Use of External Markers by Reactive Agents as an Easier Evolutionary Route Toward Memory

Shivashriganesh P. Mahato
Dept. of Comp. Sci. & Eng.
Texas A&M University
College Station, TX, USA
shivashriganesh@tamu.edu

Shreyes Kaliyur
Dept. of Comp. Sci. & Eng.
Texas A&M University
College Station, TX, USA
shreyes@tamu.edu

Ji Ryang Chung
Tilda Corp
Seoul, Korea
jiryang@gmail.com

Yoonsuck Choe
Dept. of Comp. Sci. & Eng.
Texas A&M University
College Station, TX, USA
choe@tamu.edu

***Abstract*—Memory is a key functional requirement for cognitive agents. There are three basic ways to implement memory using neural networks: (1) RNN: recurrent neural networks, (2) TDNN: time-delayed neural networks (feed-forward), and (3) DROPPER: external marker dropper/detector (feed-forward). All three have been found to be effective in prior research. In this paper, we ask which of these mechanisms could have evolved earlier/easier? To answer this question, we set up a simple ball-catching task where two balls fall from above at different speeds, and an agent at the bottom has to catch the balls using range sensors. Depending on the relative speed of the balls, sometimes the slow ball will go out of sensor range, thus to catch the fast ball first then remember to catch the second (slow) ball, memory is required. We used the Neuroevolution of Augmenting Topologies (NEAT) algorithm to evolve all three types of memory mechanisms, where not only the connection weights but also the network topologies are evolved. Our results show that the DROPPER mechanism is the fastest to evolve a successful controller, followed by TDNN and RNN. Among the feed-forward topologies, we also found that DROPPER is more robust than TDNN (less sensitive to the relative speed of the balls). These results show that a simple reactive agent could quickly evolve a rudimentary form of memory through depositing and detecting external markers, long before other internalized memory mechanisms evolve. These findings shed light on the evolutionary route toward memory in cognitive agents.**

***Index Terms*—Memory, Working Memory, Evolution, Neuroevolution, Internal vs. External**

I. Introduction

Feed-forward (FFW) neural networks can only generate reactive responses to the present input, oblivious of any inputs that may have come by in the past (Fig. 1A). As such, FFW networks are unsuitable for tasks that require memory of the past events. (Adjustment of connection weights in FFW networks could be considered as a form of memory [synaptic memory], but here we are only considering a higher level memory function like working memory.) To overcome this, some form of architectural change is needed, either (1) recurrent connections [1], [2] (Fig. 1B), or (2) delayed connections of various durations [3] (Fig 1C).

However, there is yet another way to implement some form of memory, without any architectural change to the FFW topology. Previous works [4] have shown that allowing the FFW network to drop and detect markers in the environment (dubbed the "dropper" network) can endow memory capability to these reactive agents (Fig. 1D). (Note that these new facilities [dropping and detecting markers] can be implemented in the agent with negligible overhead, by reusing existing sensors and existing mechanisms such as excretion.) In these works, a simple ball-catching task (inspired by [5], [6]) that requires memory (Fig. 2) was successfully solved using such an approach, with performance comparable to a fully recurrent neural network.

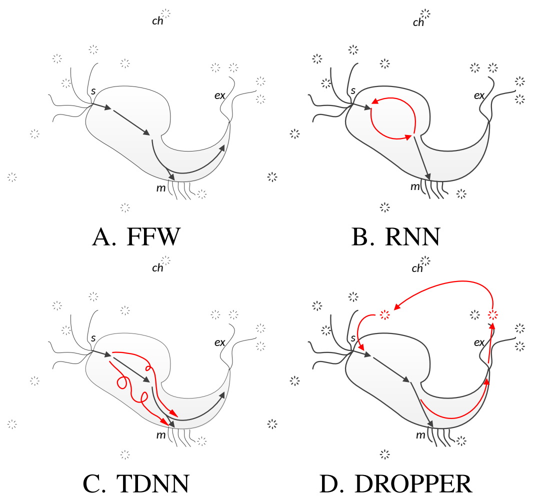


Fig. 1. **Simple Agents**. A. Reactive agent with a feed-forward pathway: sensors (s) and motor (m). (ch): chemicals. B. Recurrent agent with recurrent loop (red). C. Agent with feed-forward control pathway with delay lines (red). D. Reactive dropper agent, internally feed-forward, but with an outside loop (red) via excretion (ex) and detection.

The two basic approaches above to implement memory, i.e. (case 1) architectural adjustments (recurrent connections or delay lines) and (case 2) dropping and detecting markers in the environment, can be regarded as internal vs. external memory, respectively. (Note that in methods like Neural Turing Machine (NTM) [7] that use "external memory", memory is an integral part of the neural network architecture, thus it still falls under case 1, i.e., internal memory.)

The main research question we would like to address in this paper is which of these forms of memory evolves faster. Although there have been some works investigating the evolution of memory, to our knowledge, this specific question has not been addressed (see e.g. [8], [9], and section VI). We know that animals with limited brain capacity depend on stigmergy (".. sensing and modifying the environment ... determines the animal's behavior") to form complex behavior [10]. See [11] for a review, and [12] on a related concept (cognitive offloading). However, can we show computationally whether such an external approach is more efficient and more likely to emerge first?

To answer this question, we used the NeuroEvolution of Augmenting Topologies (NEAT) algorithm in a ball-catching task that requires memory of past events (Fig. 2). Since we are investigating evolution, especially that of neural architectures, gradient-based deep learning methods were excluded from our consideration as the main methodology.

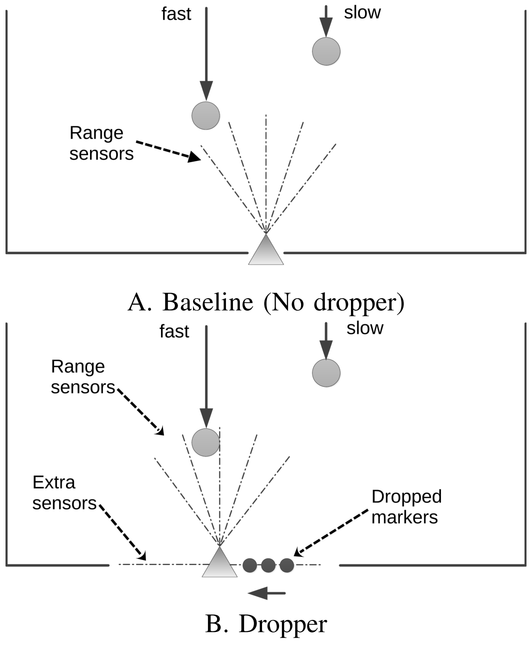


Fig. 2. **Ball-Catching Task**. The ball-catching task [4], [5] is shown in the context of two types of controllers. See section II for details. A. Task without dropper: only input is the set of range sensors. B. Task with dropper: the agent can drop markers, and in addition to the range sensors in A., another pair of extra sensors can detect the dropped markers.

