LEARNING TO RECOVER FROM FAILURES USING MEMORY

Anonymous authors
Paper under double-blind review

ABSTRACT

Learning from past mistakes is a quintessential aspect of intelligence. In sequential decision-making, existing meta-learning methods that learn a learning algorithm utilize experience from only a few previous episodes to adapt their policy to new environments and tasks. Such methods must learn to correct their mistakes from highly-correlated sequences of states and actions generated by the same policy's consequent roll-outs during training. Learning from correlated data is known to be problematic and can significantly impact the quality of the learned correction mechanism. We show that this problem can be mitigated by augmenting current systems with an external memory bank that stores a larger and more diverse set of past experiences. Detailed experiments demonstrate that our method outperforms existing meta-learning algorithms on a suite of challenging tasks from raw visual observations. Videos are available at: https://sites.google.com/view/learn-from-failures.

1 INTRODUCTION

Agents often fail to solve a new decision-making task in their first attempt. While failures do not provide good rewards, they communicate what not to do and often hint at possible solutions. Analyzing past failures to improve the current strategy is vital for adapting and solving new tasks. Because of being memory-less, popular deep reinforcement learning (DRL) algorithms (Mnih et al., 2015; Silver et al., 2017; Lillicrap et al., 2015) cannot exploit previous episodes to adjust their decisions at test time. However, recently proposed methods that learn a learning algorithm (Duan et al., 2016; Mishra et al., 2017) address this shortcoming by processing the history of multiple episodes to select the next action. Such meta-learning algorithms can learn from past failures and have achieved excellent performance on many challenging partially-observable tasks.

In this work, we utilize two observations to construct a modified training procedure for meta-learning methods that learn a learning algorithm. The proposed training scheme leads to substantial performance gains on tasks that require recovery from failures. To understand our contributions, let's first review the training process of existing methods: An agent is provided with a set of tasks to solve. Training happens in trials. In every trial, a training task is randomly sampled, and the agent acts for a fixed number of steps (say T). The agent's policy is optimized to maximize the sum of rewards collected in the trial. If the agent completes the task or the episode terminates before T steps, reset is performed, and a new episode starts. A trial typically contains multiple episodes.

Current meta-learning algorithms learn a policy that depends on all previous episodes only in the same trial. Different episodes in a single trial are the outcome of executing the policy network with the same weights and are therefore correlated. For example, consider the situation when the agent makes mistake A in the current trial that it learns to correct. Further, suppose if the agent re-encounters the same task, it makes mistake B, that it then learns to rectify. Because the agent has no memory of mistakes across trials, it may again make mistake A when it re-encounters the task. This causes instability in training and is a well-known problem in scenarios where the agent self-generates the training data. Replay buffers were introduced to largely mitigate this issue in training off-policy reinforcement learning algorithms (Mnih et al., 2015; Lillicrap et al., 2015). To overcome this shortcoming in the meta-learning context, we propose having a memory bank for each task that stores episodes across trials. By storing mistakes across trials, such memory banks encourage learning of policies that can recover from a diverse set of failures. Our second observation

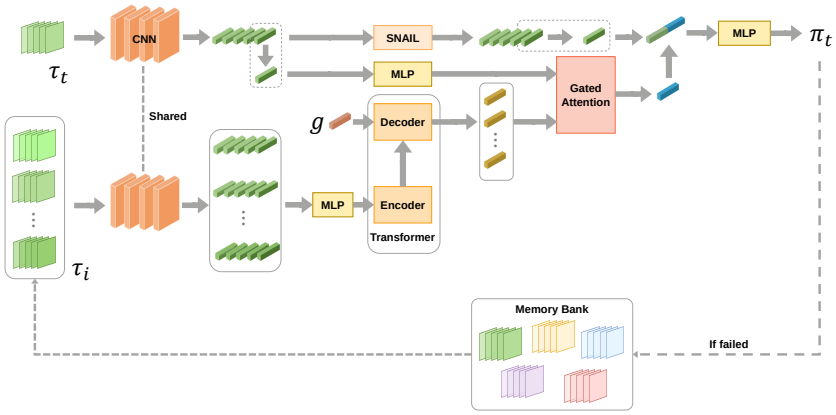


Figure 1: Policy architecture. The policy starts with an empty memory bank. An agent executes the policy in a task. If it fails, the trajectory is added to a memory bank. When the agent performs the task again, the policy adapts the output based on the information from memory. The policy uses the Transformer to extract information from each trajectory and a gated attention mechanism to merge the information of multiple trajectories.

is that despite being successful at a task, it is important to preserve memories of past failures. We humans often remember our mistakes and how we corrected them. Consider the setup where the agent only utilizes memory of episodes from the same trial. As the agent trains, it will make fewer mistakes in the training tasks. In the extreme case of perfect policy fitting, the agent may succeed in the first episode of the trial. As a consequence, the agent's memory will be populated only with successful trajectories. Lack of failures in memory might have the unintended consequence of making the agent forget how to recover from failures by the end of the training. This is undesirable because, unless generalization to new tasks is perfect at test time, the agent is likely to make mistakes from which it must recover. We show that this issue can be mitigated by explicitly storing the most recent failures (possibly from earlier trials) of the agent. As a caveat and as shown by prior works (Duan et al., 2016; Mishra et al., 2017), in partially observable environments, the agent can utilize a successful first episode to reach its goal faster in the subsequent episodes in the same trial. Our method is complementary and especially helps when the agent cannot even succeed in the first episode.

