AN ABSTRACT OF THE DISSERTATION OF

Shauharda Khadka for the degree of Doctor of Philosophy in Robotics presented on June 5, 2019.

Title: Tackling Credit Assignment Using Memory and Multilevel Optimization for Multiagent Reinforcement Learning

Abstract approved: ______________________________

Kagan Tumer

There is growing commercial interest in the use of multiagent systems in real world applications. Some examples include inventory management in warehouses, smart homes, planetary exploration, search and rescue, air-traffic management and autonomous transportation systems. However, multiagent coordination is an extremely challenging problem. First, information relevant for coordination is often distributed across the team members, and fragmented amongst each agent's observation histories (past states). Second, the coordination objective is often sparse and noisy from the perspective of an agent. Designing general mechanisms of generating agent-specific reward functions that incentivizes an agent to collaborate towards the shared global objective is extremely difficult. From a learning perspective, both difficulties can be linked to the difficulty of credit assignment - the process of accurately associating rewards with actions.

The primary contribution of this dissertation is to tackle credit assignment in multiagent systems in order to enable better multiagent coordination. First we leverage memory as a tool in enabling better credit assignment by facilitating associations between rewards and actions separated across time. We achieve this by introducing Modular Memory Units (MMU), a memory-augmented neural architecture that can reliably retain and propagate information over an extended period of time. We then use MMU to augment individual agents' policies in solving dynamic tasks that require adaptive behavior from a distributed multiagent team. We also introduce Distributed MMU (DMMU) which uses memory as a shared knowledge base across a team of distributed agents to enable distributed one-shot decision making.

Switching our attention from the agent to the learning algorithm, we then introduce Evolutionary Reinforcement Learning (ERL), a multilevel optimization framework that blends the strength of policy gradients and evolutionary algorithms to improve learning. We further extend the ERL framework to introduce Collaborative ERL (CERL) which employs a collection of policy gradient learners (portfolio), each optimizing over varying resolution of the same underlying task. This leads to a diverse set of policies that are able to reach diverse regions within the solution space. Results in a range of continuous control benchmarks demonstrate that ERL and CERL significantly outperform their composite learners while remaining overall more sample-efficient.

Finally, we introduce Multiagent ERL (MERL), a hybrid algorithm that leverages the multilevel optimization framework of ERL to enable improved multiagent coordination without requiring explicit alignment between local and global reward functions. MERL uses fast, policy-gradient based learning for each agent by utilizing their dense local rewards. Concurrently, evolution is used to recruit agents into a team by directly optimizing the sparser global objective. Experiments in multiagent coordination benchmarks demonstrate that MERL's integrated approach significantly outperforms the state-of-the-art multiagent policy-gradient algorithms.

Tackling Credit Assignment Using Memory and Multilevel Optimization for Multiagent Reinforcement Learning

by

Shauharda Khadka

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 5, 2019
Commencement June 2019

ACKNOWLEDGEMENTS

For the completion of this work I have enjoyed the assistance and support of many people, to all of whom I am greatly indebted. First, my deepest gratitude is due to my academic advisor, Professor Kagan Tumer, for his guidance and constant encouragement. This dissertation is a tribute to his mentorship.

I would also like to thank Dr. Jen Jen Chung for her guidance.

Further, I would like to thank my friends and colleagues in the Autonomous Agents and Distributed Intelligence (AADI) Laboratory for their support, camaraderie, and discussions.

A special thanks to Dr. Somdeb Majumdar, Dr. Evren Tumer, and colleagues in Intel AI for their collaboration.

Finally, my special appreciation to my father Chakra Bahadur Khadka and mother Gita Devi Khadka for their love and sacrifice through the years.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF ALGORITHMS

Chapter 1: Introduction

Reinforcement Learning (RL) offers powerful tools for addressing sequential decision making problems and holds great promise in tackling complex real world problems. Recently, its integration with powerful non-linear function approximators like deep neural networks have enabled many successful applications. Some examples of this include industrial data center cooling applications [52], control of humanoid robots [239], quadrupeds [93] and drug discovery [172].

However, most of these applications involve a single agent. This is in stark contrast to most real world applications that involve multiple agents interacting with each other. Learning in these systems brings out unique challenges different from its single agent counterparts. This stems from the fact that an individual agent action in these systems can have complex and unintended consequences to the emergent behavior of the multiagent team. For instance, air traffic control involves coordination across thousands of flights that operate within a shared US airspace every day. A small delay or deviation of plan by one flight can propagate rapidly and affect the entire system as a whole. This can lead to wide-spread delays costing billions of dollars every year for the taxpayer [33, 221, 241].

Multiagent teams are also inherently more suited for exploration. Consider the grand challenge of exploring Mars. A team of autonomous robots working together can significant improve performance over single robot missions. The multi-robot team can cover more ground, is more flexible, and offers more reliability. It is also more robust enabling graceful degradation in performance with failures. This is crucial in extreme environments like space [218, 240] or underwater settings [101, 109, 209] where the likelihood of failure in communication, sensors, actuators, or associated electronics is very high. Search and rescue operations in disaster scenarios also share this property. Rescue operations following the Fukushima nuclear plant disaster was a harrowing example [160]. Multiagent approaches hold promise in providing a possible solution and have been widely explored in recent literature [21, 86, 125].