Starting from the same single-layer FFW architecture, we evolved the topology and the connection weights of the three types of networks (1) recurrent neural networks (no dropper, recurrent), (2) time-delayed neural networks (a set of delay lines, no dropper, feed-forward), and (3) dropper neural networks (with dropper and extra sensors, feed-forward), using NEAT. Since NEAT can add or remove units (nodes) and connections (edges) in the neural network topology, it can also introduce recurrent loops in the network topology, depending on the task. For the three types of networks, we observed the resulting performance, number of generations required to pass a performance threshold (evolution speed), behavioral patterns, and robustness. This allows us to investigate the evolution of memory along two dimensions: (1) location of memory, and (2) network topology (Table I).

TABLE I
TWO ASPECTS OF MEMORY: LOCATION VS. TOPOLOGY

Location / Topology	Feed-forward	Recurrent
Internal	TDNN	RNN
External	DROPPER	*

Our results show that the dropper network evolves a successful solution much faster than TDNN and RNN, and exhibits a more efficient behavior (wandering around is minimized). The dropper network was also found to be more robust to changes in the task condition (speed of the balls). These results suggest that faced with a similar memory task (spatial memory), it is evolutionarily more efficient to have some kind of external memory implemented using a simple marker dropping/detecting mechanism, than evolving a delay lines or complex recurrent topology.

The main contribution of our work compared to existing works [4], [10] is to have shown the evolutionary advantage of using external memory (no neural architectural change), compared to internal memory (requires architectural change: recurrence or delay lines). Note that dropping and detecting markers can reuse existing metabolic processes such as excretion and the same kind of object sensors, thus no neural archtectural change is needed. See section VI for details.

The rest of the paper is organized as follows. Section II provides details regarding the task. Section III presents a brief review of the NEAT algorithm. The basic methods are outlined in section IV, and the experiment and results presented in section V. Section VI gives a more in-depth analysis of the results, followed by conclusion (section VII).

II. TASK: CATCHING FALLING BALLS

The ball-catching task [5] is shown in Fig. 2, in the context of two types of neural network controllers. The goal of the task is to catch both balls falling from above, where the two balls fall at different speeds. Either the left or the right ball may be faster than the other. The agent has a set of range sensors, which will indicate the distance to the objects that come into contact with the sensors. The agent generates left or right movement (based on its output) to move to the position where the balls can be caught. The speed of the balls is set up so that initially both balls are detectable, but when the fast-falling ball is caught, the slow-falling ball goes out of the sensor range. Thus, memory is needed to successfully catch both balls. A reactive agent (e.g., a FFW network) will only catch the fast-falling ball, since there will no longer be sensory input. The task environment is slightly modified for the dropper network, where the agent can now drop a marker in the environment, and two extra sensors are added so that the agent can sense the dropped markers (Fig. 2B).

III. NEAT: REVIEW OF THE NEUROEVOLUTION OF AUGMENTING TOPOLOGIES ALGORITHM

Topological neuroevolution methods evolve both topology and weights of neural networks. Because natural evolution includes changes in the network topology in the brain, they mimic the natural evolution better than traditional weight-only neuroevolution methods. Moreover because the functionality of a neural network can be constrained by its topology, allowing the topology to evolve will set free the structural constraints and result in significant performance gain. Amongst many variations of such an approach, we will use NeuroEvolution of Augmenting Topologies (NEAT) because of its advantages over other topological evolution methods [13]–[16].

Historical marking is the core of the NEAT algorithm. By enumerating each innovation, NEAT solves the competing conventions problem, which is one of the main problems in neuroevolution [13], [16]. The crossover operation in NEAT happens between two genomes with identical historical marking (also called "innovation number"), regardless of their locations and size in the network. Moreover, NEAT keeps the size of resulting network from growing explosively by starting with the initial populations with the minimal structure. NEAT encodes the genome in two arrays, node genes and connection genes. Innovation number is assigned to each connection gene according to the order of its appearance throughout the evolutionary stages. The connections can also be enabled or disabled through mutation. Since connections can be generated arbitrarily between any pair of neurons (the nodes), recurrent connections can and will be generated. There are several other important facilities such as speciation, where a subpopulation of individuals are isolated from other subpopulations, forming a species. This mechanism is used to encourage new topologies to have a chance to evolve. See [16] for more details on the NEAT algorithm.

IV. Methods

a) Environment: The agent is located at the bottom of a 2D environment with width $w = 400$ and height $h = 300$ (Fig. 2). At the beginning of each trial, the agent is located at $x_0 = 200$. The two balls fall from above at position $x_1 = 130$ and $x_2 = 270$, and height h. The speed of the balls v_1 and v_2 are either 1.2 or 0.4 and vice versa, depending on which ball is faster. The x position and speed were selected so that when the fast ball is caught the slow ball is out of sensor range.

b) Agents: All three types of agents (RNN, TDNN, and DROPPER) had $r = 5$ range sensors of length $\ell = 300$. The field of view was $\theta = \pi/6$. The agent can move at a fixed speed $v_a = 3$. Each range sensor, when an object (e.g. the ball) intersects it, generates an activation value of $r_i = 1 - \frac{d_i}{\ell}$ ($r_i = 0$ when there is no contact). The DROPPER agent has two additional range sensors s_1 and s_2, which report the same value when in contact with the dropped marker, normalized by the environment width w. The DROPPER agent has an additional output unit, which determines whether to drop a marker or not. The initial topology of all three agent types are shown in Fig. 3.

c) Evolution: We use NEAT to simulate the evolutionary track of the neural networks, implemented using the third-party *NEAT-Python* library. See the Appendix for the NEAT hyperparameters. Aside from the common parameters, these

