Neuroevolution of a Modular Memory-Augmented Neural Network for Deep Memory Problems

Shauharda Khadka khadkas@oregonstate.edu
Oregon State University, Corvallis, 97330, USA

Jen Jen Chung jenjen.chung@mavt.ethz.ch
Eidgenössische Technische Hochschule Zürich, Zürich, 8092, Switzerland

Kagan Tumer kagan.tumer@oregonstate.edu
Oregon State University, Corvallis, 97330, USA

Abstract

We present Modular Memory Units (MMUs), a new class of memory-augmented neural network. MMU builds on the gated neural architecture of Gated Recurrent Units (GRUs) and Long Short Term Memory (LSTMs), to incorporate an external memory block, similar to a Neural Turing Machine (NTM). MMU interacts with the memory block using independent read and write gates that serve to decouple the memory from the central feedforward operation. This allows for regimented memory access and update, administering our network the ability to choose when to read from memory, update it, or simply ignore it. This capacity to act in detachment allows the network to shield the memory from noise and other distractions, while simultaneously using it to effectively retain and propagate information over an extended period of time. We train MMU using both neuroevolution and gradient descent, and perform experiments on two deep memory benchmarks. Results demonstrate that MMU performs significantly faster and more accurately than traditional LSTM-based methods, and is robust to dramatic increases in the sequence depth of these memory benchmarks.

Keywords

RNNS, Memory-Augmented Neural Networks, Neuroevolution

1 INTRODUCTION

Remembering key features from information observed in the past, and using it towards future decision making is a crucial part of most living organisms that exhibit intelligent adaptive behavior. Agents that can use memory to identify and retain variable-scale temporal patterns dynamically, and retrieve them in the future have a distinct advantage in most real world scenarios where observations in the past affect future decision making (Baddeley and Hitch, 1974; Grabowski et al., 2010). The neuroscience literature has increasingly emphasized the importance of explicit working memory as a pillar for intelligent behavior (Hassabis et al., 2017; Tulving, 2002).

The most prominent method of incorporating memory into a neural network is through Recurrent Neural Networks (RNNs), a class of neural networks that store information from the past within hidden states of the network. Long short term memory (LSTM) (Hochreiter and Schmidhuber (1997)) is a type of RNN that uses gated

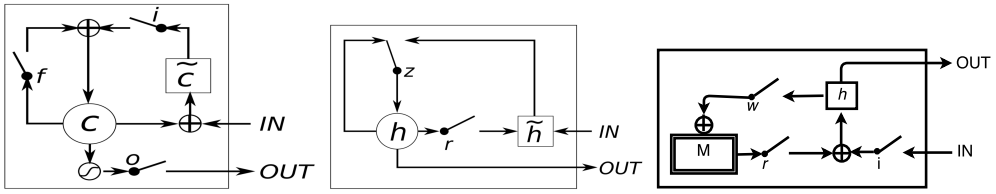
connections to stabilize the back-flow of error during training, enabling application to long temporal sequences. The gated recurrent unit (GRU) (Cho et al. (2014)) is a consequently proposed simplification of the LSTM architecture which uses fewer gated connections enabling easier training. Both GRU and LSTM have been widely applied to a range of domains and represent the state-of-the-art in many sequence processing tasks (Graves et al. (2013b); Sundermeyer et al. (2014)).

However, a common structure across all existing RNN methods is the intertwining of the central feedforward computation of the network and its memory content. This is because the memory is stored directly within the hidden states (GRU) or the cell state (LSTM). Since the output is a direct function of these hidden states/cell state, the memory content is tied to be updated alongside its output. An alternative approach is to record information in an external memory block, which is the basic idea behind Neural Turing Machines (NTMs) (Graves et al. (2014)) and Differentiable Neural Computers (DNCs) (Graves et al. (2016)). Although they have been successfully applied to solve some complex structured tasks (Greve et al., 2016), these networks are typically quite complex and unwieldy to train (Jaeger (2016)). NTMs and DNCs use "attention mechanisms" to handle the auxiliary tasks associated with memory management, and these mechanisms introduce their own set of parameters further exacerbating this difficulty.

The key contribution of our work is to introduce a new memory-augmented network architecture called Modular Memory Unit (MMU), which unravels the memory and central computation operations but does not require the costly memory management mechanisms of NTMs and DNCs. MMU borrows closely from the GRU, LSTM and NTM architectures, inheriting the relative simplicity of a GRU/LSTM structure while integrating a memory block which is selectively read from and written to in a fashion similar to an NTM. MMU adds a write gate that filters the interactions between the network's hidden state and the updates to its memory content. The effect of this is to decouple the memory content from the central feedforward operation within the network. The write gate enables refined control over the contents of the memory, since reading and writing are now handled by two independent operations, while avoiding the computational overhead incurred by the more complex memory management mechanisms of an NTM.

The MMU topology does not place any constraints on the network training procedure. Furthermore, the modular decomposition of the sub-structures within the MMU architecture makes it an ideal candidate for training via neuroevolutionary methods. In our experiments, we compared the learning performance of the MMU with the current state-of-the-art LSTM-based architecture on two deep memory benchmark domains, a sequence classification task and a sequence recall task (Rawal and Miikkulainen (2016)). In both domains, MMU demonstrated superior performance with fewer training generations, and was robust to dramatic increases in the depth of these tasks. Additionally, results also demonstrate that MMU is not only limited to learning input-output mappings, but is able to learn general methods in solving these benchmarks.

This paper builds on (Khadka et al., 2017) which introduces the Gated Recurrent Unit with Memory Block (GRU-MB) architecture. MMUs, introduced in this paper is a generalization of the GRU-MB architecture, that detaches the size of memory from the size of hidden nodes. Additionally, in this paper we implement a fully differentiable version of the MMU. We the compare and contrast the training performance of MMU with gradient descent and neuroevolution. Results demonstrate that gradient descent has comparative advantage over neuroevolution for lower depths, while neuroevolution significantly outperforms gradient descent as the depth of the temporal sequence



(a) LSTM(Chung et al., 2014) (b) GRU(Chung et al., 2014) (c) MMU

Figure 1: Comparative illustration of information flow in LSTM, GRU and MMU. (a) i, f and o are the input, forget and output gates, respectively while c and $\tilde{c}$ represents the current memory cell and candidate memory cell content. (b) r and z are the reset and update gates, while h and $\tilde{h}$ are the activation and the candidate activations (Chung et al., 2014). (c) i, r and w are the input, read and write gates, respectively while h and M denote the hidden activation and memory content. Note that for LSTM and GRU, the information passage from input to output is constrained to go through memory: hidden activation h in GRU and memory cell c in LSTM. However, this is not the case for MMU as its decoupled topology enables selective memory access: choice of reading from it, writing to it, or simply ignoring it.

grows. Additionally, neuroevolution is more repeatable and generalizable across both tasks. Further, a detailed case study probing the operation of various gates within the MMU's architecture is performed to elucidate the inner workings of MMU's memory handling protocols. We probe the limits of the MMU's memory representations and derive an approximate pseudo-algorithm that it uses to solve the tasks.

2 BACKGROUND AND RELATED WORK

RNNs can roughly be divided into two groups: those that store memory internally within the hidden nodes of the network, and those that use an external memory block with auxiliary structures to manage the associated housekeeping tasks. As described previously, current examples of networks classified in the latter are normally trained using a strong learning signal and have also relied on the differentiability of each component in the network in order to efficiently perform backpropagation (Graves et al. (2014, 2016)). In this work, we are interested in solving problems where only a weak learning signal is available, thus neuroevolutionary methods are more suitable to solving these types of problems. While it is possible to learn NTMs and DNCs via evolutionary methods (Merrild et al., 2018; Greve et al., 2016; Lüders et al., 2017), they are difficult to train reliably and efficiently, in part due to the large number of parameters characterizing their computational framework.

A standard RNN operates by recursively applying a transition function to its internal hidden states at each computation over a sequence of inputs. Although this structure can theoretically capture any temporal dependencies in the input data, in practice, the error propagation over time can become problematic over long sequences. This becomes apparent when training via gradient-based methods, where the back-flow of error can cause the gradient to either vanish or explode (Hochreiter (1998); Goodfellow et al. (2016)). The introduction of gated connections in an LSTM (Hochreiter and Schmidhuber (1997)) was aimed at directly addressing this issue.

2.1 LONG SHORT TERM MEMORY

The general concept of using gated connections is to control the flow of information through the hidden states of an RNN. Gates are neural structures that selectively filter information that passes through them. They use a sigmoidal activation and output within the range $[0, 1]$. They then operate element wise multiplicatively on information that passes through them, enabling selective filtering of information. Since its inception two decades ago, a number of gated RNN architectures have been proposed and most of these have been variants on the LSTM structure (Greff et al. (2016)). A typical LSTM unit consists of a memory cell and the associated *input*, *forget*, and *output* gates. The rate of change to the memory content stored in the cell is regulated by the input and forget gates, while the output gate controls the level of exposure.

LSTMs have been very successful at solving sequence-based problems such as analysis of audio (Graves and Schmidhuber (2005)) and video (Srivastava et al. (2015)) data, language modeling Sundermeyer et al. (2012) and speech recognition (Graves et al. (2013a)). Traditionally LSTMs have been trained via backpropagation through time (Werbos (1990)), however more recent applications of LSTMs have demonstrated that they can be trained via evolutionary methods (Bayer et al. (2009)).

