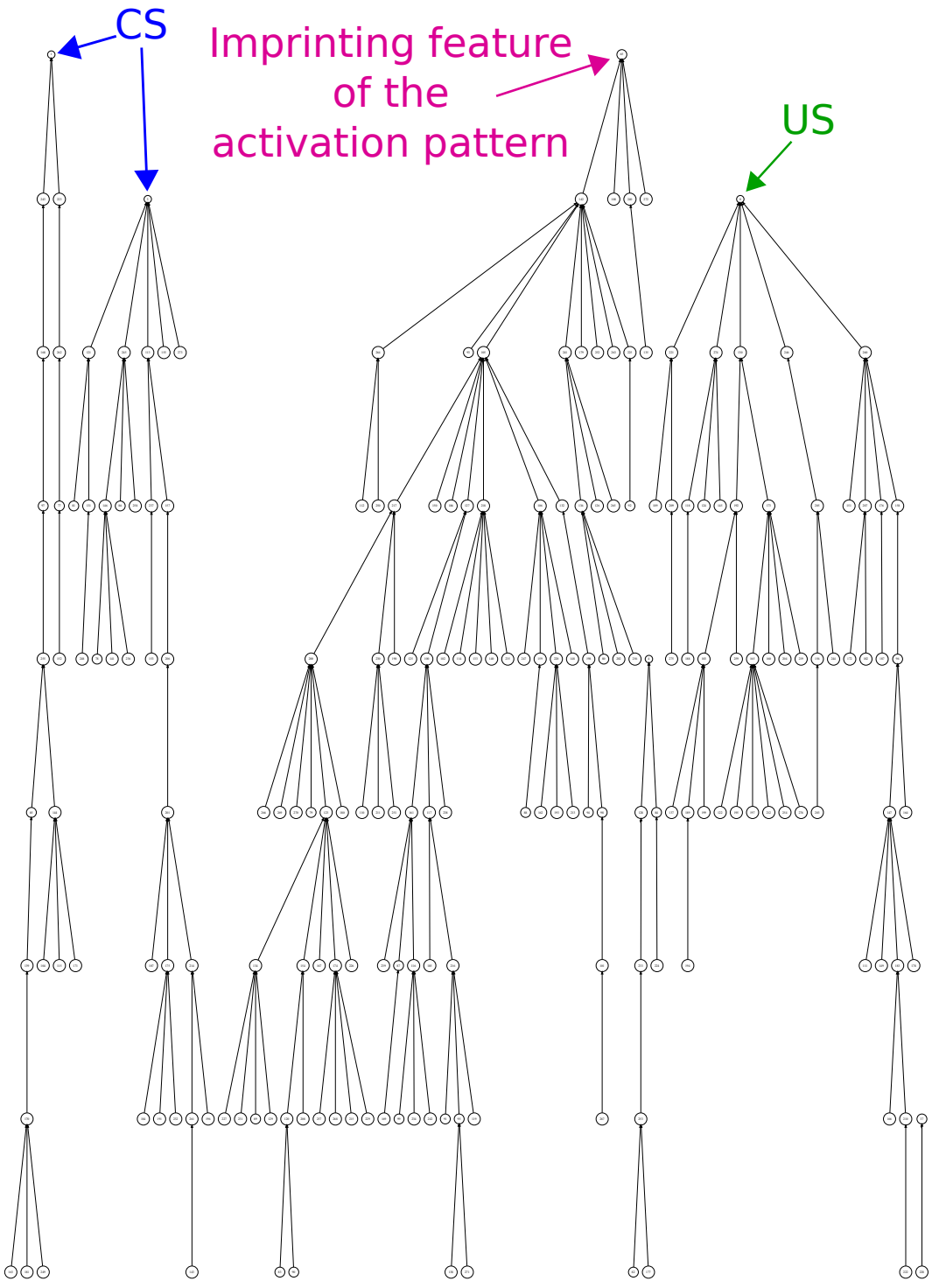

Figure 5.8: Dependency of the deep trace features at the final trial of the experiment. Most of the deep trace features directly or indirectly trace the imprinting feature representing the activation pattern. Some deep trace features are tracing the CSs. The CSs are correlated with the US and are useful, though not as valuable as the imprinting feature of the activation pattern.

