SELF-ACTIVATING NEURAL ENSEMBLES FOR CONTINUAL REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

The ability of an agent to continuously learn new skills without catastrophically forgetting existing knowledge is of critical importance for the development of generally intelligent agents. Most methods devised to address this problem depend heavily on well-defined task boundaries which simplify the problem considerably. Our task-agnostic method, Self-Activating Neural Ensembles (SANE), uses a hierarchical modular architecture designed to avoid catastrophic forgetting without making any such assumptions. At each timestep a path through the SANE tree is activated; during training only activated nodes are updated, ensuring that unused nodes do not undergo catastrophic forgetting. Additionally, new nodes are created as needed, allowing the system to leverage and retain old skills while growing and learning new ones. We demonstrate our approach on MNIST and a set of grid world environments, demonstrating that SANE does not undergo catastrophic forgetting where existing methods do.

1 INTRODUCTION

Lifelong learning is of critical importance for the field of robotics; an agent that interacts with the world should be able to continuously learn from it, acting intelligently in a wide variety of situations. In marked contrast to this ideal, most standard deep reinforcement learning methods are centered around a single task. First, a task is defined, then a policy is learned to maximize the rewards the agent receives in that setting. If the task is changed, a completely new model is learned, throwing away the previous model and previous interactions. Task specification therefore plays a central role in current end-to-end deep reinforcement learning frameworks. But is task-driven learning scalable?

In contrast, humans do not require concrete task boundaries to be able to effectively learn separate tasks – instead, we perform continual (lifelong) learning. The same model is used to learn new skills, leveraging the lessons of previous skills to learn more efficiently, without forgetting old behaviors. However, when placed into continual learning settings, current deep reinforcement learning approaches do neither: the transfer properties of these systems are negligible and they suffer from catastrophic forgetting (McCloskey & Cohen, 1989; French, 2006).

The core issue of catastrophic forgetting is that a neural network trained on one task starts to forget what it knows when trained on a second task, and this issue only becomes exacerbated as more tasks are added. The problem ultimately stems from training one network end-to-end sequentially; the shared nature of the weights and the backpropagation used to update them mean that later tasks overwrite earlier ones (McCloskey & Cohen, 1989; Ratcliff, 1990).

To handle this, past approaches have attempted a wide variety of ideas: from task-based regularization (Kirkpatrick et al., 2017), to learning different sub-modules for different tasks (Rusu et al., 2016), to dual-system slow/fast learners inspired by the human hippocampus (Schwarz et al., 2018). The core problem of continual learning, which none of these methods address, is that the agent needs to autonomously determine how and when to adapt to changing environments, as it is infeasible for a human to indefinitely provide an agent with task-boundary supervision. Specifically, these approaches rely on the notion of tasks to identify when to spawn new sub-modules, when to freeze weights, when to save parameters, etc. Leaning on task boundaries is unscalable and side-steps the core problem. While recent experience-based approaches such as Rolnick et al. (2018) forgo explicit task IDs, they still maintain explicit experience replay of all the tasks to avoid overfitting. However, large replay buffers are unscalable with an ever-increasing number of tasks.

Instead, our method approaches this problem head-on by dynamically adapting to changing environments. Our task-agnostic hierarchical method, Self-Activating Neural Ensembles (SANE), depicted in Figure 1, is the core of the method. Every node in the tree is a separate module, none task-specific. At each timestep a single path through the tree is activated to determine the action to take. Only activated nodes are updated, leaving unused modules unchanged and therefore immune to catastrophic forgetting. Critically, our tree is dynamic; new nodes are created when existing nodes are found to be insufficient. This allows for the creation of new paths through the tree for novel scenarios, preventing destructive parameter updates to the other nodes.

SANE provides the following desirable properties for continual reinforcement learning: (a) It mitigates catastrophic forgetting by only updating relevant modules; (b) Because of its task-agnostic nature, unlike previous approaches, it does not require explicit supervision with task IDs; (c) It achieves these targets with bounded resources and computation. In our results, we demonstrate the ability of SANE to learn and retain MNIST digits when presented fully sequentially, a challenging task our baselines struggle with. We also demonstrate SANE on a series of three grid world tasks, showing that SANE works in the reinforcement learning setting.

2 MODEL

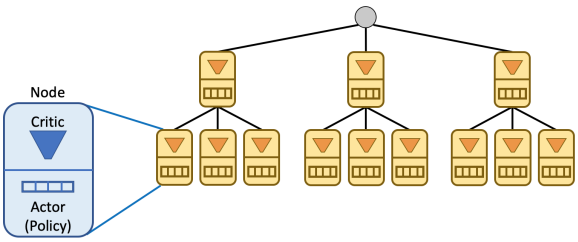

Figure 1: The overall tree structure of the SANE system, here with 2 layers of nodes. Each node contains an actor and a critic.

The structure of our Self-Activating Neural Ensembles (SANE) system is a tree, as shown in Figure 1. Each node in the tree contains both an actor and a critic, described in 2.1. At each timestep we activate a path through the tree from the root to a leaf, at each layer selecting the node whose critic estimates the highest *activation score* (described in 2.2). We then sample an action from that leaf node's policy to execute in the environment. During training, the activated nodes are the only ones updated; all other nodes are unchanged. This enables behaviors selected for by other paths through the tree to remain unchanged.

