Modern Hopfield Networks for Return Decomposition for Delayed Rewards

Anonymous Author(s)
Affiliation
Address
email

Abstract

Delayed rewards, which are separated from their causative actions by irrelevant actions, hamper learning in reinforcement learning (RL). Especially real world problems often contain such delayed and sparse rewards. Recently, *return decomposition for delayed rewards* (RUDDER) employed pattern recognition to remove or reduce delay in rewards, which dramatically simplifies the learning task of the underlying RL method. RUDDER was realized using a long short-term memory (LSTM). The LSTM was trained to identify important state-action pair patterns, responsible for the return. Reward was then redistributed to these important state-action pairs. However, training the LSTM is often difficult and requires a large number of episodes. In this work, we replace the LSTM with the recently proposed continuous modern Hopfield networks (MHN) and introduce Hopfield-RUDDER. MHN are powerful trainable associative memories with large storage capacity. They require only few training samples and excel at identifying and recognizing patterns. We use this property of MHN to identify important state-action pairs that are associated with low or high return episodes and directly redistribute reward to them. However, in partially observable environments, Hopfield-RUDDER requires additional information about the history of state-action pairs. Therefore, we evaluate several methods for compressing history and introduce *reset-max history*, a lightweight history compression using the max-operator in combination with a reset gate. We experimentally show that Hopfield-RUDDER is able to outperform LSTM-based RUDDER on various 1D environments with small numbers of episodes. Finally, we show in preliminary experiments that Hopfield-RUDDER scales to highly complex environments with the Minecraft *ObtainDiamond* task from the MineRL NeurIPS challenge.

Introduction

Recent advances in reinforcement learning (RL) have resulted in impressive models that are capable of surpassing humans in games [22, 13, 19]. However, RL is still waiting for its breakthrough in real world applications, which are often characterized by delayed and sparse rewards [4]. Delayed rewards are given later than their causative action, separated by irrelevant state-action pairs [20]. This delay hampers and slows down learning [1, 15, 12]. Recently, [1] propose *return decomposition for delayed rewards* (RUDDER) to use pattern recognition to detect the causative actions, which can be used to remove or alleviate these delays. As a consequence, RUDDER dramatically simplifies and speeds up learning of the underlying RL algorithms. RUDDER can be realized in different ways, including a long short-term memory (LSTM) [6] in [1] or sequence alignment in [14].

But these realizations of RUDDER come with drawbacks. LSTMs require a large number of samples, while sequence alignment is difficult to scale to large state-action spaces. Thus, there is a need for scalable and sample efficient realization of RUDDER.

A key idea of RUDDER is to identify important patterns in the state-action sequence. This is a task that associative memories, such as the recently introduced continuous modern Hopfield networks (MHN) [16, 23], excel at. In this work, we introduce Hopfield-RUDDER, a realization of RUDDER based on MHN to identify important state-action pairs. We show that Hopfield-RUDDER is scalable and can reduce the reward delay even with a small number of samples.

In related work, adapted Transformer [21] architectures, which share characteristics of MHN, for RL have been proposed. In [2], RL is reformulated as a conditional sequence modelling task for *offline* RL that can be learned by a *Decision Transformer*. [8] uses a *linear Transformer* [9] as an outer-product-based fast-weight programmer [17, 18]. However, in contrast to Hopfield-RUDDER, these approaches require large amounts of data, as they are based on Transformer architectures. Furthermore, Hopfield-RUDDER can speed up any learning algorithm that relies on a delayed reward signal.

Our main contributions are: We (i) introduce modern Hopfield networks for return decomposition for delayed rewards (Hopfield-RUDDER), (ii) propose and compare different methods, such as *reset-max history*, for history compression for Hopfield-RUDDER in POMDP environments, (iii) show that Hopfield-RUDDER outperforms LSTM-based RUDDER on 1D environments with small numbers of samples, and (iv) perform preliminary experiments in the Minecraft environment indicating that Hopfield-RUDDER scales to large and complex environments.

