Learning Agent State Online with Recurrent Generate-and-Test

Amir Samani
Department of Computing Science
University of Alberta
samani@ualberta.ca

Richard S. Sutton
Department of Computing Science
University of Alberta
rsutton@ualberta.ca

Abstract

Learning continually and online from a continuous stream of data is challenging, especially for a reinforcement learning agent with sequential data. When the environment only provides *observations* giving partial information about the state of the environment, the agent must learn the *agent state* based on the data stream of experience. We refer to the state learned directly from the data stream of experience as the agent state. Recurrent neural networks can learn the agent state, but the training methods are computationally expensive and sensitive to the hyper-parameters, making them unideal for online learning. This work introduces methods based on the *generate-and-test* approach to learn the agent state. A generate-and-test algorithm searches for state features by generating features and testing their usefulness. In this process, features useful for the agent's performance on the task are preserved, and the least useful features get replaced with newly generated features. We study the effectiveness of our methods on two online multi-step prediction problems. The first problem, *trace conditioning*, focuses on the agent's ability to remember a cue for a prediction multiple steps into the future. In the second problem, *trace patterning*, the agent needs to learn patterns in the observation signals and remember them for future predictions. We show that our proposed methods can effectively learn the agent state online and produce accurate predictions.

1 Introduction

Online continual learning refers to learning from a never-ending data stream without reusing past data points. A reinforcement learning agent interacts with its environment by performing actions and receiving observations. This interaction results in the agent's data stream of experience. In many cases of interest, the agent does not have access to the underlying state of the environment and, when interacting with the environment, only receives observations that provide partial information. One of the challenges that the agent has to overcome is representing its state based on the data stream of experience, which we call the agent state. Similarly, natural intelligent agents only receive limited information about the state of the environment; for instance, objects are not visible in the dark or when distant. Nevertheless, studies in classical conditioning show that animals can make accurate multi-step predictions, suggesting that animals make representations that summarize their interaction with the environment.

In reinforcement learning, learning the state is essential to the agent as the state is used in the agent's policy, value functions, and the environmental model. Historically, domain experts designed the state based on their knowledge of the environment. Relying on human input to learn the state is at odds with a significant strength of reinforcement learning which is the ability to learn directly from the data stream of experience. Therefore, we would like online algorithms to learn the agent state using the data stream of experience.

Preprint. Under review.

Modern deep learning algorithms based on gradient descent, such as real-time recurrent learning (RTRL) and backpropagation through time (BPTT), learn the agent state based on the data stream of experience. However, these algorithms are expensive in memory and computation and sensitive to the selection of their hyper-parameters. We want learning algorithms that are inexpensive in terms of computation and memory, which naturally fit with reinforcement learning algorithms for prediction and control.

A simple approach for learning the agent state is the generate-and-test. Mahmood and Sutton [2013] propose the generate-and-test approach to search for features that improve the performance. This search method for finding features generates candidate features and tests them for their utility on a given task. A *generator* would generate features, and a *tester* preserves the more useful features and deletes the least useful ones. Mahmood and Sutton [2013] apply the generate-and-test to a synthetic supervised learning problem with a feed-forward network, and we would like to extend generate-and-test to reinforcement learning and sequential data and thus with a recurrent network to learn the agent state for a reinforcement learning agent.

Classical conditioning experiments on animals such as dogs and rabbits have shown that an unconditioned stimulus (US) such as food can get associated with a conditioned stimulus (CS) such as a bell after several pairings [Pavlov and Anrep, 1927]. For instance, after multiple pairings, a dog would start salivating after hearing the bell in anticipation of the food. These experiments suggest that animals can make accurate multi-step predictions. Inspired by these experiments on animals, Rafiee et al. [2020] introduce benchmarks for partially observable multi-step online prediction problems. In this paper, we focus on the trace conditioning problem and trace patterning problem. In the trace conditioning problem, the agent must predict the US multiple time steps in the future using a CS, similar to predicting the arrival of food using the sound of the bell. In trace patterning, the agent still needs to predict the US multiple time step in the future. However, there are several CSs, and only a specific configuration of the CSs results in the arrival of the US. For instance, the dog would receive the food if only the bell sound and the light were present.