Most recently, Rawal and Miikkulainen (2016) proposed a method combining NEAT (Neuroevolution of Augmenting Topologies) (Stanley and Miikkulainen (2002)) with LSTM units. NEAT-LSTM allows the NEAT algorithm to discover the required number of memory units to best represent the sequence-based task. In order to overcome scalability issues as the memory requirements of the task increased, the authors also proposed using two training phases. The "pre-training" phase evolves the LSTM networks using an information objective, the goal of which is to discover the network topology that would encapsulate the maximally independent features of the task. Following this, the stored features could then be used to solve the memory task itself. The results in Rawal and Miikkulainen (2016) showed that NEAT-LSTM can be trained to solve memory tasks using only a weak signal with a graceful decay in performance over increased temporal dependencies.

2.2 GATED RECURRENT UNIT

The GRU was first proposed in Cho et al. (2014) as an alternative to the LSTM unit with a simpler gating architecture. Compared to the multiple gate connections in an LSTM, the GRU only contains *reset* and *update* gates to modulate the data stored in the hidden unit. It has been empirically shown to improve upon the performance of LSTMs in various time-sequence domains (Jozefowicz et al. (2015); Chung et al. (2014)) and can also be incorporated into hierarchical RNN structures (El Hihi and Bengio (1995)) to simultaneously capture time-sequence memory of varying window sizes (Chung et al. (2015)).

The reset gate r and update gate z are computed by

$$r = \sigma\left(\mathbf{W}_r\mathbf{x} + \mathbf{U}_r\mathbf{h}^{\langle t-1\rangle}\right), \tag{1}$$

$$z = \sigma\left(\mathbf{W}_z\mathbf{x} + \mathbf{U}_z\mathbf{h}^{\langle t-1\rangle}\right), \tag{2}$$

where σ is the logistic sigmoid function, $\mathbf{W}$ and $\mathbf{U}$ are the learned weight matrices, and $\mathbf{x}$ and $\mathbf{h}^{\langle t-1\rangle}$ are the input and the previous hidden state, respectively. The activation of an individual hidden unit h_j is then computed by

$$h_j^{\langle t\rangle} = z_j h_j^{\langle t-1\rangle} + (1 - z_j)\,\tilde{h}_j^{\langle t\rangle}, \tag{3}$$

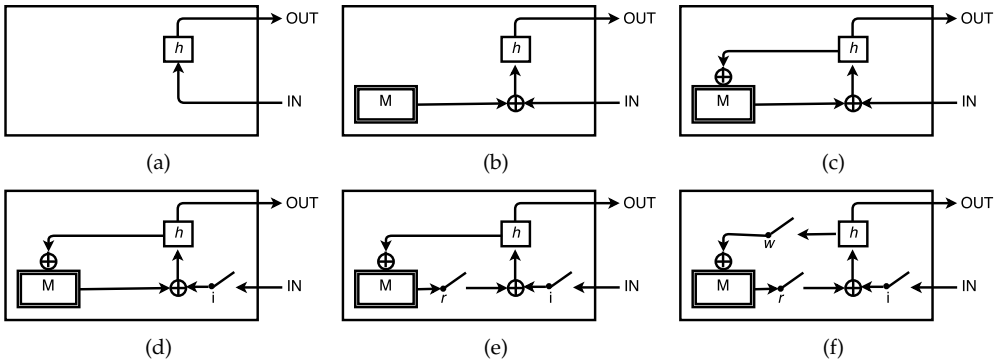


Figure 2: Illustration of a bottom up construction of a MMU: (a) We start with a simple feedforward neural network, whose hidden activations are marked as h. (b) We add a disconnected memory block, which now contributes information to the hidden activation. (c) We close the loop, and allow the hidden activation to modulate the memory block. (d) We add the input gate, which acts as a filter on information flow into the network. (e) We add a read gate, which filters the amount of information read from memory at each step. (f) Finally, we introduce a write gate, which controls the information flow that modulates the content within the memory block.

where

$$\tilde{h}_j^{\langle t \rangle} = tanh\left(\left[\mathbf{W}\mathbf{x} \right]_j + \left[\mathbf{U}\left(\mathbf{r} \odot \mathbf{h}^{\langle t-1 \rangle} \right) \right]_j \right). \tag{4}$$

The function of the reset gate is to allow the hidden unit to flush information that is no longer relevant, while the update gate controls the amount of information that will carry over from the previous hidden state.

With this gating structure, the reset and update operations of the GRU are activated during each feedforward computation of the network. Moreover, this means that accessing and editing the hidden unit become inherently tied into a single operation. In our work, we propose an architecture that builds upon the GRU by introducing separate read and write gates to filter the information flow between the memory storage and feedforward components of the network. This network structure enables greater control over memory access and update which can ultimately improve learning performance on tasks with long term dependencies. Figure 1 illustrates the core difference in neural architecture between LSTM, GRU and MMU.

3 MODULAR MEMORY UNIT

We introduce a new memory-augmented neural network architecture called Modular Memory Unit (MMU), which is designed to tackle the problem of solving deep memory tasks using only a weak signal. MMU uses input, read and write gates, each of which contribute to filtering and directing information flow within the network. The read and write gates control the flow of information into and out of the memory block, and effectively modulate its interaction with the hidden state of the central feedforward operation. This feature allows the network to have finer control over the contents of the memory block, reading and writing via two independent operations. Figure 2 details the bottom-up construction of the MMU neural architecture.

Each of the gates in the MMU neural architecture uses a sigmoid activation whose output ranges between 0 and 1. This can roughly be thought of as a filter that modulates

the part of the information that can pass through. Given the current input x^t, memory cell's content m^{t-1} and last output y^{t-1}, the input gate i^t and intermediate block input p^t is computed as

$$\begin{aligned} i^t &= \sigma\left(K_i x^t + R_i y^{t-1} + N_i m^{t-1} + b_i\right) && \textit{Input gate} \\ p^t &= \phi\left(K_p x^t + N_p m^{t-1} + b_p\right) && \textit{Block input} \end{aligned} \tag{5}$$

Here, K, R, N, Z represent trainable weight matrices while b represents the trainable bias weights. σ represents the sigmoidal activation function while ϕ and θ represents any non-linear activation function. Next, the read gate r^t and decoded memory d^t is computed as:

$$\begin{aligned} r^t &= \sigma\left(K_r x^t + R_r y^{t-1} + N_r m^{t-1} + b_r\right) && \textit{Read gate} \\ d^t &= \theta\left(N_d m^{t-1} + b_d\right) && \textit{Memory decoder} \end{aligned} \tag{6}$$

The hidden activation is then computed by combing the information from the environment filtered by the input gate and information stored in memory filtered by the read gate.

$$h^t = r^t \odot d^t + p^t \odot i^t \qquad \textit{Hidden activation} \tag{7}$$

The hidden activation is then encoded to the dimensionality of the memory cell using the memory encoder f^t, filtered using the write gate w^t and is then used to update the memory cell content m^t. α is a hyperparameter that controls the nature of the memory update: ranging from a strictly cumulative update ($\alpha = 0$) to an interpolative update ($\alpha = 1$). w^t is used to interpolate between the last memory content and the new candidate content given by f^t.

$$\begin{aligned} w^t &= \sigma\left(K_w x^t + R_w y^{t-1} + N_w m^{t-1} + b_w\right) && \textit{Write gate} \\ f^t &= \theta\left(Z_f h^t + b_f\right) && \textit{Memory encoder} \\ m^t &= (1-\alpha)\left(m^{t-1} + w^t \odot f^t\right) + \alpha\left(w^t \odot f^t + \left(1 - w^t\right) \odot m^{t-1}\right) && \textit{Memory update} \end{aligned} \tag{8}$$

The new output y^t is then computed as

$$y^t = \phi\left(Z_y h^t + b_y\right) \qquad \textit{New output} \tag{9}$$

Equations (5)-(9) specify the computational framework that constitute the MMU neural architecture. Qualitatively, a MMU can be thought of as a standard feedforward neural network with five major additions.

1. **Input Gate:** The input gate filters the flow of information from the environment to the network. This serves to shield the network from the noisy portions of each incoming observation and allows it to focus its attention on relevant features within its observation set.
2. **Memory Decoder:** The memory decoder serves to decode the contents of memory to be interpreted by the network's central computation. Specifically, the decoder transcribes the memory's contents to a projection amenable to incorporation with the network's hidden activations. This decouples the size of the network's hidden activations to the size of the external memory.

3. **Selective Memory Read:** The network's input is augmented with content that the network selectively reads from the decoded external memory. The network has an independent read gate which filters the content decoded from external memory. This serves to shape the contents of memory as per the needs of the network, protecting it from being overwhelmed with noisy information which might not be relevant for the timestep.
4. **Memory Encoder:** The memory encoder serves to encode the contents of the network's central computation so that it can be used to update the external memory. Specifically, the encoder transcribes the network's hidden activations to a projection amenable to updating the external memory.
5. **Selective Memory Write:** The network engages a write gate to selectively update the contents of the external memory. This gate allows the network to report salient features from its observations to the external memory at any given time step. The write gate that filters this channel of information flow serves to shield the external memory from being overwhelmed by updates from the network.