Review

We define our setting as a finite Markov decision process (MDP) or a finite partially observable Markov decision process (POMDP) to be a 4-tuple of $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p)$ of finite sets $\mathcal{S}$ with states s (random variable S_t at time t), $\mathcal{A}$ with actions a (random variable A_t at time t), $\mathcal{R}$ with rewards r (random variable R_t at time t), and state-reward transition distribution $p(S_{t+1} = s_{t+1}, R_{t+1} = r_t \mid S_t = s_t, A_t = a_t)$. The return of a sequence of length T at time $t = \{1, \ldots, T\}$ is defined as $g_t = \sum_{k=0}^{T-t} r_{t+k}$.

RUDDER: Complex tasks are often hierarchically composed of sub-tasks. Hence, the Q-function of an optimal policy often resembles a step function [1]. Such steps indicate patterns like achievements, failures, accomplished sub-tasks, or changes of the environment. RUDDER reduces the delay in the rewards by identifying these patterns and moving (redistributing) the rewards to the causative state-action patterns. [1] propose an LSTM as realization of RUDDER. The LSTM predicts the return at the end of an episode as early as possible. These predictions $\widehat{g}$ are then used to redistribute the reward: $r_t = \widehat{g}_t - \widehat{g}_{t-1}$.

The redistributed reward serves as reward for a subsequent learning method and can be used to optimize a policy. However, training an LSTM network requires a large amount of episodes, which are often difficult or expensive to obtain.

Modern Hopfield networks: Recently, continuous modern Hopfield networks (MHN) have been introduced in [16] as a continuous generalization of binary classical [7] and binary modern [10, 3, 11] Hopfield networks. MHN can be directly integrated into deep learning architectures. There they serve as powerful associative memories with large memory capacity and excel at pattern-recognition as they associate inputs with similar patterns in their memory [16, 23].

Hopfield-RUDDER

The application of MHN for RUDDER naturally follows from the idea of identifying patterns: Patterns, e.g. state-action pairs, of observed episodes with known returns are stored in the memory of the MHN. A new pattern can then be used to query this memory for similar patterns and their returns. This yields an estimated return of the new pattern, which can be used to redistribute the rewards.

Fig. 1 illustrates this in detail: The MHN associates N *stored patterns* $\boldsymbol{Y} = \{\boldsymbol{y}_i\}_{i=1}^N$ from previously observed episodes and their corresponding observed returns $\boldsymbol{G} = \{g_i\}_{i=1}^N$ with M new *state patterns* $\boldsymbol{U} = \{\boldsymbol{u}_i\}_{i=1}^M$. As such, it directly associates the unknown state patterns $\boldsymbol{U}$ with estimated returns $\widehat{\boldsymbol{G}} = \{\widehat{g}_i\}_{i=1}^M$. β is a hyper-parameter controlling the temperature of the $\mathrm{softmax}$ function.

The raw state patterns $\boldsymbol{U}$ and stored patterns $\boldsymbol{Y}$ may be transformed by functions m_{state} and m_{stored}, for example with linear mappings, as in [16, 23].