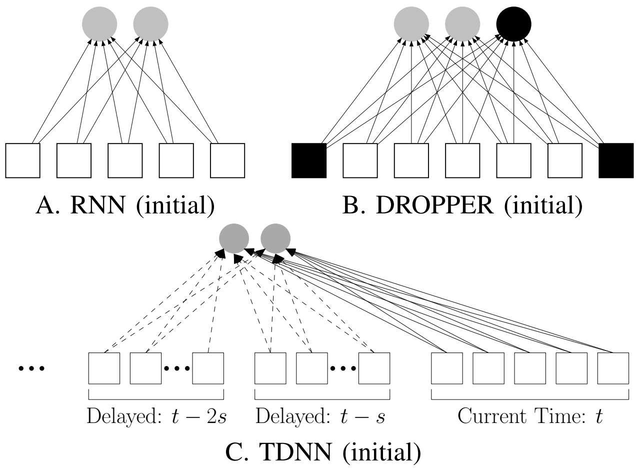


Fig. 3. Initial Network Topology. A. RNN. White = input (5 range sensors), Gray = output (movement left, right). B. DROPPER. Same as A, but with additional inputs (black squares) and additional output (black circle: drop marker). C. TDNN. Same as A., but with additional inputs with time delay. The delay inputs are initially inactivated (dashed).

were the agent-specific differences: (1) RNN: recurrent connections allowed. (2) TDNN: additional inputs due to delayed connections (but the delayed connections were initially inactivated), and recurrent connections disallowed. (3) DROPPER: additional inputs due to marker sensors, additional output due to marker dropping output, and recurrent connections disallowed.

In each trial, the agent is given two tasks: (1) left ball is falling fast, (2) right ball is falling fast. So, the agent has to catch a total of 4 balls to get maximum fitness. The fitness is:

$$f = c + \frac{1}{max(\sum_{i=1}^{4} |b_i - x_i|, 1)},$$

where c is the number of balls caught, and b_i is the ball's horizontal position and x_i the agent's position. The second term is included to encourage ball-seeking behavior prior to successful catching.

For each agent, the population size was 200. Each evolutionary run was limited to 100 generations. If the run failed to produce a successful controller to catch all 4 balls, the run was considered a failure. For each agent type, we ran a total of 150 evolutionary runs, and the performance metrics were collected on all 150 runs. Behavioral analysis and robustness data were collected from the last $2/3$ of the runs. For performance, we used (1) success rate and (2) evolution speed. Success rate is the percentage of runs where a successful controller emerged. Evolution speed means is measured as the earliest generation and average generation where a successful controller emerged. These performance metrics allowed us to test which agent/memory type evolves faster.

For the TDNN agents, we further tested with different number of delays, and distinguished them as TDNN$_d$, where d indicates how many sets of delayed inputs were used. We tested with $d \in \{5, 10, 15, 20\}$. (This was done to make the comparison fair: DROPPER and RNN may have a finer-

TABLE II
ARCHITECTURES CONSIDERED IN THE EXPERIMENTS. FOR $TDNN$: d IS THE NUMBER OF DELAYED INPUT SETS AND s THE INTERVAL BETWEEN THE DELAYED SETS.

Abbrev.	Architecture	# Inp.	# Out.
$DROPPER$	FFW with Dropper	7	3
$TDNN_5$	FFW w/delay $d = 5$, $s = 48$	30	2
$TDNN_{10}$	FFW w/delay $d = 10$, $s = 24$	55	2
$TDNN_{15}$	FFW w/delay $d = 15$, $s = 16$	80	2
$TDNN_{20}$	FFW w/delay $d = 20$, $s = 12$	105	2
RNN	Recurrent NN	5	2

grained memory than TDNN with only a few delay lines.) The delay time interval between each set for a specific TDNN$_d$ was set to $s = 48, s = 24, s = 16, s = 12$, respectively, where for each set, the full 5 inputs were delayed as $t-s$, $t-2s$, $t-3s$, etc. where t is the current time (see Fig. 3C). The maximum delay was fixed to 240 (the maximum sensory silence interval we observed), and s was selected to evenly divide this, so, d effectively controls the time resolution of the delay. Table II shows all agent types tested.

V. RESULTS AND ANALYSIS

a) Success rate: The success rate measured how many evolutionary trials out of the 150 was each agent type successful in evolving a controller that caught all four balls. Table III summarizes our findings. The results show that DROPPER is the most successful, followed by the four TDNNs, then by RNN.

TABLE III
SUCCESS RATES OF THE CONTROLLER AND 95% CONFIDENCE INTERVALS, OVER $n = 150$ TRIALS EACH.

	Success Rate (%)	CI (%, %)
$DROPPER$	100.00	$(100, 100)$
$TDNN_5$	60.67	$(53, 68)$
$TDNN_{10}$	74.67	$(68, 82)$
$TDNN_{15}$	72.67	$(66, 80)$
$TDNN_{20}$	76.67	$(70, 83)$
RNN	36.00	$(28, 44)$

b) Speed of Evolution: To measure how fast a viable solution emerges for each agent type, we checked the generation when a successful controller emerged. Table IV summarizes the results. Again, DROPPER was the best, evolving a solution the earliest (generation 3), and was the fastest, on average (mean = 11.48 generations). TDNN followed next, and RNN last. This shows that it is easier and faster to evolve a solution with the dropper. Also see Fig. 4 for the fitness over generations.

c) Speed of Milestones Appearance: One interesting observation regarding the hidden unit activations is that some of the evolved hidden nodes show constant activity regardless of the changes in sensory input. Some of these may be dangling hidden nodes that at some point had incoming connections but at a later time lost all incoming connections (NEAT can remove, as well as sprout new connections: see Fig. 5B for an example). All other hidden units play a functional role in the network, and connections attached to these functional

TABLE IV
GENERATION WHERE SUCCESSFUL SOLUTION EMERGED FOR THE DIFFERENT CONTROLLER TYPES AND THEIR 95% CONFIDENCE INTERVALS, OVER $n = 150$ TRIALS EACH.