A crucial feature of the MMU neural architecture is its separation of the memory content from the central feedforward operation (Fig. 2). This allows for regimented memory access and update, which serves to stabilize the memory block's content over long network activations. **The network has the ability to effectively choose when to read from memory, update it, or simply ignore it. This ability to act in detachment allows the network to shield the memory from distractions, noise or faulty inputs; while interacting with it when required, to remember and process useful signals.** This decoupling of the memory block from the feedforward operation fosters reliable long term information retention and retrieval.

The memory encoder and decoder collectively serve to decouple the information representations held within the external memory, and the representations within the hidden activations of the MMU network. This allows separation between the volume of computation required for reactive decision making (processing the current input) and the volume of information that needs to be retained in memory for future use.

For example, consider a recurrent convolutional network processing video input to count the number of frames where a dog appeared. For each frame (element in sequence), a vast number of features have to be extracted and processed to classify whether a dog appeared or not. This requires a large number of hidden activations, often in the scale of thousands. However, the volume of information that has to be retained in memory for future use in this task is simply the current running count of the number of frames where the dog appeared. This is significantly smaller when compared to the volume of information that has to be fed forward through the network's hidden activations for instance classification. Additionally, the representation of information between computing image features, and keeping a running count can be very different. Constraining the size and representation of these two modes of information is undesirable. The memory encoder and decoder collectively address this issue by decoupling the size and representation between memory and hidden activations.

In this work, we test the MMU's efficacy in retaining useful information across long temporal activations, while simultaneously shielding it from noise. Our focus here is to test the MMU's capability in dealing with the difficulties defined by the depth of the temporal sequence (number of timesteps the information has to selectively retained for), rather than the volume of the information that has to be propagated in memory. The tasks we use to test our MMU network is designed to challenge this axis of diffi-

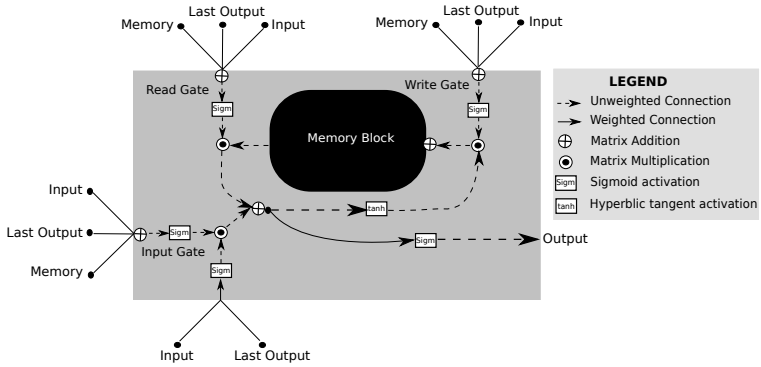


Figure 3: MMU neural architecture detailing the weighted connections and activations used in this paper. *Last Output* represents the recurrent connection from the previous network output. *Memory* represents the peephole connection from the memory block.

culty. The MMU network is designed to be a highly reconfigurable and modular framework for memory-based learning. We exploit this reconfigurability and set a prior for our network to expedite training. We set α to 0 while setting the memory encoder (f^t) and memory decoder (d^t) from Equation 6 and Equation 8 as identity matrices, and freeze their weights from training. This virtually eliminates the decoding and encoding operation from our MMU network which we deem surplus to requirements for the depth-centered experimentation in this paper. Figure 3 shows the detailed schematic of the MMU instance.

Initialize a population of k neural networks
Define a random number generator $r()$ with output $\in [0, 1)$
foreach *Generation* **do**
  foreach *Network* **do**
    Compute fitness
  Rank the population based on fitness scores
  Select the first e networks as elites where e = int(ψ*k)
  Select $(k - e)$ networks from population, to form Set S using tournament selection
  foreach *Network* $N \in$ *Set* S **do**
    foreach *weight matrix* $W \in N$ **do**
      if $r() < mut_{prob}$ **then**
        Randomly sample and perturb weights in W with 10% Gaussian noise

Algorithm 1: Neuroevolutionary algorithm used to evolve the MMU network

Training We use neuroevolution to train our MMU network. However, memory-augmented architectures like the MMU, can be a challenge to evolve, particularly due to the highly non-linear and correlated operations parameterized by its weights. To address this problem, we decompose the MMU architecture into its component weight matrices and implement mutational operations in batches within them. Algorithm 1 details the neurovolutionary algorithm used to evolve our MMU architecture. The size the population k was set to 100, and the fraction of elites ψ to 0.1.

4 EXPERIMENT 1: SEQUENCE CLASSIFICATION

Our first experiment tests the MMU architecture in a Sequence Classification task, which has been used as a benchmark task in previous works (Rawal and Miikkulainen (2016)). Sequence Classification is a deep memory classification experiment where the network needs to track a classification target among a long sequence of signals interleaved with noise. The network is given a sequence of 1/-1 signals, interleaved with a variant number of 0's in between. After each introduction of a signal, the network needs to decide whether it has received more 1's or -1's. The number of signals (1/-1s) in the input sequence is termed the depth of the task. A key difficulty here are distractors (0's) that the network has to learn to ignore while making its prediction. This is a difficult task for a traditional neural network to achieve, particularly as the length of the sequence (depth) gets longer.

Note that this is a sequence to sequence classification task with a **strict success criteria**. In order to successfully classify a sequence, the network has to output the correct classification at each depth of the sequence. For instance, to correctly classify a 10-deep sequence, the network has to make the right classification at each of the intermediary depths: 1,2,..,9 and finally 10. If any of the intermediary classification are incorrect, the entire sequence classification is considered incorrect.

Input Sequence	Target Output
-1 0....0 1 0....0 -1	-1 .... 1 .... -1
-1 0....0 -1 0....0 1	-1 .... -1 .... -1
-1 0....0 1 0....0 1	-1 .... 1 1
1 0....0 1 0....0 -1	1 .... 1 1

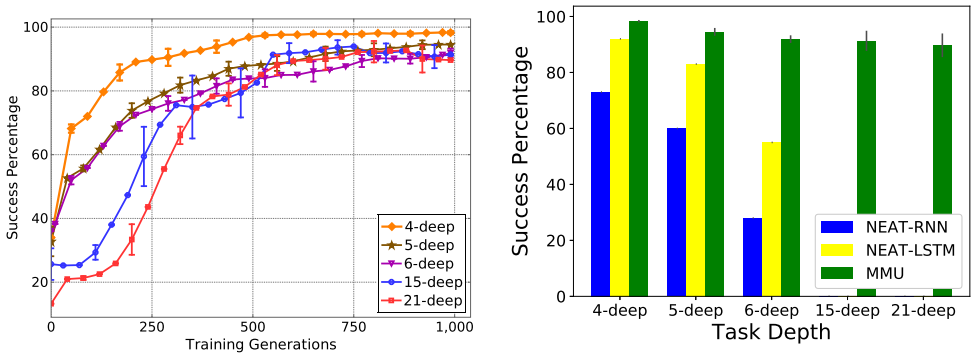
Figure 4: Sequence Classification: The network receives a sequence of input signals interleaved with noise (0's) of variable length. At each introduction of a signal (1/-1), the network must determine whether it has received more 1's or -1's. MMU receives an input sequence and outputs a value between [0,1], which is translated as -1 if the value is between [0,0.5) and 1 if between [0.5,1].

Further, the number of distractors (0's) is determined randomly following each signal, and ranges between 10 and 20. This variability adds complexity to the task, as the network cannot simply memorize when to pay attention and when to ignore the incoming signals. The ability to filter out distractions and concentrate on the useful signal, however, is a key capability required of an intelligent decision-making system. Mastering this property is a necessity in expanding the integration of memory-augmented agents in myriad decision-making applications. Figure 4 describes the Sequence Classification task and the input-output setup to our MMU network.

4.1 RESULTS

During each training generation, the network with the highest fitness was selected as the champion network. This network was then tested on a separate test set of 50 experiments, generated randomly for each generation. The percentage of the test set that the champion network solved completely was then logged as the success percentage and reported for that generation. Ten independent statistical runs were conducted, and the average with error bars reporting the standard error in the mean are shown.

Figure 5a shows the success rate for the MMU neural architecture for varying depth experiments. MMU is able to solve the Sequence Classification task with high accuracy and fast convergence. The network was able to achieve accuracies greater than 80% within 500 training generations. The performance also scales gracefully with



(a) (b)

Figure 5: (a) Success rate for the Sequence Classification task of varying depth using MMU. (b) Comparison of MMU with NEAT-RNN and NEAT-LSTM (extracted from Rawal and Miikkulainen (2016)). NEAT-RNN and NEAT-LSTM results are based on 15,000 generations with a population size of 100 and are only available for task depth up to 6. MMU results are based on 1,000 generations with the same population size and tested for task depths of up to 21.

the depth of the task. The 21-deep task that we introduced in this paper has an input series length ranging between $\{221, 441\}$, depending on the number of distractors (0's). MMU is able to solve this task with an accuracy of $87.6\% \pm 6.85\%$. This demonstrates MMU's ability to capture long term time dependencies and systematically handle information over long time intervals.