In this paper, we use the observations of labeling and storing failures across trials to build a meta-learning algorithm that outperforms the existing state-of-the-art (Mishra et al., 2017). This was achieved by constructing task-specific memory banks that replay trajectories from multiple previous trials of the same task. We first validate our method on a 2D gridworld, where we present extensive ablation studies. We then show that our method works well at the task of finding objects from the first-person view in a 3D world and the task of touching objects using visual observation with a UR robot.

2 METHOD

Our setup is as follows: An agent is required to solve a set of tasks $\mathcal{M}$. Training is performed in trials. In each trial, the agent executes a fixed number of actions, say T. A trial typically consists of multiple episodes, where each episode refers to the trajectory taken by the agent starting from the initial state until the environment is reset due to either failure or success of the agent or a timeout. During training, suppose the agent encounters a task $\mathcal{M}_i$ in each episode in j^{th} trial. Let the set of episodes in this trial be $\{\tau_k^{i,j}; k \in K_j\}$, where K_j is the total number of episodes in j^{th} trial. Let the k^{th} episode be $\tau_k^{i,j}$: $\left(o_{t^k}^{i,j}, a_{t^k}^{i,j}, r_{t^k}^{i,j}, ..., o_{t^{k+1}-1}^{i,j}, a_{t^{k+1}-1}^{i,j}, r_{t^{k+1}-1}^{i,j}\right)$, where $o_{t^k}^{i,j}, a_{t^k}^{i,j}, r_{t^k}^{i,j}$ represent the observation, action and the reward obtained by the agent at a time step t^k. Further assume that the agent is equipped with $\|\mathcal{M}\|$ memory banks, one for each task: $\mathcal{B}^{\mathcal{M}_i}$. Each memory bank stores a maximum of N trajectories. At every time step in the trial, the agent chooses actions $(a_t^{i,j})$ using the

policy,

$$a_t^{i,j} = \pi_\theta(o_t, h_{t-1}^{i,j}, \mathcal{B}^{\mathcal{M}_i}) \tag{1}$$
$$h_{t-1}^{i,j} = \{o_{1:t-1}^{i,j}, a_{1:t-1}^{i,j}, r_{1:t-1}^{i,j}\}$$

where, $h_{t-1}^{i,j}$ represents the agent's trajectory in the current trial until time $t-1$ and θ represents the policy parameters. The policy π_θ depends on the current observation, the entire history of the current trial ($h_{t-1}^{i,j}$), and the trajectories stored in the memory bank ($\mathcal{B}^{\mathcal{M}_i}$). The policy is optimized to maximize the sum of rewards in the trial.

The memory bank is populated in the following way: Let $f(\tau)$ be a binary function that returns whether the episode τ results in a success ($f(\tau) = 1$) or failure ($f(\tau) = 0$). We assume that we have access to a f that can tell whether the agent fails the tasks or not. In the case of sparse rewards, it is trivial to define f based on the reward signal. Let $\mathcal{F}^{i,j} = \{k : f(\tau_k^{i,j}) = 0; k \in [1, K_j]\}$ be the set of failure episodes in the j^{th} trial. At the end of the trial, the failure episodes $\mathcal{F}^{i,j}$ are added to the memory bank $\mathcal{B}^{\mathcal{M}_i}$. If $\mathcal{B}^{\mathcal{M}_i}$ is out of space, the earliest episodes stored in the memory are dropped. For some tasks, it is possible that the last L steps in the episode carry sufficient information about the cause of failure. In these scenarios, instead of storing the entire episodes, only the last L steps can be stored in $\mathcal{B}^{\mathcal{M}_i}$ to reduce RAM memory usage. In the general case, L can be a hyperparameter to be tuned for a specific task or determined by domain knowledge.

The previous state-of-the-art algorithms, RL^2 (Duan et al., 2016) and SNAIL (Mishra et al., 2017), are special cases of our formulation described in Equation (1). In particular, the policy $a_t^{i,j} = \pi_\theta(o_t, h_{t-1}^{i,j})$ learned by these methods does not depend on the memory bank. Due to the lack of this dependence, training of these prior methods can be unstable. To see why, consider the training process. At the end of every trial, the policy parameters are updated to rectify the agent's mistakes to maximize the agent's rewards. Updated parameters may make the same mistake that the agent made in some past trial. In such a scenario, the agent will oscillate between mistakes in subsequent trials, which will lead to high-variance in gradients and slower convergence. One way to overcome these challenges is to remember a diverse set of mistakes from the past to prevent the agent from making the same mistake. Our method achieves this by maintaining a memory of failures from different past trials. Because the parameters of the policy change across trials, it is possible that the agent will commit different mistakes in different trials. Therefore, accumulating failures across trials is a way to collect a diverse set of past mistakes made by the agent. We show that conditioning the policy on the memory bank stabilizes training and makes our method more effective at correcting mistakes and thereby achieving higher rewards in previously unseen environments.

In the remainder of this section, we provide details about how we condition the policy on $\mathcal{B}^{\mathcal{M}_i}$. First, we describe the method for computing a fixed-size feature vector for each trajectory. Next, we describe how the information is aggregated across trajectories and finally, how it is combined with trajectory information from the current trial. Figure 1 visually illustrates the policy architecture.