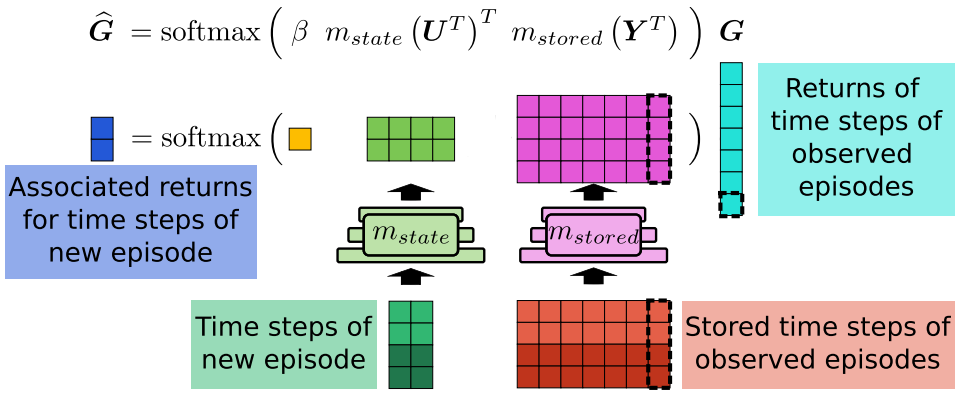

Figure 1: Illustration of modern Hopfield networks for return decomposition in Hopfield-RUDDER. The associative memory of modern Hopfield networks associates 2 time steps (raw state patterns) $\boldsymbol{U}$ of a new episode with 7 time steps (stored patterns) $\boldsymbol{Y}$ of previously observed episodes and their corresponding observed returns $\boldsymbol{G}$. Thereby, the state patterns $\boldsymbol{U}$ are associated with estimated returns $\widehat{\boldsymbol{G}}$. $\boldsymbol{U}$ and $\boldsymbol{Y}$ are transformed via m_{state} and m_{stored} before association, respectively, and can include information about previous state-action pairs, such as via reset-max history. Dotted rectangles show a single raw and transformed stored pattern with its observed return.

This results in a return decomposition function ψ in the form of

$$\widehat{\boldsymbol{G}} = \psi(\boldsymbol{Y}, \boldsymbol{G}, \boldsymbol{U}; \beta) \;=\; \text{softmax}\left(\beta \, m_{state}\left(\boldsymbol{U}^T\right)^T \; m_{stored}\left(\boldsymbol{Y}^T\right)\right) \boldsymbol{G}. \tag{1}$$

The redistributed reward r_t at time t can then be obtained by computing the differences of the estimated return values as

$$r_t = \widehat{g}_t - \widehat{g}_{t-1}. \tag{2}$$

For state patterns $\boldsymbol{u}_t$ and $\boldsymbol{u}_{t-1}$ at times t and $t-1$ this results in

$$r_t \;=\; \psi(\boldsymbol{Y}, \boldsymbol{G}, \boldsymbol{u}_t; \beta) - \psi(\boldsymbol{Y}, \boldsymbol{G}, \boldsymbol{u}_{t-1}; \beta), \tag{3}$$

where $\boldsymbol{Y}$ contains a representation of the stored state-action pairs $\{(s_i, a_i)\}_{i=1}^N$ with their known corresponding returns $\boldsymbol{G}$ and $\boldsymbol{u}_t$ represents a state-action pair (s_t, a_t) without known return.

In the simplest case, a raw state pattern $\boldsymbol{u}$ or stored pattern $\boldsymbol{y}$ is a vector containing the observation s_t and action a_t at time t in an episode. The estimated return for a single state pattern $\boldsymbol{u}_t$ at time t in this case is $\widehat{g}_t = \psi(\boldsymbol{Y}, \boldsymbol{G}, s_t, a_t; \beta)$ with the redistributed reward r_t computed from state patterns $\boldsymbol{u}_{t-1}$ and $\boldsymbol{u}_t$ as $r_t = \widehat{g}_t - \widehat{g}_{t-1} = \psi(\boldsymbol{Y}, \boldsymbol{G}, s_t, a_t; \beta) - \psi(\boldsymbol{Y}, \boldsymbol{G}, s_{t-1}, a_{t-1}; \beta)$.

History compression and reset-max history: Assume a key event at time t_1, e.g. the collection of a key, that can result in a reward at time t_2, e.g. the opening of a chest. In a POMDP, this key event might only be observable at t_1 but not at t_2, e.g. the observation space contains no information whether a key has been collected already. Assume further that observation s_t and action a_t are used as state or stored pattern: The association of the pattern at t_2 with a return, which depends on whether a key was collected at t_1, may severely suffer from the missing information.

To address this problem, the state and stored patterns can be augmented by the history of the previous state-action pairs. However, retaining the complete history of a state-action pair and including it in the state and stored patterns would not be feasible for environments with large observation spaces or long episode sequences.

