Addressing Partial Observability in Streaming Deep Reinforcement Learning

Arber Shala, Hrithick Chakraborty, Rana Thind, Romanshu Bedi, Jastegh Singh, Viraj Murab, Jassidak Singh, Swastik Sharma, Lubomyr Soroka, Bhrugu Nilesh Rajput

Department of Computing Science, University of Alberta
{arber, hrithick, ranasunj, romanshu, jastegh, murab, jassidak, swastik, lubomyr,bhruguni}@ualberta.ca

Abstract

In real world environments, reinforcement learning (RL) agents operate in partially observable environments. This means that agents are only given partial observations, where limited information is available about the environment at each time step. Recent advances in streaming reinforcement learning in the form of the stream-x algorithms have allowed for deep reinforcement learning (DRL) agents to learn from a continuous stream of experience. The increased adaptability that stream-x DRL agents have grants greater capabilities in learning using partial observations. However, traditional training algorithms like Backpropagation Through Time (BPTT) are not well suited for real-world problems because they require storing and revisiting entire sequences which makes them impractical for stream learning in resource constrained systems. In this paper, we explore using Real Time Recurrent Learning (RTRL) as a more memory-efficient alternative to BPTT for training agents in partially observable environments.

1 Introduction

Real-world applications of reinforcement learning (RL) operate in partially observable conditions, where an agent receives only fragmentary or indirect information about the environmental state (1). Traditional RL algorithms that commonly assume full observability can struggle to extract meaningful features in these scenarios, leading to suboptimal learning or failure to learn altogether. Moreover, many RL implementations rely on batch-based methods that store extensive past experiences in replay buffers. Although effective in certain settings, this reliance on stored past states becomes infeasible for scenarios demanding online or streaming updates where data arrives continuously. In streaming or online settings, maintaining a large buffer is often impossible. So, we pursue two goals: a performance objective, where we maximize expected reward in the Partially Observable Markov Decision Process (POMDP), which are decision process with hidden underlying states, and a learning objective where the model is updated on-the-fly without storing past transitions.

Streaming learning is an approach to RL that updates the agent's internal state after every step instead of collecting large batches of past experiences in a replay buffer (2). Dropping the buffer cuts the memory use and reduces delay, thus helping learning happen directly on small, low-power devices even when the data keeps changing overtime. The initial efforts proved to be volatile, affectionately referred to as the stream barrier, until Elsayed et al. presented the stream-x family (Stream-Q/AC/TD) which uses per-step normalization alongside adaptive eligibility traces. Because no old samples are stored, the agent has to build an internal memory on the fly, making recurrent networks and fully-online training rules crucial (2). Real-Time Recurrent Learning (RTRL) computes exact gradients step-by-step avoiding the heavy "unroll" of Backpropagation Through Time (BPTT). When paired with element-wise Long Sort Term

Preprint. Under review.

Memory (eLSTM) or the light Recurrent Trace Units (RTUs), it keeps memory small while still tracking long-range dependencies in partially observable settings (1). Recurrent Neural Networks (RNNs) let DRL agents build internal state representations from their interactions, filling the information gap inherent in partially observable environments (1). This paper explores a gated RNN architecture: eLSTMs, and a non-gated RNN architecture: RTUs to see which approach works better in learning streams of data. A sample-efficient back-propagation method is therefore essential, as it reduces training memory load and allows learning on portable hardware (1).

Currently, BPTT remains the standard for training RL agents. BPTT requires unrolling of the RNN back to previous states, the memory requirements for this unrolling prohibits BPTT from being used to train RNNs in most hardware ubiquitous in common devices. Methods such as Truncated Backpropagation through time (T-BPTT) have been used to address the memory inefficiency of BPTT (3). However, no single truncation window fits all tasks, and choosing an optimal length is time and resource-intensive. RTRL computes gradients online at each step, avoiding the need to store past activations (4). RTRL is thus more memory-efficient than BPTT and, unlike T-BPTT, handles fully untruncated sequences. Although both BPTT and RTRL can train RNNs in partially observable environments, RTRL with its reduced memory requirements can be used as a more generalizable method to train streaming DRL agents on less powerful hardware than traditional BPTT methods.