2.1 Computing Feature Representation of Trajectories in the Memory Bank

Previous work has shown that the *Transformer architecture* (Vaswani et al., 2017) outperforms purely recurrent networks in modelling long-range temporal correlations (Ott et al., 2019). Taking inspiration, we use the *Transformer* to compute the feature representation of each trajectory in the memory bank. Let the trajectories in $\mathcal{B}^{\mathcal{M}_i}$ be $\{\tau_n^i; n \in [1, N]\}$. The transformer consists of an encoder that computes information per time-step of the trajectory and a decoder that aggregates this information.

$$e_{\tau_n^i} = \text{Encoder}(\Phi(\tau_n^i)) \qquad v_{\tau_n^i} = \text{Decoder}(g, e_{\tau_n^i})$$

where Φ is a function (CNN and MLP layers) that merges the information of observation, action, and reward at each time step, $e_{\tau_n^i} \in \mathbb{R}^{L \times E}$ is an array of the embedding vectors for each time step in the trajectory, g is the query vector that decodes the per time-step embedding into a per-trajectory embedding, and $v_{\tau_n^i} \in \mathbb{R}^E$ is the embedding of the entire trajectory. $g \in \mathbb{R}^E$ is randomly initialized and learned via back-propagation. g is the same for all tasks in $\mathcal{M}$ and learns a decoding vector that decodes the information for a class of tasks. The embedding of each history trajectory is independent of the current observation o_t. Therefore, it only needs to be computed once in every trial, which speeds up the inference.

2.2 AGGREGATING INFORMATION ACROSS TRAJECTORIES IN THE MEMORY BANK

Next, the policy aggregates information from all trajectories in the memory bank $\mathcal{B}^{\mathcal{M}_i}$ using *multi-head self-attention* (Vaswani et al., 2017) with *GRU gating* (Chung et al., 2014; Parisotto et al., 2019). Each trajectory is embedded into a vector of length E as in Section 2.1, resulting in a set of trajectory vectors $v_{\tau^i} \in \mathbb{R}^{N \times E}$. This set is reduced into a single vector $m_o^i \in \mathbb{R}^E$ that represents the contribution of the memory. To enable the agent to weight different failures according to the current observation, the aggregation operation is conditioned on the current observation o_t.

$$m_d^i = \text{MultiHead}(Q, K, V) = \text{MultiHead}(v'_{o_t}, v_{\tau^i}, v_{\tau^i})$$
$$r = \sigma(\text{FC}(m_d^i) + \text{FC}(v'_{o_t}))$$
$$z = \sigma(\text{FC}(m_d^i) + \text{FC}(v'_{o_t}))$$
$$y = \tanh(\text{FC}(m_d^i) + \text{FC}(r \odot v'_{o_t}))$$
$$m_o^i = (1 - z) \odot y + z \odot v'_{o_t}$$

where $v'_{o_t} = \text{MLP}(\text{CNN}(o_t))$ is the embedding vector of the current observation, each FC represents a different linear transformation, $\odot$ is the Hadamard product. As shown in Section 2.1 and the above derivation of computing m_o^i, we have two levels of attention operations to extract information from all the trajectories in the memory bank, which is different from Mishra et al. (2017). The first level is the attention over each frame within a trajectory. The second level is the attention over all the trajectory embedding vectors. The advantage of this is the reduction of time complexity (Appendix B).

2.3 MEMORY-CONDITIONED POLICY

To encode information from past steps of the trial, we use SNAIL (Mishra et al., 2017) architecture:

$$v_{o_t} = \text{SNAIL}(v_{\tau_t})_t$$

where v_{τ_t} is the featurization of current trajectory τ_t.

The final policy π merges information from current trial and memory bank.

$$\pi : \text{MLP}\left[\text{MLP}(v_{o_t}) \oplus m_o^i\right]$$

where m_o^i is the feature representation of the memory $\mathcal{B}^{\mathcal{M}_i}$, $\oplus$ represents concatenation. If $\mathcal{B}^{\mathcal{M}_i}$ is empty, $m_o^i = \mathbf{0}$.

3 EXPERIMENTS

We evaluate our method on three platforms: 2D Gridworld (Zuo, 2018), 3D Miniworld (Chevalier-Boisvert, 2018) and UR robot reacher. We compare our method against the following baselines:

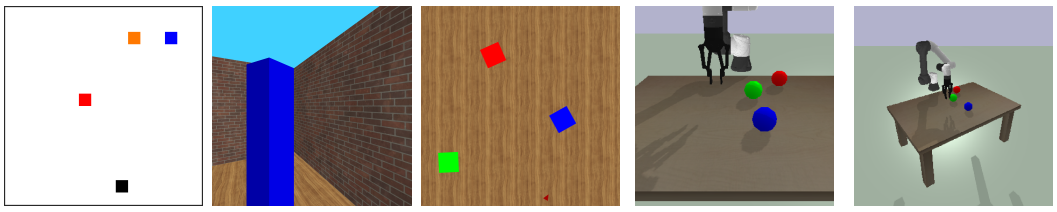
1. *SNAIL*: SNAIL (Mishra et al., 2017) is a meta-learning algorithm that accumulates information across episodes in the same trial as described in Section 2. SNAIL improved over RL^2 (Duan et al., 2016) by replacing RNNs with temporal convolutions and attention [1].
2. *PPO-finetune*: While memory-based methods adapt to a new task by incorporating past episodes, it is also possible to update the policy by fine-tuning. To compare our method against fine-tuning based adaption, we first trained a memoryless base-policy, which only takes as input the current observation. At test time, we fine-tune this policy after every episode using PPO (Schulman et al., 2017) and the reward gathered by the agent. A variant of this baseline (*PPO-finetune + Attention*) includes an attention operation (similar to our method and SNAIL) on the past steps in the same episode such that the policy can use temporal information from the past.