As the purpose of this history is to enable the detection and storage of key events, we propose a fast and light-weight history compression method: the *reset-max history*. Assuming a vector ${\boldsymbol{v}^o}_t \in \mathbb{R}^{K \times 1}$ with K features that contains a representation of the state-action pair (s_t, a_t) at time $t = \{1, \ldots, T\}$, the reset-max history features ${\boldsymbol{v}^h}_t \in \mathbb{R}^{J \times 1}$ with J features, and the final state or stored pattern $\boldsymbol{v}_t \in \mathbb{R}^{(J+K) \times 1}$ is computed as follows:

$$\boldsymbol{v}_t \;=\; \left[{\boldsymbol{v}^h}_t; {\boldsymbol{v}^o}_t\right] \;\; ; \quad {\boldsymbol{v}^h}_t \;=\; \max_{i=1,\ldots,(t-1)} \left({\boldsymbol{v}^o}_i\right) \cdot f_{reset}\left({\boldsymbol{v}^h}_{t-1}, {\boldsymbol{v}^o}_t\right) \tag{4}$$

with ${\boldsymbol{v}^h}_0$ initialized with $-\infty$ and

$$f_{reset}\left({\boldsymbol{v}^h}_{t-1}, {\boldsymbol{v}^o}_t\right) \;=\; \sigma\left(\boldsymbol{W}\left[{\boldsymbol{v}^h}_{t-1}; {\boldsymbol{v}^o}_t\right]\right), \tag{5}$$

where $[;]$ is the operator for vertical concatenation, the $\max$ operator is used to store the previously observed maximum feature values, and the reset gate f_{reset} utilizes the sigmoid activation function σ and learned weight matrix $\boldsymbol{W} \in \mathbb{R}^{J \times (J+K)}$ to reset the stored maximum feature values. Notably, this changes the return decomposition function ψ for a state pattern $\boldsymbol{u}_t$ to be dependent on all previous and current state-action pairs $\{(s_i, a_i)\}_{i=1}^{t}$ in the episode: $\widehat{g}_t = \psi(\boldsymbol{Y}, \boldsymbol{G}, s_{1,\ldots,t}, a_{1,\ldots,t}; \beta, \boldsymbol{W})$.

Experiments

In this section, we first compare different history compression methods on multiple 1D toy environments. Subsequently, we compare the performance of Hopfield-RUDDER to the originally proposed LSTM-RUDDER. Finally, we visually analyse the reward redistribution of Hopfield-RUDDER on the difficult and complex task of collecting a diamond in the Minecraft environment.

1D key-chest environment. We designed 16 different versions of a 1D environment, the *1D key-chest environment*, in order to evaluate the performance of Hopfield-RUDDER, different history compression methods, and to compare Hopfield-RUDDER to LSTM-RUDDER. We use a reward redistribution score rr_score, which is based on the known optimal policy, for this evaluation and comparison. For more details see App. A1.2.

Environment details. The 1D key-chest environment is illustrated in Fig. 2. The agent starts at position *s* in the middle. At each position it can move either one position to the right or to the left, except for the left-most and right-most position, where further movement to the left and right, respectively, is ignored. If the agent visits position *k*, it collects 1 *key*. If the agent visits position *s*, it loses all collected *keys* with a probability of p_l. If the agent visits position *c* while holding n_k *keys*, it will receive 1 reward at the end of the episode. All episodes have the same fixed length. A reward of 0 or 1 is given at the end of each episode, depending on whether the chest was opened or not.

We evaluate the performance of RUDDER on different versions of this environment with $p_l = \{0, 0.5\}$ and $n_k = \{1, 3\}$. We consider fully observable MDP and partially observable POMDP versions of the environment. Furthermore, we add $n_{rnd} = \{0, n_{obs}\}$ features that contain random values $\{0, 1\}$ to the observation space, where n_{obs} is the number of features of the original observation space. In total, we created 16 different versions of this environment (see Tab. 1 and App. A1).

To redistribute reward in this environment successfully, RUDDER has to identify that position *k* needs to be visited n_k times before visiting position *c*. Furthermore, RUDDER has to identify that the *key*-collection process has to be restarted if the *keys* are lost at *s*.