Autonomous driving is another example of a multiagent system with significant potential for impact in the world [54]. The past decade has seen tremendous progress in enabling technologies associated with autonomous vehicles such as sensors, control algorithms, perception, and actuators [31, 130, 182]. However, integrating these vehicles in our roadways still faces significant challenges [54]. Most of these difficulties stem from the uncertainty and unpredictability of interaction between multiple autonomous agents. To fully realize the potential impact of autonomous driving technologies, it is crucial to address these multiagent challenges. Efforts towards this end have been widely explored in recent literature [18, 190, 238].

However, multiagent coordination is an extremely challenging problem. First, information relevant for effective coordination is often distributed across the members of the team (different agents holding part of the key information). Additionally, in most real world settings this information is also fragmented over each agent's observation history (past states of the agent). Retrieving information by integrating across both time and across other agents is necessary for effective multiagent coordination.

For instance, real time traffic updates through an autonomous car network requires both inference on the sequence of sensor data from each car, and consolidation between multiple data streams acquired concurrently from numerous cars distributed in space. If an autonomous car observes an unanticipated blockage in a specific location, this observation has to be quickly identified among a plethora of normative information. It has to be processed, and dynamically retained for a period of time, so that the network can quickly adapt navigation routes for other cars approaching this location.

Second, the coordination objective is often vague, sparse, and extremely noisy from the perspective of an agent in the team. Consider the game of soccer where a team of agents coordinate to achieve a global objective of winning. Directly optimizing this objective to train individual agents is sub-optimal due to two reasons. First, it fails to encapsulate the contributions of individual agents to the final result. Second, it is usually very sparse - a single scalar capturing performance of an entire team operating across an extended period of time. This makes it a weak metric to learn on. In practice, domain knowledge is often used by an expert to design agent-specific rewards [43, 237]. A simple example would be to reward the defenders for keeping clean sheets while the strikers get rewarded for scoring. However, this type of agent-specific proxy rewards are not very generalizable. For example, a team that is winning may benefit from protecting its lead by temporarily being more defensive. This objective now becomes misaligned with the local objectives of the strikers that prioritize scoring. This leads to sub-optimal coordination overall. Further, these kind of shaped rewards can change the underlying problem itself [162].

From a reinforcement learning perspective, both of these problems can be linked to a common root cause - the difficulty of credit assignment, defined as the ability to accurately associate rewards with actions. Addressing this problem has been a long-time research focus in reinforcement learning [42, 187, 186] and has led to its most successful innovations. An example of this is temporal difference (TD) learning [211], which is the principal component behind model-free reinforcement learning. While temporal credit assignment in single-agent applications still remains the key area of focus for reinforcement learning research [7, 42, 187, 186], credit assignment for multiagent systems has also received a lot of attention in recent times [91, 146]. This is driven by the rapid progress in the development of deep reinforcement learning and the expansion of these techniques to real world applications which are often inherently multiagent. Continued progress will require addressing the joint complexities in settings where credit needs to be assigned across temporal separations and amongst multiple agents.

1.1 Contributions

The primary contribution of this dissertation is to improve learning in multiagent systems by tackling credit assignment: establishing relationships between action and reward across time and among a team of agents. First we develop novel memory-augmented neural network architectures and leverage it to improve learning in distributed multiagent teams. Then we formulate a multilevel optimization framework, that can integrate reward functions across multiple hierarchies to improve multiagent coordination. Collectively, these developments enable effective learning in scenarios where the reward is conditioned on a series of joint-action taken by a team of distributed agents operating through time. The four research contribution of this dissertation are to:

1. Introduce Modular Memory Unit (MMU), a novel memory-augmented neural network topology that enables reliable retention and propagation of information over an extended period of noisy operation enabling improved temporal credit assignment. [113, 115].

2. Introduce Distributed MMU (DMMU), where an external shared memory is collectively read and written to by a team of agents to enable distributed one-shot decision making [114, 119, 120, 121].

3. Introduce Evolutionary Reinforcement Learning (ERL), a hybrid framework that combines the strengths of an Evolutionary Algorithm (EA) with fast gradient-based algorithms for effective deep reinforcement learning [118]. Further, we extend ERL to introduce Collaborative ERL (CERL), which employs a collection of policy gradient learners (portfolio), each optimizing over varying resolution of the same underlying task for improved exploration of the search space [116].

4. Introduce Multiagent ERL (MERL), which leverages the multilevel optimization framework of ERL to tackle sparse multiagent coordination problems by leveraging dense local reward even when there is no guarantee of alignment between the two [117].

1.2 Research Outline

Each of the contributions and their associated chapters are outlined below:

Contribution 1: Modular Memory Unit (MMU) The first thread of research tackles temporal credit assignment by using memory to identify and associate reward with its associated actions. This is crucial for settings where the action and its associated reward is separated by a long period of time. Most real world applications fall under this category. For instance, the action of packing your umbrella in the morning might only result in a reward when it rains in the evening. To properly associate the reward with its associated action, the agent needs to remember the action taken in the morning, shield it from other decisions (actions) taken during the day, and associate it with its related reward that occurs only in the evening. Humans inherently excel at using memory to make these associations.