3.1 GRIDWORLD

3.1.1 ENVIRONMENT SETUP

The gridworld is $W \times W$ in size and has $X + 1$ blocks in different colors (Figure 2a). The black block represents the agent. One block is the goal. The other $X - 1$ blocks are traps. The agent gets a

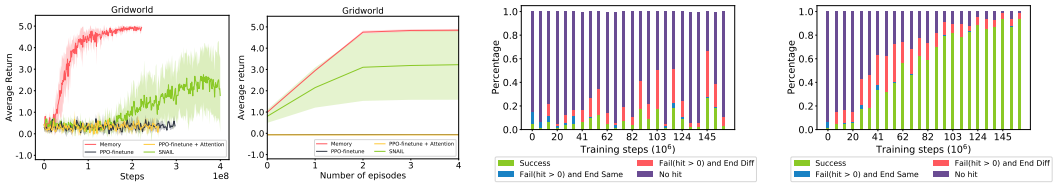
[1] Since Mishra et al. (2017) shows that SNAIL outperforms RL^2, we only compare our method against SNAIL.



(a) (b) (c) (d) (e)

Figure 2: Examples of the 2D Gridworld, 3D Miniworld, UR Robot Reacher environments. **(a)**: the top-down view of an example of 2D Gridworld. **(b)** and **(c)** show the 3D miniworld environment. **(b)**: first-person view (observation input) for the agent. **(c)**: top-down view of the environment, which is not available to the agent. **(d)** and **(e)** show the UR robot reacher environment. **(d)** is an example observation input to the policy. **(e)** shows UR robot reacher setup.

$+5$ reward if it reaches the goal, a -1 reward if it reaches any trap, and a 0 reward otherwise. The color of the goal and the positions of the blocks vary across tasks. The agent is not provided with the color of the goal block, which makes the environment partially-observable. *Observation*: The agent takes as input the rendered top-down view images of the gridworld (Figure 2a). *Action*: The action set is {move left, move right, move up, move down}. All the actions move the agent by one cell. *Training*: We use PPO (Schulman et al., 2017) to train all the policies. The number of steps in a trial is 80 for $X = 3$. We store maximally 8 failed trajectories for each task in the memory bank with $L = 6$. More experiment details are in the Appendix A.



(a) (b) (c) SNAIL (d) Memory

Figure 3: Learning in the Gridworld environment. **(a)**: learning curves in training. Our method (*Memory*) learns significantly faster than *SNAIL*. *PPO-finetune* and *PPO-finetune + Attention* fail to learn to solve the tasks. **(b)**: testing results on new tasks in the Gridworld. **(c)** and **(d)**: testing results (5 episodes) on 200 new tasks (2D gridworld) as the training progresses. *Success*: tasks that are successful. *Fail(hit > 0) and End Diff*: tasks where the agent failed, hit at least one trap within 5 episodes, and ended up in a different state (a different trap or does not hit any block) in each episode. *Fail(hit > 0) and End Same*: tasks where the agent failed and hit the same trap in all 5 episodes. *No hit*: tasks where the agent hit no block in all 5 episodes. Our method quickly learns to avoid hitting the wrong blocks while *SNAIL* oscillates in hitting the wrong blocks, or does not learn to hit the blocks even after 100M training steps.

3.1.2 Evaluation

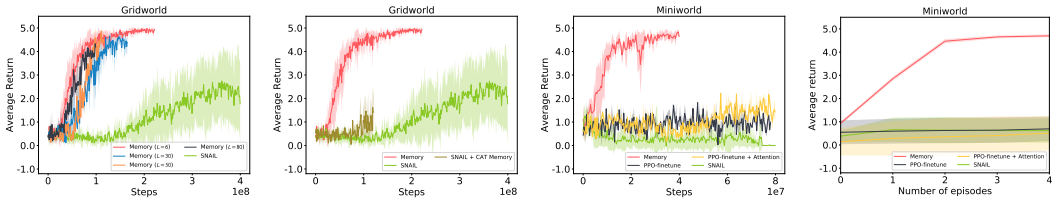
Our testing procedure is as follows: At the beginning of a trial in test time, the memory bank is empty, and the agent has no task information. The agent acts using its policy, and if an episode fails, the last 6 frames of the episode (*i.e.*, $L = 6$) is added to the memory bank. The agent rolls out 5 episodes. *SNAIL* is evaluated in the same way, except that in a manner consistent with the original work, all frames from all previous episodes (and not just failures) in the trial are processed to predict the action. For *PPO-finetune* and *PPO-finetune + Attention*, the policy is finetuned with PPO after each episode.

Figure 3a shows the learning curves in training for different methods. We hypothesize that it is because SNAIL does not remember the experience from previous trials, the agent suffers from making the same mistakes repeatedly, which slows down the learning. To validate our hypothesis, we evaluated the kind of mistakes made by our and SNAIL agents on 200 held-out environments while the agents were training. We categorize the outcomes into 4 cases: agent fails in all 5 episodes and does not hit any block (*No hit*), agent fails and ends up in different state in 5 episodes (*Fail(hit*

> 0) and End Diff*), agent fails and ends up in the same state (*Fail(hit > 0) and End Same*) in all 5 episodes, agent succeeds (*Success*).

Figure 3c and Figure 3d shows the distribution of error modes of *SNAIL* and our method (*Memory*) during training. As shown in Figure 3c, for a significant fraction of tasks the error mode for the SNAIL agent is *Fail(hit > 0) and End Diff*. This is especially salient in the late stage of the training as the agent learns to hit the block, but keeps hitting the wrong block. Because the SNAIL agent ends up getting a negative reward for hitting blocks quite often, we hypothesize it never learns to hit any block resulting in very slow training and poor performance even at the end of the training. However, using memory (Figure 3d) quickly reduces the number of such cases during training as the memory can replay the past failures and help the policy identify the correct goal block much more quickly. This analysis quantifies the intuition developed in the introduction – current SOTA methods suffer from oscillating between failure modes, which our method can overcome.