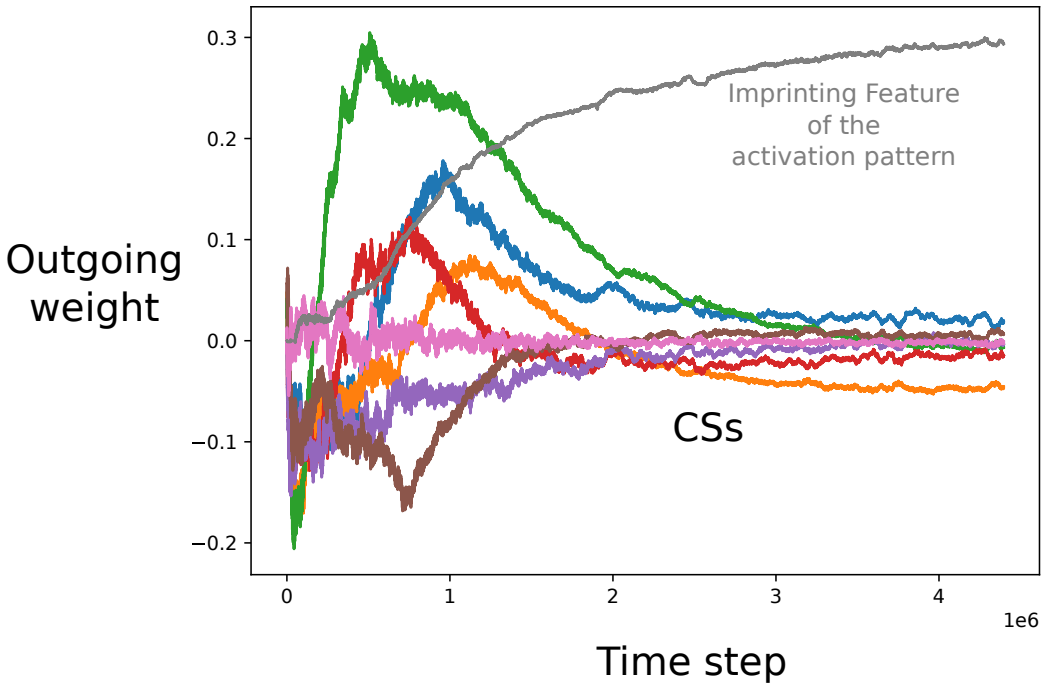

Figure 5.9: At the beginning of the experiment, the agent uses the CSs for predicting the US. However, after the agent learns the imprinting feature representing the activation pattern, the CSs lose their outgoing weight. The imprinting feature is perfectly correlated with the US, but the CSs are not always a good indicator of the arrival of the US.

Chapter 6

Related Work

This chapter discusses existing works related to learning the agent state in this thesis. In Section 6.1, we discuss approaches based on the generate-and-test method. In Section 6.2, we summarize methods based on gradient descent for learning an RNN, and the complications of using the gradient descent. We also include the recent advancement of addressing the practical and computational issues. In Section 6.3, we describe methods that directly use the predictions as the representation.

6.1 Generate-and-Test

Representation search refers to learning the representation by searching through the representation space. Generate-and-test methods can be considered as representation search methods as these methods learn the representation by continually searching for more useful features. Applying search to finding features and representation has been widely investigated in the supervised learning setting (Blum and Langley, 1997; Guyon and Elisseeff, 2003; Vamplew and Ollington, 2005; Whiteson and Stone, 2006). As in the reinforcement learning setting, Kaelbling (1993) proposes a generate-and-test algorithm for learning Boolean functions that represent high performance actions in the environment. Representation search fits directly with continual learning, and the

generate-and-test algorithm offers a fully online representation search method (Mahmood and Sutton, 2013). Mahmood and Sutton (2013) propose searching for representation by generating and testing features. Mahmood and Sutton (2013) show the effectiveness of the proposed generate-and-test algorithm on a synthetic supervised learning task. The generator produces random LTU features, and the tester evaluates the utility of the features and eliminates features with the least utility.

Mahmood and Sutton (2013) propose three testers. The first tester uses the magnitude of the outgoing weight as the measure for utility. The tester only considers features with *age*—time steps since generation—beyond a certain *maturity threshold* for deletion to protect the newly generated features. The second tester keeps a moving average of the outgoing weight magnitude for each feature—similar to our proposed tester. Finally, the third tester uses the learned step-size and the magnitude of outgoing weight to measure the utility of each feature. Learning individual step-size for each feature ensures certainty in the outgoing learned weight.