Our paper asks: Can a streaming deep RL agent that uses exact RTRL with a lightweight recurrent core match the performance of its BPTT counterpart while operating within a small, constant memory budget? To answer this question we (1) pair Stream-Q(λ) with eLSTM back-ends, (2) train using exact RTRL and benchmark against a BPTT baseline, (3) evaluate on the T-maze POMDP tasks (lengths 10, 100, 1000), reporting episodic return and peak memory across ten random seeds to achieve statistical significance.

In Section 2 we review BPTT and RTRL in detail, including trace recursion. Related work in streaming RL and recurrent learning is covered in Section 3. The setup for our experiment is explained in Section 4. Results and discussion are provided in Section 5. The last section presents conclusions, shortcomings, and directions for further research.

2 Background

We will consider the Markov Decision Process (MDP) and the POMDP frameworks for modeling the sequential decision process with continual interactions. Formally, a MDP is defined by a 6-tuple: $(S, A, P, R, \gamma, d_0)$ where S is a set of states, A is a set of actions, $P : S \times A \to \Delta(S \times \mathbb{R})$ is a set of conditional transition probabilities between states , $R : S \times A \to \mathbb{R}$ is the reward function. $\gamma \in (0, 1]$ is the discount factor, and d_0 is the distribution over initial states (2). The POMDP is the same as the MDP except that it has two extra elements: Ω the set of observations, $O : S \times A \to \Delta(\Omega)$ is a set of conditional observation probabilities. With POMDPs, when transitioning to the next state, the agent additionally receives an observation o_{t+1} with probability $\Omega(o_{t+1} \mid S_{t+1})$. The agent only receives a partial or limited view of the state of each S_t. The agent's objective in the control setting is to find the policy π that maximizes the expected sum of discounted rewards $\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1}\right]$ (1).

2.1 Back Propagation Through Time (BPTT)

BPTT trains RNNs by unfolding the network over T steps and applying the chain rule in reverse (4). With pre-activations $\mathbf{s}(t) \in \mathbb{R}^N$, inputs $\mathbf{x}(t) \in \mathbb{R}^D$ and total loss $L_{1:T} = \sum_{t=1}^{T} L(t)$. An example BPTT gradient of a loss function is as follows:(4)

$$\frac{\partial L_{total}(1,T)}{\partial W_{i,j}} = \sum_{t=1}^{T} \frac{\partial L(t)}{\partial W_{i,j}} = \sum_{t=1}^{T} (\sum_{k=1}^{N} \frac{\partial L_{total}(1,T)}{\partial s_k(t)} * \frac{\partial s_k(t)}{\partial W_{i,j}})$$

Where L_{total} is the total loss, W is the weights, T is the length of the input sequence, N is the number of trainable parameters, and s_k are intermediate variables of an internal state vector. Using the sensitivity matrix s allows RTRL to update gradients using only forward recursion, facilitating full online learning.

2.2 Real-Time Recurrent Learning (RTRL)

RTRL is a gradient-based online learning algorithm that updates the weights of a network immediately after consuming each new input (4). The key to RTRL making immediate updates is the computation and storage of a sensitivity matrix. The sensitivity matrix is a Jacobian matrix that represents the partial gradients of the network output with respect to the weights and biases of the networks. The high computational and storage costs of computing this sensitivity matrix made only approximations of RTRL feasible. This approximation of RTRL is unattractive compared to BPTT even with the obvious memory advantages of making immediate updates without unrolling the network as it introduces bias to the network. An example RTRL gradient of a loss function is as follows:(4)

$$\frac{\partial L_{total}(1,T)}{\partial W_{i,j}} = \sum_{t=1}^{T} \frac{\partial L(t)}{\partial W_{i,j}} = \sum_{t=1}^{T} (\sum_{k=1}^{N} \frac{\partial L(t)}{\partial s_k(t)} * \frac{\partial s_k(t)}{\partial W_{i,j}})$$

Where L is the component loss, all other parameters are the same as the BPTT equation in section 2.1. A key difference between RTRL and BPTT is that it takes the gradient of the component loss with respect to elements of the sensitivity matrix rather than directly differentiating on the full loss function (L_{total}) (4).