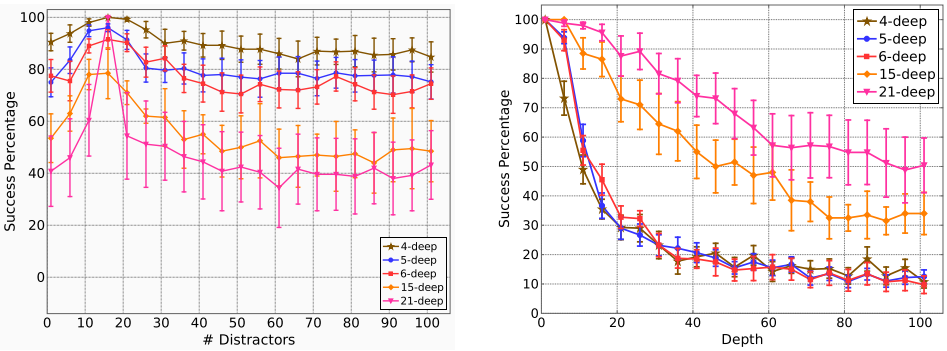
To situate the results obtained using the MMU architecture, we compare them against the results from Rawal and Miikkulainen (2016) for NEAT-LSTM and NEAT-RNN (Fig. 5b). These results for NEAT-LSTM and NEAT-RNN represent the success percentage after 15,000 generations while the MMU results represent the success percentage after 1,000 generations. MMU demonstrates significantly improved success percentages across all task depths, and converges in 1/15th of the generations.

4.2 GENERALIZATION PERFORMANCE

Section 4.1 tested the networks for the ability to generalize to new data using test sets. In this section, we take this notion a step further and test the network's ability to generalize, not only to new input sequences, but to tasks with longer depths or noise sequences than what they were trained for.

4.2.1 GENERALIZATION TO NOISE

Figure 6(a) shows the success rate achieved by the networks from the preceding section when tested for a varying range of noise sequences (number of distractors), extending up to a length of 101. All the networks tested were trained for distractor sequence length between 10 and 20. This experiment seeks to test the generalizability of the strategy that the MMU network has learned to deal with noise. Overall, the networks demonstrate a decent ability for generalization towards varying length of noise sequences. The network trained for a task depth of 21, was able to achieve $43.2\% \pm 13.2\%$ success when tested with interleaving noise sequences of length 101. This length of noise sequence results in the total input sequence length of 2041, which is an extremely long sequence to process. The peak performance across all the task depths is seen be-



(a) (b)

Figure 6: (a) MMU network's generalization to an arbitrary number of distractors (length of interleaving noise sequences) within the task. All MMUs shown were trained for tasks of their respective depths with variable length of distractors (noise) randomly sampled between 10 and 20. These were then tested for interleaving noise sequences ranging from 0 to 101. (b) MMU network's generalization to arbitrary depths of the task. We tested MMU networks across a range of task depths that they were not originally trained for. The legend indicates the task depth actually used during training.

tween noise sequences of length between 10 and 20. This is expected as this is the length of noise sequences that the networks were trained for. The performance degrades as we vary the length of the noise sequence away from the $10-20$ range. However, this degradation in performance is graceful and settles to an **equilibrium** whereafter increases in noise sequence length do not affect performance. This equilibrium performance reflects the core generalizability of the MMU's strategy to deal with noise.

Interestingly, decreasing the length of the noise sequence leads to virtually the same rate of degradation as increasing it. This suggests that the loss of performance suffered by MMU when tested with noise sequence lengths beyond its training, is caused more by the MMU's specialization to the expected 10-20 range, rather than its inability to process variable length noise sequences. In other words, the MMU is exploiting the consistency of the noise sequence lengths being between 10-20. Additionally, the equilibrium performance that the MMU network settles on, also seem to be higher for tasks with lower depths. The severity in loss of generalization to lengths of noise sequences grows with the depth of the task. This can be attributed to the sheer volume of additional noise added. For example, a 4-deep task consists 4 signals interleaved with 3 noise sequences while a 21-deep task consists of 21 signals interleaved with 20 noise sequences. An increase in the length of each of the noise sequences thus introduces significantly more noise to the 21-deep case when compared with the 4-deep case.

4.2.2 GENERALIZATION TO DEPTH

Figure 6(b) shows the success rate achieved by the networks from the preceding section when tested for a varying range of task depths, extending up to a depth of 101. Overall, the networks demonstrate a strong ability for generalization. The network trained for a task depth of 21, was able to achieve $50.4\% \pm 9.30\%$ success in a 101 deep task. This is a task with an extremely long input sequence ranging from $\{1111, 2121\}$. This shows that MMU is not simply learning a mapping from an input sequence to an output classification target sequence, but a specific method to solve the task at hand.

Interestingly, training MMU on longer depth tasks seem to disproportionately improve its generalization to even larger task depths. For example the network trained on a 5-deep task achieved $36.7\% \pm 4.32\%$ success in a 16-deep task which is approximately thrice as deep. A network trained trained on the 21-deep task, however achieved $56.4\% \pm 10.93\%$ success in its corresponding 66-deep task (approximately thrice as deep). This shows that, as the depth of the task grows, the problem becomes increasingly complex for the network to solve using just the mapping stored within its weights. Even after ignoring the noise elements, a task depth of 5 means discerning among 2^5 possible permutations of signal, whereas a depth of 21 means discerning among 2^{21} permutations. Therefore, training a network on deeper tasks forces it to learn an algorithmic representation, akin to a 'method', in order to successfully solve the task. This lends itself to better generalization to increasing task depths.

4.3 CASE STUDY

The selective interaction between memory and the central feedforward operation of the network is the core component of the MMU architecture. To investigate the exact role played by the external memory, we performed a case study to probe the memory contents as well as the protocols for reading and writing.

We analyzed evolved MMU networks and identified one particular network within the stack trained on the 21 deep task. This specific individual network exhibited perfect generalization (100% success) to all but the 96-deep and 101-deep task sets, in which it achieved 98.0% and 96.0% respectively. Additionally, this network also achieved perfect generalizability to increasing the length of noise sequences, achieving 100% success for all noise sequence lengths. We will refer to this specific network as **21-champ** and use this network to perform case studies at multiple levels of abstraction. First, we perform a case study for a specific depth of task, highlighting all of its possible configurations (sequence of signals) and the corresponding final memory contents achieved while solving them. Secondly, we perform a finer study where we use a trained MMU network to solve a singular instance of a task, and investigate the magnitude of read-write interactions at every time step. Finally, we will test the memory representations that the network uses to encode its intermediate variables, operations that use them, and decode the pseudo algorithm that the network has seemingly learned.

4.3.1 TASK LEVEL STUDY

For every depth of the sequence classification task, there are a large but finite number of unique permutations for the input sequence. A primary contributor to this large possible set of input sequences is the variable number of distractors interleaved between signals. If we ignore the variable number of distractors (noise), the number of permutations of inputs for each task is simply 2^d where d is the depth of the task (1 or -1 for each signal). For example, in a 2-deep task we can have either of $\{[1,-1],[1,1],[-1,-1],[-1,1]\}$ as our input sequence, ignoring the distractors (0's) in between. We will refer to each of these unique input sequences as a **permutation**.

The task-level case study seeks to identify the distinctions within the memory content in response to different permutations of input sequences spanning a specific task depth. Since the number of unique permutations even after ignoring the variable number of distractors scales exponentially, we use a smaller task depth of 3 to conduct our analysis. This leads to 8 unique permutations of the input sequence. We set the number of distractors to a constant of 15 while testing to allow for a fair comparison.

The 21-champ network is able to solve all the permutations successfully. The 21-champ network has 5 units of memory, 3 of which it holds constant. However, this is

deceiving. We tested the marginal value of each memory unit to the network's operation by masking each memory unit. We found that although the network held 3 units constant and only updated 2, it needed 3 memory units to be able to classify all 8 permutations successfully. Out of the 3 memory units that the network held constant, a specific one was necessary to maintain the network's 100% success performance. Overall, the network required the second, third and fifth units (M2, M3, M5) to solve all 8 permutations. The network modified the values of M3 and M5 as it observed signals while M2 was held constant. The network is seemingly using M2 unit as a bias while the other two units encode the current state of the cumulative sum.

#	Permutation	Sum	Final Memory State	[M2, M3, M5]	Signature
1	[-1,-1,-1]	-3	[-0.33,-1.51, 2.72, 0.84, -0.80]	[-1.51, 2.72, -0.80]	[0,2,1]
2	[-1,-1,1]	-1	[-0.33,-1.51, 1.73, 0.84, -1.46]	[-1.51, 1.73, -1.46]	[0,2,1]
3	[-1,1,-1]	-1	[-0.33,-1.51, 1.73, 0.84, -1.46]	[-1.51, 1.73, -1.46]	[0,2,1]
4	[1,-1,-1]	-1	[-0.33,-1.51, 1.73, 0.84, -1.46]	[-1.51, 1.73, -1.46]	[0,2,1]
5	[-1,1,1]	+1	[-0.33,-1.51, 0.78, 0.84, -2.36]	[-1.51, 0.78, -2.36]	[2,0,1]
6	[1,-1,1]	+1	[-0.33,-1.51, 0.78, 0.84, -2.36]	[-1.51, 0.78, -2.36]	[2,0,1]
7	[1,1,-1]	+1	[-0.33,-1.51, 0.78, 0.84, -2.36]	[-1.51, 0.78, -2.36]	[2,0,1]
8	[1,1,1]	+3	[-0.33,-1.51, 0.01, 0.84, -3.34]	[-1.51, 0.01, -3.34]	[2,0,1]

Table 1: Final memory states for a fully trained MMU network across all unique permutations of the 3-deep Sequence classification task (number of distractors set to a constant value of 15).