The generate-and-test can be used both as an alternative or alongside the backpropagation algorithm. Dohare et al. (2021) show that the initial randomness in the weights is critical to the performance of backpropagation. In continual learning settings, the backpropagation performance is severely reduced after the initial randomness in the weights is lost during the training. Dohare et al. (2021) suggest using a generate-and-test algorithm alongside backpropagation to mitigate this issue. Similarly, Rahman (2021) uses a generate-and-test method for fast continual feature discovery.

The Cascade correlation algorithm learns a network by adding features one by one (Fahlman and Lebiere, 1989). In each iteration, a pool of candidate features is generated, and each feature is separately trained to maximize its correlation with the error. The candidate feature with the highest correlation

with the error is added to the network, and the rest of the candidate features are discarded. The Cascade correlation algorithm is similar to the generate-and-test approach. The generator produces candidate features, and the tester keeps the one with the highest correlation with the error and eliminates the rest. Note that after a feature is added to the network, it will no longer get removed. Unlike the generate-and-test algorithm by Mahmood and Sutton (2013), the Cascade correlation network is offline.

6.2 Recurrent Neural Networks

RNNs are heavily used in sequence modelling tasks where the most recent observation is insufficient to perform well on a task. Similar to the agent state architecture, the hidden state of an RNN feeds back to itself, making it possible for the RNN to retain information from the past.

The challenge is how to learn the weights of RNNs. The backpropagation algorithm utilizes gradient descent to learn the weights in a multi-layer feed-forward neural network (Rumelhart et al., 1986). Similarly, by unfolding RNNs across time, BPTT learns the weight of an RNN to minimize the prediction error (Robinson and Fallside, 1987; Werbos, 1988). Figure 6.1 shows how an RNN is unrolled and effectively becomes a multi-layer feed-forward neural network. BPTT may become extremely slow as the computational cost of unrolling scales each time step—at time step t, the unrolling includes all the time steps from 0 to $t - 1$.

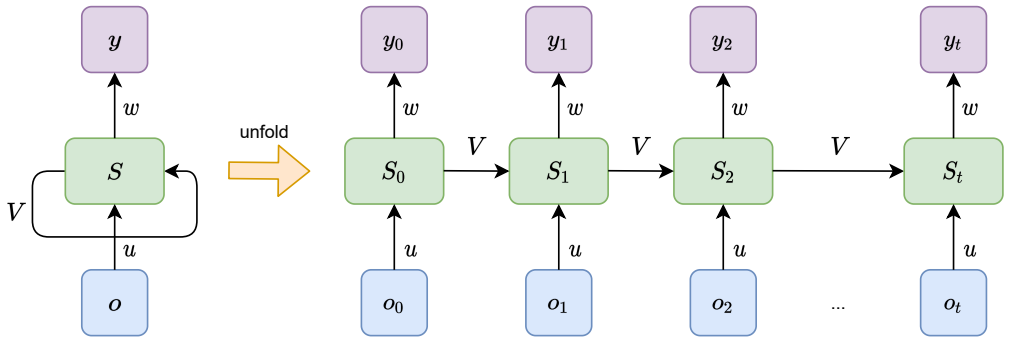

Figure 6.1: BPTT unrolls the RNN to update the weights. The gradients are computed back in time to make temporally distant associations. The unrolling becomes computationally more expensive with time since BPTT unrolls back to the first time step (figure inspired by Olah (2015)).

A well-known approximation to BPTT is truncated BPTT (T-BPTT), in which the unrolling for the gradient calculation only considers the past T time steps $t-1$ to $t-T$ (Williams and Peng, 1990). Since the gradient information for steps beyond T time steps in the past is ignored, temporal dependencies further than T time steps cannot be captured (Williams and Zipser, 1989). T-BPTT is considered one of the online learning algorithms for training RNNs, but its computational complexity is far from ideal—$O(n^2h)$ for n units and truncation of h.