The tree structure is critical to the success of the system; a node's children can be seen as partitioning the input space that activates the node, specializing their policies to these subspaces. When they differ sufficiently from their parent in performance, they are promoted to a higher level of the tree and obtain children of their own, further partitioning the space. Nodes are constantly being created, promoted to higher levels, or merged as the situation demands (described in Section 2.4).

Static vs. Dynamic Trees: One critical question to ask is why use a dynamic structure and not just a huge static tree with only parameter updates? In the case of the static tree, if a path through the network is beneficial for one task, it is likely to be chosen as a starting point for the subsequent task, and updated in-place for this new setting. An example of this is in Jacobs et al. (1991), where specialized losses are necessary to distribute activation across the ensemble of experts, even in the case of a stationary distribution. With SANE, the change of environment would be detected and a new path created, thus minimizing disruption of the previously beneficial behaviors.

2.1 NODE CONTENTS

Every node i contains the following: a policy $\boldsymbol{p}_i$, a critic $\mathcal{C}_i$, a replay buffer containing a list of (state s, discounted reward r_{disc}), and a count of the number of times this node has been used (usage count).

The policy $\boldsymbol{p}_i$ presented here is a vector of n logits, where n is the number of actions available to the agent. The critic $\mathcal{C}_i$ is a neural net mapping a state s to two scalars: the value estimate of the

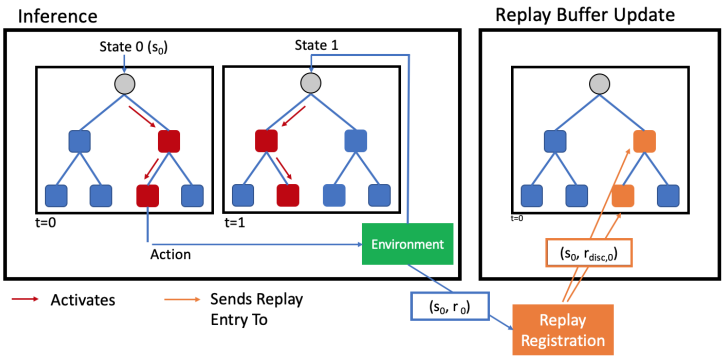

Figure 2: On the left is an example of inference, which results in the generation of actions to execute in the environment. On the right is the associated adding of replay entries to the nodes that activated at $t = 0$, after the collection phase.

discounted reward received if the current node is activated, $v_i(s)$, and an estimate of the absolute error:

$$u_i(s) \approx |r_{disc} - v_i(s)| \tag{1}$$

With these we can define an optimistic estimate (upper confidence bound) and a pessimistic estimate (lower confidence bound) for what reward a node can achieve:

$$v_{UCB,i}(s) = v_i(s) + \alpha * u_i(s) \tag{2}$$

$$v_{LCB,i}(s) = v_i(s) - \alpha * u_i(s) \tag{3}$$

α is a hyper-parameter representing how wide a margin around the expected value we allow. v_{UCB} is analogous to an upper confidence bound in multi-armed bandit literature: as training proceeds and v_i becomes more accurate, u_i will decrease, providing a natural transition from exploration to exploitation for each node as it learns.

Each node's replay buffer contains the history of experiences per node and is used to train the node's critic. The usage count is the number of times a node has been activated and is used in the merging of nodes. These are described more in sections 2.3 and 2.4.

2.2 Inference

For each observed state s_t from the environment, we activate a path through the tree from the root to a leaf node, whose policy is used to determine the next action to take in the environment, a_t. The environment then responds with a reward r_t and a new observation s_{t+1} (See Figure 2).

From an activated node, the next node to activate is selected from its children by taking the node with the highest optimistic upper bound of reward. This process continues until a leaf node is reached. After the set of activated nodes is determined, an action is selected stochastically from the activated leaf's policy, and executed in the environment. New nodes are not created mid-episode to keep the tree idempotent, which allows us to readily run environments in parallel.

During inference, three things happen at each timestep to support updating the tree. All active nodes:

- are registered to receive the replay entry for that timestep after the episode completes.
- compare their critic's estimate to their parent's. If they differ sufficiently (Section 2.4), the child will be promoted during the next tree structure update.
- increment their usage count.

2.3 Parameter Updates

SANE training updates consist of two steps: parameter updates (replay buffer, policy, and critic) and tree structure updates (creating new nodes, merging nodes, etc.). After T steps of data collection have occurred, we trigger both a parameter update and a tree structure update.

Replay Buffer Update: Each node's replay buffer contains the history of the times that node was activated, and is used to train the node's critic. Each entry in the replay buffer contains an input

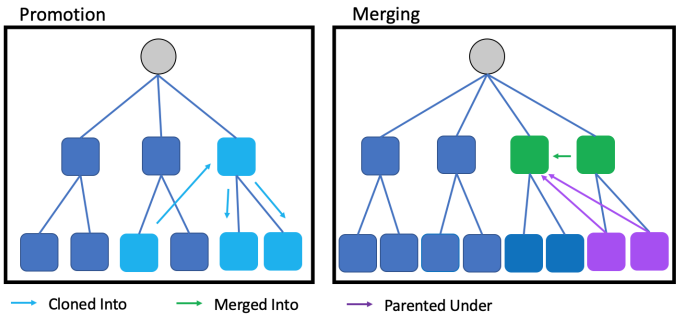

Figure 3: The two parts of updating the structure of the tree: promoting nodes to a higher level, via cloning, and the merging of nodes together when the maximum size has been exceeded.

state s and a discounted reward received r_{disc}, which is also used in the training of the policies. The discounted reward is computed in the standard fashion, as in Espeholt et al. (2018).