Chapter 3 introduces MMU, a new neural network topology designed to effectively retain and propagate information over an extended period of time. MMU's independent read and write gates serve to decouple memory from the central feedforward operation of the network, administering the ability to choose when to read from memory, update it, or simply ignore it. This capacity to act in detachment enables MMU to shield the memory from noise and other distractions, while simultaneously remembering connections between reward and associated action dispersed across time. Results on two deep memory benchmarks demonstrate that MMU performs significantly faster and more accurately than traditional memory-based methods, and is robust to dramatic increases in the length of time between reward and its associated action [113, 115].

Contribution 2: Distributed Modular Memory Unit (DMMU) The second thread of research builds on the MMU topology and applies it towards improving multiagent coordination in settings that require rapid adaptation. This is crucial in most real world settings where a multiagent team has to change its joint behavior based on a singular observation by one of its agents. For instance, if an autonomous car observes an unanticipated blockage in a specific location, this observation has to be quickly identified among a plethora of normative information. It has to be processed, and dynamically retained for a period of time, so that the autonomous fleet can quickly adapt navigation routes for other cars approaching this location.

Chapter 4 details two such applications. The first application uses memory to augment individual agent policies in solving dynamic tasks that require adaptive behavior. Results on a T-Maze benchmark demonstrate that the memory-augmented agents are able to leverage their memory to integrate information across time to learn adaptive behaviors while reactive agents fail entirely to solve the task [114]. The second application introduces DMMU where an external memory is used as a shared knowledge base across a team of agents for distributed one-shot decision making. Each agent can selectively and asynchronously access the shared memory to accept and/or disseminate pertinent information as they are observed. This allows for rapid consolidation of salient information enabling distributed one-shot decision making [119, 120, 121].

Contribution 3: Evolutionary Reinforcement Learning (ERL) Chapter 5 introduces ERL, a multilevel optimization algorithm that integrates reward signals across multiple hierarchies to improve credit assignment in reinforcement learning. This is particularly crucial for settings where the learning goal is conditioned on a series of actions (long chain of sequential decisions) with sparse reward signals. This is a defining feature of most real world applications and is a fundamental difficulty within reinforcement learning.

ERL hybridizes the strengths of two distinct learning techniques: namely Evolutionary Algorithms (EAs) and policy gradients to achieve the best of both approaches. Specifically, ERL inherits EAs capability for temporal credit assignment with sparse rewards, effective exploration with a diverse set of policies, and stability of a population-based approach and complements it with policy gradients ability to leverage gradients for higher sample efficiency and faster learning. Experiments in a range of challenging continuous control benchmarks demonstrate that ERL significantly outperforms prior DRL and EA methods [118].

Chapter 6 extend the ERL framework to introduce Collaborative ERL (CERL) [116] which employs a collection of policy gradient learners (portfolio), each optimizing over varying resolution of the same underlying task. This leads to a diverse set of policies that are able to reach diverse regions within the solution space. A shared replay buffer pairs this improved exploration with collective exploitation for improved learning. Results in a range of continuous control benchmarks demonstrate that CERL significantly outperforms its composite learner while remaining overall more sample-efficient.

Contribution 4: Multiagent Evolutionary Reinforcement Learning (MERL) The final thread of research builds on the ERL framework and applies it towards improving multiagent coordination where the team's learning goal is sparse and noisy. This is a common occurrence is most real world multiagent systems. Some examples include soccer, extraterrestrial exploration, or search and rescue. The team goal (global reward) in all these scenarios is too sparse and noisy to learn effectively from.

Chapter 7 introduces MERL [117] which splits multiagent problems into two sub-parts: learning to manipulate the environment using high-fidelity and dense local rewards, while concurrently leveraging the skills learned towards tackling the global reward. MERL achieves this using two classes of optimizers that operate over varying levels of purview. The local optimizer (policy gradient) leverages local rewards to learn with high-fidelity information. Concurrently, an evolutionary algorithm is used to recruit agents into a team by directly optimizing the sparser global reward. Results in a multiagent coordination task demonstrate that MERLs integrated approach significantly outperforms the state-of-the-art multiagent policy-gradient algorithms.

Chapter 2: Background

This dissertation tackles credit assignment in multiagent reinforcement learning through two distinct threads of research. The first thread leverages memory-augmented neural networks to augment the agent's capability in identifying and storing action-reward associations through an extended period of time. A comprehensive survey on memory-based techniques is presented in Section 2.1. The second thread leverages hybrid reinforcement learning techniques that combine the strengths of multiple optimizer operating at varying levels of purview to enable effective credit assignment. A comprehensive survey on relevant reinforcement learning techniques is described in Section 2.2.

2.1 Memory

Memory provides the capacity to recognize and recall similar past experiences in order to improve current decision making. Increasingly, adaptive systems are required to operate in dynamic environments where critical events occur stochastically and affect the system only after a period of time. In these settings, memory allows an agent to bridge information across time enabling associations to be made between reward and its causal action. In a learning setting, the most widely used avenue to impart memory faculty to an agent is through the use memory augmented neural networks.

Artificial Neural Networks (ANN) are universal functional approximators [102] that were originally conceived to mimic the functionality of the human brain. It has since been established that these systems are extremely simplistic, in relation to the human brain, and resemble more of a highly flexible statistical device that can be mathematically configured to approximate functions. ANNs have since become widely popular and have been successfully applied in a broad range of disciplines and real world tasks [155, 78, 13, 219].