	Earliest	Mean	SD	CI (gen, gen)
$DROPPER$	3	11.48	4.03	$(10.83, 12.13)$
$TDNN_5$	9	48.18	24.35	$(43.17, 53.18)$
$TDNN_{10}$	9	49.88	25.39	$(45.17, 54.58)$
$TDNN_{15}$	3	52.98	23.66	$(48.54, 57.42)$
$TDNN_{20}$	9	47.37	23.45	$(43.08, 51.65)$
RNN	9	58.74	26.63	$(51.64, 65.84)$

units may have significance. Given a successful controller, it would be interesting to see when a particular connection included in this successful controller appeared first during the evolution. This can be easily tracked down, post-hoc, by simply referencing the innovation number of such connections. We call the first appearance of such a functional connection a *milestone*. We can count how many milestones are included in the final solution, and at what interval these milestones appear throughout evolution (milestone speed). The milestone speed is computed as

$$\nu = \left(\frac{1}{N} \sum_{i}^{N-1} (\tau_{i+1} - \tau_i) \right)^{-1},$$

where N is the number of milestones, τ_i is the i-th milestone generation. so higher number means shorter intervals between milestones. Table V shows these results. The main finding is that DROPPER is about 3$\times$ faster in sprouting milestones (0.524 vs. $\sim$0.15). The number of milestones are comparable (note: larger networks may contain more milestones).

TABLE V
MILESTONE DATA OF DIFFERENT CONTROLLERS (H.M. = HARMONIC MEAN).

	Avg. # Milestones	H.M. Milestone Spd ν
$DROPPER$	5.00	0.524
$TDNN_5$	6.99	0.168
$TDNN_{10}$	7.22	0.154
$TDNN_{15}$	7.83	0.153
$TDNN_{20}$	6.91	0.147
RNN	7.13	0.124

d) Evolved Topology: Representative evolved neural network topology of each agent type is shown in Fig. 5. It is hard to relate these topologies directly to behavioral performance, but there are several interesting things to note. Of course the RNN has recurrent loops (see the loops between the output units [green] and the hidden units below them [orange]). The RNNs tend to evolve more hidden units than the TDNN or DROPPER, and many delay lines in TDNN are unused. DROPPER shows the simplest topology (shallowest and least number of hidden units), which is understandable, given how early in the generation they solve the task.

e) Behavior (DROPPER): The DROPPER behavior is shown in Fig. 6A. Compared to other agent types, DROPPER tracks the balls most closely, with minimum overshoot, undershoot, or oscillatory behavior. All successful agents showed

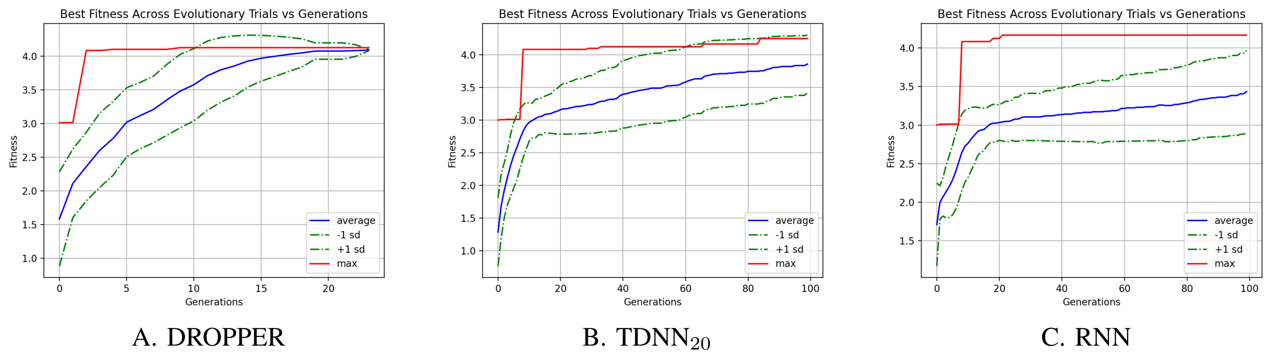


Fig. 4. Fitness Over Generations. x-axis: Generations. $y-$axis: Fitness. Each plot shows population average, standard deviation, and best fitness shown. Note: DROPPER's x-axis is up to 20. Others go up to 100. Also, DROPPER's average fitness reaches 4 (max) while others do not. Note: TDNN$_5$ and other TDNNs were similar to TDNN$_{20}$ shown in B.

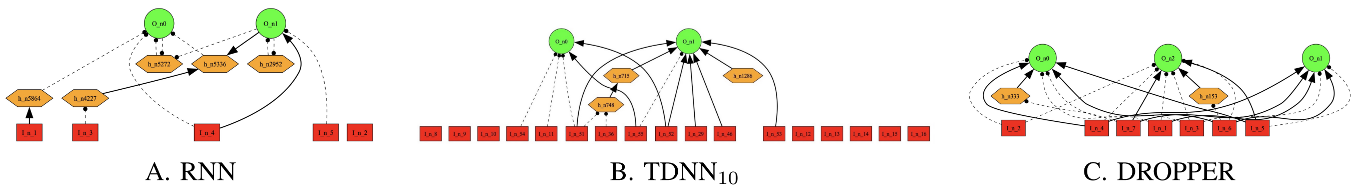


Fig. 5. Evolved Topologies. Red: input, Green: output, Orange: hidden. Solid line: positive weight. Dashed line: negative weight. Some unused input nodes are omitted in B. Some hidden units are dangling (B, h_{n1286}, the right-most one) due to connections removed by the NEAT algorithm. Note: due to the use of graphviz, the orderings of the nodes are not sequential.

a very similar trajectory. Although the movement behaviors were rather uniform, the marker usage strategy showed some divergence. Fig. 7 shows the marker dropping behavior (see section VI for more on this).

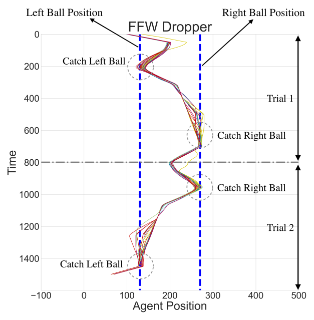
f) Behavior (RNN): The RNN behavior was similar to the DROPPER, but with slightly higher deviation from the optimal path (Fig. 6B). As DROPPER, all agents exhibited similar behavior.

g) Behavior (TDNN): Unlike DROPPER or RNN, TDNN exhibited diverse behavior. We were able to observe some repeated behaviors, so we conducted a clustering analysis of the agent trajectories, where the similarity was measured using dynamic time warping (DTW; [17]).