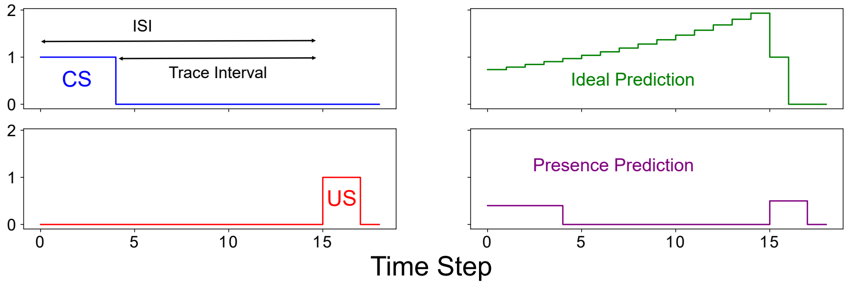

Figure 1: An example of the trace conditioning problem in which the CS is active for 4 time steps (top right) the US arrives 15 time steps after the onset of the CS (bottom right). The time from the onset of the CS to the onset of the US is called the inter-stimulus interval (ISI). The time from the offset of the CS to the onset of the US is called the trace Interval. During the trace interval, there are no relevant observation signals available to the agent. The agent needs to learn the agent state to predict the arrival of the US accurately. Studies have shown that the predictions made by the animals are similar to the discounted return (top right). A simple form of representing the state, the presence representation, uses the observation as the agent state thus can not make accurate predictions, especially when the trace interval gap is long (bottom right).

This paper proposes two online generators and a simple tester for learning the agent state. First, we discuss the architecture for representing the agent state. Second, we propose the deep trace generator that allows the agent to fill the trace interval gap by making features that hold fading memory of features and observation signals. We study the effectiveness of the deep trace generator on the trace conditioning problem. We show that the agent can fill the trace interval gap using the feature generated by the deep trace generator. Finally, we enable the agent to make features representing a non-linear configuration of the stimuli using our second generator, the imprinting generator. The imprinting generator makes features that respond to a particular configuration in the observation signals. We show that combining the deep trace generator and the imprinting generator allows the agent to make accurate predictions in the trace patterning problem.

2 Agent State Architecture

The data stream of experience for a reinforcement learning agent contains the history of all the interactions between the agent and the environment—actions performed by the agent and the observations received from the environment. We denote the action at time step t by $\mathbf{a}_t \in \mathbb{R}^d$ and the observation at time t by $\mathbf{o}_t \in \mathbb{R}^m$. The data stream of experience is the sequence

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

going forever for the life the reinforcement learning agent. Storing the whole sequence and using it as the agent state is computationally impractical since the length of the sequence grows with time. Let us denote the agent state at time step t as $\mathbf{s}_t \in \mathbb{R}^n$. We prefer the agent state to be updated 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 call this update function *state-update* function and denote it by u:

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

We want to learn the agent state by learning the state-update function u. We consider the case in which the agent state consists of features. Since the features are learned, we may talk about the usefulness of each feature. We can compare features based on their usefulness for predicting or controlling the data stream of experience. Our method to learning the state is based on the generate-and-test approach. We search for more useful features by generating features and putting them through testing. As the agent performs on the original task, the generator makes new features. The agent may use these features for prediction or controlling its data stream of experience. The tester preserves features that the agent found to be useful and eliminates the least useful features. With the deletion of the least useful feature, the generator can then make more features. Although similar ideas have been studied before, our approach extends representation search to partially observable sequential multi-step prediction setting for learning the agent state.

Each feature $s_t^i \in \mathbb{R}$ may be connected to x_t^j with the weight of $V_t^{i,j}$ where $\mathbf{x}_t = [\mathbf{s}_{t-1}, \mathbf{o}_t, \mathbf{a}_{t-1}] \in \mathbb{R}^{m+n+d}$ is the concatenation of previous time step state $\mathbf{s}_{t-1}$ and the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$. The feature s_t^i is then computed as follows:

$$s_t^i = \sum_{j=1}^{m+n+d} V_t^{i,j} x_t^j.$$

In the case of a single scalar prediction, the prediction $y_t \in \mathbb{R}$ at time step t is connected to f_t^k with the weight of w_t^k where $\mathbf{f}_t = [s_t, o_t, a_{t-1}] \in \mathbb{R}^{m+n+d}$ is the concatenation current time step state $\mathbf{s}_t$ and the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$. The final prediction y_t is compute as follows:

$$y_t = \sum_{k=1}^{m+n+d} w_t^k f_t^k = \mathbf{f}_t^\intercal \mathbf{w}_t.$$

There can also be an always on bias bit in the agent in the agent state vector. An abstract view of agent state architecture is shown in Figure 2. Note that the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$ are used to compute the current time step state $\mathbf{s}_t$ and are also directly connected to the final prediction y_t.