Figure 3b shows the testing performance on 500 new tasks. Our method (*Memory*) performs considerably better than *SNAIL* and has a lower variance in the testing performance. We also found that policies that take as input only the observations at the current time step (*PPO-finetune*) or the history within the episode (*PPO-finetune + Attention*) are unable to learn the training tasks.



(a) (b) (c) (d)

Figure 4: **(a)**: learning curves of the policies that store failures of different length in the memory. **(b)**: comparing learning curves of our method (*Memory*), *SNAIL*, and *SNAIL* with concatenated memory. **(c)**: learning curves in 3D miniworld **(d)**: testing performance in 3D Miniworld.

3.1.3 Effect of the length L of trajectories in memory

In the previous experiments, only the last 6 time steps of each trajectory are stored in the memory. For the gridworld environments, this is privileged information as observing the last few time steps in the episode is sufficient to infer if the touched block is the goal or the distractor. However, in general, it is not known apriori how many time-steps of an episode should be stored in the memory bank. It is expected that storing longer trajectories can make learning harder. This raises a potential confound in the results – did our method outperform *SNAIL* because it had access to privileged information of storing only the last few frames in an episode, which made the training easier? To test this, we trained different versions of our system, each storing a different number of last L steps of the episode. Since the episode length is 80 steps at maximum, it is guaranteed that the entire trajectory is stored in the memory bank when $L = 80$. Figures 4a show that even if we store a much longer portion of the past trajectories, even the entire trajectory, in the memory bank, our method still learns significantly faster than *SNAIL*. Therefore, our method does not require precise knowledge of how many steps of the failed episode should be stored in the memory.

3.1.4 Benefits of using a hierarchical memory processing module

As described in Section 2.1 and Section 2.2, our method uses a hierarchical memory processing module: first, we use an attention module to extract each trajectory's information into an embedding vector; second, we use another attention module to merge the information from multiple trajectories. Another way to make use of the memory is to concatenate all the trajectories along with the current trial trajectory, and use *SNAIL* to process the entire sequence (*SNAIL + CAT Memory*). While our hierarchical architecture is less computationally expensive (Section B), we further investigate whether using hierarchical architecture is important. Figure 4b shows that applying SNAIL on a concatenated memory (*SNAIL + CAT Memory*) learns much slower than our method (*Memory*) suggesting that the proposed architecture is also crucial. However, note that the architecture by itself is not sufficient to

explain the superior performance of our agents – storing failures in a memory bank is also critical (see Figure 4b).

3.2 3D MINIWORLD

In the 2D gridworld environment, the agent uses a simple action space and has access to the top-down view. We want to test if our method can work in more visually complex environments. For this, we conducted experiments in a 3D navigation environment, with the same task as in the gridworld environment. There are 3 boxes in the room with different colors. Only one of the boxes gives a $+5$ reward, and the other two boxes give a -1 reward. The boxes' positions vary across tasks, and the target box is chosen randomly from the 3 boxes for each task. The agent's action set is {move forward by 0.3m, turn left by $18°$, turn right by $18°$}. The setting resembles the situation where a TurtleBot moves around the room to find the target object. The observation is a first-person view as shown in Figure 2b. The policies are trained on 10000 tasks and tested on 1000 new tasks. Figure 4c shows that our method learns much faster than all the baselines and achieves the highest training performance. *SNAIL* policies quickly get stuck in local optimums: the agent learns to avoid getting close to any box to avoid getting negative rewards. In contrast, our method does not suffer from this issue. From Figure 4d, we can see that our method can successfully identify the correct target box within 3 episodes in most tasks, while the baselines perform much worse. The advantage of using a memory bank is even more salient in this case.

3.3 UR ROBOT REACHER

We also evaluated our method on a simulated manipulator from Universal Robots (UR) in Pybullet (Coumans & Bai, 2017). There are 3 balls on the table, and the robot's task is to touch the target ball (Figure 2e). If the robot touches the true ball, it gets a $+5$ reward. If it touches the other balls, it gets a -1 reward. It gets a 0 reward, otherwise. More details about the setup are described in Appendix A. Figure 5 shows that our method outperforms *SNAIL* by a large margin.

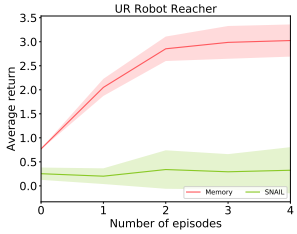


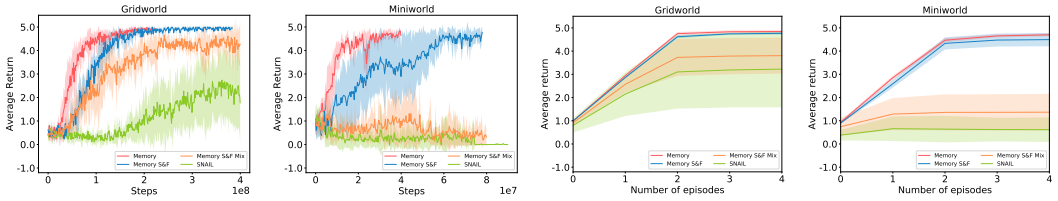
Figure 5: Testing results for UR Robot Reacher tasks

3.4 WHY DO WE ONLY STORE FAILURES?

Since we only store failures in the memory bank, a natural question to ask is that why do we only store failures? To investigate whether we should store successful experiences, we extend our method (*Memory*) to make use of success in the policy. We evaluate two possible ways to add the successful experience: (a) *Memory S&F*: store the successful experience in a separate memory bank $\mathcal{B}_s^{\mathcal{M}_i}$ and use a different memory processing neural module for the failures and successes (see Figure C.2 in Appendix C), (b) *Memory S&F Mix*: use the same architecture as *Memory*, but add both the successes and failures to the same memory bank. We performed the ablation experiments in the 2D Gridworld and 3D Miniworld environments.