2.3 Streaming learning vs Batch learning

Most traditional RL algorithms use batch learning, where agents store past experiences in a replay buffer and train on them over time.(2) While this helps with stability, it takes a lot of memory. In contrast, streaming learning avoids this by updating the agent step-by-step with each new experience and then discarding it. This makes it much more efficient and better for low-resource settings. Methods like stream-Q(λ) and RTRL help make streaming updates stable, even without a buffer.

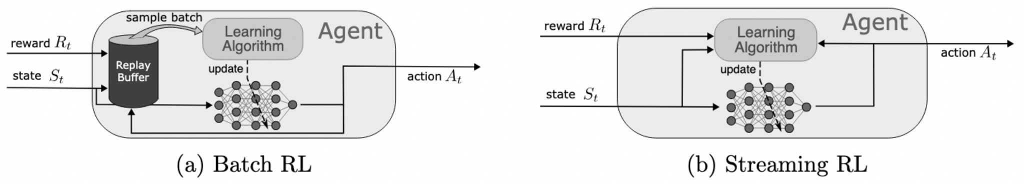

Figure 1: Image from Elsayed et al. (2)

3 Related Works

Elsayed's et al.'s paper is similar to ours in that it explores streaming DRL (2). They introduce streaming versions of several popular RL algorithms including stream Actor Critic(λ), stream Q(λ) and stream TD(λ) They compare these algorithms to existing streaming and non-streaming algorithms. For example, they compare stream AC (λ) to Proximal Policy Optimization (PPO) and PPO1 (the streaming version of PPO). Our paper differs in that it considers streaming learning with partially observable environments. We will implement stream Q(λ) with RTU and eLSTM using RTRL to compute the gradients. More information can be found in methods.

Our paper relates to Elelimy et al.'s paper because both of our papers discuss partially observable environments and also the use of RNNs with RTRL for training the network in the partially observable context (1). Both of our papers also consider the RTU architecture of RNNs. The RTU is an extension of the Linear Recurrent Unit (LRU). The LRU was introduced by Zucchet et al. and was created with the purpose of " allowing exact, tractable online gradient calculation" (5). One difference between our papers was in our implementations of RTUs. In our implementation of RTUs, we do not use imaginary numbers, whereas they do. Also, in their experiments, they used PPO as the control algorithm, whereas we decided to use stream Q(λ). PPO is a batch RL method and is therefore not considered a streaming learning algorithm. However, their results showed that PPO trained with RTUs using RTRL outperformed methods using LRUs and Gated Reccurent Units (GRUs) and

T-BPTT. We will compare RTUs with eLSTMs, to see how RTUs perform against more modern gated architectures than GRUs.

Irie et al.'s paper also discusses using RTRL with RNNs. They use a modified version of actor-critic which is called Real-Time Recurrent Actor Critic (R2AC) as their main reinforcement learning algorithm (4). They compare the results between using their algorithm and using eLSTM and RTRL with another actor-critic algorithm that uses LSTM and T-BPTT. This paper differs from ours in that Actor-critic is a batch RL method, which is not compatible with streaming RL.

4 Methods

In this section, we clearly define the models and algorithms used to address partial observability in Streaming Deep Reinforcement Learning (DRL).

4.1 Recurrent Neural Network (RNN) Architectures

4.1.1 LSTM with Element-Wise Recurrence (eLSTM)

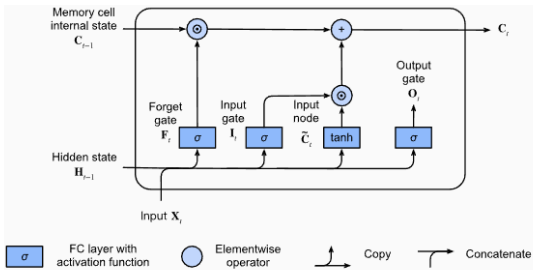

Figure 2: Diagram of LSTM gated memory unit structure. Image from chapter 10 of Zhang et al. (6)

LSTMs are an RNN architecture specialized in remembering sequences of data after a time delay. The LSTM architecture takes advantage of unique gate units which together form a memory unit that controls the flow of information through the RNN (7). In vanilla LSTM implementations there are three gates, an input gate, a forget gate, and an output gate that protect important information from previous states from being perturbed by irrelevant information in the sequence of data. In particular, the input gate decides which information from the previous state is important, the forget gate controls which information from the input should be discarded, and the output gate decides which information should be fed as input to the next state (8).