When the addition of new entries results in the replay buffer exceeding its max length, instead of removing the oldest entries, random entries are removed instead. This is to mitigate each critic from developing a recency bias.

Policy Update: The policy for each node is trained on-line using REINFORCE (Williams, 1992) with Monte Carlo returns.

Training the Critics: The critic for each node is trained off-line using the replay buffer. More specifically, we split a node's replay buffer into batches each of size k. The critic has two functions to minimize: $v(s)$ (the expected value) and $u(s)$ (the uncertainty as defined in equation 1). We use L2 loss for both reward and error prediction. Note that this is the slowest step in the training of the SANE tree. Since the nodes all use separate critics, this step can be effectively parallelized. We only train a critic when its node was activated at least once since the last time nodes were trained.

2.4 Structure Updates

One of the critical properties of SANE is that the tree structure updates dynamically. Promoting nodes enables existing behaviors to stay unchanged, and merging nodes allows us to keep memory and computation bounded, both depicted in Figure 3. The term promotion is used to refer to cloning a node and placing it in the tree one level up from the original. Merging is the process of combining nodes and is necessary to cap the computational and memory costs that scale with tree size.

Node Promotion: Drift is the term for when an environment is non-stationary, e.g. when the reward distribution or the state transition distribution are changing over time. Often where drift occurs, catastrophic forgetting follows because networks update to the new setting, forgetting the old.

To recognize drift, non-leaf nodes at the time of creation have their critic cloned and frozen, creating a static critic called the *anchor*. We compare the prediction of a node's critic to the prediction its parent's anchor makes. We say that sufficient change has occurred when the reward bounds predicted by a node's critic do not include the value predicted by its parent's anchor, which functions as a static baseline. In such a scenario, the node should be promoted up a level of the tree, because we wish to capture the new situation being observed, and minimize disruption of previously learned behavior (catastrophic forgetting).

Concretely, we say that sufficient change has occurred when either of the following inequalities hold, for a given input state s, node n, and parent's anchor a: $v_{UCB,n}(s) < v_a(s)$, $v_{LCB,n}(s) > v_a(s)$. In order to avoid an extraneous number of promotions for outliers, a leaf node must have been used more than k_m times since it was created or last promoted.

Merging Nodes: To prevent unlimited memory consumption and increasingly long activation times, we limit the number of children each node can have. The limit is defined per level of the tree: level k has limit ℓ_k. We start by finding the two entries in the set that are closest in policy space. To do this we convert the policies from logits to probabilities, then find the two policies that have the smallest L2 distance. The usage counts, policies, and replay buffers are combined by doing a weighted

average of the nodes' contents by usage count. The critic for the new node is trained using the merged replay buffer, after being initialized from the critic of the selected node with higher usage.

Non-leaf nodes additionally update the anchor by combining the nodes' anchors' replay buffers and retraining once. Merging nodes is the exception to the general rule of never updating the anchors.

Initialization: The first node in the tree is initialized as a child of the root and has a random critic and a uniform policy. That node is cloned twice, which become the first node's children. This continues until the tree has reached the pre-specified maximum height. From now on, the height of the tree remains fixed.

To maintain this, whenever a node is promoted, that node is recursively cloned to generate children until the maximum height is re-achieved.

3 EXPERIMENTS

We present two types of experiments that demonstrate the behavior of SANE. The first trains on the digits of MNIST sequentially, to show recall under complete domain shift. The second learns a series of grid world environments, to show that the method works in a reinforcement learning setting.

In these experiments we only consider two-layer SANE trees, with their sizes ($[\ell_0, \ell_1]$) indicating respectively the maximum number of nodes at the first layer, and the maximum number of children each of those may have. A [20, 4] tree therefore has 20 top layer nodes and 80 second layer nodes, for a total of 100 nodes. The experiments are all run over 4 seeds, and the error bars represent the standard error of the mean.

In both cases we use IMPALA (Espeholt et al., 2018) as the non-continual learning baseline, to demonstrate the effects of catastrophic forgetting, and CLEAR (Rolnick et al., 2018) as the continual learning baseline. CLEAR is a variant of IMPALA that adds an extensive replay buffer to maintain experiences from old tasks. In particular, we selected CLEAR because it is one of the few other methods that does not require task IDs as input. We chose for CLEAR the same number of entries in its replay buffer as the upper bound of the number SANE uses, across all nodes: 409,600 frames. All network and implementation details are given in the Appendix.

As SANE uses one network per critic, whereas IMPALA and CLEAR each use only one network total, we can see that there are two versions of each of our baselines: one where they all use the same base network architecture, and one where IMPALA and CLEAR both use significantly larger networks, such that the total number of parameters is approximately equal to the number across all nodes of the SANE tree. For MNIST we increase the network size of IMPALA and CLEAR by a factor of 75 (indicated as "75x" in the graphs), and for MiniGrid by a factor of 40 ("40x" in the graphs), for approximate parity with the number of nodes at steady state in the [20, 4] and [10, 4] trees respectively.