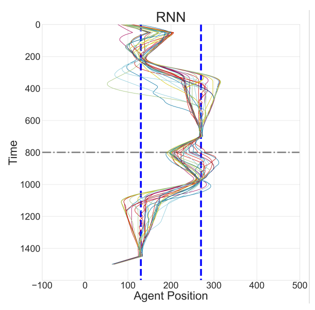
We first convolved the time series, then conducted the clustering. Given a time series T, we apply the transformation ϕ_K: $T \rightarrow^{\phi_K} T * \frac{1}{K}\mathbf{1}_K$ where K is the kernel size, $\mathbf{1}_K$ is the K-vector of all ones, and "$*$" represents the convolution operator. At each step of DTW a distance function is used to compare pairs of values across the two time series, and by default it uses $d(x, y) = |x - y|$. If we use a higher power pairwise distance function, then greater weight is placed on higher amplitude effects, which is exactly what we need to extenuate larger positional values away from the center [18]. We will denote DTW on time series T_1 and T_2, using the pairwise distance function $d(x, y) = |x - y|^\gamma$, as $DTW_\gamma(T_1, T_2)$. We found that $\gamma = 2$ works best for minimizing the within-cluster to between-cluster ratios. Next we parameterize a simple sigmoid as follows to center it and allow it to be scaled to our preference: $\sigma_{\alpha,\beta,\kappa}(x) = \kappa \left(\frac{1}{1+\exp(\frac{\alpha-x}{\beta})} - \frac{1}{2} \right)$ Next, given a center μ, our preprocessing of a time series T is defined by a function ψ: $\psi_{\mu,\beta,\kappa(T)} = \frac{\sigma_{-\frac{\mu}{2},\beta,\kappa}(x-\mu)+\sigma_{\frac{\mu}{2},\beta,\kappa}(x-\mu)}{2}$

In our environment, the center is $\mu = 200$ and we will let $\kappa = 400$. κ is set to the size of the viewing window, which is also approximately 3 times the distance between the balls. This fairly accommodates movement to the left and right of the balls. We found that $\beta = 15$ works best for our purposes. Given two agents $\mathcal{E}_1, \mathcal{E}_2$ and respective x position series T_1, T_2, we now define the distance between the corresponding agents in the path space as: $d(\mathcal{E}_1, \mathcal{E}_2) = DTW_\gamma(\psi_{\mu,\beta,\kappa}(T_1), \psi_{\mu,\beta,\kappa}(T_2))$ After computing distance values as a precomputed matrix, we use standard clustering algorithms to generate the clusters of behavior. We found that Agglomerative clustering works best for our purposes. Using the Elbow method, we find that an appropriate number of clusters is $k = 4$. The clustering results are shown in Fig. 6C. This only shows TDNN$_{20}$, but other TDNN$_d$ showed similar qualitative results. The four clusters show different overshooting behavior. Cluster 1 (Fig. 6C, top left) shows overshooting to the left only when the left ball falls fast. Cluster 2 (Fig. 6C, top right) shows overshooting to the right only when the right ball falls fast. Cluster 3 (Fig. 6C, bottom left) does not show overshooting, and Cluster 4 (Fig. 6C, bottom right) tends to overshoot to the right, regardless of which ball falls fast.

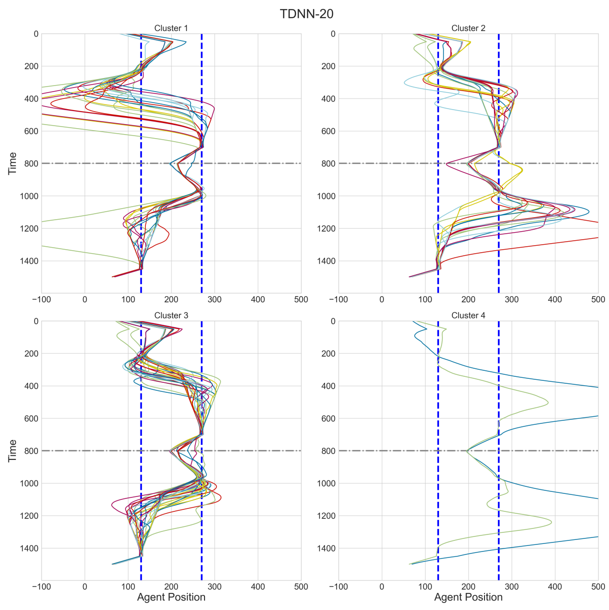
h) Robustness to Ball Speed Variation: Since each agent type may have their own strengths and limitations in the context of the task, our main results could be due to a certain type of agent particularly tuned to the specific simulations parameters. To guard against this possibility, we conducted a



A. DROPPER



B. RNN



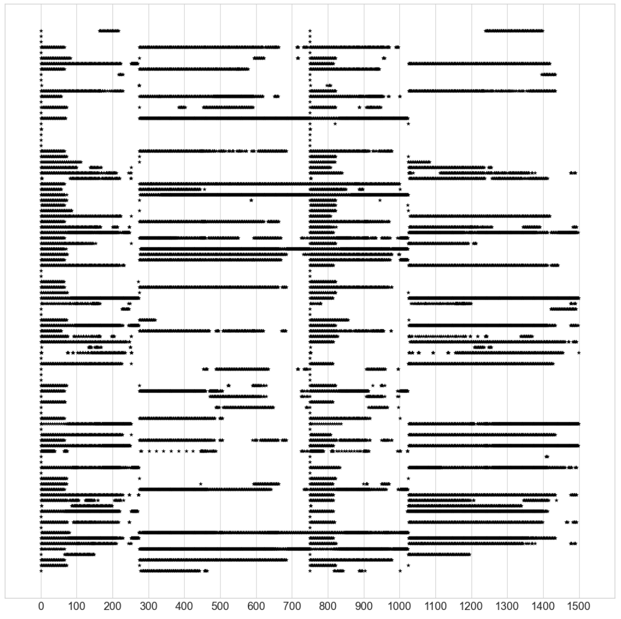
C. TDNN_{20} Behavioral Clusters.

Fig. 6. Agent Behavior. The x-axis represents horizontal position of the agent, and the y-axis time. The two blue vertical lines show the two balls' positions, respectively. The gray horizontal line marks the boundary between the left ball fast (top) vs. right ball fast tasks (bottom) conditions. The height of the balls are reset to h at this boundary.