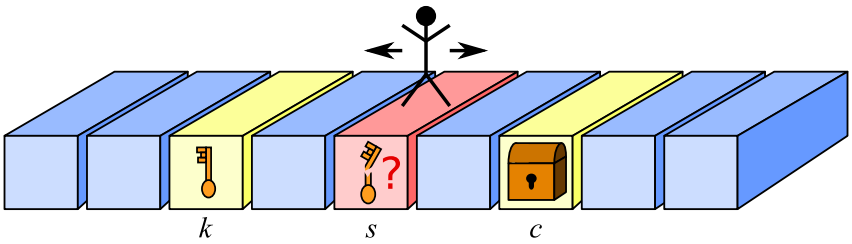

Figure 2: Illustration of 1D key-chest environment with special states *k*, *s*, *c*, and 9 states in total.

Experimental setup. We sample a large test set of $1,000$ episodes and small training sets of $\{8, 16, 32, 64, 128, 256, 512\}$ episodes, using a random agent. Half of the episodes have a return of 0 and 1 in each set, with fixed sequence lengths between 32 and 148 times per episode (see App. A1).

Reset-max history outperforms other history compression methods. We first analyze the performance of Hopfield-RUDDER, where the training set is used as stored patterns. For this, we evaluate Hopfield-RUDDER without history information and with 5 different versions of history compression: (i) feature-wise max-pooling, (ii) feature-wise sum-pooling, (iii) fully-connected LSTM (LSTMf), (iv) sparsely-connected LSTM (LSTMs), and (v) reset-max history. Furthermore, we evaluate each setting with a linear mapping $m_{state} = m_{stored}$ and without any mapping. As shown

in Tab. 1 and A2, reset-max history consistently outperforms all other Hopfield-RUDDER versions, except for one POMDP setting and two MDP settings, in which it is the runner-up method.

Hopfield-RUDDER with reset-max history consistently outperforms LSTM-RUDDER. Subsequently, we analyze the performance of LSTM-RUDDER with reset-max history and without history information. As shown in Tab. 1, Hopfield-RUDDER with reset-max history consistently outperforms all LSTM-RUDDER versions on all environment versions.

More details on results, history compression methods, and training can be found in App. A1.