A generate-and-test algorithm learns the weight matrix V. Generating a new feature s_t^i corresponds to selecting which input signals to connect to in $\mathbf{x}_t$ and its connection weight in the weight matrix V. When we remove a feature, we delete its connections to the $\mathbf{x}_t$. In the case of learning an online multi-step prediction task, we use semi-gradient TD(λ) [Sutton, 1988]. Semi-gradient TD(λ) is computationally efficient and can learn predictions online [Modayil et al., 2014]. We use semi-gradient TD(λ) to learn the weight vector w to predict the future value of a cumulant c [Sutton et al., 2011]—in the case of classical conditioning problem we can consider the US as the cumulant [Sutton and Barto, 1990]. Since the agent state is learned separately by a generate-and-test algorithm, we can use linear semi-gradient TD(λ) to update the weight vector $\mathbf{w}$

$$\mathbf{z}_t = \gamma\lambda \mathbf{z}_{t-1} + \mathbf{s}_t \tag{1}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha(c_{t+1} + \gamma \mathbf{f}_{t+1}^\intercal \mathbf{w}_t - \mathbf{f}_t^\intercal \mathbf{w}_t)\mathbf{z}_t \tag{2}$$

where $\alpha \in (0, 1]$ is the step size and $\gamma \in [0, 1)$ is the discount factor that determines the horizon of the prediction of the cumulant. $\mathbf{z}_t \in \mathbb{R}^{m+n+d}$ is the eligibility trace, and $\lambda \in [0, 1]$ is the decay of the eligibility trace.

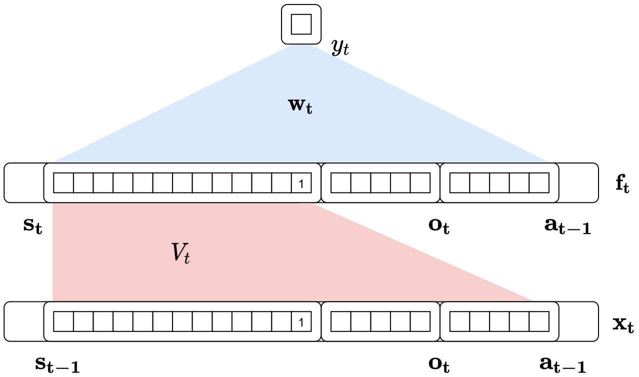

Figure 2: The agent state $\mathbf{s}_t$ at time step t is computed using the previous time step state $\mathbf{s}_{t-1}$ and the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$ using weight matrix V. The current agent state can then be used by the agent for prediction or control. For example, for the case of a single scalar prediction, the current state $\mathbf{s}_t$ and the most recent observation $\mathbf{o}_t$ and action $\mathbf{a}_{t-1}$ are mapped to the final prediction y_t using weight vector $\mathbf{w}_t$. The weight matrix V is learned by a generate-and-test algorithm, and the weight vector $\mathbf{w}$ is learned by semi-gradient TD(λ).

3 Deep Trace Generate-and-Test for Trace Conditioning

The first problem we focus on is how to generate features that enable the agent to associate temporally distant events. Using the trace conditioning problem, we can focus on this problem in isolation. The trace conditioning problem considers the problem of uncontrolled multi-step prediction in which two stimuli that have no prior association are presented to the animal in a particular order. First, a CS is presented to the animal, followed by an US. The animal responds to the US by generating an unconditioned response (UR). After several trials, the animal generates a conditioned response (CR) when presented with the CS. The UR happens before the arrival of the US, which suggests that the animal associated the CS with the US. For example, a dog would naturally salivate (UR) in the presence of food (US). Suppose the dog receives a previously unassociated stimulus such as a tone (CS) before the arrival of the food. After enough pairings, the dog will start salivating (CR) after hearing the tone and before the arrival of the food. There can be a gap between the offset of the CS and the onset of the US. This gap is called the trace interval. There is no immediate relevant observation available during the trace interval; thus, the agent should make features that fill the gap to predict the US accurately.

The TD model of classical conditioning can make predictions similar to predictions observed by animal experiments [Sutton and Barto, 1990]. The TD model of classical conditioning uses TD(λ) to learn predictions [Sutton, 1988]; however, to make accurate predictions, the state should have features representing the trace interval [Ludvig et al., 2012]. To make features representing the trace interval, we introduce our first generator, the *deep trace generator*. The deep trace generator makes features that enable the agent to remember helpful information from the past to predict the future. We call these features *deep trace features*. Deep trace features are *traces* from either observation signals or other features—including other deep traces.

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

$$s_t^i = \psi s_{t-1}^i + (1 - \psi) x_t^j \tag{3}$$

in which ψ is the *decay rate* and x^j the *source* of the deep trace feature s^i.

For example, let s^a to be a deep trace feature of the observation signal o^j with a decay rate of 0.9. At time step 2, o_j^2 becomes 1 for one time step. Assuming $s_1^a = 0$ (since $o_1^j = 0$), we can calculate s_1^a using Equation 3

$$s_1^a = 0.9 s_0^a + (1 - 0.9) o_2^j = 0.9(0) + 0.1(1) = 0.1.$$

At time step 2, the observation o_2^j is no longer present ($o_j^2 = 0$); however, since $s_1^i = 0.1$, s_2^i becomes 0.09. Even if o^j remains inactive for more time steps, the deep trace feature s^a will have non-zero activity for many more time steps. The trace decay rate ψ controls how rapidly the deep trace fades away. The self-connection weight ψ and source connection weight $(1 - \psi)$ ensure the value of the deep trace remains between 0 and 1—assuming the source of the deep trace is binary or is between 0 and 1.