A standard LSTM cell is defined as follows:(6)

$$\begin{aligned} I_t &= \sigma(X_t W_{xi} + H_{t-1} W_{hi} + b_i) \qquad & F_t &= \sigma(X_t W_{xf} + h_{t-1} W_{hf} + b_f) \\ O_t &= \sigma(X_t W_{xo} + H_{t-1} W_{ho} + b_o) \qquad & H_t &= O_t \odot \tanh(C_t) \\ C_t &= F_t \odot C_{t-1} + I_t \odot \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c) \end{aligned}$$

where I_t, F_t, and O_t are input, forget, and output gates respectively, c_t is the memory cell state, and h_t is the hidden state. The gradient of h_t is whats propagated through the RNN. The variables b_i, b_f, b_o are the bias terms of the input, forget, and output gates respectively while σ is the sigmoid activation function used (6).

We use a modified version of LSTM networks optimized for capturing long-range temporal dependencies.(4). LSTM with element-wise recurrence (eLSTM) modifies the conventional LSTM architecture by enhancing the gate structures to facilitate improved gradient propagation. This improvement lies in using element-wise recurrence when calculating gradients rather than doing recurrence on full components. This element-wise recurrent structure allows for tractable non-approximated RTRL due to it using partial gradients on the component loss function rather than the full loss which is why we use eLSTM as our gated RNN architecture in our experiments.(4).

4.1.2 Recurrent Trace Units (RTUs)

Recurrent Trace Units (RTUs) are another class of recurrent neural architectures which are an extension of Linear Recurrent Units (LRUs). LRUs were introduced as an alternative to nonlinear RNNs, it was found that LRUs are more performant in terms of test-accuracy and are more computationally efficient than nonlinear RNNs (9). The increase in computational efficiency comes from the complex diagonal recurrent matrices, which allow for a highly parallelizable unrolling of the recurrence using parallel scans to substantially improve training speeds (10). LRUs are primarily represented as:

$$h_{t+1} = \lambda \odot h_t + Bx_{t+1}, \qquad y_t = \mathrm{Re}\left[Ch_t\right] + Dx_t,$$

where $\odot$ denotes the element-wise product. Here, $x_t \in \mathbb{R}^H$ represents the input received by the LRU at time t, $h_t \in \mathbb{C}^N$ denotes its internal state, and $y_t \in \mathbb{R}^H$ its output. The parameters of the unit include $\lambda \in \mathbb{C}^N$, $B \in \mathbb{C}^{N\times H}$, $C \in \mathbb{C}^{H\times N}$, and $D \in \mathbb{R}^{H\times H}$ (11).

RTUs, being an extension of LRUs, offer the same benefits as LRUs such as being lightweight and computationally more efficient. RTUs are implemented as a real valued representation of LRUs by representing the complex weight matrix in the real form:

$$\Lambda = \begin{bmatrix}\mathbf{c}_1 & \cdots & \mathbf{c}_n\end{bmatrix}, \quad \mathbf{c}_k = \mathbf{r}_k \begin{bmatrix}\cos(\theta_k) & -\sin(\theta_k)\\ \sin(\theta_k) & \cos(\theta_k)\end{bmatrix},$$

where $\mathbf{r}_k \in \mathbb{R}^2$ and $\theta_k \in \mathbb{R}$. The forward pass $\mathbf{h}_t = \Lambda \mathbf{h}_{t-1} + W_x \mathbf{x}_t \in \mathbb{R}^{2n}$ has two components, $\mathbf{h}_{t,c_1}$ and $\mathbf{h}_{t,c_2}$, which are parametrized and updated recursively:

$$\mathbf{h}_{t,c_1} = \mathbf{g}(\nu_{\log}, \theta_{\log}) \odot \mathbf{h}_{t-1,c_1} - \varphi(\nu_{\log}, \theta_{\log}) \odot \mathbf{h}_{t-1,c_2} + \gamma \odot W_{x,c_1}\mathbf{x}_t,$$
$$\mathbf{h}_{t,c_2} = \mathbf{g}(\nu_{\log}, \theta_{\log}) \odot \mathbf{h}_{t-1,c_2} + \varphi(\nu_{\log}, \theta_{\log}) \odot \mathbf{h}_{t-1,c_1} + \gamma \odot W_{x,c_2}\mathbf{x}_t.$$