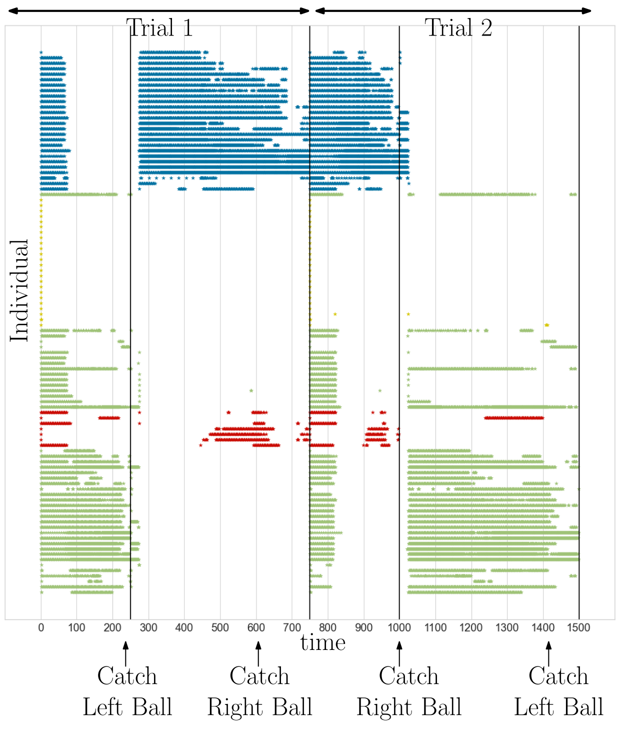
robustness study, by varying the task parameters. The main variation was in the speed of the balls.

In our evolution simulations reported above, we used the same ball speed conditions. The robustness test was done by using the previously evolved controllers, and changing the speed of the balls by scaling them with a multiplier. The chosen grid of speed multipliers was $\{0.5 + 0.1j : j \in [\![0..20]\!]\} \setminus \{1.0\}$. That is from 0.5 to 2.0, half the speed up to twice the speed, omitting the identical speed 1.0. All three agent types were tested.

The robustness was measured based on three criteria: (1) success rate, (2) average number of balls caught, and (3) average distance from the ball and the agent when the balls hit the ground (lower the better). Fig. 8A shows the results. The success rate drops significantly for most of the agent types, but DROPPER maintains performance for slower conditions (multiplier < 1.0) and for some faster conditions (multiplier > 1.0). The DROPPER also does best in terms of the number of balls caught and the distance to the balls. RNN is the next up, followed by the various TDNNs.



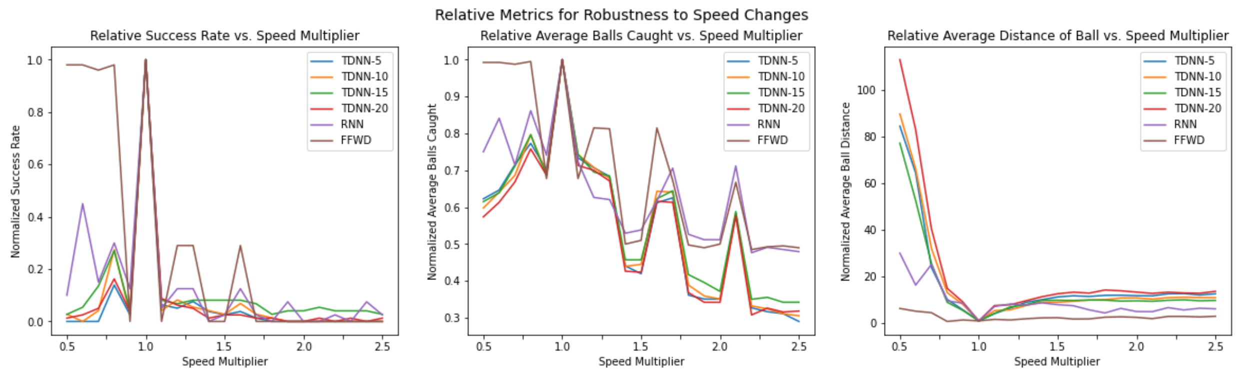
A. Markers



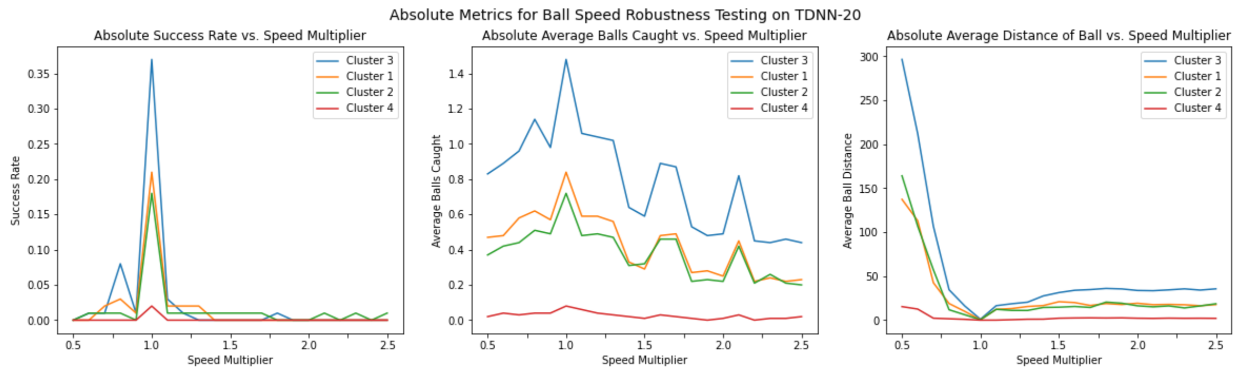
B. Markers (Clustered)

Fig. 7. DROPPER Agent Marker Dropping Behavior. A. shows the unordered marker dropping data ('*' marks the dropped marker). The x-axis is time and the y-axis represents individual agents (their index). B. is the clustered version of A. The four vertical lines at $x = 250, 750, 1000, 1500$ mark when the balls touch the ground.

Since TDNN showed high sensitivity to ball speed, we conducted further analysis, measuring the robustness by behavioral category. The results are shown in Fig. 8B. Interestingly, cluster 3 did the best on the first two metrics, but did the worst



A. RNN vs. TDNN vs. DROPPER (marked as FFWD)



B. TDNN$_{20}$: Comparison by behavioral cluster

Fig. 8. Relative Metrics for Robustness to Ball Speed Variation. In each plot, the x-axis is the ball speed multiplier and the y-axis, from left to right, (1) success rate, (2) Average number of balls caught, and (3) Average distance from the ball when the balls hit the ground. In A, relative metrics are used due to the large difference among the three agent types. The results in B use unnormalized absolute metrics since these are all from the same agent type.

for the third metric. Cluster 3 is the behavior that is the most similar to DROPPER and RNN, without much overshoot in any direction (Fig. 6C, bottom left).