Table 1 shows the results for running our 21-champ network across all 8 permutation of 3-deep sequence classification task (setting the number of distractors to a constant value of 15). The **sum** refers to the cumulative sum of real signals (input sequences). While comparing the final memory states explicitly against each other is extremely difficult, we observed some interesting patterns. The final memory state was entirely predictive of the cumulative sum of the corresponding input permutation. In other words, permutations with the same sum ended up with the same final relevant memory state regardless of the ordering of the signals in the input sequence. The final memory state is a coarse record of the cumulative central feedforward-memory interactions within that run. The unique mapping from cumulative sum to final memory state indicates that this coarse representation is predictive of the interactions that are necessary for representing that cumulative sum.

The two memory units that vary across different permutations (M3 and M5) also demonstrated a monotonic decrease in the magnitude of final activation value as the cumulative sum increased. Additionally, we sorted the three relevant final memory states (M2, M3 and M5) in ascending order and recorded the *sorted index* as its **signature**. The correct classification at any depth of this task depends on the cumulative sum. In other words, the permutations with the negative cumulative sum should be classified as -1 while the ones with a positive cumulative sum as 1. The memory signature observed at this depth is entirely predictive of this binary classification. The network seems to be interpreting the memory signature (relative values of the unit's activations) directly to make a classification.

Collectively, the network is using the specific magnitude of memory states to encode the magnitude of the cumulative sum, while using the memory state's relative relationship (signature) to make classification decisions. This is akin to tracking the cumulative sum and using its sign (whether positive or negative), to make classification decisions for each depth. This is an optimal policy to solve the Sequence classification

task, and is representative of what a human solver might perhaps do. The MMU seems to be executing this very policy using a distributed representation of the cumulative sum.

4.3.2 ROLE OF SELECTIVE MEMORY ACCESS

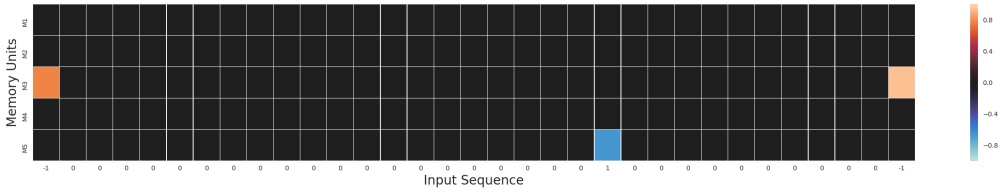


Figure 7: Illustration of the write interaction between memory and the central feedforward network in a 3-deep #3 permutation task from Table 1. The input sequence contains three signals [-1,1,-1] with 0s interleaved between them (x-axis of the heatmap). The color scheme corresponds to a scale of memory update magnitudes. Write activations in response to each individual input is shown for each of the five memory units.

We investigate the role of selective memory access facilitated by the read/write gates. Figure 7 illustrates the write interaction between memory and central feedforward part of the MMU network during the course of one experiment where the input sequence contains the signals [-1,1,-1] interleaved with variable-length distractors. A clear pattern emerges from the heatmap of the write activations. Memory Units M3 and M5 are the only units written to during the entire run. M3 and M5 are written to when a -1 and $+1$ is encountered, respectively. When a distractor is encountered, the memory is shut off and nothing is written to it. **This is demonstrative of the MMU's mechanism to shield memory from noisy activations, while selectively updating it when a signal is encountered.** The selective shielding of memory content from noise was consistent across all other depths (not shown here). This behavior seems to have evolved as a general protocol for dealing with the variable length distractors.

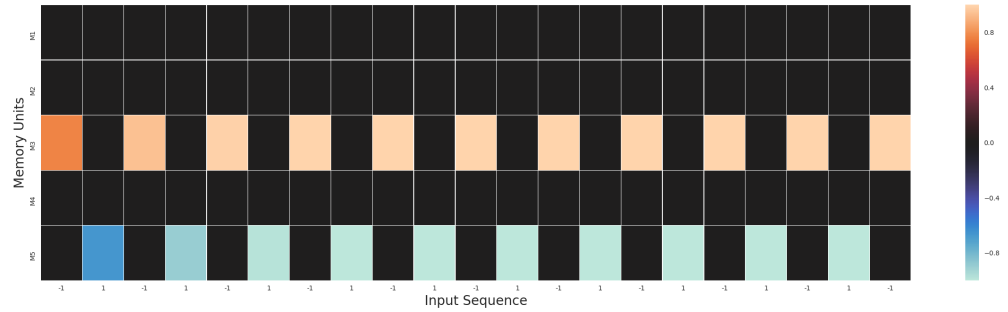


Figure 8: Illustration of the write interaction between memory and the central feedforward network in a 21-deep task with input sequence that followed the format [-1,1,-1,1,-1...] (flipping subsequent signal) interleaved with a variable number of 0s. The activations for the noise inputs (0s) are not shown but were all constantly 0s.

Figure 8 illustrates the write activations during the course of a 21-deep task instance. The input sequence in this task contain 21 signals in the format [-1,1,-1,1,-1....]. The pattern is that the signal starts with -1 and each subsequent signal flips the sign from its predecessor. The signals are interleaved with variable number of 0s ranging

between 10 and 20 as usual. The write activation for the part of the sequence with 0s (distractors) were consistently zero (similar to Figure 7) and were thus omitted from the plot. Memory Unit (M3) is written to when a -1 is encountered while M5 is written to when a $+1$ is encountered. This pattern is consistent across the entire sequence. This is indicative of the network's protocol for **memory consolidation**, tracking and retaining the number of times -1 and $+1$ is encountered, separately and independently.

4.3.3 MEMORY REPRESENTATION

The memory consolidation protocol demonstrates that the 21-champ network systematically updates specific memory units in response to specific inputs. M3 and M5 track the number of -1s and +1s observed, respectively. However, to successfully solve the sequence classification task, it is necessary to reliably represent the cumulative sum of the signals at any time. For example, in a 21-deep task, consider an extreme input sequence which has 10 counts of 1s followed by 11 counts of -1s (disregarding interleaving distractors). For this task, the correct classification sequence would be 1 for the first 20 depths and -1 for the 21st depth. This is because the cumulative sum in this task instance is always positive, except for the final depth. The cumulative sum is at its maximum value of +10 at depth 10, 0 at depth 20, and finally -1 at depth 21. To solve this task instance, it is necessary for the network to precisely represent the cumulative sum at any given timestep, discriminating between their magnitudes reliably.

#-1s	#1s	Memory State	M3	M5
1	0	[-0.33, -1.51, 0.78, 0.84, -0.80]	0.78	-0.80
2	0	[-0.33, -1.51, 1.72, 0.84, -0.80]	1.72	-0.80
5	0	[-0.33, -1.51, 4.72, 0.84, -0.80]	4.72	-0.80
100	0	[-0.33, -1.51, 99.72, 0.84, -0.80]	99.72	-0.80
1000	0	[-0.33, -1.51, 999.72, 0.84, -0.80]	999.72	-0.80
10000	0	[-0.33, -1.51, 1244.72, 0.84, -0.80]	1244.72	-0.80
100000	0	[-0.33, -1.51, 1244.72, 0.84, -0.80]	1244.72	-0.80
100000	1	[-0.33, -1.51, 1244.72, 0.84, -1.46]	1244.72	-1.46
100000	2	[-0.33, -1.51, 1244.72, 0.84, -2.36]	1244.72	-2.36
100000	5	[-0.33, -1.51, 1244.72, 0.84, -5.34]	1244.72	-5.34
100000	100	[-0.33, -1.51, 1244.72, 0.84, -100.34]	1244.72	-100.34
100000	1000	[-0.33, -1.51, 1244.72, 0.84, -1000.34]	1244.72	-1000.34
100000	10000	[-0.33, -1.51, 1244.72, 0.84, -10000.34]	1244.72	-10000.34
100000	1000000	[-0.33, -1.51, 1244.72, 0.84, -1000000.34]	1244.72	-1000000.34

Table 2: Memory states at varying counts of -1 and 1s fed within the input sequence

The 21-champ network represents the cumulative sum using M3 and M5, tracking the number of -1s and 1s observed, respectively. We tested the memory representation that was used to encode these respective counts. Interestingly, we found that the encoding M3 and M5 use is simply a slightly perturbed negation of the actual count. Table 2 shows the memory representation when the input sequence contains up to 100,000 counts of -1s, followed by 1,000,000 1s interleaved with variable number of 0s.

As shown in Table 2, M3 and M5 directly encode the number of -1 and 1 encountered, respectively. The reliability of this representation is remarkably robust. M3 was able to successfully count the number of -1s encountered upto 1,244 introductions, after which it saturated. M5 on the other hand demonstrated virtually indefinite generalizability, and was able to successfully count 1,000,000 introductions of 1s reliably.