Feedforward neural networks (FNNs) are perhaps the most popular variant of ANNs. FNNs, however, are limited to processing stationary input-output patterns, and assume that their inputs are static and independent of each other. This is an unrealistic assumption in most real world applications, as past events often influence future events. The key to predicting or acting optimally in the present often is conditioned on events that led up to the current point. One simple way of integrating past events to make prediction in the present is to concatenate past states to compute the current input (project temporal data in a spatial dimension). However, the choice of depth in time to concatenate is a free parameter that needs to be set manually. Additionally, the size of the neural connections also scales poorly with increasing window of time to look through.

Time-delay networks [228, 227, 132] attempted to solve this problem by using 1-D convolution across the concatenated temporal sequence. The convolution kernels, similar to the one used in the hugely successful Convolutional Neural Networks (CNNs) [111, 128, 134] today, allow for sharing of parameters for input variables spread across time. However, the convolution operation is shallow and only considers a set of neighboring variables while computing an output. Recurrent Neural Networks (RNNs) are a class of ANNs that provides a more natural approach to sharing parameters across multiple temporal scales. The same update rule is applied for the input variables across all temporal scales but the output for each subsequent temporal processing loop is conditioned on the previous output. This feeding back of past output is often referred to as *recurrency*, and this approach leads to a more efficient exploitation of regularity between input variables separated by time [73].

The recurrent input is the key to a RNN's ability to capture temporal dependencies and process sequences effectively. In a way, this recurrent input can be viewed as a type of *memory* that condenses salient observations from the past and allows for the current processing loop to condition its output based on these observations. It is necessary to precisely define what memory refers to. All ANNs can be represented as computational graphs and store its input-output mapping as dense vectors in its *weights* that parameterize this computational graph. The dense vector representation of how an ANN computes output, given a certain input, can in itself be thought of as memory. In order to disambiguate, we will **not** refer to these weights as memory in this paper, but reserve the term for structures like recurrent hidden activations, cell states or memory banks that explicitly serve across multiple temporal scales and propagate information between them.

RNNs have grown to be extremely popular in the last decade and are part of the state of the art in many sequence processing tasks [207, 78, 73, 185]. This ubiquity has led to development of numerous variants to RNNs. In this article, we classify and review these developments through the perspective of the memory structure employed and highlight three major types. Section 2.1.1 discusses vanilla recurrent neural networks which are the most simplest form of RNNs. Section 2.1.2 highlights RNNs that employ gates to filter infromation flow and use cell states to propagate recurrent information. Section 2.1.3 discusses RNNs which introduce memory banks external to the controller itself and utilize attentional processes to interact with it. The developments of these structures are coextending but their first introduction roughly follow the chronological order of their presentation.

2.1.1 Vanilla Recurrent Neural Networks

In this article, vanilla recurrent neural networks (VRNN) refers to the original and simplest formulation of RNNS that use a fixed size recurrent vector to propagate information between multiple temporal layers of input. This can be represented as a deep computational graph that unfolds to form a multiple layer FNN. Each layer of this FNN would deal with input data corresponding to that temporal location within the sequence and the recurrent connection from the preceding FNN. Figure 1 (extracted from [73]) depicts the unfolding of a VRNN into a series of FNNs. The recurrent connection depicted as solid black square serves to propagate information from earlier parts of the temporal sequence. The sequence input is represented by x, while h represents a fixed hidden state vector which captures the recurrence. Each FNN unfolded maps a temporal slice of input and the current values of h

into the next value of h. The output, not shown is a layer that sits on top of h that maps it to an output of the network.

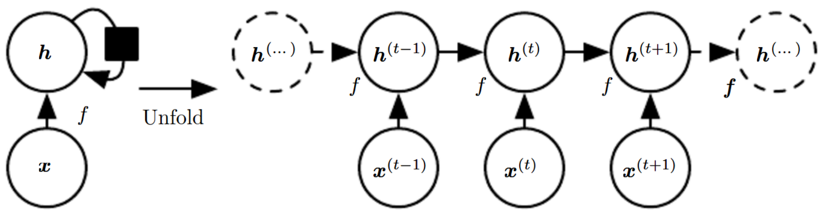


Figure 2.1: Illustration of unfolding a vanilla recurrent neural network. Output is not shown. Graphic extracted from [73]

The *recurrence* employed by VRNNs make them a form of deep neural network that in principle is Turing complete [193, 194]. This property means that a VRNN, in theory, is able to approximate any program and is a type of general computer. Turing completeness makes VRNN an universally applicable structure, but in practice, this same universality means that the space of programs that it can describe is virtually infinite. This makes searching for a set of network parameters which describe a specific desired program very difficult. Finding these parameters, commonly referred to as training the network, is a limiting factor on its applicability, and has thus been a principal component of VRNN research.

Training a VRNN: One of the primary reasons for the renaissance seen in ANN research was integration of efficient backpropagation (BP) in training them around the 1980's. Minimization of errors through gradient descent for a complex nonlinear and differentiable parameters space had existed for a long time [25, 112, 46, 26]. Explicit configuration of gradient descent to perform efficient error BP exploiting sparse connections in ANN-like structures was described however, in [142, 143]. This was tasked for ANN-specific applications in [230, 136] and popularized especially by demonstration of internal representations within hidden layers by Rumelhart [180]. The same work also formulated BP for sequential computation graphs like VRNNs which formed the basis for backpropagation through time (BPTT) [231]. Slight variations of gradient based algorithms for VRNNs like real-time recurrent learning (RTRL) [236] and recurrent backpropagation algorithm [170] were devised [234] and improved iteratively [168, 183, 235].