A deep trace feature can be the source of another deep trace feature . Let us extend our example by adding two more deep trace features. Deep trace features s^b and s^c trace deep trace feature s^a and s^b, respectively—both of these new deep trace features have a decay rate of 0.9. Consequently, deep trace features s^b and s^c are indirect traces of the observation o^j. Figure 3 shows the value of deep trace features s^a, s^b, and s^c over time. Since deep trace a is directly connected to the observation o^j, it quickly jumps up and fades away faster than deep trace features s^b and s^c. Deep trace features s^b and s^c are slower in growth but last longer. Multiple traces of an observation signal (both indirect and direct) let the agent have a longer-lasting memory of that observation signal. In the trace conditioning problem, deep trace features enables the agent to fill the trace interval gap and remember the CS.

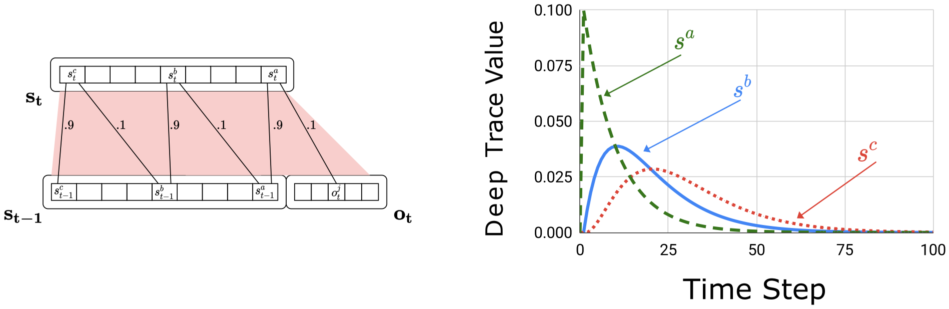

Figure 3: The abstraction of deep trace features s^a, s^b, and s^c that trace o^j, s^a, and s^b, respectively (right figure). When the observation signal o_2^j gets activated at time step 2 for one time step, deep trace feature s^a quickly jumps up and starts to fade away. Deep trace feature s^b and s^c indirectly trace the observation signal o^j and are slower to respond. The figure on the right shows the level of deep trace feature s^a, s^b, and s^c over time. These deep trace features provide a rich memory of the observation signal o^j that can allow the agent to fill the trace interval gap and make accurate predictions.

The deep trace generator makes a new deep trace feature by choosing the source and the decay rate. The generator chooses the decay rate randomly (between 0 and 1). The generator also selects the source randomly from the $\mathbf{x}_t$. However, the probability of each x_t^i to be selected is proportional to its outgoing weight magnitude. In other words, x_t^i with a larger outgoing weight magnitude w_t^i is more likely to be selected as the source for a new deep trace feature.

The maximum number of features is fixed as the computation and memory cost should remain constant. To generate new features, the tester should remove the least useful features. At each time step, the tester partitions the features by half based on the exponential moving average of their weight magnitude. A certain number of features in the bottom half of the partitioning are subjected to deletion to make space for the generator to make more features. However, if a feature is the source of another deep trace, the tester refrains from removing it. This protection happens since a feature that ranks low in the partitioning can be a source for a useful feature.The tester protects a newly generated feature by initializing its moving average of the weight magnitude as the median of all moving averages of the weight magnitude.

To show the effectiveness of the deep trace generator, we experiment on the trace conditioning problem [Rafiee et al., 2020]. Trace conditioning problem consists of several trials. Each trial starts with the CS lasting for 4 time steps followed by the US lasting for 2 time steps. The time from the onset of the CS and the onset of the US is called inter-stimulus interval (ISI). The time onset of the US and beginning of the next trial is the inter-trial interval (ITI) which we uniformly sampled from (80,120). In addition to the CS and the US, other uninformative stimuli happen randomly during each trial. These uninformative stimuli are called *distractors* and provide no information about the CS or the US. In our experiments, there are 10 distractors that occur with a Poisson distribution and the rate of $\frac{1}{10}, \frac{1}{20}, ..., \frac{1}{100}$, respectively and lasted for 4 time steps. The discount factor γ is set to $1 - \frac{1}{ISI}$. To evaluate the performance of the agent we use the Mean Squared Return Error (MSRE) over bins of 1000 time steps. The Squared Return Error (SRE) at time t is calculated by $(y_t - G_t)^2$ in which $G_t = \sum_{k=0}^{\infty} \gamma^k c_{t+k+1}$ is the return.