The length of the temporal sequence at this stage was approximately 20,000,000 (including the interleaving 0s). This ability to represent count is particularly remarkable when we consider that the 21-champ network was trained exclusively on 21-deep tasks, whose input sequences can only generate cumulative sums between -21 and 21. The 21-champ network is instead able to represent counts ranging from -1244 to 1,000,000 (at least, but likely more). Additionally, representing counts of -1 and 1 encountered is not a specific objective the 21-champ was explicitly trained on. Instead, these abilities were developed by the 21-champ in order to solve the Sequence Classification task.

4.3.4 PSEUDO ALGORITHM

The finer grain investigation into the 21-champ MMU network has provided insight into its organization and operations that allows it to robustly and generally solve the Sequence Classification Task. We put all these insights together and derive an approximate **pseudo-algorithm** (shown in Algorithm 2) that the MMU network uses. M2 is used as a constant bias while M3 and M5 are used to count the -1 and 1s encountered. The classification decision at any time step is given by a function **f** that maps the values of M2, M3, and M5 to a binary output.

Three core components lie at the heart of 21-champ's successful operation. Firstly, the read and write gates serve to selectively access memory, blocking noisy activations (distractors) while allowing real signals to be registered. Secondly, selective memory access facilitates reliable representations within memory that are able to encode the count of each input encountered separately in two different memory units. Finally, the feedforward operation of the network is able to predict the correct binary classification at each time step by learning the function to infer from the memory states. These components, acting independently from each other collectively define a general strategy to solve the Sequence Classification Task.

```
foreach Input Sequence do
    foreach Item i in input do
        if i == -1 then
            M3 = M3 + 1̃
        else if i == 1 then
            M5 = M5 - 1̃
        else
            Block any update to memory
        Compute Classification C = f(M2, M3, M5)
```

Algorithm 2: Approximate pseudo-algorithm derived from the operations of the 21-champ MMU network

5 EXPERIMENT 2: SEQUENCE RECALL

For our second experiment, we tested our MMU architecture in a Sequence Recall task (Fig. 9), which is also sometimes referred to as a T-Maze navigation task. This is a popular benchmark task used extensively to test agents with memory (Bakker (2001); Charles et al. (2012); Rawal and Miikkulainen (2016)). In a simple Sequence Recall task, an agent starts off at the bottom of a T-maze and encounters an instruction stimulus (e.g sound). The agent then moves along a corridor and reaches a junction where the path splits into two. Here, the agent has to pick a direction and turn left or right, depending on the instruction stimulus it received at the start. In order to solve the task, the agent has to accurately memorize the instruction stimulus received at the beginning of the

trial, insulate that information during its traversal through the corridor, and retrieve it as it encounters a junction.

The simple Sequence Recall task can be extended to formulate a more complex deep Sequence Recall task (Rawal and Miikkulainen (2016)), which consist of a sequence of independent junctions (Fig. 9). Here, at the start of the maze, the agent encounters a series of instruction stimuli (e.g. a series of sounds akin to someone giving directions). The agent then moves along the corridor and uses the instruction stimuli (directions) in correct order at each junction it encounters to reach the goal at the end. The length of the corridors following each junction is determined randomly and ranges between 10 and 20 for our experiments. This ensures that the agent cannot simply memorize when to retrieve, use and update its stored memory, but has to react to arriving at a junction.

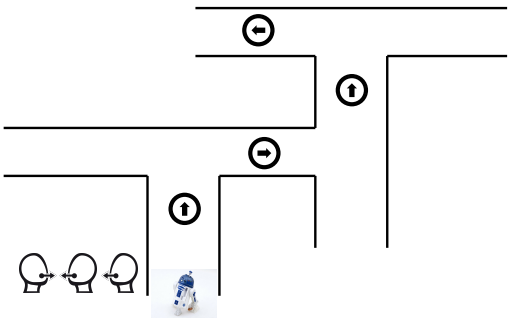


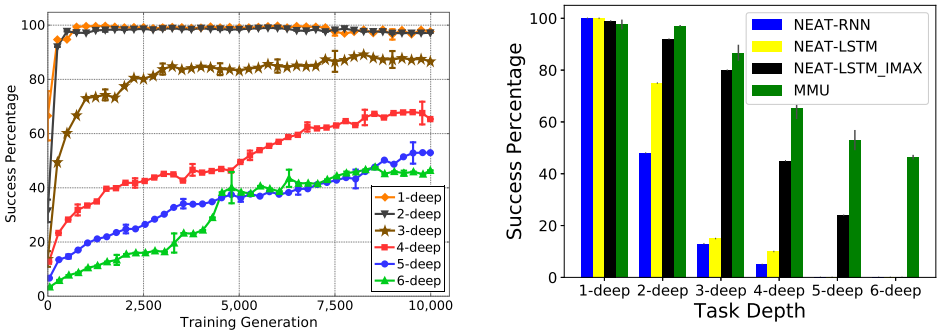
Figure 9: Sequence Recall task: An agent listens to a series of instructional stimuli (directions) at the start of the trial, then travels along multiple corridors and junctions. The agent needs to select a turn at each junction based on the set of directions it received at the beginning of the trial.

The agent receives two inputs: distance to the next junction, and the sequence of instructional stimuli (directions) received at the start of the trial. The second input is set to 0 after all directions have been received at the start of the trial. The agent's action controls whether it moves right or left at each junction. In order to successfully solve the deep Sequence Recall task, the agent has to accurately memorize the instruction stimuli in its correct order and insulate it through multiple corridors of varying lengths. Achieving this would require a memory management protocol that can associate a specific ordered item within the instruction stimuli to the corresponding junction that it serves to direct. A possible protocol that the agent could use is to store the instruction stimuli it received in a first in first out memory buffer, and at each junction, dequeue the last bit of memory and use it to make the decision. This is perhaps what one would do, if faced with this task in a real world scenario. This level of memory management and regimented update is non-trivial to a neural network, particularly when mixed with a variable number of noisy inputs.

5.1 RESULTS

Figure 10a shows the success rate for the MMU neural architecture for varying depth experiments. MMU is able to find good solutions to the Sequence Recall task, achieving greater that 45% accuracy on all choices of task depths within 10,000 training generations. Figure 10b shows a comparison of MMU success percentages after 10,000 generations of evolution with the results obtained using NEAT-RNN, NEAT-LSTM and NEAT-LSTM-Info-max from Rawal and Miikkulainen (2016) after 15,000 generations of evolution. The NEAT-LSTM-Info-max is a hybrid method that first uses an unsupervised pre-training phase where independent memory modules are evolved using an info-max objective that seeks to capture and store highly informative sets of features. After completion of this phase, the network is then trained to solve the task.

As shown by Fig. 10b, MMU achieves significantly higher success percentages for



(a) (b)

Figure 10: (a) Success rate for the Sequence Recall task of varying depth. (b) Comparison of MMU with NEAT-RNN and NEAT-LSTM (extracted from Rawal and Miikkulainen (2016)). NEAT-RNN and NEAT-LSTM results are based on 15,000 generations with a population size of 100 and are only available for task depth up to 5. MMU results are based on 10,000 generations with the same population size and are tested for extended task depths of up to 6.

all task depths, with fewer generations of training except for the 1-deep case. The difference is increasingly apparent as the depth of the task is increased. MMU's performance scales gracefully with increasing maze depth while other methods struggle to achieve the task. MMU was able to achieve a $46.3\% \pm 0.9\%$ success rate after 10,000 generations on a 6-deep Recall Task that we introduced here.

An interesting point to note here is that neither MMU's architecture, nor its training method, have any explicit mechanism to select and optimize for maximally informative features like NEAT-LSTM-Infomax. The influence of the unsupervised pre-training phase was shown in Rawal and Miikkulainen (2016) to significantly improve performance over networks that do not undergo pre-training (also shown in Fig. 10b). MMU's architecture is designed to reliably and efficiently manage information (including features) over long periods of time, shielding it from noise and disturbances from the environment. The method used to discover these features, or optimize their information quality and density, is orthogonal to MMU's operation. Combining an unsupervised pre-training phase using the Info-max objective with a MMU can serve to expand its capabilities even further, and is a promising area for future research.

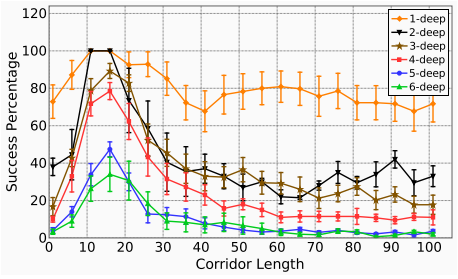


Figure 11: MMU network's generalization to arbitrary corridor lengths in the Sequence Recall task. All MMUs shown were trained with corridor lengths randomly sampled between 10 and 20. These were then tested for corridor lengths ranging from 0 to 101.

5.2 GENERALIZATION TO NOISE

Figure 11 shows the success rate achieved when tested for a varying range of corridor lengths, extending up to a length of 101. All the networks tested were trained for

corridor length between 10 and 20. This experiment tested the generalizability of the strategy that the MMU network learns to deal with noisy corridors.