While BPTT in theory is capable of propagating error gradients backwards up to arbitrary depths of network layers, in practice, it was able to do so effectively for only a couple of layers backwards. For FNNs this was not a principal problem as a FNN with a single layer of enough hidden units is shown to be able to approximate any continuous functions with arbitrary precision [127, 97]. VRNNs, however, are compactly represented architectures which unfold into a multiple layer FNN with number of layers equaling the temporal depth of the input sequence as shown in Figure 1.

Vanishing/Exploding Gradients: This difficulty faced by BPTT in training deep VRNN architectures was highlighted in the seminal dissertation by Hochreiter [99] (in German) as summarized in [73]. The phenomenon commonly referred to as the problem of vanishing or exploding gradient today, refers to the rapid shrinking or explosion of backpropagated error signals as they traverse across multiple layers of nonlinear activation functions. Goodfellow [73] expressed this in terms of credit assignment path lengths which describe the length of the path a backpropagated error signal has had to traverse before reaching its target weight. The credit assignment path length was measured in the number of layers which strongly correlated to number of activations, and caused an exponential decay or explosion in the quality of the error signal. A large volume of subsequent research ranging from unsupervised pre-training [184], Kalman filtering [174], simulated annealing and pseudo-Newton optimization [20] focused on addressing this problem with intermittent success.

A major breakthrough in addressing the problem of exploding/decaying backpropagated errors in deep credit assignment paths came through the introduction of Long Short Term Memory (LSTM) [100]. The principal idea of LSTMs is the introduction of Constant Error Carousel Units (making up a memory cell) with an identity activation function and a self-referential connection with a weight value of 1. This leads to a constant derivative of 1 that serves to shield the errors propagated by the constant error carousel units from decaying/exploding exponentially [100, 73]. This property allows LSTMs to effectively backpropagate errors through long credit assignment paths spanning across hundreds of layers, and by extension, memorize and learn long term temporal dependencies between events. In addition to the constant error carousel units, LSTM uses gates that are essential in learning nonlinear behavior when used in conjunction with linear identity activations. The next section will focus on the role of gates in protecting and propagating information across long timescales.

2.1.2 RNNs with Gating Units

Gating units were introduced in LSTM [100] to protect the memory cell from unintended perturbations while being able to interact with other nonlinear multiplicative components within the architecture. An input gate protects the memory cell's content from irrelevant or noisy inputs while an output gate protects other components of the LSTM from irrelevant or outdated memory content. Gating units are multiplicative and often use sigmoid activation function which output in the interval [0-1]. In a way, gates can be thought of as a mask that filters information passing through it. In an LSTM unit, the gates effectively modulate access to the constant error flow through the memory cells. This allows for LSTM to learn important features and remember it across long time lags, effectively creating shortcut paths that bypass multiple time steps [34].

A forget gate was added into the LSTM block in [70] which allowed the LSTM to reset its internal cell state adaptively. This was critical in allowing LSTMs to learn to generalize to continuous input streams instead of being limited to solving *a priori* decomposed subsequences. Weighted connections from the memory cell to the gates, termed peepholes were added in [69]. These peephole connections allowed the memory cell content to directly contribute in controlling the gates that shielded them, allowing for more precise control and timings. A final major key modification was done in [80] which implemented a full Backpropagation Through Time (BPTT) algorithm to train the LSTM instead of a combination of Real-time Recurrent Learning (RTRL) and truncated BPTT used by the previous iterations

[85].

Bidirectional LSTM: Many variations on the LSTMs has since been introduced with alterations to the topology, training algorithms and even direction of information flow. The same paper that implemented the full BPTT in LSTMs [80] also introduced Bidirectional LSTM (BLSTM) building on the earlier ideas of Bidirectional Recurrent Neural Networks (BRNNS) [16, 189]. The central idea behind bidirectionality is to use two separate networks to process the input sequence forward and backwards and then combine them to compute the final output. This means that at each point in the input sequence, a bidirectional architecture has complete information about all inputs that precede and succeed it. While this dependence on the entire sequence which includes events that are in the future seems to violate causality for an online task, many real world tasks often require an output following an input segment with defined start and endpoints[80]. Protein structure modeling is one such domain where BLSTM has seen major success [226, 94, 214]. Natural language processing (NLP) is another domain where knowing the words or speech later in the sequence is often useful in providing context for interpreting the inputs earlier. Many variations of BLSTMs have been widely used in various NLP tasks like machine translation [207], spoken language understanding [153, 152], voice conversion [206] and recognition [164, 55].

Training a RNN with gates: Full BPTT applied in conjunction with the Constant Error Carousel unit has been the key ingredient to effective information propagation through a LSTM architecture, both forward and backwards. This synergy between gradient descent and architecture engineered for effective propagation of gradients is a principal component behind the successful training of LSTMs. Many variations, alternatives, and enhancements have however been proposed on training methods to increase convergence speed and quality. Second order methods like Hessian free optimization [151] were shown to demonstrate promising results. These however came at a cost of computational complexity, especially as the size of the network grew. Connectionist Temporal Classification (CTC) [77] was proposed to train an LSTM to produce labels directly from noisy unsegmented sequences. This obviated the need for pre-segmentation of sequences and post processing to collate the LSTM outputs for sequence labeling [76].