				Hopfield-RUDDER												LSTM-RUDDER			
History:				reset-max		LSTMf		LSTMs		max		sum		none		reset-max		none	
Mapping:				linear	none	linear	none	linear	none	linear	none	linear	none	linear	none	none	none	none	none
Learning rate:				$1e-3$	-	$1e-3$	-	$1e-3$	-	$1e-3$	-	$1e-3$	-	$1e-3$	-	$1e-4$	$1e-3$	$1e-4$	$1e-3$
Environment specs																			
	n_k	p_l	n_{rnd}																
POMDP	1	0%	0	**93.08**	82.42	72.23	70.37	76.94	66.79	91.85	89.96	78.72	41.36	58.30	58.61	65.27	72.85	57.25	74.32
				±4.87	±8.80	±7.12	±6.40	±6.65	±10.29	±5.40	±5.74	±7.41	±6.09	±3.24	±3.85	±6.69	±5.97	±4.41	±5.93
POMDP	1	0%	n_{obs}	87.66	81.37	66.17	61.51	75.31	62.36	**87.90**	84.31	76.91	41.92	56.90	57.54	62.23	73.37	53.84	68.23
				±5.43	±4.88	±5.17	±5.60	±7.56	±6.42	±6.74	±5.23	±6.67	±4.15	±3.59	±2.35	±8.37	±4.04	±5.03	±7.34
POMDP	1	50%	0	**75.41**	73.57	58.22	58.75	62.77	59.93	62.85	65.52	52.80	49.28	54.83	55.99	60.21	55.25	48.85	52.11
				±10.33	±13.01	±7.55	±7.72	±9.41	±11.43	±7.57	±7.03	±6.61	±2.10	±6.01	±5.66	±4.60	±8.43	±7.76	±7.52
POMDP	1	50%	n_{obs}	**69.19**	66.19	54.87	54.98	56.33	54.33	61.47	60.13	53.07	48.90	54.43	54.64	51.34	51.80	48.79	50.14
				±7.95	±5.52	±4.37	±3.84	±5.00	±6.34	±6.31	±3.84	±3.50	±1.36	±4.01	±3.43	±6.32	±5.01	±2.59	±4.34
POMDP	3	0%	0	**80.91**	74.06	66.61	64.94	76.06	73.80	77.02	77.59	68.16	45.17	54.88	55.31	59.50	66.62	55.37	63.68
				±7.74	±8.43	±8.17	±6.98	±8.44	±13.78	±8.81	±6.17	±12.70	±5.25	±4.03	±4.08	±5.75	±7.37	±6.15	±9.36
POMDP	3	0%	n_{obs}	**77.96**	74.07	63.42	61.90	73.32	68.99	75.30	75.06	66.61	45.56	54.56	54.41	53.34	65.84	51.84	58.34
				±5.95	±4.84	±4.95	±4.01	±6.65	±8.06	±6.29	±4.19	±10.50	±3.73	±3.33	±3.42	±6.20	±7.19	±7.31	±6.18
POMDP	3	50%	0	**74.30**	69.60	56.17	58.38	59.50	62.42	57.42	60.44	54.61	55.36	54.84	56.58	52.78	52.80	47.96	49.51
				±10.68	±11.35	±7.57	±8.99	±11.27	±13.61	±7.22	±5.95	±4.72	±1.59	±7.44	±6.88	±2.72	±7.65	±9.88	±6.34
POMDP	3	50%	n_{obs}	**67.30**	63.82	53.07	53.06	55.86	54.81	55.79	55.46	55.05	55.55	53.92	53.39	52.64	50.51	46.40	49.93
				±8.50	±6.76	±4.72	±3.48	±5.84	±8.43	±5.70	±3.25	±2.70	±0.81	±4.82	±3.99	±5.68	±4.72	±3.47	±4.91

Table 1: Comparison of different Hopfield-RUDDER and LSTM-RUDDER versions w.r.t. reward redistribution score rr_score on different versions of the 1D key-chest environment. Results show the mean rr_score over all training set sizes and a 10-fold cross-validation (CV). Error bars show mean standard deviation of 10-fold CV over all training set sizes. Hopfield-RUDDER with reset-max history consistently outperforms all LSTM-RUDDER versions. For MDP versions see Tab. A2.

Minecraft. In order to show that Hopfield-RUDDER can scale to complex problems, we perform preliminary experiments within the challenging MineRL Minecraft environment [5]. Specifically, we use Hopfield-RUDDER to redistribute rewards for the task `ObtainDiamond`. As the high complexity of `ObtainDiamond` makes random exploration unfeasible, [5] provide the Minecraft environment with demonstrations of human players solving this task. Players and agents are randomly placed in a procedurally generated 3D environment without any items in their possession. The objective is to gather resources and build the tools required to obtain a diamond.

For `ObtainDiamond`, auxiliary reward is given the first time the player obtains an item, even if multiple copies of this item are required. For example, agents have to gather multiple logs in order to build all necessary tools but receive reward only for the first log they acquire. However, we omit these auxiliary rewards and only use episodic return by giving a return of 1 for demonstrations that obtain a diamond and a return of 0 for those that do not manage to obtain it.