Figure 6a and Figure 6b show that *Memory S&F* learns a little bit slower than *Memory*, which is due to the fact that *Memory S&F* has more parameters to be trained. Both *Memory S&F* and *Memory* achieve similar converged performance. Mixing the successful and failed experience in the same memory bank (*Memory S&F Mix*) compromised both the learning speed and final converged performance. As *Memory S&F Mix* still has a memory bank that stores experience, it learns faster than *SNAIL*. In the 3D Miniworld environment (Figure 6b), both *Memory S&F Mix* and *SNAIL* fail to learn to solve the tasks. Figure 6c and Figure 6d show that *Memory* and *Memory S&F* have similar testing performance, while *Memory S&F Mix* achieves lower testing performance and *SNAIL* performs worse than any variant of our method. Therefore, in these environments, it is sufficient only to store the failed experiences. There could be environments where storing successful experiences is critical, which is orthogonal and complementary to our paper.

In our experiments, we also found that if we add successful experiences into the memory bank (*Memory S&F* and *Memory S&F Mix*), it is important to clear the memory bank for the successful experiences once in a while so that the policy does not overfit to the successful experiences after the policy learns how to solve the tasks. We found that clearing the memory bank for the successes when the memory bank is full is sufficient to prevent the overfitting.



(a) (b) (c) (d)

Figure 6: Ablation on adding successful experiences. (a) and (b) show the learning curves during training for the 2D Gridworld and 3D Miniworld environments. (c) and (d) show the testing performance of the trained policies.

4 Related Work

External Memory and Learning: Integrating memory and learning is an active area of research in deep learning. Graves et al. (2014) combined RNN with differentiable memory bank to solve problems such as copying and sorting. Oh et al. (2016) used the idea of differentiable memory along with Q-learning to solve navigation tasks. Explicitly using a memory bank has been found useful in solving partially observable tasks with long-term dependencies (Parisotto & Salakhutdinov, 2017; Wayne et al., 2018) and few-shot image recognition (Santoro et al., 2016). Recent works (Fortunato et al., 2019; Hung et al., 2019) found that performance of memory modules can be improved by using auxiliary losses such as the contrastive prediction coding (CPC). These losses learn better feature representations of states stored in the memory. In a separate line of work, (Lengyel & Dayan, 2008; Blundell et al., 2016; Pritzel et al., 2017) used episodic memory to improve data efficiency of RL algorithms. While we do not propose a novel memory module, unlike prior work, we use it for learning across trials. Ritter et al. (2018) used episodic memory to augment the working memory, LSTM cell states in an episode are stored in the episodic memory and retrieved later. The stored states are less consistent after policy optimization (He et al., 2020). To avoid the issue of storing stale representation, we store the raw trajectories in the memory banks. The policy can, therefore, makes use of much earlier experience.

Meta Learning: Meta learning aims to learn new concepts or skills with a few examples in the new environments (Thrun & Pratt, 1998; Vilalta & Drissi, 2002). Gradient-based meta learning uses gradient descent to quickly adjust the network weights for adapting to the new tasks (Finn et al., 2017; Nichol & Schulman, 2018; Beaulieu et al., 2020). Metric-based meta learning learns a metric space in which the testing sample is compared to the training samples for the prediction (Koch et al., 2015; Vinyals et al., 2016). RNN-based meta learning (Duan et al., 2016; Wang et al., 2016) encodes the experience in RNN hidden states for fast adaptation in new tasks. Mishra et al. (2017) show that replacing RNN with temporal convolutions and soft attention leads to better performance in long-horizon tasks. However, these methods do not use the information of earlier trials. We show that incorporating the information from multiple trials via a memory bank significantly improve the learning both in terms of speed and test-time performance. (Rakelly et al., 2019) learned a probabilistic task context from unordered state transitions sampled from recently collected data in off-policy RL settings and showed improvement in data efficiency compared to (Finn et al., 2017; Duan et al., 2016). We show that using the old failure experience speeds up the learning.

5 Discussion

In this paper, we apply the idea of replaying failures to the meta-learning setting where an agent learns to avoid making the mistakes that it has made in the past. We show that having a memory bank of the experience across multiple previous trials and explicitly replaying the past failures can significantly improve learning speed as well as the test-time performance.

We use a simple fixed-capacity memory bank in this work. While our methods have shown good performance on all the tasks, there could be duplicate or similar trajectories in the memory, which leads to inefficiency of the memory usage. While we emphasize the importance of using failures from the past, successful experiences could be important for certain tasks as well. In the future, we would like to explore and exploit the semantics of the trajectories, improve memory efficiency, and investigate the benefits of using successes on other tasks.