Stochastic Gradient Descent (SGD) [68, 23, 243] is an iterative version of gradient descent which updates the network parameters after computing the gradient from a small sample size of training data, often referred to as mini-batches. This obviates the need for computing gradients from the entire training set before updating the network parameters, and is critical when dealing with training data that may include millions of examples. Many heuristics that adaptively tune the learning rate for SGD such as AdaGrad [49], Adam [122] and RMSprop [216] were introduced to speed up convergence. Matrix based optimization methods and parallelization using Graphics Processing Units (GPUs) [210, 28] have been popular in training of deep LSTM units, leading to up to 50 times the speedup compared to traditional CPU based methods.

Apart from gradient based methods, direct search approaches have also been extensively explored. These approaches evaluate the set of parameters (network weights) directly on the given problem without trying to establish explicit causalities or correlations between parameters and error. This is equivalent to searching directly in the weight space without heeding to the credit assignment paths that traditional gradient based methods exploit and depend upon. Evolutionary Algorithms (EAs) [61, 10] are a prime example, with a successful track record. EAs are a class of stochastic search methods inspired by biological evolution, and employ primary operators like mutation, crossover and selection iteratively on a population of individuals. Each individual is a complete solution to a problem and is often encoded into a real valued representation called genome. The sets of weights parameterizing RNNs and its topology can be encoded into a genome, and EAs can then be used to evolve them. This is often referred to as neuroevolution. Some notable neuroevolutionary methods include NeuroEvolution of Augmented Topologies (NEAT) [201] and Cooperative Synapse NeuroEvolution (CoSyNE) [72]. Since RNNs are shown to be general computers, neuroevolution of RNNs can be considered a type of Genetic Programming (GP), which is a close cousin of EAs that evolves programs. The added advantage of RNNs however is that unlike traditional GP which is often limited evolving sequential programs, RNN neuroevolution can find synergies between its distributed representation and sequential information flow leading to efficient parallel information processing [185].

Variations in network architecture Many variations on the LSTM architecture have also been proposed for varying applications. An extensive analysis and study on different LSTM variants ablating various of its components was conducted in [85]. Gated Recurrent Unit (GRU)[32] is perhaps the most popularly used variant which differs from an LSTM primarily in its level of memory exposure. Unlike an LSTM which has a memory gate modulating the exposure of the memory content, a GRU exposes its entire memory to the network for each feedforward operation.

The reset gate r and update gate z are computed by

$$r = \sigma\left(\mathbf{W}_r\mathbf{x} + \mathbf{U}_r\mathbf{h}^{\langle t-1\rangle}\right), \tag{2.1}$$

$$z = \sigma\left(\mathbf{W}_z\mathbf{x} + \mathbf{U}_z\mathbf{h}^{\langle t-1\rangle}\right), \tag{2.2}$$

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

$$h_j^{\langle t\rangle} = z_j h_j^{\langle t-1\rangle} + (1 - z_j)\,\tilde{h}_j^{\langle t\rangle}, \tag{2.3}$$

where

$$\tilde{h}_j^{\langle t\rangle} = tanh\left([\mathbf{W}\mathbf{x}]_j + \left[\mathbf{U}\left(\mathbf{r}\odot\mathbf{h}^{\langle t-1\rangle}\right)\right]_j\right). \tag{2.4}$$

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.

GRU employs one fewer gate in comparison to an LSTM, and also boasts a lighter architecture, often leading to easier and faster training. Extensive empirical evaluations between a GRU and LSTM was conducted in [34] which failed to conclusively establish one's superiority over the other. These two architectures were found to be clearly superior to VRNNs without gating units, but were comparable in relation to each other.

LSTMs and its variants, by virtue of their effective error handling and efficient training methods, have virtually become the standard in most modern learning based sequence processing tasks. Their method of formulating memory through the cell state that is able to propagate information through extended periods of activation has set many benchmarks in recent times. A stumbling block in its even broader applicability however, is its coupling of memory with computation. The sets of weights that parameterize the LSTM/GRU architecture scale severely with increasing size of the cell state (memory). The number of parameters are thus a direct function of the size of memory. This is highly limiting and problematic in tasks that may require huge amounts of memory. For example, a general NLP machine could benefit from having the entire dictionary in its memory. An AI think tank could benefit from having the entire set of Wikipedia entries in its memory. This integration of large banks of information however, is not possible if the network parameters scale directly as a function of the memory size. Some exciting developments in alleviating this issue has been made in recent times. The following section will discuss some of these in detail.

2.1.3 RNNs with External Memory Bank

One way of alleviating the harsh scaling of network parameters with size of memory is to decouple memory from computation. These category of architectures are often referred to as Memory Augmented Neural Networks (MANNs). Neural Turing Machines (NTM) [81] is perhaps the first architecture in achieving this distinction. A NTM augments a traditional ANN with an external memory it can interact with attentional processes. It is inspired by theories in cognitive sciences which suggest that humans have a central executive that interacts with a memory buffer [11].