We evaluate the effectiveness of the deep trace generator for enabling the agent to remember distant stimuli by running our experiments for ISI of 10, 20, and 30. Each experiment consists of 20000 trials for each ISI value, which is more than 20 million time steps. We use semi-gradient TD(λ) with $\lambda = 0.9$ and we use step-size adaptation with initial step-size of $\alpha = 0.01$ and meta step-size $\theta = 0.01$ [Sutton, 1992, Thill, 2015]. The maximum number of features is set to 100, 200, and 300 for ISI 10, 20, and 30, respectively. The maximum number of features to add (g_d) and remove (r_d) at each time step is set to 2. Figure 4 shows the MSRE over bins of 1000 time steps. We average MSRE over 30 runs, and the shaded area is the standard error. Figure 5 shows the learned predictions by the agent and compares them to ideal predictions based on the return. The agent is able to make accurate predictions using the features generated by the deep trace generator.

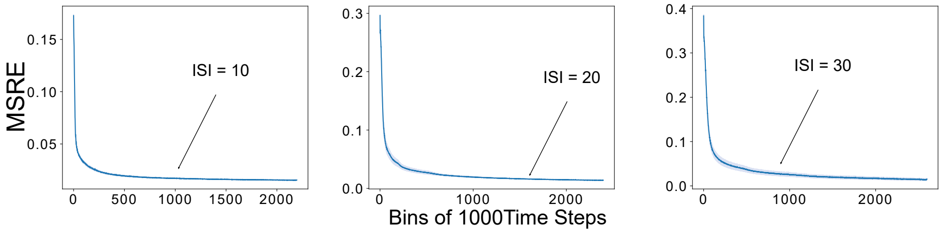

Figure 4: The MSRE over bins of 1000 time steps for the deep trace generator on the trace conditioning problem with varying ISI 10, 20 , and 30 averaged the MSRE over 30 runs. The shaded area is the standard error. The agent can make accurate predictions using the features learned by the deep trace generator.

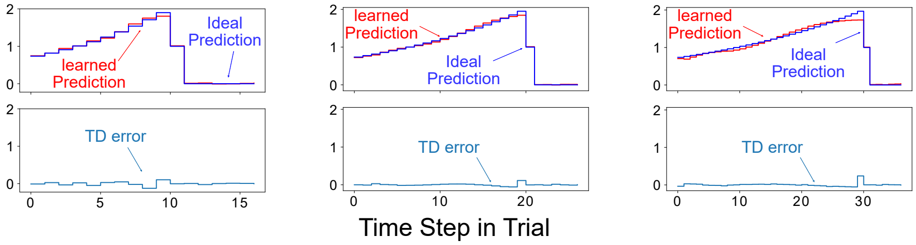

Figure 5: Prediction made using the features generated by the deep trace generator after more than 2 million time steps (red line) compared to the ideal prediction (blue line) which is the return. The bottom plots show the TD error. The agent's prediction closely matches the ideal prediction, which shows the agent could effectively fill the trace interval gap.

4 The Imprinting Generator for Trace Patterning

In the trace conditioning problem, the agent only needs to remember a single CS to predict the US. Although there are distractors, they provide no information, and the agent can ignore them and focus only on the CS and the US. In trace patterning, there are multiple CSs, and only a specific configuration of active and inactive CSs triggers the arrival of the US. For instance, the dog would receive food only if the tone is present and the light is absent. Without considering the non-linear configurations of the CSs, each CS would not be a useful predictor of the US. The deep trace generator could only generate traces of individual features or observation signals. We need to generate features that respond to configurations in the observation signals.

We propose the *imprinting generator* for generating features that respond to a particular configuration in the observation signals. An *imprinting feature* s^i is connected to observation signal o^j with a weight of +1 if o^j should be active and -1 if o^j should be inactive in the configuration. Note that not all observation signals need to be connected to s^i. The imprinting feature s^i is a non-linear map of the observations that are connected and is computed using Linear Threshold Unit (LTU) [Sutton and Whitehead, 1993]. The imprinting feature s^i is computed as follows:

$$s_t^i = \begin{cases} 1 & \sum_{j=1}^{m} V_t^{i,j} o_t^j \geq \sum_{j=1}^{m} V_t^{i,j} \\ 0 & \text{otherwise} \end{cases}$$