In order to show that Hopfield-RUDDER is able to redistribute reward to relevant state-action pairs in `ObtainDiamond`, we train a simple Hopfield-RUDDER model using the *reset-max history* history compression. We only use the sequence of inventory states and associated actions as input to Hopfield-RUDDER. Following [14], we use 10 successful and 10 unsuccessful demonstration episodes as training set. The state-action pairs from these 20 episodes are used as stored patterns $\boldsymbol{Y}$. We use the obfuscated version of the environment and demonstrations, therefore the observations s and actions a

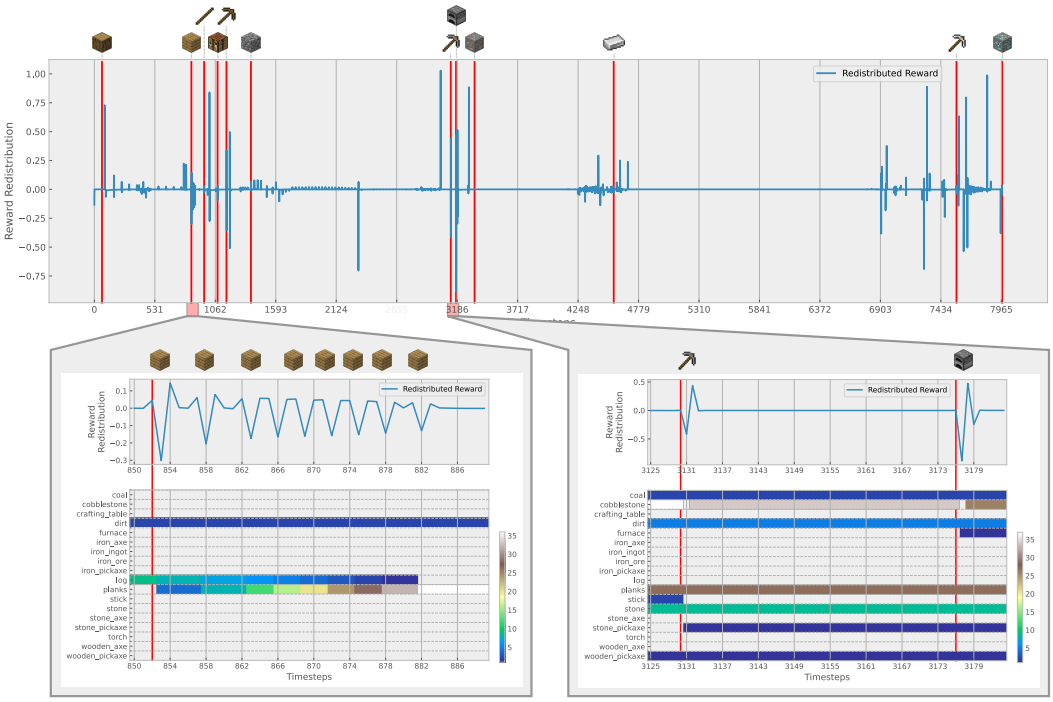

Figure 3: **Top:** example of redistributed reward of a successful demonstration using only the inventory and actions. Red vertical lines indicate the auxiliary reward an agent would receive from the environment, however the reward redistribution is trained only with reward at episode end. The reward redistribution seems to identify relevant state-action pairs. **Bottom left:** Magnified region showing acquisition of *planks* through crafting. First, logs are used up, resulting in negative redistributed reward followed by an increase in planks, which increases the reward prediction. **Bottom right:** Time steps showing the crafting of *stone pickaxe* and *furnace*. Both items require stone and show a similar behavior as crafting planks. Predicted reward decreases when the inventory state for base component decreases and increases when the crafted item appears in the inventory.

are the hidden representation of an unknown auto encoder (for further details see App. A2.3). We use a neural network with 2 hidden layers and ReLU activation as mapping functions $m_{state} = m_{stored}$.

Training Hopfield-RUDDER does not require massive compute resources. On the contrary, training a model for 100 episodes takes only minutes on an Nvidia A40 GPU. A small hyper-parameter search showed that the model is not overly sensitive to hyper-parameter changes and most models produce a similar reward redistribution.

Fig. 3 shows the redistribution for a demonstration episode, which was not used for training the model. We use the non-obfuscated inventory states for visualization. It visually seems that Hopfield-RUDDER is able to redistribute the reward to important state-action pairs. Furthermore, Hopfield-RUDDER appears to not redistribute rewards to irrelevant state-action pairs, resulting in a reward redistribution with little noise.