REFERENCES

Shawn Beaulieu, Lapo Frati, Thomas Miconi, Joel Lehman, Kenneth O Stanley, Jeff Clune, and Nick Cheney. Learning to continually learn. *arXiv preprint arXiv:2002.09571*, 2020. 8

Charles Blundell, Benigno Uria, Alexander Pritzel, Yazhe Li, Avraham Ruderman, Joel Z Leibo, Jack Rae, Daan Wierstra, and Demis Hassabis. Model-free episodic control. *arXiv preprint arXiv:1606.04460*, 2016. 8

Maxime Chevalier-Boisvert. gym-miniworld environment for openai gym. https://github.com/maximecb/gym-miniworld, 2018. 4

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. 4

Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation in robotics, games and machine learning, 2017. 7

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016. 1, 2, 3, 4, 8

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymir Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. *arXiv preprint arXiv:1802.01561*, 2018. 12

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1126–1135. JMLR. org, 2017. 8

Meire Fortunato, Melissa Tan, Ryan Faulkner, Steven Hansen, Adrià Puigdomènech Badia, Gavin Buttimore, Charles Deck, Joel Z Leibo, and Charles Blundell. Generalization of reinforcement learners with working and episodic memory. In *Advances in Neural Information Processing Systems*, pp. 12448–12457, 2019. 8

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014. 8

Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9729–9738, 2020. 8

Chia-Chun Hung, Timothy Lillicrap, Josh Abramson, Yan Wu, Mehdi Mirza, Federico Carnevale, Arun Ahuja, and Greg Wayne. Optimizing agent behavior over long time scales by transporting value. *Nature Communications*, 10(1):5223, Nov 2019. ISSN 2041-1723. doi: 10.1038/s41467-019-13073-w. URL https://doi.org/10.1038/s41467-019-13073-w. 8

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 12

Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2. Lille, 2015. 8

Máté Lengyel and Peter Dayan. Hippocampal contributions to control: the third way. In *Advances in neural information processing systems*, pp. 889–896, 2008. 8

Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. 1

Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. *arXiv preprint arXiv:1707.03141*, 2017. 1, 2, 3, 4, 8, 13

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. 1

Alex Nichol and John Schulman. Reptile: a scalable metalearning algorithm. *arXiv preprint arXiv:1803.02999*, 2:2, 2018. 8

Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, and Honglak Lee. Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128*, 2016. 8

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*, 2019. 3

Emilio Parisotto and Ruslan Salakhutdinov. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*, 2017. 8

Emilio Parisotto, H Francis Song, Jack W Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant M Jayakumar, Max Jaderberg, Raphael Lopez Kaufman, Aidan Clark, Seb Noury, et al. Stabilizing transformers for reinforcement learning. *arXiv preprint arXiv:1910.06764*, 2019. 4

Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adria Puigdomenech Badia, Oriol Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 2827–2836. JMLR. org, 2017. 8

Kate Rakelly, Aurick Zhou, Chelsea Finn, Sergey Levine, and Deirdre Quillen. Efficient off-policy meta-reinforcement learning via probabilistic context variables. In *International conference on machine learning*, pp. 5331–5340, 2019. 8

Samuel Ritter, Jane X Wang, Zeb Kurth-Nelson, Siddhant M Jayakumar, Charles Blundell, Razvan Pascanu, and Matthew Botvinick. Been there, done that: Meta-learning with episodic recall. *arXiv preprint arXiv:1805.09692*, 2018. 8

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pp. 1842–1850, 2016. 8

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 4, 5

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017. 1

Sebastian Thrun and Lorien Pratt. Learning to learn: Introduction and overview. In *Learning to learn*, pp. 3–17. Springer, 1998. 8

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017. 3, 4

Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial intelligence review*, 18(2):77–95, 2002. 8

Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *Advances in neural information processing systems*, pp. 3630–3638, 2016. 8

Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016. 8

Greg Wayne, Chia-Chun Hung, David Amos, Mehdi Mirza, Arun Ahuja, Agnieszka Grabska-Barwinska, Jack Rae, Piotr Mirowski, Joel Z Leibo, Adam Santoro, et al. Unsupervised predictive memory in a goal-directed agent. *arXiv preprint arXiv:1803.10760*, 2018. 8

Xingdong Zuo. mazelab: A customizable framework to create maze and gridworld environments. https://github.com/zuoxingdong/mazelab, 2018. 4

APPENDIX A TRAINING SETUP

We use the same network architecture for all environments. The convolutional part is an IMPALA-CNN (Espeholt et al., 2018). The shape of the observation input is $64 \times 64 \times 3$. We use 48 processes in parallel for policy rollouts. The discount factor for the reward is 0.999. The entropy coefficient in PPO is 0.004. We use Adam optimizer (Kingma & Ba, 2014) for the training, and the learning rate is initialized at 0.00025. Batch size is 12. We use 2 encoder layers and 2 decoder layers in the Transformer with embedding dimension $E = 64$. The multi-head attention uses 4 heads. The policy is parameterized by a categorical distribution. In the test time, actions with the highest probability are taken.

Gridworld: The agent succeeds if it reaches the true goal cell. The episode ends when the maximum number of episode steps is reached, or the agent reaches the traps or the goal. The total number of possible levels is around $X(X+1)!\binom{W^2}{X+1}$ [1]. An action will not be executed if the agent would cross the boundaries after taking the action. In our experiments, $W = 16, X = 3$. Since the agent cannot distinguish the goal from the traps just based on observations, even an expert agent has to fail $X - 1$ times before reaching the real goal in the worst case. The trial horizon (also the maximum number of episode steps) is 80. The memory bank stores 8 trajectories at most for each level. The trajectory length L is 6. We use 40000 training levels to avoid policy over-fitting. The average episode length of a well-trained policy is around 10 steps. We test the policies on 500 held-out new tasks. For each task, we test the models for 5 episodes.