VI. Discussion

The main contributions of this paper are in the systematic testing of the emergence of memory through evolution along two major axes (1) internal vs. external, and (2) feed-forward vs. recurrent (Table I), and the discovery of the order of ease in the three different memory strategies: DROPPER, TDNN, followed by RNN. The comparison of the DROPPER and TDNN, both restricted to feed-forward topology, explored the internal vs. external aspect. The comparison of the DROPPER and RNN, on the other hand, explored the feed-forward vs. recurrent neural network topology. Prior works have compared DROPPER and RNN, but they did not include TDNN in the comparison, and they did not examine the evolution of neural network topology [4]. Ollion et al. investigated the role of evolution in a sequence memory task (T-maze) using NEAT, but only considered RNN as the memory mechanism [9]. Blynel and Floreano showed that continuous time recurrent neural networks (CTRNNs) can solve the T-maze task, but again, the focus was on solving these memory tasks with one type of memory [19]. Brave used genetic programming for a planning task that requires memory, where both short-term memory and external buffer were used, but these were built into the agent, so the focus was the evolved agent's use of these memory, and no comparison with RNN was made [8]. Carvalho and Nolfi [12] considered "cognitive offloading" in reactive vs. recurrent controllers, but did not consider delayed connections, and did not evolve neural network topology. With our paper, we can now make an informed statement regarding the evolution of memory strategies. (See [12] for an interesting perspective on how cognitive offloading can promote cognitive development beyond the simple pairing of external markers and reative controllers.)

There are several limitations of our paper, which may lead to interesting future work. The first thing we note is that there is an empty grid in Table I, the lower right grid (marked "*"). This corresponds to Recurrent + external. This could be tested by allowing both the dropper mechanism and recurrent connections in the NEAT evolution. Since the agent now has both the dropper and the recurrent memory mechanisms, an interesting question is whether evolution would exploit one mechanism over the other. In fact, our preliminary simulations show that given this situation, the dropper mechanism will be heavily used, and recurrent connections are only used very sparingly (Fig. 9 shows 0 or only 1 recurrent loop [red] in a successful evolved controller). Note that having the dropper and detectors does not require excessive additional hardware. As mentioned in the introduction, existing sensors and existing excretion mechanism can be easily repurposed for this function, without much overhead, and as we can see, it can be readily used.

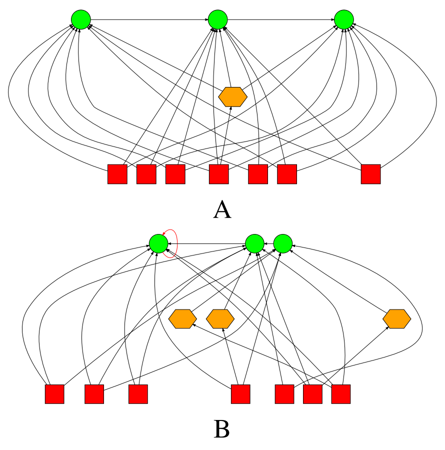


Fig. 9. Evolved Topologies for RNN+Dropper (preliminary results). No or only few recurrent connections evolved.

Another possible criticism is that the task itself is too simple. For example, what if we add more balls simultaneously falling from the top, etc.? In such a case, do we expect the same results to hold: DROPPER $>$ TDNN $>$ RNN? The main reason we adopted this simple task is to devise a minimal task that captures the essence of memory phenomena so that we can more easily identify key mechanisms: the "minimally cognitive" system, inspired by [5], [6]. Our careful expectation is that the results would carry over to more complex tasks. In fact, [4] showed that the DROPPER strategy works in a 2D foraging task, as well as the ball-catching task. One possible outcome for the foraging task is that TDNN might fail, due to the longer horizon in that task.

This brings us to another interesting question. Are there some other sorts of tasks where the results of this paper hold and yet other kinds of tasks where the results generally fail to hold? We expect that the DROPPER mechanism will work as long as the task is spatial (e.g. foraging, as mentioned above), where the markers can be deposited at a spatial coordinate in the environment. The mechanism may not work well for other types of memory tasks, such as remembering sequences or memories that require hierarchy (however, it is possible that if different types of markers can be deposited, these more complex tasks could be handled).

Also, when the task becomes more complex with multiple agents, the standard DROPPER strategy may not be sufficient. For example, when there are multiple agents, e.g., predator and prey, they may drop their own external markers, and the agents must distinguish between their own marker and that of the others' (see [20]). The current strategy used in this paper cannot handle such cases.

Finally, we would like to comment on interesting behavioral patterns of DROPPER agents in terms of the marker strategy. Let us reconsider Fig. 7. The first plot (Fig. 7A) does not seem to show any meaningful strategy, but once we clustered the marker dropping patterns, we were able to see distinct strategies emerging (Fig. 7B). There are four clusters: Blue, Yellow, Red, and Green. In the plot, the horizontal center marks the boundary between the left ball fast vs. right ball fast condition. With this in mind, we can see that the Blue and Green clusters are basically the same. For one condition, drop the marker a bit when chasing the fast ball, and once the slow ball is caught, leave a long trail (Case 1). For the other condition, immediately start dropping markers and keep on dropping them until the fast ball is caught, then go silent (Case 2). The Yellow strategy is a minimalistic version of Case 2. The Red strategy is more ad hoc.

VII. Conclusion

In this paper, we aimed to explore the evolution of memory mechanisms in solving a memory-dependent task. We compared three memory strategies: dropper networks (DROPPER), time-delay neural networks (TDNN), and recurrent neural networks (RNN). Our findings indicate that DROPPER exhibit the fastest evolution speed among the three strategies, followed by TDNN, and RNN evolves the slowest overall. Analysis of hidden activation suggests that the emergence of key topological features (milestones), facilitated by structural mutations, contributes significantly to the evolution speed. Further, in investigating emerged behaviors, we used time series analysis to cluster trajectories and identified distinct behavioral patterns in various strategies. Regarding robustness between the feed-forward networks, DROPPER appears to be more resilient to perturbed ball speeds. Further research can explore more complex environments and adaptive strategies to continue enhancing our understanding of memory evolution in neural networks, and hopefully push our understanding of cognition as a whole further.

Acknowledgments

This work is largely based on the dissertation and theses by the first three co-authors, not published elsewhere [20]–[22]. The implementation was based on NEAT Python: https://github.com/CodeReclaimers/neat-python.

VIII. Appendix

The NEAT Python hyperparameters and the computing resources can be found in the next page.