Conclusion

We have introduced Hopfield-RUDDER, a novel realization of RUDDER. Hopfield-RUDDER replaces the LSTM in the original LSTM-RUDDER with a powerful auto-associative memory, the modern Hopfield network. In contrast to LSTM-RUDDER, Hopfield-RUDDER has less hyper-parameters and can be trained with a drastically lower number of samples.

References

[1] J. A. Arjona-Medina*, M. Gillhofer*, M. Widrich*, T. Unterthiner, J. Brandstetter, and S. Hochreiter. RUDDER: Return decomposition for delayed rewards. In *Advances in Neural Information Processing Systems*, pages 13544–13555, 2019.

[2] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *arXiv preprint arXiv:2106.01345*, 2021.

[3] M. Demircigil, J. Heusel, M. Löwe, S. Upgang, and F. Vermet. On a model of associative memory with huge storage capacity. *Journal of Statistical Physics*, 168(2):288–299, 2017.

[4] F. Dulac-Arnold, D. J. Mankowitz, and T. Hester. Challenges of real-world reinforcement learning. *CoRR*, abs/1904.12901, 2019.

[5] W. H. Guss, C. Codel, K. Hofmann, B. Houghton, N. Kuno, S. Milani, S. P. Mohanty, D. P. Liebana, R. Salakhutdinov, N. Topin, M. Veloso, and P. Wang. The MineRL competition on sample efficient reinforcement learning using human priors. *arXiv*, 2019.

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

[7] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, 1982.

[8] K. Irie, I. Schlag, R. Csordás, and J. Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. *arXiv preprint arXiv:2106.06295*, 2021.

[9] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. In *Proceedings of the 37th International Conference on Machine Learning*, pages 5156–5165. PMLR, 2020.

[10] D. Krotov and J. J. Hopfield. Dense associative memory for pattern recognition. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, pages 1172–1180. Curran Associates, Inc., 2016.

[11] D. Krotov and J. J. Hopfield. Dense associative memory is robust to adversarial inputs. *Neural Computation*, 30(12):3151–3167, 2018.

[12] J. Luoma, S. Ruutu, A. W. King, and H. Tikkanen. Time delays, competitive interdependence, and firm performance. *Strategic Management Journal*, 38(3):506–525, 2017.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, , and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[14] V. P. Patil, M. Hofmarcher, M.-C. Dinu, M. Dorfer, P. M. Blies, J. Brandstetter, J. A. Arjona-Medina, and S. Hochreiter. Align-rudder: Learning from few demonstrations by reward redistribution. *CoRR*, abs/2009.14108, 2020.

[15] H. Rahmandad, N. Repenning, and J. Sterman. Effects of feedback delay on learning. *System Dynamics Review*, 25(4):309–338, 2009.

[16] H. Ramsauer, B. Schäfl, J. Lehner, P. Seidl, M. Widrich, T. Adler, L. Gruber, M. Holzleitner, M. Pavlović, G. K. Sandve, V. Greiff, D. Kreil, M. Kopp, G. Klambauer, J. Brandstetter, and S. Hochreiter. Hopfield networks is all you need. *published at ICLR 2021; arXiv preprint arXiv:2008.02217*, 2020.

[17] J. Schmidhuber. Learning To Control Fast-Weight Memories: An Alternative To Dynamic Recurrent Networks, 1991.

[18] J. Schmidhuber. Learning to Control Fast-Weight Memories: An Alternative to Dynamic Recurrent Networks. *Neural Computation*, 4(1):131–139, 1992.

[19] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[20] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Dept. of Comp. and Inf. Sci., 1984.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[22] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, J. Quan, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. P. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing. Starcraft II: A new challenge for reinforcement learning. *ArXiv*, 2017.

[23] M. Widrich, B. Schäfl, M. Pavlović, H. Ramsauer, L. Gruber, M. Holzleitner, J. Brandstetter, G. K. Sandve, V. Greiff, S. Hochreiter, and G. Klambauer. Modern hopfield networks and attention for immune repertoire classification. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 18832–18845. Curran Associates, Inc., 2020.