Figure 6 shows an example in which the imprinting feature s_t^i is connected to the observation signal o_t^1 and o_t^2 with a weight of +1 and -1, respectively. The imprinting generator needs to decide which observation signals to connect to. A simple solution would be selecting the observation signals randomly, but the space of possible imprinting features is 3^m (connecting with +1 or -1 or no connection), which can slow down learning significantly. Since the observation vector $\mathbf{o}_t$ has direct weights to the prediction y_t, we use those weights to make the observation signals with a larger weight magnitude more likely to participate in imprinting features. Observation o_t^i participates in the creation of a imprinting feature at time step t if

$$\frac{|w_t^{i+n}|}{\sum_{j=n+1}^{m+n} |w_t^j|} \geq \frac{1}{m} + \epsilon \qquad (4)$$

where $\epsilon \sim \mathcal{N}(0, \frac{1}{m})$ is small random noise to give observation signals with small weight a chance to be selected. The connection weight is +1 if the observation o_t^i is active and -1 if the observation is inactive at time t. There is also the question of when we generate imprinting features. The imprinting generator monitors the observation signals. Suppose there is a non-zero activity in the observation signals. In that case, the imprinting generator makes g_c new imprinting features (if there is capacity in the network) and add these features if they are new—not to add duplicate features to the network. The imprinting generator works alongside the deep trace generator. Together, these generators would make non-linear combinations of observations and remembers them to make temporally distant associations.

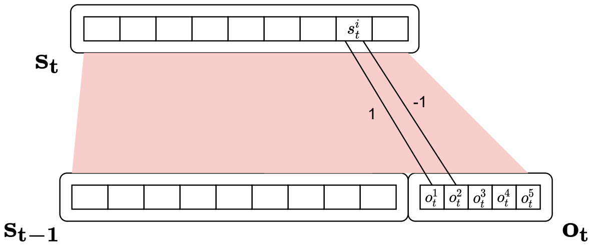

Figure 6: Imprinting feature s^i is connected to the observation signal o^1 with a weight of +1 and the observation signal o^2 with a weight of -2. The imprinting feature s_t^i would become activated (1) if the $o_t^1 = 1$ and $o_t^2 = 0$. The imprinting feature s_i represents a non-linear configuration of its connected observation signals.

The tester is the same as our tester for the trace conditioning problem. However, the maximum number of features in the network is divided between the two generators. Otherwise, the deep trace generator would end up using almost all of the capacity. The tester also applies to the deep traces and imprinting features separately. Thus, each feature would be compared only to the features of the same type. Algorithm 1 demonstrate the details of our proposed generators and tester.

We study the effectiveness of the imprinting generator on the trace patterning problem. In our setup, there are 6 CSs and 10 distractors. All CSs and distractors have a duration of 4 time step if they become active. A specific configuration of 3 active and 3 inactive CSs would result in the arrival of the US, which is referred to as the activation pattern. In our experiments, the activation pattern occurs in half of the trials. The distractors are also presented to the agent simultaneously with the CSs. Each distractor occurs independently with a probability of 0.5. If the activation pattern is present during a trial, the US arrives after 10 time steps (ISI=10) and remains active for 2 time steps. The discount factor γ is set to 0.9. We evaluate the performance of the agent using MSRE over bins of 1000 time steps. All the hyper-parameters are the same as the trace patterning experiments, except the maximum

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

```
Initialize: Set the state-update function $u$ with no initial features and consider the agent state
  $\mathbf{s}_{-1} \in \mathbb{R}^n$ as zeros
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 $\alpha$, $\theta$, $\lambda$, $c_d$, $g_d$, $r_d$, $\mu$, $p_d$, $c_i$, $g_i$, $r_i$, $\mu$, and $p_i$

for each observation $\mathbf{o}_t$ and $\mathbf{US}_t \in \mathbb{R}$ do
    if there is non-zero activity in $\mathbf{o}_t$ and $c_i$ is not reached then
        Generate $g_i$ number of imprinting features
        for each generated feature $i$ do
            Select the observations using Equation ??
            Add the feature $i$ if it is a new feature—not to add duplicate features.
        end
    end
    Compute the current state: $\mathbf{s}_t = u(\mathbf{s}_{t-1}, \mathbf{o}_t)$
    Compute the prediction: $y_t = \mathbf{w}_t^T \mathbf{s}_t$
    $\delta_t = US_t + \gamma y_t - y_{t-1}$
    $\mathbf{z}_t = \gamma\lambda\mathbf{z}_{t-1} + \nabla_{\mathbf{w}} v_t$
    $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t$
    if $c_d$ is not reached then
        Generate $g_d$ deep trace features
        for each generated feature $i$ do
            Set the decay rate $\psi$ randomly
            Choose the source $j$ randomly
            Set $V^{i,j}$ to $1 - \psi$
            Set $V^{i,i}$ to $\psi$
        end
    end
    if $c_d$ is reached then
        Remove $r_d$ features from the bottom $1 - p_d$ portion of the deep trace features—based on
          the weight magnitude—that are not a source for other features
        for each removed feature $i$ do
            Set the outgoing weight $w^i$ to 0
            Set the corresponding eligibility trace $z^i$ to 0
        end
    end
    if $c_i$ is reached then
        Remove $r_i$ features from the bottom $1 - p_i$ portion of the imprinting features—based on
          the weight magnitude—that are not a source for other features
        for each removed feature $i$ do
            Set the outgoing weight $w^i$ to 0
            Set the corresponding eligibility trace $z^i$ to 0
        end
    end
end
```

number of deep traces is set to 200, and the maximum number of imprinting features is set to 60. The maximum number of imprinting features to add (g_c) and remove (r_d) at each time step is set to 2.