Where $\nu = \exp(\nu_{\log}), \quad \mathbf{r} := \exp(-\nu), \quad \theta_{\log} := \log(\theta), \quad \gamma_k := \big(1 - \exp\big(-\exp(\nu_{k,\log})\big)^2\big)^{\frac{1}{2}}$ and

$$g(\nu_k, \theta_k) := \exp\big(-\exp(\nu_{k,\log})\big)\cos\big(\exp(\theta_{k,\log})\big),$$
$$\phi(\nu_k, \theta_k) := \exp\big(-\exp(\nu_{k,\log})\big)\sin\big(\exp(\theta_{k,\log})\big).$$

Thus giving us the final update: $\mathbf{h}_t = \big[\mathbf{h}_t^{c_1}; \mathbf{h}_t^{c_2}\big]$.

These are known as linear RTUs, as the non-linearity occurs after the recurrence. This nonlinearity and slightly modified parameterization is the primary reason behind RTUs outperforming LRUs GRUs (12). We only implemented Linear RTUs in our paper and compared it to eLSTMs.

4.2 Stream Q (λ)

Stream Q(λ) is a fully online adaptation of Q-learning tailored for streaming, partially observable environments.(2) Unlike traditional off-line Q-learning, which relies on replay buffers to store and sample past experiences, stream Q(λ) processes each transition exactly once before discarding it. This approach maintains a constant memory footprint, making it suitable for resource-constrained or embedded systems. We use stream-Q(λ) paired with an eLSTM RNN architecture on our reinforcement learning experiments.

4.2.1 Training via RTRL and BPTT

We explore two gradient propagation methods:

Real-Time Recurrent Learning (RTRL): Computes exact online gradients through the RNN at each time step, avoiding sequence unrolling and supporting unbounded histories. **Backpropagation Through Time (BPTT):** Unrolls the recurrent network over a truncated window and performs batch updates; used here as a baseline to evaluate the trade-off between memory usage and gradient fidelity.

4.3 Experiments

We need a good RNN architecture that can effectively learn when given long sequences of continuous information. Furthermore, we need a memory-efficient gradient update method so the agent can train on lightweight hardware like phones or Household Vacuum Bot. As such, our experiments are centered around two parts: **(1)** a classic time-series prediction problem on the Copy Task, and **(2)** a reinforcement learning problem in the T-maze environment.

Experiment (1) gauges the relative performance of the two RNN architectures (eLSTM and RTU) to remember sequences of data after long delay when trained using RTRL. This experiment uses the Copy Task (4.3.1), which is meant to simulate how the RNN structure of a DRL agent should remember past information when exposed streams of data in a partially observable environment. We also modify the hidden size between runs. Increasing the hidden size should increase the capacity of the RNN to remember more information so we want to see how changes in the hidden size influence the performance of eLSTM and RTU. We only use RTRL in these experiments since we specifically want to see how a gated verusus non-gated RNN architecture performs when trained using RTRL.

Experiment (2) compares the relative performance of BPTT and RTRL as gradient update methods used to train an RL agent to learn in a partially observable environment. Here, we use the best performing RNN architecture from experiment (1) in both the BPTT and RTRL implementations. Given RTRL's memory footprint, we test whether its performance approaches or matches that an agent trained with BPTT. The T-maze environment, explained in section 4.5.2, is a simple partially observable environment where we will run these experiments in. We use Stream Q(λ) as our learning algorithm.