3D Miniworld: The room size is $4\text{m} \times 4\text{m}$. The trial horizon is 80. The memory bank stores 8 trajectories at most for each level. The trajectory length L is 6. We train all the policies on 10000 levels and test them on 1000 new levels. Our methods learn significantly faster and better than baselines. This task is much harder than the 2D gridworld task as the agent only has access to the first-person view instead of the top-down view of the environment. The agent moves like a mobile robot with a differential drive. The search space is much bigger as the agent's action space includes the heading of the agent. The average episode length of a well-trained policy is around 15 steps.

UR Robot Reacher: There are 3 balls on the table, and the robot's task is to touch the target ball. The color of the target ball and the locations of the balls vary across tasks. The xy locations of the balls are randomly sampled within a $0.5\text{m} \times 0.6\text{m}$ region. Like the other two environments, the robot does not know which ball is the target ball before its interaction with the balls. If the robot touches the true ball, it gets a $+5$ reward. If it touches the other balls, it gets a -1 reward. It gets 0 reward, otherwise. The policy takes as input the RGB images (Figure 2d) from a camera that looks at the table. We do not include state information such as robot's joint positions and ball positions in the policy input. The robot takes an action $a_t = (\Delta x, \Delta y, \Delta z)$ at each time step, where $\Delta x \in \mathcal{X}, \Delta y \in \mathcal{X}, \Delta z \in \mathcal{X}$, and $\mathcal{X} = \{-0.04, 0, 0.04\}$cm. The policy outputs a categorical distribution on the action selection, which has a dimension of $3^3 = 27$. The trial horizon is 80. The memory bank stores 8 trajectories at most for each level. The trajectory length L is 6. The policy is trained on 1000 tasks and tested on 200 new tasks. The learning curves are shown in Figure A.1.

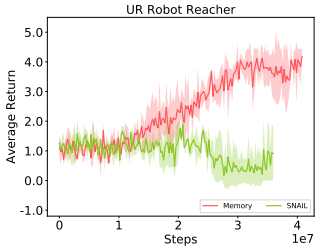


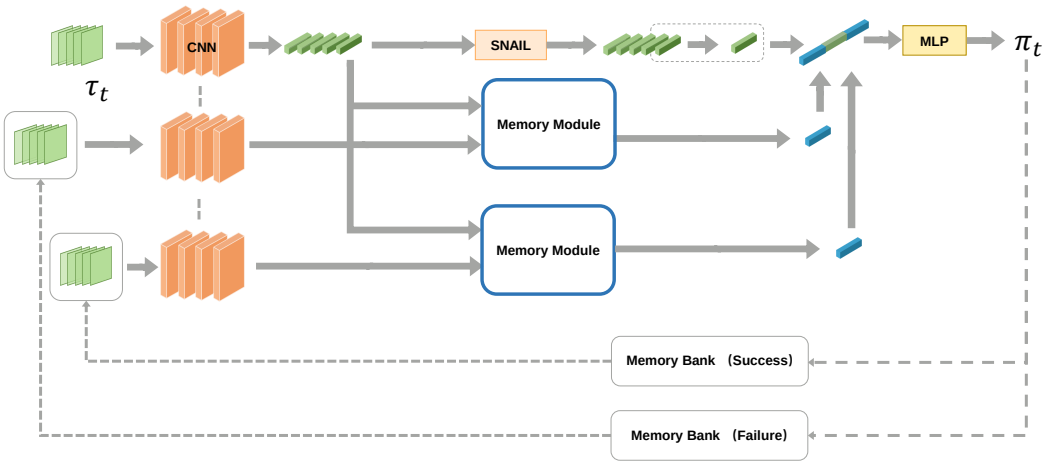
Figure A.1: Learning curves for UR Robot Reacher tasks. Our method (*Memory*) learns much faster than *SNAIL*.

[1] This is a rough estimation as this formula does not exclude the cases where there is no feasible path from the agent's start position to the goal position. The number of such cases is small when $X \ll W^2$.

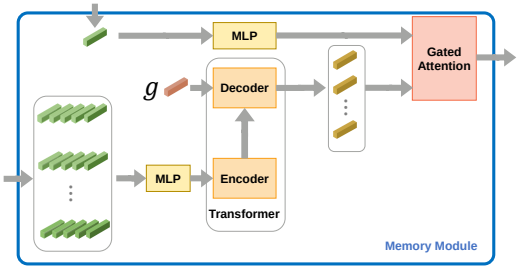
APPENDIX B TIME COMPLEXITY OF HIERARCHICAL TRAJECTORY EMBEDDING

Assume a sequence length is T, and the embedding dimension is d, then the time complexity of a self-attention layer is $\mathcal{O}(T^2d)$. If there are N such trajectories, processing trajectories with two-level attention (Section 2.1, 2.2) has a time complexity of $\mathcal{O}(NT^2d + N^2d)$. However, if the attention operation is over all the frames in N trajectories (Mishra et al., 2017), the time complexity becomes $\mathcal{O}(N^2T^2d)$. the former one typically requires much less computation.

APPENDIX C NETWORK ARCHITECTURE WITH BOTH SUCCESSFUL AND FAILED EXPERIENCE



(a) Policy architecture with both successful and failed experience.



(b) Memory module.

Figure C.2: Policy architecture with both successful and failed experience. **(a)** is the policy architecture. **(b)** is the memory module that is used in **(a)**.