Environments/Datasets: We conduct two experiments: MNIST and MiniGrid. MNIST allows us to test network behavior in the presence of huge domain shifts as well as a relatively large number of tasks, to highlight how approaches like CLEAR struggle in these scenarios. MiniGrid allows us to demonstrate how SANE avoids forgetting in the reinforcement learning setting. Note that while previous efforts have used Atari, we do not use Atari for a couple of reasons: (a) this setting is particularly ill-suited for continual learning per the desiderata given in (Farquhar & Gal, 2018), for instance as there is little cross-task resemblance, task IDs can be readily inferred (b) the required compute is high (a bottleneck in new model development).

3.1 MNIST EXPERIMENTS

MNIST (LeCun et al., 2010) is a dataset of the digits 0-9, each represented by a grayscale 28x28 image. Demonstrating continual learning with MNIST has an extensive history; standard baselines such as permuted MNIST (Kirkpatrick et al., 2017; Schwarz et al., 2018; Goodfellow et al., 2015) train on the full set of digits concurrently, adding significant pixel-level shifts to change the appearance between tasks. However, while the domain may shift dramatically, the distribution over actions does not. We introduce sequential MNIST, an alternative task in which digits are trained completely sequentially: first the 0s, then the 1s, etc. Aside from being a sequence of tasks with a fundamentally different distribution over desirable outputs, we believe that this setting is significantly more

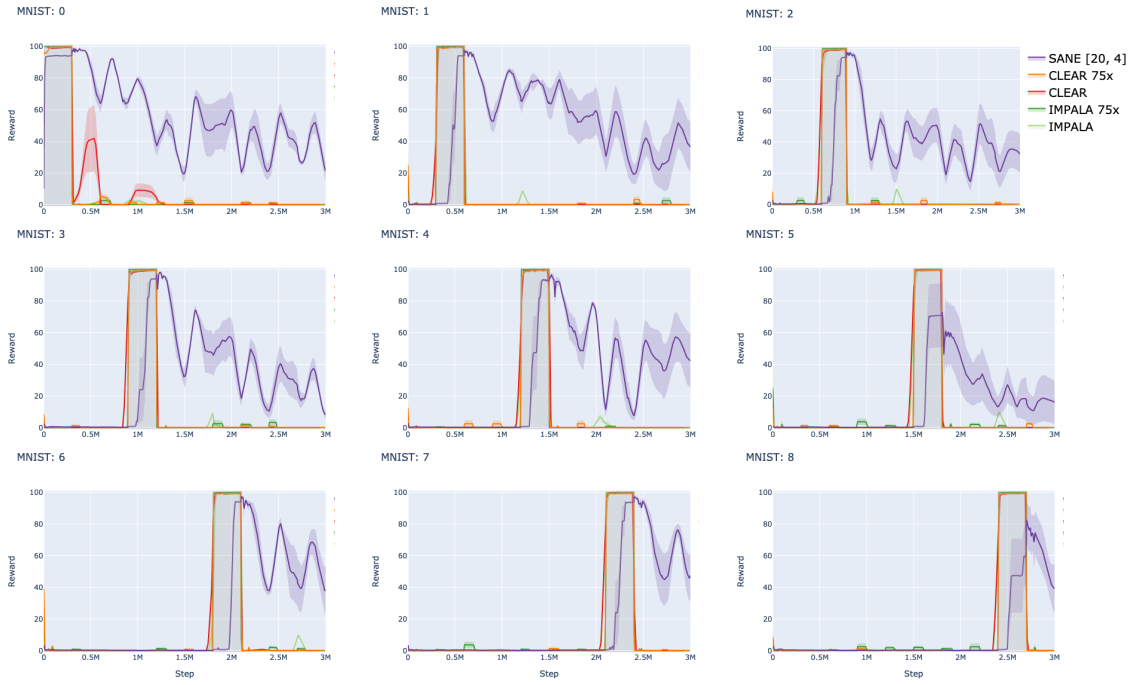

Figure 4: Sequential MNIST results: Each graph visualizes all networks on the given digit. The x-axis is consistent across graphs. Training on a newly introduced digit occurs in the shaded regions, and evaluation to test recall happens everywhere else. SANE is the only method to not catastrophically forget. Digit 9 can be found in the Appendix.

Method	0	0-1	0-2	0-3	0-4	0-5	0-6	0-7	0-8	0-9
SANE	97.5	**82.2**	**75.0**	**57.3**	**47.1**	**50.6**	**29.9**	**26.8**	**28.7**	**33.0**
IMPALA	**100.0**	53.2	32.0	24.8	19.2	14.9	14.0	12.9	11.0	9.7
IMPALA 100x	**100.0**	53.3	31.9	25.1	19.0	15.4	14.0	13.0	10.9	9.9
CLEAR	**100.0**	64.8	32.0	24.7	19.1	15.1	14.0	13.0	10.8	10.1
CLEAR 100x	**100.0**	53.4	32.0	24.8	19.0	15.2	14.0	12.9	10.7	10.0
Mem Last	100	50	33.3	25	20	16.7	14.3	12.5	11.1	10

Table 1: The scores of the methods on the MNIST Test dataset, on all digits trained so far, e.g. column "0-3" occurs after training on digit 3 and tests cumulative recall on all four digits to that point. "Mem Last" is the score achieved if the answer given is always the last number seen.

realistic. We train for 300,000 frames per digit. No normalization is done other than scaling pixel intensities to between [0, 1].