4.3.1 The Copy Task

The Copy Task is a prediction task used as a benchmark to gauge how well an RNN architecture can retain information when exposed to long, continuous, sequences of data (13).
The task definition is as follows: given a long sequence of bits (0 or 1), the RNN model should remember the first n bits of the sequence defined by a sequence length after being exposed to a long sequence of irrelevant data known as blank space. When the model encounters a delimiter (a number other than 0 or 1, such as 2), the model is expected to remember the n bits that it encountered at the beginning (13).

4.3.2 The T-maze Environment

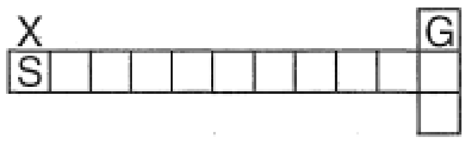

Figure 3: T-maze environment. Image from Bakker. (7)

We will consider the episodic T-maze environment for use in our task which was used in the paper by Allen (14). This environment is made up of a corridor with a junction state at the end, from which there are transitions to two different terminating states. One state gives a positive reward and the other a negative one. The agent starts on the left-hand end of the corridor where they make an initial observation that will tell them which action will lead to the positive award at the junction state. The environment can be defined by a POMDP as follows: The state space consists of the position of the agent. The transitions are deterministic; the agent moves exactly where they want to go. At the beginning of the episode we define: the length of the corridor, the observation given on the start state, and the rewards for the transitions from the junction state (14).

5 Results

5.1 Results of experiments on the Copy Task

Figure 4 presents the results of eLSTM and RTUs learning to remember a sequence of 50 bits. We do 3 runs for each hidden size for both eLSTM and RTU. We see that for both hidden sizes eLSTMs

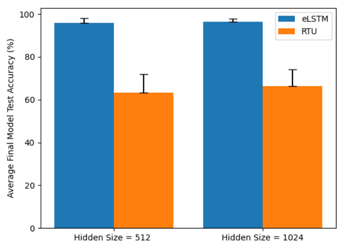

Figure 4: Performance of eLSTM vs RTU measured by average final model test accuracy in remembering a sequence length of 50 when trained using different hidden sizes.

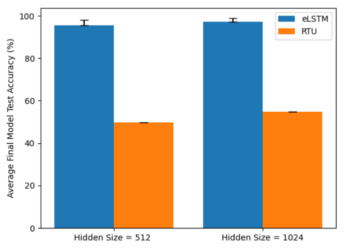

Figure 5: Performance of eLSTM vs RTU measured by average final model test accuracy in remembering a sequence length of 500 when trained using different hidden sizes. RTUs have one run for each hidden size due to time constraints.

outperform RTUs by a significant margin. The eLSTM architecture has an average final model test accuracy around 95% while RTUs perform at around the 60% range measured by the same metric. We see that changing hidden size from 512 to 1024 has little effect in improving the performance of either eLSTM or RTU in remembering a sequence length of 50. Lastly, we see that the RTU has a lot more variance in performance denoted by the larger error bars calculated using the standard deviation. **Figure 5** presents the results of eLSTM and RTUs learning to remember a sequence of 500 bits. Due to the long time it takes to train, we have only one run of the RTUs learning a sequence of 500 bits for hiddens sizes 512 and 1024 so the results in this section are not as strong as those in figure (a). We do 3 runs of eLSTMs for each hidden size. From our RTU run, we see that it performed worse for both hidden sizes when compared to learning a sequence of 50 bits. This degradation of performance is not present in the eLSTM.

5.2 Results of experiments on the T-maze

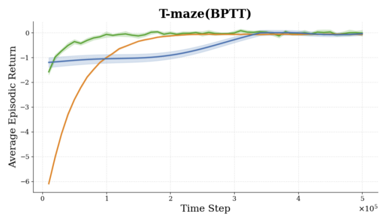

Figure 6: Graph of Stream Q(λ) trained using BPTT with average episodic return at each time step for corridor lengths 10 (green), 100 (orange), 1000 (blue).The results are averaged over 10 independent runs. The shaded area represents a 90% confidence interval