Another approach to learning the weights of RNNs in online fashion is to use forward-mode differentiation methods such as RTRL (Williams and Zipser, 1989). RTRL computes the gradient online by updating the Jacobian matrix of the recurrent cell, which is prohibitively expensive, making it infeasible to use in practice—time complexity of $O(n^4)$ for n units. There have been efforts to approximate and reduce the computational cost of using RTRL. Tallec and Ollivier (2018) use stochastic unbiased estimates of the gradient at the expense of high variance, resulting in poor performance in practice. Mujika et al. (2018) approximate RTRL using Kronecker product decomposition. Menick et al. (2020) propose an approximation to RTRL by only tracking the parameters that have a non-zero influence on the hidden state within a fixed number of

time steps. Javed et al. (2021) propose a method to reduce the computational cost of training RNNs by modularizing the network into columns with scalar states and tracking the influence of parameters within their columns.

Apart from computational issues, other problems arise when training RNNs with gradient descent. When the spectral radius of the recurrent weight matrix is greater than 1, the gradient components may increase exponentially. This problem is referred to as the exploding gradient problem. The vanishing gradient is the opposite, which means that the gradient components decay exponentially to 0 and happens when the spectral radius of the recurrent weight matrix is less than 1. These issues significantly hinder the ability of RNNs to learn long-term dependencies. Pascanu et al. (2013) propose gradient clipping to mitigate the exploding gradient problem.

The solution to the vanishing gradient problem is an active area of study. Alternative recurrent architectures such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are proposed to alleviate the vanishing gradient problem (Hochreiter and Schmidhuber, 1997; Chung et al., 2014). Incorporating skip connections into RNNs can help with the vanishing gradient problem and capturing long-term dependencies (He et al., 2016; Chang et al., 2017). The choice of the activation function also plays a role in the vanishing gradient problem. For instance, the Rectified Linear Unit is less likely to be affected by the vanishing gradient problem (Glorot et al., 2011).

6.3 Predictive Representation of State

Intelligent agents acquire knowledge about the world by interactions and grounding the knowledge based on predictions about possible future experiences. Sutton et al. (2011) propose using a collection of *general value functions* (GVFs) to capture predictive knowledge of the environment. The *predictive representation hypothesis* proposes that representation of the state in terms of predictions

about the future experiences generalizes well (as in Rafols et al. (2005)). Predictive representation of the state use this idea and form the representation in terms of predictions (Littman et al., 2001). There have been several attempts to implement such a representation learning method which we discuss here.

Sutton and Tanner (2005) propose TD Networks to learn the representation based on interrelated predictions. TD networks represent the predictive questions in a network of nodes. Each node represents a single prediction. The nodes are interconnected, and the connections represent relationships between predictions and other actions, observations, and predictions (Tanner and Sutton, 2005a). The agent then uses the predictions learned while interacting with the environment as the agent state. Tanner and Sutton (2005b) extended TD Networks to include a history of fixed-length past observations. Sutton et al. (2006) propose forming the predictive questions based on *options*, which allows the TD Network to condition a predictive question on a series of actions instead of only one action. Schlegel et al. (2021) introduce General Value Function Networks (GVFN) by formulating the hidden state of RNNs as predictions about the future experience and represent the predictions using GVFs. Schlegel et al. (2021) show that GVFNs are less sensitive to the truncation parameter compared to traditional RNNs when trained with T-BPTT.

The central challenge of predictive representation methods is referred to as the *discovery problem* (Singh et al., 2003). The discovery problem is concerned with the choice of the predictions used in the representation—the question network in the TD networks and the GVFs in the GVFN.

Chapter 7

Conclusion

Learning the agent state is essential for a reinforcement learning agent. The agent state summarizes the agent's interaction with the environment in a useful way for predicting and controlling future experiences. In this thesis, we proposed methods for learning the agent state online using the generate-and-test approach. The idea behind the generate-and-test approach is to continuously generate features and offer them to the agent for prediction and control, and then replace the least useful features with newly generated features.

One of the challenges that the agent needs to address is to associate temporally distant events. When the cue for a prediction of interest has happened in the past, considering the current observation signals would not be enough to make accurate predictions. We proposed the deep trace generate-and-test algorithm to learn features that enable the agent to remember past events and make accurate multi-step predictions. We studied the effectiveness of the deep trace generate-and-test algorithm on the trace conditioning problem. In the trace conditioning problem, the agent needs to remember the CS in order to predict the arrival of the US accurately. Our experiments show that the deep trace generate-and-test algorithm learns useful features, enabling the agent to remember the CS for accurate prediction of the US.