References

[1] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[3] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang, "Phoneme recognition using time-delay neural networks," *IEEE transactions on acoustics, speech, and signal processing*, vol. 37, no. 3, pp. 328–339, 1989.

[4] J. R. Chung and Y. Choe, "Emergence of memory in reactive agents equipped with environmental markers," *IEEE Transactions on Autonomous Mental Development*, vol. 3, pp. 257–271, 2011.

TABLE VI
NEAT HYPERPARAMETERS.

Common Hyperparameter	**Value**
`fitness_criterion`	`max`
`fitness_threshold`	4
`pop_size`	200
`reset_on_extinction`	`True`
DROPPER Hyperparameter	**Value**
`num_inputs`	7
`num_outputs`	3
`initial_connection`	`full_direct`
`feed_forward`	`True`
RNN Hyperparameter	**Value**
`num_inputs`	5
`num_outputs`	2
`initial_connection`	`full_direct`
`feed_forward`	`False`
TDNN Hyperparameter	**Value**
`num_inputs`	$5(d+1)$
`num_outputs`	2
`initial_connection`	`deterministic` $5d+1$ $5d+2$ $5d+3$ $5d+4$ $5d+5$
`feed_forward`	`True`

TABLE VII
NEAT OTHER HYPERPARAMETERS.

Category	**Hyperparameter**	**Value**
`DefaultSpeciesSet`	`compat_threshold`	3.0
`DefaultStagnation`	`species_fitness_func`	`max`
`DefaultStagnation`	`max_stagnation`	20
`DefaultReproduction`	`elitism`	2
`DefaultReproduction`	`survival_threshold`	0.2

[5] R. Ward and R. Ward, "2006 special issue: Cognitive conflict without explicit conflict monitoring in a dynamical agent," *Neural Networks*, vol. 19, no. 9, pp. 1430–1436, 2006.

[6] R. D. Beer, "The dynamics of active categorical perception in an evolved model agent," *Adaptive Behavior*, vol. 11, no. 4, pp. 209–243, 2003.

[7] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[8] S. Brave, "The evolution of memory and mental models using genetic programming," *Genetic Programming*, pp. 261–266, 1996.

[9] C. Ollion, T. Pinville, and S. Doncieux, "With a little help from selection pressures: evolution of memory in robot controllers," in *ALIFE 2012: The Thirteenth International Conference on the Synthesis and Simulation of Living Systems*. MIT Press, 2012, pp. 407–414.

[10] R. Beckers, O. E. Holland, and J. L. Deneubourg, "From local actions to global tasks: Stigmergy and collective robotics," in *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, R. Brooks and P. Maes, Eds. The MIT Press, 1994, pp. 181–189.

[11] G. Theraulaz and E. Bonabeau, "A brief history of stigmergy," *Artificial life*, vol. 5, no. 2, pp. 97–116, 1999.

[12] J. T. Carvalho and S. Nolfi, "Cognitive offloading does not prevent but rather promotes cognitive development," *PloS one*, vol. 11, no. 8, p. e0160679, 2016.

[13] D. Whitley, "Genetic reinforcement learning for neurocontrol problems," *Machine Learning*, vol. 13, no. 2-3, pp. 259–284, 1993.

[14] F. Gruau, D. Whitley, and L. Pyeatt, "A comparison between cellular encoding and direct encoding for genetic neural networks," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Cambridge, MA: The MIT Press, 1996, pp. 81–89.

[15] X. Yao, "Evolving artificial neural networks," in *Proceedings of the IEEE*, vol. 87, no. 9, 1999, pp. 1423–1447.

[16] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002.

TABLE VIII
SHARED NEAT GENOME HYPERPARAMETERS (DEFAULTGENOME).

Hyperparameter	**Value**
`num_hidden`	0
`compatibility_disjoint_coefficient`	1.0
`compatibility_weight_coefficient`	1.0
`conn_add_prob`	0.40
`conn_delete_prob`	0.05
`node_add_prob`	0.40
`node_delete_prob`	0.05
`activation_default`	`tanh`
`activation_options`	`tanh`
`activation_mutate_rate`	0.02
`aggregation_default`	`sum`
`aggregation_options`	`sum`
`aggregation_mutate_rate`	0.0
`bias_init_mean`	0.0
`bias_init_stdev`	1.0
`bias_replace_rate`	0.1
`bias_mutate_rate`	0.7
`bias_mutate_power`	0.5
`bias_max_value`	30.0
`bias_min_value`	-30.0
`response_init_mean`	1.0
`response_init_stdev`	0.0
`response_replace_rate`	0.0
`response_mutate_rate`	0.0
`response_mutate_power`	0.0
`response_max_value`	30.0
`response_min_value`	-30.0
`weight_max_value`	30
`weight_min_value`	-30
`weight_init_mean`	0.0
`weight_init_stdev`	1.0
`weight_mutate_rate`	0.8
`weight_replace_rate`	0.1
`weight_mutate_power`	0.5
`enabled_default`	`True`
`enabled_mutate_rate`	0.01

TABLE IX
COMPUTING RESOURCES

CPU	Apple M1 Pro 8-Core
GPU	Not used
RAM	32GB
OS	macOS 12.2
Language	Python 3.9.7
Libraries	NEAT Python 0.92
Run time	DROPPER $\sim$ 1 hr, TDNN $\sim$ 3.5 hrs, RNN $\sim$ 5 hrs for 100 generations

[17] M. Müller, "Dynamic time warping," *Information retrieval for music and motion*, pp. 69–84, 2007.

[18] M. Herrmann, C. W. Tan, and G. I. Webb, "Parameterizing the cost function of dynamic time warping with application to time series classification," *Data Mining and Knowledge Discovery*, pp. 1–22, 2023.

[19] J. Blynel and D. Floreano, "Exploring the t-maze: Evolving learning-like robot behaviors using ctrnns," in *Workshops on applications of evolutionary computation*. Springer, 2003, pp. 593–604.

[20] J. R. Chung, "Evolution of memory in reactive artificial neural networks," Ph.D. dissertation, Texas A&M University, 2011. [Online]. Available: https://www.proquest.com/docview/1037338991

[21] S. Kaliyur, "Evolved neural network memory strategies," Undergraduate Research Scholars Thesis, Texas A&M University, 2023.

[22] S. Mahato, "Comparison of evolved memory mechanisms in neural networks," Master's thesis, Texas A&M University, 2023.