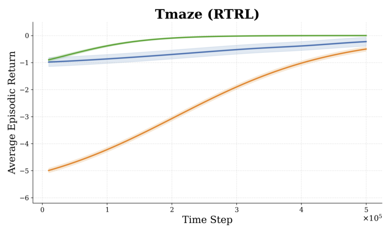

Figure 7: Graph of Stream Q(λ) trained using RTRL with average episodic return at each time step for corridor lengths 10 (green), 100 (orange), 1000 (blue).The results are averaged over 10 independent runs. The shaded area represents a 90% confidence interval

Figure 6: shows that for a corridor length of 10, the BPTT trained stream Q (λ) converges very quickly, while increasing the corridor length to 100 greatly reduces the rate of convergence. The length of 10000 takes a lot of steps to reach expected average episodic return.
Figure 7: The corridor length 10 approaches average episodic return 0, as does corridor length 100. Corridor length 1000 struggles to reach the average episodic return 0.

The BPTT algorithm reaches the average episodic return of 0 for all corridor lengths, however, RTRL struggles to reach the average episodic return of 0 for corridior length 1000. This shows that BPTT performed better than RTRL for more complex tasks, however, with longer time steps, we would expect RTRL to converge to average episodic return of 0.

6 Discussion

6.1 Prediction Task

Our results on the prediction task show a clear increase in performance when using eLSTM on the Copy Task versus RTUs when measured by average final model test accuracy. The difference in performance between eLSTM and RTUs changes little when exposed to changes to hidden size and the length of the sequence the RNN needs to remember. Furthermore, eLSTMs display a smaller standard deviation than RTUs when exposed to these changes. This suggests that the eLSTM as a gated RNN structure is more consistent when recalling previous information when exposed to changes in its environment than RTUs. The results of our prediction task show that eLSTMs are a promising direction when choosing an RNN architecture to train streaming DRL agents using RTRL.

6.2 Reinforcement Learning Problem

As for T-maze, BPTT showed a slight advantage on corridor length 10, where its unroll easily fit within available memory. On the other hand, RTRL performed better in longer corridor lengths. BPTT must calculate gradients from the reward, which is at the end of the corridor all the way back to the cue at the start of the Maze. The longer the corridor, the weaker or more memory hungry those gradients become. Given these constraints, BPTT has difficulty forming a cue-reward association in the 100 and 1000 step mazes. This is clearest with corridor length 1000, where RTRL reached the same final return around 40% faster. To summarize, when the temporal dependencies exceed BPTT windows, eLSTM-RTRL agent yields equal or greater reward under tight memory updates. On the other side, BPTT remains competitive only in very short-horizon tasks.

7 Conclusion

7.1 Limitations

A significant limitation of our study is that we did not procure enough samples in our experiments which hurts the strength of our results. A greater variety of hidden state size comparisons in learning the Copy Task could reveal greater contrasts in performance within different runs of eLSTM and RTUs. For the reinforcement learning experiment, we did not do experiments on different hidden sizes affecting performance of RTRL and BPTT. Due, to time and resource limitation, we could only do 3 runs for RTRL. This could explain why it performed poorly for corridor length 1000, as one bad seed could have skewed our plots.

7.2 Future Works

While we obtained good results using stream-Q(λ) with the eLSTM, experiments on stream-Q with an RTUs could support or disprove evidence collected in the prediction task about eLSTMs and RTUs. Experiments should be done using the other stream-x algorithms on learning T-maze with eLSTMs and RTUs to see how some other streaming DRL algorithms interact with gated and non-gated RNN architectures. In the prediction task, modifying the delay in the Copy Task could provide further insights to how well eLSTMs and RTUs remember information. Also, more repetitions of the prediction task experiments mentioned in this paper, especially with RTUs learning a sequence length of 500 should be done to produce stronger results. Once we have stronger results on the experiments defined on this paper, experiments on environments more complicated than T-maze could signal how generalizable our results are. As a start, Pendulum-V modified for partial observability is a good environment for testing how the methods in this paper learn in a continuous-action setting.

7.3 Final Words

In conclusion, we found that stream-Q(λ) trained using RTRL has performance approaching that of our BPTT baseline in the experiments we ran for corridor lengths 10 and 100. With the memory advantages it has compared to BPTT, RTRL opens the door for applications of streaming DRL agents in hardware common in our daily lives. While more work mentioned in the limitations and future works sections should be done on this topic, we hope this paper provides some preliminary insights into the compatibility of RTRL with novel streaming DRL methods.