The results are presented in Figure 4. The networks are trained on the given digit during the timesteps indicated by the shaded region. In the non-shaded regions of the plots, the networks are evaluated on that digit alone, to analyze recall. As this figure visualizes recall and not generalization, these evaluation steps are done on the MNIST Train dataset.

Additionally we test the methods by evaluating them on the Test dataset for the digits cumulative to the most recent digit seen during training. These results are presented in Table 1. "Mem Last" is provided as a baseline for what the expected results would be if the network always identifies the last-seen digit.

Only the SANE method consistently recalls digits after their training period, not experiencing catastrophic forgetting. This setting is particularly challenging for typical methods, because in each training regime there is always only one correct action to take, and it is easy to overfit to providing the singular desired answer. It is worth noting that SANE did not perform flawlessly; there are a few digits (5, 8, 9) that SANE learns inconsistently in the allotted number of frames.

There are a few specific behaviors to note. (a) CLEAR's added replay buffer does provide benefit for recalling "0" during the training of future digits, though the benefit quickly tapers off. (b) SANE's

curves are reliably cyclic: when SANE begins to do well on a digit, its performance on prior digits decreases. However, when the next new digit is presented, the performance on prior digits increases again. The network is overfitting while SANE masters a digit, then the negative examples from the start of training the next digit compensate for the previous overfitting, giving us some reverse transfer. (c) In the cumulative test experiment, SANE not only consistently produces the highest scores, but the results are not monotonically decreasing. (d) Increasing the size of the baseline networks did not improve performance.

3.2 MINIGRID EXPERIMENTS

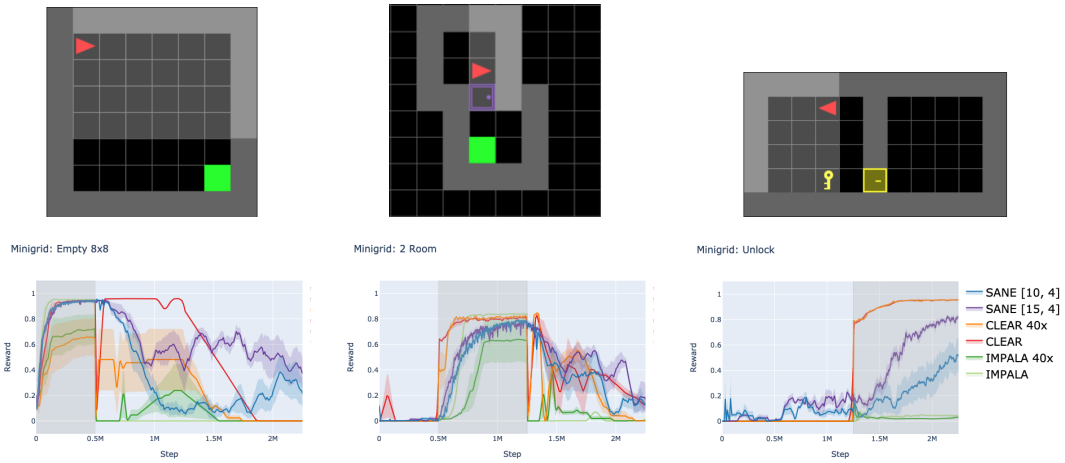

Figure 5: Results for the MiniGrid experiments, in the order in which they were trained. Training occurs in the shaded region, and the rest of the graph depicts recall while the other environments are trained.

MiniGrid (Chevalier-Boisvert et al., 2018) is a lightweight grid world library designed to procedurally generate environments. We train for 500,000 frames on the Empty environment, followed by 750,000 frames on the 2-Room environment, and finally 1 million frames on the Unlock environment, all depicted in Figure 5. The Empty environment is always initialized the same way, whereas 2-Room and Unlock are randomized at the completion of each episode. The observation is an egocentric representation of the environment, capturing state information about the 7x7 square in front of the agent.

The training results are presented in Figure 5. As with MNIST, the shaded region indicates when that environment was being trained, while the unshaded parts are evaluation steps on that environment, to visualize recall.

The Empty8x8 graph is the most insightful: CLEAR has almost perfect recall of this environment while 2-Room is training, but catastrophically forgets it during the training of Unlock. In contrast, both SANE trees do see a decrease in performance while 2-Room is training, but the larger [15, 4] tree in particular does not exhibit catastrophic forgetting, exhibited by its continued positive reward during recall. We can also see the benefit that larger SANE trees can provide; it is likely the smaller one ran out of capacity. It suggests that as we continue to scale SANE up, we may see increases in performance.

The 2-Room training curve simply shows that both SANE methods and both CLEAR methods are performing about the same in this setting, with IMPALA catastrophically forgetting as expected. The Unlock graph shows that the larger SANE tree learns more sample efficiently than its smaller sibling, though not as rapidly as CLEAR. Interestingly, CLEAR learns readily while IMPALA does not. Again, as with MNIST, we see that increasing the size of the baseline networks provides no benefit.

Finally, we provide some intuition for the behavior of the SANE network by visualizing a few states that activate the same node in Figure 6, located in the Appendix.

4 RELATED WORKS

The goal of continual learning, or lifelong learning, is to create an agent that can learn to accomplish a wide variety of tasks sequentially (Lesort et al., 2019). To accomplish this a system must balance stability (the extent to which existing knowledge is retained) and plasticity (how readily new knowledge is acquired) (Parisi et al., 2018).