Overall, the networks demonstrate a modest ability for generalization towards varying corridor lengths. The peak performance across all the task depths is seen between noise sequences of length between 10 and 20. This is expected as this is the length of noise sequences that the networks were trained for. The performance degrades as we vary the length of noisy sequence (corridors) away from the 10-20 range. However, this degradation settles to an **equilibrium** whereafter increase in noise sequence length does not affect performance. The degradation observed here is more severe than the ones observed for the sequence classification task detailed in Section 4.2.1. This is reflective of the added complexity of the Sequence Recall task relative to the Sequence Classification task.

Similar to Section 4.2.1, decreasing the length of the noise sequence leads to virtually the same rate of degradation as increasing it. This suggests that the loss of performance suffered by MMU when tested with noise sequence lengths beyond its training, is caused more by the MMU's specialization to the expected $10 - 20$ range, rather than its inability to process variable length noise sequences. Further, the severity in loss of generalization to lengths of noise sequences grows with the depth of the task. Similar to the Sequence Classification case, this can be attributed to the growth in the sheer volume of the noise added as the depth of the task is increased.

6 DIFFERENTIABILITY

Experiments in Sections 4 and 5 demonstrated the MMU architecture's comparative advantage in retaining information over extended periods of noisy temporal activations when compared to other RNN architectures. However, the training approach utilized was limited to neuroevolution. Gradient descent is a staple training tool in machine learning and thus it is critical to demonstrate MMU's applicability within this context. In this section we implement a differentiable version of the MMU and use gradient descent to train it. We then compare and contrast its performance with neuroevolution within the context of the experiments from Section 4 and 5.

PyTorch (Paszke et al., 2017), a Python based open source library for symbolic differentiation was used to implement the differentiable version of the MMU network. Adam (Kingma and Ba, 2014) optimizer with a learning rate of 0.01 and L2 regularization 0.1 was used. Smooth L1 loss was used to compute the loss function while a batch size of 1000 was used to compute the gradients. GPU acceleration using CUDA was used to expedite the matrix operations responsible for gradient descent. The weights were initialized using the Kaiming normal protocol (He et al. (2015)). Since the sequence lengths for both of our tasks vary dynamically, we pad the sequence with zeros to make the final length equal before using GPU acceleration. The operation protocol for each training sample and hardware resources required for gradient descent and neuroevolution vary widely. This poses a difficulty in defining a common experimentation framework to rigorously compare these two approaches. For example, a generation in neuroevolution is not readily comparable to a gradient descent epoch. The former involves evaluating multiple solutions within the context of some training examples, and selecting the best performing one, while the latter involves one solution learning marginally from all available training examples.

In this experiment, we perform these comparisons by controlling the number of **Reinforcement Instances** for each training algorithm. We define reinforcement instances as the number of times the method uses a training example to obtain rein-

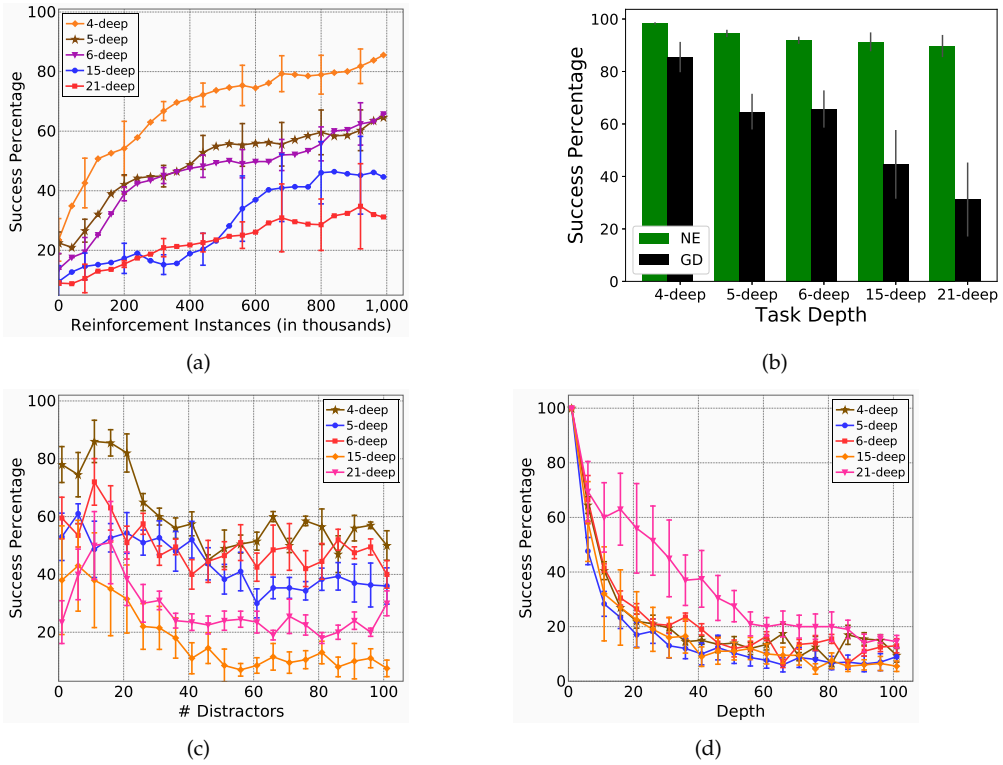


Figure 12: (a) Success rate for the Sequence Classification task of varying depth for a MMU trained using gradient descent. 1000 reinforcement instances are approximately equivalent to a generation. (b) Comparison of gradient descent (GD) and neuroevolution (NE) in training a MMU. (c/d) MMU network's generalization to distractor lengths (number of interleaving noise sequences)(c), and task depths (d) within the Sequence Classification task. All MMUs shown were trained for tasks of their respective depths with distractor lengths randomly sampled between 10 and 20. These were then tested for distractor lengths (c) and task depths (d) ranging from 0 and 101.

forcement about its attempted solution. For neuroevolution, a reinforcement instance is defined as an individual network being evaluated in one training example and getting a fitness score. For gradient descent, a reinforcement instance is defined as one feedforward and backpropagation loop through a training example. In essence, we are comparing the two training approaches by controlling the number of training samples that each method uses to optimize its solution. Note that the training samples are not guaranteed nor required to be unique.

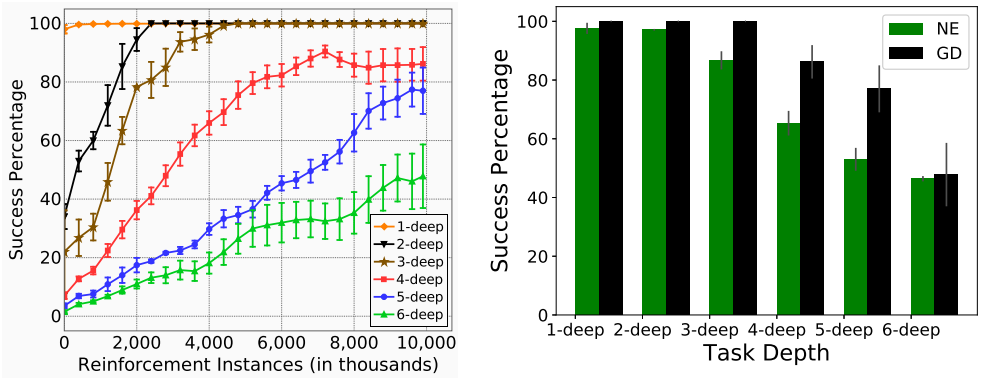
6.1 SEQUENCE CLASSIFIER

Figure 12 show the success rate for the MMU neural architecture trained using gradient descent for varying depth experiments, and its comparative performance with neuroevolution. MMU is able to solve the Sequence Classification task with good accuracy. However, the rate of convergence and the final success rates after 1000 reinforcement instances (comparable to a generation) is significantly lower than the MMU trained with neuroevolution (Figure 5a in Section 4). In particular, the performance of gradient descent scales poorly with the depth of the task. This is not surprising, as the length

of the input sequence increases rapidly with increasing task depth. Propagating error gradients backwards becomes increasingly difficult as the length of the input sequence grows. The MMU trained through neuroevolution (Figure 5a) sidesteps this problem by not relying on propagating gradients backwards through the network's activations. Additionally, the final success achieved by the MMU trained using gradient descent has higher variance than the one trained with neuroevolution. This can be attributed to gradient descent's sensitivity to varying weight initialization. The gradient descent method converges to different local minima depending on varying weight initializations. Neuroevolution on the other hand is more robust to weight initializations, and leads to more repeatable solutions.

Furthermore, compared to the MMU trained with neuroevolution (Section 4), generalizability to larger numbers of interleaving noise, and depths is significantly reduced (shown in Figure 12c and 12d). A network trained using gradient descent in the 21-deep task achieved $14.5\% \pm 2.30\%$ on a 101-deep task while a similar experiment for a network trained with neuroevolution yielded $50.4\% \pm 9.30\%$ (Section 4.2.2). The networks trained with gradient descent tend to specialize more to the range of distractors, and depth they were trained for. When either of these variables changed, the network's performance degraded quickly.

6.2 SEQUENCE RECALL



(a) (b)

Figure 13: (a) Success rate for the Sequence Recall task of varying depth for a MMU trained using gradient descent. A reinforcement instance is approximately equivalent to a generation.(b) Comparison of MMU trained with gradient descent (GD) with one trained using neuroevolution (NE) from Section 5.