The focus of this experiment is to show the effectiveness of the imprinting generator in finding the correct configuration. There are 3^{16} possible configurations (6 CSs and 10 distractors). We run the experiment for 20000 trials which is about 20 million time steps. We report the results for this experiment in Figure 7. Figure 7 compares the agents learned prediction and ideal prediction based on the return. The agent can accurately predict the arrival of the US in the case that the activation pattern occurs. The agent also makes accurate predictions when the US would not be presented.

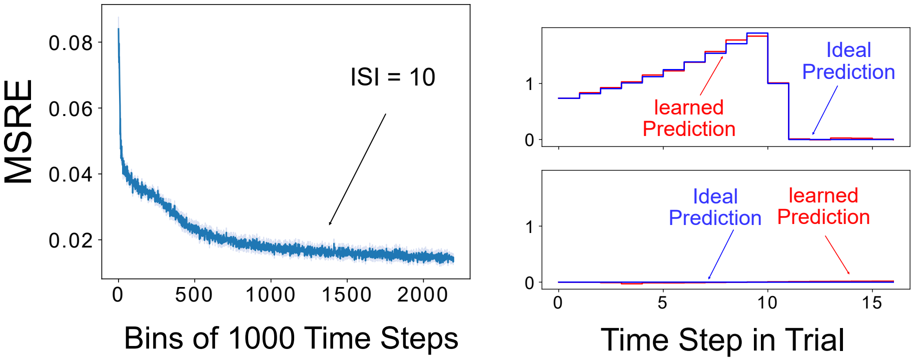

Figure 7: The MSRE over bins of 1000 time steps for trace patterning problem averaged over 30 runs and the shaded area is the standard error (left figure). The prediction learned using the features generated by the imprinting generator and the deep trace generator after more than 2 million time steps (red line) compared to the ideal prediction (blue line) based the return. The agent can accurately predict the arrival of the US (top right) and its absence (bottom right), suggesting that the agent represent the useful non-linear configuration of observation signals and fill the trace interval gap.

5 Related Studies

Employing search for finding representation has been studied in the supervised learning setting. Sutton and Whitehead [1993] introduce the random representation for online learning, and Mahmood and Sutton [2013] introduce generate-and-test by searching for random features that improve the performance of the base system. Later on, Dohare et al. [2021] show that training neural networks using stochastic gradient descent is not ideal for the continual learning setting and suggest that when the initial randomness in the weights is lost, the performance drastically degrades. This issue is mitigated by performing generate-and-test alongside Backpropagation. The Cascade correlation learning architecture introduced by Fahlman and Lebiere [1989] constructively add layers to construct a neural network. Studies mentioned above focus on feed-forward networks, and they are not directly applicable to learning the agent state for a reinforcement learning agent.

Learning the state for a reinforcement learning agent using the data stream of experience is a challenging problem. Recurrent neural networks are often used to learn the agent state. The difficulty is how to train these recurrent neural networks. Backpropagation through time (BPTT) [Elman, 1990] is a solution based on stochastic gradient descent that updates the weight of recurrent neural network to minimize the error. BPTT requires storing all previous network activations to unroll and update the weight, making it prohibitively expensive for online learning. Truncated BPTT (TBPTT) [Williams and Peng, 1990] only store the past t number of activations which makes the complexity of training constant, but it is still expensive. The truncation parameter t is where we decide not to consider further dependencies. Longer truncation enables the network to consider temporal association further in the past at the expense of memory and computation. RTRL [Williams and Zipser, 1989] is another alternative to BPTT that allows for online training, but the cubic computation makes it intractable for larger networks. Approximating the gradient for RTRL makes it more computationally feasible; though, these approximations introduce parameters that their choice influences how far back in time we can make associations [Menick et al., 2020, Tallec and Ollivier, 2018]. Gated RNNs [Chung et al., 2014, Hochreiter and Schmidhuber, 1997] modify the architecture of the RNNs to mitigate problems such as vanishing gradients that make long temporal associations difficult [Hochreiter et al., 2001]; however, These architectures still need training algorithms such as TBPTT or RTRL, which are not suited for online learning.

Predictive representation methods can learn the state by answering predictive questions. TD networks learn the agent state by learning a network of predictions using TD methods [Sutton and Tanner, 2005, Tanner and Sutton, 2005]. OTD network [Rafols et al., 2006] is an extension of the TD network that includes temporal abstracted predictive questions also known as options [Sutton et al., 1999]. More recently, General Value Function Networks [Schlegel et al., 2021] restrict the hidden state of a recurrent neural network to be predictive questions. The challenging part of the predictive representation methods is discovering what predictive questions the agent should consider.