The former in particular has posed a substantial problem for neural networks, which undergo catastrophic forgetting, tending to completely forget old tasks when presented with new ones, often quite abruptly (French, 2006). Attempts to correct for this tend to fall into these broad categories, which we discuss below: dynamic networks, dual memory, regularization, rehearsal, sparse-coding, and ensembles (Parisi et al., 2018; Kemker et al., 2017). Each direction has received a significant amount of work, and our overview here will be non-exhaustive. In particular we largely focus here on methods that have been demonstrated on reinforcement learning environments.

Dynamic architectures and dual memory: One example of dynamic architectures, where resources are added as-needed, is CHILD (Ring, 1997), which determines when to create new units by analyzing the average errors observed in the weights of the network. Another dynamic architecture method is Progressive Networks (Rusu et al., 2016), where new neural networks ("columns") are added as new tasks are encountered, with lateral connections created to the networks of previously learned tasks. Progress and Compress (Schwarz et al., 2018) adapts this into a dual-memory method by only maintaining two columns and regularly distilling the active column into the knowledge-base column.

Regularization: Elastic Weight Consolidation (Kirkpatrick et al., 2017) and Synaptic Intelligence (Zenke et al., 2017) are examples of regularization. Both automatically learn an importance measure for the individual neuron weights for particular tasks, such that new tasks minimally influence weights that were important for prior tasks.

Rehearsal: One rehearsal method is CLEAR (Rolnick et al., 2018), which maintains a replay buffer over old tasks, using reservoir sampling and a behavioral cloning loss to ensure the network continues to respond appropriately to the inputs of old tasks, giving this method some regularization as well. CLEAR is built on top of IMPALA (Espeholt et al., 2018), an asynchronous non-continual learning actor-critic method.

Sparse coding and ensembles: Fixed Expansion Layer Networks (Coop et al., 2013) (FEL) is an example of both a sparse-coding and an ensemble method, though not one demonstrated in RL environments; it introduces a layer that is sparsely activated, such that the error signal does not propagate through all weights, and creates an ensemble of these to improve results. PathNet (Fernando et al., 2017), which may also be considered an ensemble method (Kemker et al., 2017), uses an evolutionary strategy to select a path through the network. The usage of specific paths instead of the whole network is a similarity PathNet shares with SANE. However, the method still requires task boundaries to determine when to freeze paths, a key difference from SANE.

5 CONCLUSION

The core challenge behind continual learning is knowing how and when to adapt to novel situations without overwriting previous behavior. Most existing methods side-step the problem by allowing the task boundaries to be specified, giving a clear signal for when to add parameters, freeze them, reinitialize them, etc. Our method, Self-Activating Neural Ensembles, needs no such crutch. It automatically detects and adapts to drift in the environment by adding new nodes to the SANE tree. Additionally, since only the necessary set of nodes is activated at each timestep, nodes that did not participate are not updated with the new interaction data and therefore are not susceptible to catastrophic forgetting. Finally, our method automatically merges nodes as necessary, ensuring the resources consumed remain bounded.

The biggest roadblock to deploying SANE on more and harder problems is that presently during every update, all nodes' critics are trained. While this step is parallelizable, it still forms the primary bottleneck and limits the size of the networks we can train. Improving this is one clear direction for future work. Other directions include experimenting with trees with more layers, sharing parameters between nodes, having a common pre-encoder akin to the visual cortex, actively mining negative examples from other nodes, and many others.

REFERENCES

Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. https://github.com/maximecb/gym-minigrid, 2018.

Robert Coop, Aaron Mishtal, and Itamar Arel. Ensemble learning in fixed expansion layer networks for mitigating catastrophic forgetting. *Neural Networks and Learning Systems, IEEE Transactions on*, 24:1623–1634, 10 2013. doi: 10.1109/TNNLS.2013.2264952.

Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. *CoRR*, abs/1802.01561, 2018. URL http://arxiv.org/abs/1802.01561.

Sebastian Farquhar and Yarin Gal. Towards robust evaluations of continual learning. *arXiv preprint arXiv:1805.09733*, 2018.

Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A. Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *CoRR*, abs/1701.08734, 2017. URL http://arxiv.org/abs/1701.08734.

Robert French. *Catastrophic Forgetting in Connectionist Networks*, volume 3. 01 2006. ISBN 9780470018866. doi: 10.1002/0470018860.s00096.

Ian J. Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks, 2015.

Robert Jacobs, Michael Jordan, Steven Nowlan, and Geoffrey Hinton. Adaptive mixtures of local experts. *Neural Computation*, 3:78–88, 02 1991. doi: 10.1162/neco.1991.3.1.79.

Ronald Kemker, Angelina Abitino, Marc McClure, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. *CoRR*, abs/1708.02072, 2017. URL http://arxiv.org/abs/1708.02072.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017. ISSN 0027-8424. doi: 10.1073/pnas.1611835114. URL https://www.pnas.org/content/114/13/3521.

Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. TorchBeast: A PyTorch Platform for Distributed RL. *arXiv preprint arXiv:1910.03552*, 2019. URL https://github.com/facebookresearch/torchbeast.

Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Timothée Lesort, Vincenzo Lomonaco, Andrei Stoian, Davide Maltoni, David Filliat, and Natalia Díaz Rodríguez. Continual learning for robotics. *CoRR*, abs/1907.00182, 2019. URL http://arxiv.org/abs/1907.00182.

Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation - Advances in Research and Theory*, 24(C):109–165, 1 1989. ISSN 0079-7421. doi: 10.1016/S0079-7421(08)60536-8.

German Ignacio Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *CoRR*, abs/1802.07569, 2018. URL http://arxiv.org/abs/1802.07569.

Roger Ratcliff. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 97 2:285–308, 1990.

Mark Ring. Child: A first step towards continual learning. *Machine Learning*, 07 1997. doi: 10.1023/A:1007331723572.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy P. Lillicrap, and Greg Wayne. Experience replay for continual learning. *CoRR*, abs/1811.11682, 2018. URL `http://arxiv.org/abs/1811.11682`.

Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *CoRR*, abs/1606.04671, 2016. URL `http://arxiv.org/abs/1606.04671`.

Jonathan Schwarz, Wojciech Czarnecki, Jelena Luketina, Agnieszka Grabska-Barwinska, Yee Whye Teh, Razvan Pascanu, and Raia Hadsell. Progress & compress: A scalable framework for continual learning. In *International Conference on Machine Learning*, pp. 4528–4537, 2018.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

Friedemann Zenke, Ben Poole, and Surya Ganguli. Improved multitask learning through synaptic intelligence. *CoRR*, abs/1703.04200, 2017. URL `http://arxiv.org/abs/1703.04200`.

A APPENDIX

Code is provided in the supplementary.

A.1 NODE VISUALIZATIONS

Here we see four instances of states the same node was activated. These examples were simply the last four usages we recorded where this node was activated, not cherry picked from a larger set. In all cases we can see that the ideal next action is to move forward, and indeed that is what this node's policy encodes for.

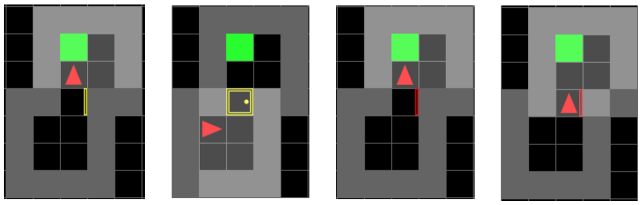

Figure 6: An example of four instances where the same node was triggered. When this node fires, the action taken is for the agent to move forward.

A.2 SANE IMPLEMENTATION DETAILS

There are a number of details left out of the main SANE explanation, to streamline the explanation down to the most significant concepts. These details are explained here, roughly in decreasing order of significance.

- The entries in the replay buffer are importance-weighted when being used to train the critic. In particular they are scaled by the simple ratio: $\frac{P_{cur}(a)}{P_{old}(a)}$, or in other words the ratio of the probability of the selected action using the node's current policy to that if the node's policy at the time the action was originally taken.
- When merging nodes, only the first 3/4 of the set of nodes is eligible to be merged. This is under the premise that the most recent nodes were likely newly created, and we're giving them a little longer to be used before we collapse it into another node.
- During training SANE operates in an epsilon-greedy manner, selecting a random action some ϵ fraction of the time, to encourage exploration.
- All nodes' policies are scaled as $3tanh(P)$, meaning each logit falls in the range (-3, 3). This is for a few reasons. (1) Even if a node is exceptionally confident in an action, when it encounters a new situation it must still have some non-zero chance of trying another action (related to epsilon-greedy). (2) Raising the probability of a previously-disadvantaged action to a range where it gets selected frequently should not take a large number of examples.
- We maintain 3 sets of usage counts: (1) The total number of times the node has been activated across its entire ancestry (i.e. including the count of the node it was cloned from), roughly representing how many unique replay entries the critic has been exposed to. During merge this count becomes the average of the two nodes'. This is the usage used in weighting policies and replay buffers during merge. (2) The usage count since this node was created or last cloned. Used for determining whether the node is eligible to be cloned again. (3) The usage count since the last time the node was trained, only used to determine whether to train the node's critic during the update step.
- Usage count (1) from the above point is not used directly in determining a node's weight; instead the weight is given by $max(1, tanh(usage_count/20000) * 100)$. These constants have not been highly tuned, and work surprisingly well regardless of task. Ensuring a minimum of 1 means that under-used nodes are never completely ignored during a merge.
- To adjust for the fact that rewards are not consistently scaled across tasks, we maintain a max_reward constant, set to the absolute value of an observed reward that exceeds the

current max_reward. This value is multiplied by .999 every timestep. All rewards are scaled as $10 * reward/max_reward$, to scale their values between [-10, 10].

- The critic network's weights are initialized according to a normal distribution with mean 0 and standard deviation $0.001/in_features$. In other words, the values are very close to 0.
- To adjust for the fact that the very first node's anchor is initialized randomly, and thus not effective as a baseline, we have an edge-case way to initiate the promotion of a node: when it is used more than 10,000 times. (Usage count 2 above.)
- We take a leaf out of IMPALA's book and, since our critics are updated asychronously, every node has a duplicate critic that is what gets trained, and then its weights are copied into the shared critic that is used during data collection. This does mean critic weights change mid-data collection, but not mid-training.
- Every training step we update the non-leaf nodes' policies to be the weighted average of their children's. These policies are used for two purposes: (1) if the node is cloned, its policy will end up as the policy of a leaf, and thus used, and (2) the policies are used to determine which nodes to merge, so a reasonably meaningful one is beneficial. These aspects are primarily important in trees with more than two layers.
- Periodically (every 100 updates) we send to each non-leaf node a batch of entries from each of its children. This is in case one of the children has been getting activated to the exclusion of the others, ensuring the parent still represents its other children.
- When bootstrapping the estimated value for the last entry in the replay buffer to compute the discounted rewards, we found it helpful to multiply that value by 0.9. This is so if the estimate is wrong, it doesn't perpetuate itself via the discounted rewards entered into the replay buffer.