Figure 13 shows the success rate for the MMU neural architecture trained using gradient descent for varying depth experiments, and its comparative performance with neuroevolution. MMU is able to solve the Sequence Recall task with a very high level of accuracy. Interestingly, the rate of convergence and the final success rates after 1000 reinforcement instances (comparable to a generation) is better than the MMU trained with neuroevolution (Figure 10a in Section 5). Unlike the Sequence Classification task in the previous section, the range of the depths tested for Sequence Recall is relatively low. The primary limitation to applying gradient descent is the difficulty of propagating errors backwards across long temporal activations. An input to the 6-deep task can have a sequence length between $\{72, 132\}$. While this is a long sequence, it is much shorter than the $\{221, 441\}$ range of the 21-deep task. In the depth ranges tested, gra-

dient descent is thus able to propagate errors backwards and successfully solve the Sequence Recall task.

Figure 14 shows the success rate achieved by the MMU trained using gradient descent, when tested for a varying range of corridor lengths, ranging up to 101. The capability for generalization to larger corridor lengths is comparable to the MMU trained with neuroevolution (Section 5 Figure 11).

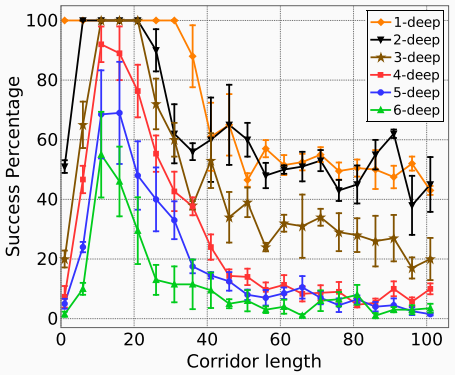


Figure 14: MMU network's generalization to arbitrary corridor lengths in the Sequence Recall task. All MMUs shown were trained for tasks of their respective depths with corridor lengths randomly sampled between 10 and 20.

7 DISCUSSION

Memory is an integral component of high level adaptive behaviors. The ability to store relevant information in memory and identify when to retrieve that data is critical for effective decision-making in complex time-dependent domains. Moreover, knowing when information becomes irrelevant, and being able to discard it from memory when such circumstances arise, is equally important for tractable memory management and computation.

In this paper, we introduced MMU, a new memory-augmented neural network architecture. The key feature of the MMU architecture is its separation of the memory block from the central feedforward operation, allowing for regimented memory access and update. This provides the network with the ability to choose when to read from memory, update it, or simply ignore it. This capacity to act in detachment from the memory block allows the network to shield the memory content from noise and other distractions, while simultaneously using it to read, store and process useful signals over an extended time scale of activations.

We used neuroevolution to train our MMU, and tested it on two deep memory benchmarks. Our results demonstrate that MMU significantly outperforms traditional memory-based methods. Further, MMU is shown to be robust to dramatic increases in the depth of these memory tasks, exhibiting graceful degradation in performance as the length of temporal dependencies increase. Further, in the Sequence Classification task, MMU exhibited good task generalization where a network trained via neuroevolution on a 21-deep memory task is able to achieve $50.4\% \pm 9.30\%$ success rate on a 101-deep task. This suggests that MMU, with its regimented control over memory, is not limited to learning direct mappings, but is able to learn general methods for solving the task. We probed the interactions between memory and the central feedforward operation of a specific individual network and identified its robust memory representation protocol which represented extremely large numbers reliably. We also leveraged the modularity of the MMU architecture to analyze its interior operations, and derived an approximate pseudo-algorithm that the network had learned to solve sequence classification.

Additionally, we implemented a fully differentiable version of the MMU and compared its performance with neuroevolution. Results show that gradient descent holds an advantage over neuroevolution for the Sequence Recall task, while neuroevolution significantly outperforms gradient descent in the Sequence Classification task. Sequence Recall requires a more complex memory management protocol than Sequence

Classification, but was tested for lower depths (length of temporal sequences). This demonstrates that gradient descent holds a comparative advantage in solving complex problems faster but suffers rapid degradation of performance as the length of the temporal sequence grows. Neuroevolution on the other hand is more robust and exhibits graceful degradation to increasing temporal lengths. Neuroevolution also leads to more repeatable and generalizable solutions across both tasks.

Future work will integrate an unsupervised pre-training method, such as infomax optimization, as described in Rawal and Miikkulainen (2016), in training MMU. Combining the discovery of maximally informative features with MMU's capability to reliably hold and process information over long periods of time can expand the capabilities of MMU even further. Additionally, in this paper, we used a neuroevolutionary algorithm with slight modifications to exploit the natural decomposition of MMU architecture. A principled approach to exploiting this sub-structure decomposition, and expanding to hybrid training methods such as Evolutionary Reinforcement Learning (Khadka and Tumer, 2018) is another promising area.

The experiments conducted in this paper tested reliable retention, propagation and processing of information over an extended period of time. However, the size and complexity of the information that needed to be retained was small and relatively simple. Future experiments will develop and test MMU in tasks that require retention and manipulation of larger volumes of information. The size of the information being propagated (memory size) also approximately matched the size of features necessary to be extracted (hidden nodes size). This excluded the memory encoder and decoder aspects of the MMU from being used and tested. Future work will include experiments where there is a mismatch between these two aspects of network operation, to test the memory encoder and decoder functionalities of the MMU architecture.

References

Baddeley, A. D. and Hitch, G. (1974). Working memory. *The Psychology of Learning and Motivation*, 8:47–89.

Bakker, B. (2001). Reinforcement learning with long short-term memory. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, pages 1475–1482. MIT Press.

Bayer, J., Wierstra, D., Togelius, J., and Schmidhuber, J. (2009). Evolving memory cell structures for sequence learning. In *International Conference on Artificial Neural Networks*, pages 755–764.

Charles, O., Tony, P., and Stéphane, D. (2012). With a little help from selection pressures: evolution of memory in robot controllers. *Artificial Life*, 13:407–414.

Cho, K., Gulcehre, B. v. M. C., Bahdanau, D., Schwenk, F. B. H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1724–1734.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. Preprint at https://arxiv.org/abs/1412.3555.

Chung, J., Gülçehre, C., Cho, K., and Bengio, Y. (2015). Gated feedback recurrent neural networks. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 2067–2075.

El Hihi, S. and Bengio, Y. (1995). Hierarchical recurrent neural networks for long-term dependencies. In *Advances in Neural Information Processing Systems*, pages 493–499.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Grabowski, L. M., Bryson, D. M., Dyer, F. C., Ofria, C., and Pennock, R. T. (2010). Early evolution of memory usage in digital organisms. In *ALIFE*, pages 224–231.

Graves, A., Jaitly, N., and Mohamed, A.-r. (2013a). Hybrid speech recognition with deep bidirectional lstm. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 273–278.

Graves, A., Mohamed, A.-r., and Hinton, G. (2013b). Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE.

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5):602–610.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural Turing machines. Preprint at https://arxiv.org/abs/1410.5401.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., Badia, A. P., Hermann, K. M., Zwols, Y., Ostrovski, G., Cain, A., King, H., Summerfield, C., Blunsom, P., Kavukcuoglu, K., and Hassabis, D. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., and Schmidhuber, J. (2016). LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*.

Greve, R. B., Jacobsen, E. J., and Risi, S. (2016). Evolving neural turing machines for reward-based learning. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 117–124. ACM.

Hassabis, D., Kumaran, D., Summerfield, C., and Botvinick, M. (2017). Neuroscience-inspired artificial intelligence. *Neuron*, 95(2):245–258.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116.

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

Jaeger, H. (2016). Artificial intelligence: Deep neural reasoning. *Nature*, 538(7626):467–468.

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2342–2350.

Khadka, S., Chung, J. J., and Tumer, K. (2017). Evolving memory-augmented neural architecture for deep memory problems. In *In Proceedings of the Genetic and Evolutionary Computation Conference 2017*. ACM.

Khadka, S. and Tumer, K. (2018). Evolutionary reinforcement learning. *arXiv preprint arXiv:1805.07917*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lüders, B., Schläger, M., Korach, A., and Risi, S. (2017). Continual and one-shot learning through neural networks with dynamic external memory. In *European Conference on the Applications of Evolutionary Computation*, pages 886–901. Springer.

Merrild, J., Rasmussen, M. A., and Risi, S. (2018). Hyperntm: Evolving scalable neural turing machines through hyperneat. In *International Conference on the Applications of Evolutionary Computation*, pages 750–766. Springer.

Paszke, A., Gross, S., Chintala, S., and Chanan, G. (2017). Pytorch.

Rawal, A. and Miikkulainen, R. (2016). Evolving deep LSTM-based memory networks using an information maximization objective. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 501–508.

Srivastava, N., Mansimov, E., and Salakhutdinov, R. (2015). Unsupervised learning of video representations using LSTMs. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 843–852.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127.

Sundermeyer, M., Alkhouli, T., Wuebker, J., and Ney, H. (2014). Translation modeling with bidirectional recurrent neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 14–25.

Sundermeyer, M., Schlüter, R., and Ney, H. (2012). LSTM neural networks for language modelling. In *Interspeech*, pages 194–197.

Tulving, E. (2002). Episodic memory: from mind to brain. *Annual review of psychology*, 53(1):1–25.

Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.