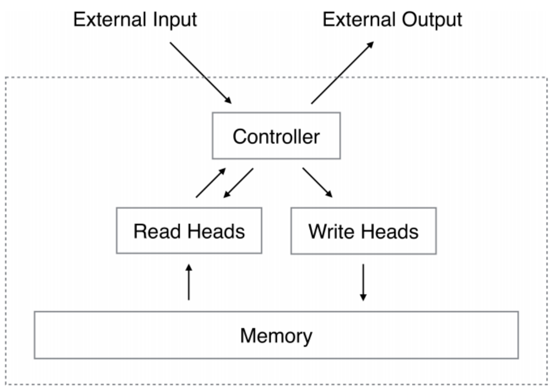


Figure 2.2: High level Overview of a NTM architecture. Extracted from [81]

Neural Turing Machines: An NTM consists of two main parts: a neural network controller and a memory bank. The controller can be any type of ANN including VRNNs, FNNs and LSTMs. In addition to processing input to output activations like most ANNs, NTM also interacts with an external memory bank using read and write operations. These are termed as read/write heads in analogy with a traditional Turing Machine. The memory bank is comprised of a large two dimensional matrix and is detached from the details of the controller. The interaction between controller and memory is entirely facilitated by the read and write heads.

In general, the process starts with the controller emitting a value and a read/write key. This value and key, alongside the state information of the network, is then mapped onto a distribution of weightings projected onto all the locations available in the memory by an addressing mechanism. This process takes into account the last location of memory that was addressed (location based addressing), and the similarity of the value emitted by the controller and the entries in the memory bank (content-based addressing) to produce a weighting that spans the length of the memory. A distributed weighting is often referred to as soft attention which indicates a distributed attention model over multiple memory locations. A weighting can also be made to focus entirely on one slot of memory, often referred to as hard attention. The read/write heads use this weightings to then read from memory or write to memory respectively.

In an analogy with Von Neumann architecture, the controller can be thought of as a Central Processing Unit (CPU) and the memory bank as Random Access Memory (RAM). The main addition however is the adaptability of a NTM which can be trained end to end with input-output examples. Each part of the controller, memory bank and the heads that modulate their interaction is fully differentiable and thus can be trained using gradient descent. The authors in [81] demonstrated the ability of an NTM using five tasks including copy, sort and associative recall whose solutions required chains of operations akin to an algorithm. NTM was shown to significantly outperform LSTMs in solving these algorithmic tasks and also demonstrated generalization beyond the size of task that it was trained for. This is a promising result which demonstrates signs of inductive reasoning that were facilitated by leveraging a large memory bank detached from the controller.

Differential Neural Computers: Subsequent work has introduced many variants of MANNs. Differential Neural Computers (DNCs) [84] expand the addressing mechanism of a NTM using dynamic memory allocation that tracks the usage of each memory location. The usage metric can be incremented and decremented for each write and read operation adaptively. This ensures that the allocated memory does not overlap and interfere with each other and also allows for freeing of memory that is no longer needed. An important feature here is that the tracking and allocation mechanism is independent of the size of memory, thus the DNC can be trained to solve a task on one size of memory and then later be upgraded to a larger size memory without the need for retraining. Additionally, a temporal link matrix tracks the locations in memory that were written preserving the order of writing. This is akin to 'pointers' and provides DNC with a natural way of recovering sequence in the order they were written in. The DNC was tested on synthetic question answering task meant to mimic reasoning and inference problems in natural language, and shown to solve them exhibiting these qualities. Additionally, DNC was also shown to successfully perform inference on randomly generated graphs, figuring out missing links on paths given starting and end positions. Results also demonstrated its ability to find shortest routes between multiple graph nodes. These results demonstrated DNC's ability to leverage memory to store a representation of the graph, interpret it, and conduct effective inference and reasoning using this knowledge.

Both NTMs and DNCs offer a parametric way to interact with an external memory component formulated to retain differentiability. This allows for gradient descent algorithms to train them end to end using input-output examples. The decoupled memory and controller structure also allows for reinforcement learning algorithms to be applied directly to train these structures [84, 242]. Direct search methods like EAs have also been explored to augment training. NEAT was modified to evolve NTMs in [87] with some success. Other variants in training procedure and architecture are explored in [205, 129, 232] for various question answering tasks using large memory sizes.

2.2 Reinforcement Learning

Markov Decision Process: A standard reinforcement learning setting is formalized as a Markov Decision Process (MDP) and consists of an agent interacting with an environment E over a number of discrete time steps. At each time step t, the agent receives a state s_t and maps it to an action a_t using its policy π. The agent receives a scalar reward r_t and moves to the next state s_{t+1}. The process continues until the agent reaches a terminal state marking the end of an episode. The return $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$ is the total accumulated return from time step t with discount factor $\gamma \in (0, 1]$. The goal of the agent is to maximize the expected return. The state-value function $\mathcal{Q}^{\pi}(s,a)$ describes the expected return from state s after taking action a and subsequently following policy π.

Markov Games: A standard reinforcement learning (RL) setting is often formalized as a Markov Decision Process (MDP) and consists of an agent interacting with an environment over a finite number of discrete time steps. This formulation can be extended to multiagent systems in the form of partially observable Markov games littman1994markov, lowe2017multi. An n-agent Markov game is defined by a full state $\mathcal{S}$ describing the global state of the world, a set of observations $\mathcal{O}_1, \mathcal{O}_2.., \mathcal{O}_n$ for each agent and their corresponding actions $\mathcal{A}_1, \mathcal{A}_2.., \mathcal{A}_n$, respectively. At each time step t, each agent observes its corresponding observation O_i^t and maps it to an action A_i^t using its policy π_i.