A.3 SANE Experiment Details

The majority of SANE's constants are common between experiments:

Description	Value
Error scale α	1.0
Policy learning rate	1e-4
Critic learning rate	1e-3
Replay buffer size	4096
Critic train epochs	2
Merge ratio	3/4
Num parallel envs	1
Timesteps between updates (T)	1000
Min usages for promotion k_m	1500

MiniGrid constants:

Description	Value
Replay batch size (k)	4096
Reward discount γ	.99
Epsilon ϵ	0.02

MNIST constants:

Description	Value
Replay batch size (k)	1024
Reward discount γ	0
Epsilon ϵ	0.04

MNIST gets divided into more batches per training cycle, MNIST's discount factor must be 0 because the images presented to the agent have no correlation to each other in time, and MNIST gets a higher exploration rate.

A.4 Baseline Implementation Details

For IMPALA we used the Torchbeast implementation (Küttler et al., 2019). We built CLEAR on top of this base. We used the same constants for both baselines.

Code is available in `continual_rl/policies/impala/torchbeast/monobeast.py`.
torchbeast's default values are available in `continual_rl/policies/impala/impala_policy_config.py`
Listed here are the ones we changed:

MiniGrid constants

Description	Value
Unroll length (k)	20
Entropy cost	0.005
Learning rate	0.006
Epsilon ϵ	0.02
Num learner threads	1
Discounting	0.99

MNIST constants

Description	Value
Unroll length (k)	20
Entropy cost	0.01
Learning rate	0.001
Epsilon ϵ	0.04
Num learner threads	1
Discounting	0

A.5 MiniGrid Remaining Result

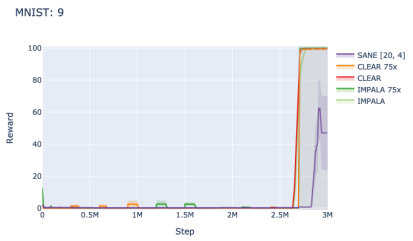

Figure 7: The result of training on digit 9, annexed here for want of space.

A.6 Network Specifications

To simplify things, we adopt the convention of describing 2d convolutional layers with $Conv(c, w, s)$, indicating there are c output channels, window size $w \times w$, and stride s. Similarly, we denote max pooling over a window size of $w \times w$ with $MaxPool(w)$. Finally, we denote a linear layer with output c channels with $Lin(c)$.

In all cases the non-linearity used is ReLU, and for simplicity remains implied.

A.6.1 BASE NETWORKS

For MiniGrid the base encoder, shared by SANE, CLEAR, and IMPALA is

$$Conv(16, 2, 1) \rightarrow MaxPool(2) \rightarrow Conv(32, 2, 1) \rightarrow Conv(64, 2, 1) \rightarrow Flatten \rightarrow Lin(32) \tag{4}$$

For MNIST the base encoder is:

$$Conv(24, 5, 1) \rightarrow MaxPool(2) \rightarrow Conv(48, 5, 1) \rightarrow MaxPool(2) \rightarrow Flatten \rightarrow Lin(32) \tag{5}$$

For CLEAR and IMPALA, the output of these encoders is concatenated with a one-hot of the last action and the last reward, and projected using a linear layer to the final output size: the total number of actions (10 for MNIST, 7 for MiniGrid).

For SANE, the encoder is followed by

$$encoder \rightarrow Lin(128) \rightarrow Lin(128) \rightarrow Lin(2) \tag{6}$$

Where the the 2 output values are the expected value v and the uncertainty u for that node. This gives a total of about 75,000 parameters for MNIST and around 35,000 for MiniGrid.

A.6.2 INCREASED PARAMETER NETWORKS

A number of iterations were tried for these larger networks, particularly with emphasis on maintaining certain bottleneck layer dimensions, though we do not claim to have spanned the whole space of options. These are what were used in the paper:

For MiniGrid, giving around 1.2M parameters:

$$\begin{aligned} &Conv(1024, 2, 1) \rightarrow Conv(1024, 1, 1) \rightarrow MaxPool(2) \rightarrow Conv(32, 2, 1) \rightarrow Conv(64, 2, 1) \\ &\rightarrow Flatten \rightarrow Lin(32) \end{aligned} \tag{7}$$

For MNIST, giving around 5.5M parameters:

$$\begin{aligned} &Conv(24, 5, 1) \rightarrow Conv(2048, 1, 1) \rightarrow Conv(2048, 1, 1) \rightarrow Conv(24, 1, 1) \rightarrow MaxPool(2) \\ &\rightarrow Conv(48, 5, 1) \rightarrow Conv(1024, 1, 1) \rightarrow Conv(1024, 1, 1) \rightarrow Conv(48, 1, 1) \rightarrow MaxPool(2) \\ &\rightarrow Flatten \rightarrow Lin(32) \end{aligned} \tag{8}$$