In many cases of interest, remembering individual observation signals may

not be sufficient to make accurate predictions. For instance, in the trace patterning problem, a particular configuration of active and inactive CSs triggers the arrival of a temporally distant US. In order to predict the arrival of the US, the agent must make features representing the specific configuration of the observation signal and remember the configuration for future predictions. The imprinting generator produces features that learn non-linear configurations of observation signals. Our experiments show that the agent can learn non-linear configurations of the CSs and accurately predict a temporally distant US by utilizing the imprinting and the deep trace generate-and-test algorithm.

Our generate-and-test algorithms suggest that it is possible to develop methods to learn the agent state online using the agent's data stream of experience. Nevertheless, many aspects and questions are not discussed and remain for future studies. We only showed that the agent state is used for prediction. Reinforcement learning agents also may control the future trajectory. It is interesting to study how to learn the agent state online when the agent also needs to learn a policy and act in the environment.

In addition to the action-selection policy for controlling the future trajectory, the agent state may be the input to multiple value functions or the environmental model. Generally, the agent state may have multiple users, and it is still unclear how to generate features that are useful to multiple users of the agent state. Likewise, testing the features can be even more challenging since a feature that seems useless to a particular user of the agent state may be crucial for another user.

Another exciting aspect of online and continual learning is to deal with non-stationary environments. The world is ever-changing and vast compared to the agent and what it can learn. Different parts of the environment may require drastically different behaviours and predictions, and an online learning agent needs to adapt its predictions and policy when needed. Future studies

are required to understand better how to adapt the agent state when dealing with such non-stationarities.

Finally, in this thesis, we propose a simple tester that measures feature utility based on the magnitude of the outgoing weight. Intuitively, features with larger weight magnitude contribute more to the final predictions than features with smaller weight magnitude. Features can be indirectly useful, and the tester should consider various types of usefulness. To simplify the tester, we refrain from removing the features that are the basis for other features; however, it is limiting as the tester only may remove features with no dependent features. Future studies should investigate how to measure the utility of features more thoroughly.

References

Blum, A. L., and Langley, P. (1997). Selection of relevant features and examples in machine learning. *Artificial intelligence*, *97*(1-2), 245–271.

Chang, S., Zhang, Y., Han, W., Yu, M., Guo, X., Tan, W., Cui, X., Witbrock, M., Hasegawa-Johnson, M., and Huang, T. S. (2017). Dilated recurrent neural networks. *arXiv: 1710.02224*.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv: 1412.3555*.

Dickinson, A. (1980). *Contemporary animal learning theory*. Cambridge University Press.

Dohare, S., Mahmood, A. R., and Sutton, R. S. (2021). Continual backprop: Stochastic gradient descent with persistent randomness. *arXiv: 2108.06325*.

Fahlman, S. E., and Lebiere, C. (1989). The cascade-correlation learning architecture, In *Advances in neural information processing systems*.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks, In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*.

Guyon, I., and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of machine learning research*, *3*(Mar), 1157–1182.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition, In *Proceedings of the IEEE conference on computer vision and pattern recognition*.

Hochreiter, S., and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Javed, K. (2021). Step-size adaptation for TD(λ)–comparing two algorithms. https://khurramjaved.com/reports/stepsize_traces.pdf.

Javed, K., White, M., and Sutton, R. S. (2021). Scalable online recurrent learning using columnar neural networks. *arXiv: 2103.05787*.

Kaelbling, L. P. (1993). *Learning in embedded systems*. MIT press.

Kearney, A., Veeriah, V., Travnik, J. B., Sutton, R. S., and Pilarski, P. M. (2018). TIDBD: Adapting temporal-difference step-sizes through stochastic meta-descent. *arXiv: 1804.03334*.

Littman, M. L., Sutton, R. S., and Singh, S. P. (2001). Predictive representations of state, In *Advances in neural information processing systems*.

Ludvig, E. A., Sutton, R. S., and Kehoe, E. J. (2012). Evaluating the TD model of classical conditioning. *Learning & behavior*, *40*(3), 305–319.

Mahmood, A. R., and Sutton, R. S. (2013). Representation search through generate and test, In *Proceedings of the 12th AAAI conference on learning rich representations from low-level sensors*, AAAI Press.

Menick, J., Elsen, E., Evci, U., Osindero, S., Simonyan, K., and Graves, A. (2020). A practical sparse approximation for real time recurrent learning. *arXiv: 2006.07232*.