6 Discussion

Learning the agent state online is an essential step towards more general reinforcement learning agents. This work demonstrates how to learn the agent state by searching for features that improve the agent's performance on online partially observable multi-step prediction tasks. We focused on two benchmark problems introduced by Rafiee et al. [2020]. First, using the trace conditioning problem, we focus on the agent's ability to predict an upcoming stimulus based on a temporally distant cue. For this problem, we introduce the deep trace generator. We show the effectiveness of the deep trace generator and tester on three instances of trace conditioning problems by varying the ISI. Second, the trace patterning problem is used as an extension of the trace conditioning problem to the case where a non-linear configuration of stimuli results in the arrival of the US. For this problem, we propose the imprinting generator. The imprinting generator makes features that only get activated when a particular configuration in the observation signals occurs. We show that the deep trace generator and imprinting generator can learn useful non-linear configurations of observation signals and remember them for accurate predictions of the US.

The agent state is used in the agent's policy, value functions, and model of the environment. We only studied learning the agent state when there is only a single user for it. Our simple tester used the weight magnitude for the prediction as an indicator of the usefulness of a feature. When there are multiple users for the agent state, it is unclear which features to preserve and which ones to delete. Generate-and-test is a promising approach towards learning the agent state, and future research should investigate how it scales to other more complicated problems.

References

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Shibhansh Dohare, A. Rupam Mahmood, and Richard S. Sutton. Continual backprop: Stochastic gradient descent with persistent randomness, 2021.

Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

Scott E Fahlman and Christian Lebiere. The cascade-correlation learning architecture. In *Advances in neural information processing systems*, 1989.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997.

Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. *A Field Guide to Dynamical Recurrent Neural Networks*, 2001.

Elliot A. Ludvig, Richard S. Sutton, and E James Kehoe. Evaluating the td model of classical conditioning. *Learning & behavior*, 40(3):305–319, 2012.

Ashique Rupam Mahmood and Richard S. Sutton. Representation search through generate and test. In *Proceedings of the 12th AAAI Conference on Learning Rich Representations from Low-Level Sensors*, AAAIWS'13-12, page 16–21. AAAI Press, 2013.

Jacob Menick, Erich Elsen, Utku Evci, Simon Osindero, Karen Simonyan, and Alex Graves. A practical sparse approximation for real time recurrent learning. *CoRR*, abs/2006.07232, 2020.

Joseph Modayil, Adam White, and Richard S. Sutton. Multi-timescale nexting in a reinforcement learning robot. *Adaptive Behavior*, 22(2):146–160, 2014.

Ivan P. Pavlov and Gleb Vasåilâevåich Anrep. *Conditioned reflexes: an investigation of the physiological activity of the cerebral cortex*, volume 3. London: Oxford University Press, 1927.

Banafsheh Rafiee, Zaheer Abbas, Sina Ghiassian, Raksha Kumaraswamy, Richard S. Sutton, Elliot Ludvig, and Adam White. From eye-blinks to state construction:diagnostic benchmarks for online representation learning. *CoRR*, abs/2011.04590, 2020.

Eddie Rafols, Anna Koop, and Richard S. Sutton. Temporal abstraction in temporal-difference networks. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems*, volume 18. MIT Press, 2006.

Matthew Schlegel, Andrew Jacobsen, Zaheer Abbas, Andrew Patterson, Adam White, and Martha White. General value function networks. *Journal of Artificial Intelligence Research*, 70:497–543, 2021.

Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3 (1):9–44, 1988.

Richard S. Sutton. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 171–176. MIT Press, 1992.

Richard S. Sutton and Andrew G. Barto. Time-derivative models of pavlovian reinforcement. In M. Gabriel and J. Moore (Eds.). *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, pages 497–537, 1990.

Richard S. Sutton and Brian Tanner. Temporal-difference networks. In *Advances in neural information processing systems*, pages 1377–1384, 2005.

Richard S. Sutton and Steven D. Whitehead. Online learning with random representations. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 314–321, 1993.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

Richard S. Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M. Pilarski, Adam White, and Doina Precup. Horde: a scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan, May 2-6, 2011, Volume 1-3*, pages 761–768. IFAAMAS, 2011.

Corentin Tallec and Yann Ollivier. Unbiased online recurrent optimization. In *International Conference on Learning Representations*, 2018.

Brian Tanner and Richard S. Sutton. Temporal-difference (λ) networks: temporal-difference networks with eligibility traces. In *Proceedings of the 22nd international conference on Machine learning*, pages 888–895, 2005.

Markus Thill. 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, 2015.

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

Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.