References

[1] E. Elelimy, A. White, M. Bowling, and M. White, “Real-time recurrent learning using trace units in reinforcement learning,” 2024. [Online]. Available: https://arxiv.org/abs/2409.01449

[2] M. Elsayed, G. Vasan, and A. R. Mahmood, “Streaming deep reinforcement learning finally works,” 2024. [Online]. Available: https://arxiv.org/abs/2410.14606

[3] Y. O. Corentin Tallec, “Unbiasing truncated backpropagation through time,” 2017. [Online]. Available: https://arxiv.org/pdf/1705.08209

[4] K. Irie1, A. Gopalakrishnan, and J. Schmidhuber, “Exploring the promise and limits of real-time recurrent learning,” 2024. [Online]. Available: https://arxiv.org/pdf/2305.19044

[5] N. Zucchet, R. Meier, S. Schug, A. Mujika, and J. Sacramento, “Online learning of long-range dependencies,” 2023. [Online]. Available: https://arxiv.org/abs/2305.15947

[6] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, “Dive into deep learning,” 2021. [Online]. Available: https://d2l.ai/index.html

[7] B. Bakker, “Reinforcement learning with long short-term memory,” 2001. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2001/file/a38b16173474ba8b1a95bcbc30d3b8a5-Paper.pdf

[8] S. Hochreiter and J. Schmidhuber, “Long short-term memory.” 1997. [Online]. Available: https://www.researchgate.net/publication/13853244_Long_Short-Term_Memory

[9] A. Orvieto, S. L. Smith, A. Gu, A. Fernando, C. Gulcehre, R. Pascanu, and S. De, “Resurrecting recurrent neural networks for long sequences,” 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.06349

[10] E. Martin and C. Cundy, “Parallelizing linear recurrent neural nets over sequence length,” 2017. [Online]. Available: https://doi.org/10.48550/arXiv.1709.04057

[11] N. Zucchet, R. Meier, S. Schug, A. Mujika, and J. a. Sacramento, “Online learning of long-range dependencies,” 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.15947

[12] E. Elelimy, A. White, M. Bowling, and M. White, “Real-time recurrent learning using trace units in reinforcement learning,” 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2409.01449

[13] J. Ravey, “The copying task,” 2018. [Online]. Available: https://wiki.jamesravey.me/books/benchmarks-and-exercises/page/the-copying-task

[14] C. Allen, A. Kirtland, R. Y. Tao, S. Lobel, D. Scott, N. Petrocelli, O. Gottesman, R. Parr, M. L. Littman, and G. Konidaris, “Mitigating partial observability in sequential decision processes via the lambda discrepancy,” 2024. [Online]. Available: https://arxiv.org/abs/2407.07333

8 Appendix

8.1 Acknowledgments

We would like to thank our professor: Russell Greiner, our TA: Sam Scholnick-Hughes for giving valuable feedback throughout the semester. We would also like Rupam Mahmood for introducing this project to us as well as our domain expert: **Dr.** Mohamed Elsayed for taking the time to guide us through our implementations and explaining important concepts to us.

8.2 Full Prediction Task Results

eLSTM learning a sequence length = 50

	eLSTM (hidden size = 512)	**eLSTM (hidden size = 1024**
	96.2	97.17
	93.71	94.99
	98.45	97.58
AVERAGE	95.94	96.49
STDEV	2.1	1.30

RTU learning a sequence length = 50

	RTU (hidden size = 512	**RTU (hidden size = 1024**
	60.02	73.03
	73.1	68.45
	56.72	57.94
AVERAGE	63.28	66.47
STDEV	8.66	7.74

eLSTM learning a sequence length = 500

	eLSTM (hidden size = 512)	**eLSTM (hidden size = 1024**
	95.77	97.58
	92.08	98.47
	96.56	97.89
AVERAGE	94.80	97.98
STDEV	2.39	0.45

RTU learning a sequence length = 500

	RTU (hidden size = 512	**RTU (hidden size = 1024**
	49.72	54.67
AVERAGE	49.72	54.67
STDEV	0	0