Modayil, J., White, A., and Sutton, R. S. (2014). Multi-timescale nexting in a reinforcement learning robot. *Adaptive Behavior*, *22*(2), 146–160.

Mujika, A., Meier, F., and Steger, A. (2018). Approximating real-time recurrent learning with random Kronecker factors. *arXiv:1805.10842*.

Olah, C. (2015). Understanding LSTM networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks, In *Proceedings of the 30th international conference on machine learning*.

Pavlov, I. P., and Anrep, G. V. (1927). *Conditioned reflexes: An investigation of the physiological activity of the cerebral cortex* (Vol. 3). Oxford University Press.

Rafiee, B., Abbas, Z., Ghiassian, S., Kumaraswamy, R., Sutton, R. S., Ludvig, E., and White, A. (2020). From eye-blinks to state construction: Diagnostic benchmarks for online representation learning. *arXiv: 2011.04590*.

Rafols, E. J., Ring, M. B., Sutton, R. S., and Tanner, B. (2005). Using predictive representations to improve generalization in reinforcement learning, In *Proceedings of the international joint conference on artificial intelligence*.

Rahman, P. (2021). *Toward generate-and-test algorithms for continual feature discovery* (Master's thesis). University of Alberta.

Robinson, A., and Fallside, F. (1987). *The utility driven dynamic error propagation network*. University of Cambridge Department of Engineering Cambridge.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning internal representations by error propagation. In *Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1: Foundations*. Cambridge, MA, USA, MIT Press.

Schlegel, M., Jacobsen, A., Abbas, Z., Patterson, A., White, A., and White, M. (2021). General value function networks. *Journal of Artificial Intelligence Research*, *70*, 497–543.

Singh, S. P., Littman, M. L., Jong, N. K., Pardoe, D., and Stone, P. (2003). Learning predictive state representations, In *Proceedings of the 20th international conference on machine learning*.

Sutton, R. S. (1992). Adapting bias by gradient descent: An incremental version of delta-bar-delta, In *Proceedings of the tenth national conference on artificial intelligence*. MIT Press.

Sutton, R. S., and Barto, A. G. (1990). Time-derivative models of Pavlovian reinforcement. In M. Gabriel and J. Moore (Eds.) *Learning and computational neuroscience: foundations of adaptive networks*, 497–537.

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

Sutton, R. S., Koop, A., and Silver, D. (2007). On the role of tracking in stationary environments, In *Proceedings of the 24th international conference on machine learning*.

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 *Proceedings of the 10th international conference on autonomous agents and multiagent systems*.

Sutton, R. S., Rafols, E., and Koop, A. (2006). Temporal abstraction in temporal-difference networks, In *Advances in neural information processing systems*.

Sutton, R. S., and Tanner, B. (2005). Temporal-difference networks, In *Advances in neural information processing systems*.

Sutton, R. S., and Whitehead, S. D. (1993). Online learning with random representations, In *Proceedings of the 10th international conference on machine learning*.

Tallec, C., and Ollivier, Y. (2018). Unbiased online recurrent optimization, In *International conference on learning representations*.

Tanner, B., and Sutton, R. S. (2005a). TD(λ) networks: Temporal-difference networks with eligibility traces, In *Proceedings of the 22nd international conference on machine learning*.

Tanner, B., and Sutton, R. S. (2005b). Temporal-difference networks with history, In *Proceedings of the international joint conference on artificial intelligence*.

Thill, M. (2015). *Temporal difference learning methods with automatic step-size adaption for strategic board games: Connect-4 and dots-and-boxes* (Master's thesis). Cologne University of Applied Sciences.

Vamplew, P., and Ollington, R. (2005). Global versus local constructive function approximation for on-line reinforcement learning, In *Australasian joint conference on artificial intelligence*. Springer.

Wagner, A. R. (1978). Expectancies and the priming of STM. *Cognitive Processes in Animal Behavior*, 177–209.

Werbos, P. J. (1988). Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, *1*(4), 339–356.

Whiteson, S., and Stone, P. (2006). Evolutionary function approximation for reinforcement learning. *Journal of machine learning research*, *7*(31), 877–917.

Williams, R. J., and Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, *2*(4), 490–501.

Williams, R. J., and Zipser, D. (1989). A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, *1*(2), 270–280.