2.2.1 Deep Deterministic Policy Gradient (DDPG)

Policy gradient methods frame the goal of maximizing return as the minimization of a loss function $L(\theta)$ where θ parameterizes the agent. A widely used policy gradient method is Deep Deterministic Policy Gradient (DDPG) [141], a model-free RL algorithm developed for working with continuous high dimensional actions spaces. DDPG uses an actor-critic architecture [211] maintaining a deterministic policy (actor) $\pi : \mathcal{S} \rightarrow \mathcal{A}$, and an action-value function approximation (critic) $\mathcal{Q} : \mathcal{S} \times \mathcal{A} \rightarrow R$. The critic's job is to approximate the actor's action-value function $\mathcal{Q}^{\pi}$. Both the actor and the critic are parameterized by (deep) neural networks with θ^{π} and $\theta^{\mathcal{Q}}$, respectively. A separate copy of the actor π' and critic

$\mathcal{Q}'$ networks are kept as target networks for stability. These networks are updated periodically using the actor π and critic networks $\mathcal{Q}$ modulated by a weighting parameter τ.

A behavioral policy is used to explore during training. The behavioral policy is simply a noisy version of the policy: $\pi_b(s) = \pi(s) + \mathcal{N}(0,1)$ where $\mathcal{N}$ is temporally correlated noise generated using the Ornstein-Uhlenbeck process [223]. The behavior policy is used to generate experience in the environment. After each action, the tuple (s_t, a_t, r_t, s_{t+1}) containing the current state, actor's action, observed reward and the next state, respectively is saved into a **cyclic replay buffer** $\mathcal{R}$. The actor and critic networks are updated by randomly sampling mini-batches from $\mathcal{R}$. The critic is trained by minimizing the loss function:

$$L = \tfrac{1}{T}\textstyle\sum_i (y_i - \mathcal{Q}(s_i, a_i|\theta^{\mathcal{Q}}))^2$$
$$\text{where } y_i = r_i + \gamma \mathcal{Q}'(s_{i+1}, \pi'(s_{i+1}|\theta^{\pi'})|\theta^{\mathcal{Q}'})$$

The actor is trained using the sampled policy gradient:

$$\nabla_{\theta^\pi} J \sim \tfrac{1}{T}\textstyle\sum \nabla_a \mathcal{Q}(s, a|\theta^{\mathcal{Q}})|_{s=s_i, a=a_i} \nabla_{\theta^\pi} \pi(s|\theta^\pi)|_{s=s_i}$$

The sampled policy gradient with respect to the actor's parameters θ^π is computed by backpropagation through the combined actor and critic network.

2.2.2 Twin Delayed Deep Deterministic Policy Gradients

Policy gradient methods re-frame the goal of maximizing the expected return as the minimization of a loss function $L(\theta)$ where θ encapsulates the agent parameters. A widely used policy gradient method is Deep Deterministic Policy Gradient (DDPG) [141], a model-free RL algorithm developed for working with continuous, high dimensional actions spaces. Recently, Fujimoto et al. extended DDPG to Twin Delayed DDPG (TD3), [64] addressing the well-known overestimation problem of the former. TD3 was shown to significantly improve upon DDPG and is the state-of-the-art, off-policy algorithm for model-free deep reinforcement learning in continuous action spaces. TD3 uses an actor-critic architecture [211] maintaining a deterministic policy (actor) $\pi : \mathcal{S} \rightarrow \mathcal{A}$, and two distinct action-value function approximations (critics) $\mathcal{Q} : \mathcal{S} \times \mathcal{A} \rightarrow R_i$.

Each critic independently approximates the actor's action-value function $\mathcal{Q}^{\pi}$. The actor and the critics are parameterized by (deep) neural networks with θ^{π}, $\theta_a^{\mathcal{Q}}$, and $\theta_b^{\mathcal{Q}}$ respectively. A separate copy of the actor π' and critics: $\mathcal{Q}_a'$ and $\mathcal{Q}_b'$ are kept as target networks for stability. These networks are updated periodically using the actor π and critic networks: $\mathcal{Q}_a$ and $\mathcal{Q}_b$ regulated by a weighting parameter τ and a delayed policy update frequency d.

A behavioral policy is used to explore the environment during training. The behavioral policy is simply a noisy version of the policy: $\pi_b(s) = \pi(s) + \mathcal{N}(0, 1)$ where $\mathcal{N}$ is white Gaussian noise. After each action, the tuple (s_t, a_t, r_t, s_{t+1}) containing the current state, actor's action, observed reward and the next state, respectively, is saved into a **replay buffer** $\mathcal{R}$. The actor and critic networks are updated by randomly sampling mini-batches from $\mathcal{R}$. The critic is trained by minimizing the loss function:

$$L_i = \frac{1}{T} \sum_i (y_i - \mathcal{Q}_i(s_i, a_i|\theta^{\mathcal{Q}}))^2$$

$$\text{where } y_i = r_i + \gamma \min_{j=1,2} \mathcal{Q}'_j(s_{i+1}, \underset{a}{\sim} | \theta^{\mathcal{Q}'_j})$$

where $\underset{a}{\sim}$ is the noisy action computed by adding Gaussian noise clipped to between $-c$ and c. $\underset{a}{\sim} = \pi'(s_{i+1}|\theta^{\pi'}) + \epsilon,\ clip\big(\epsilon \sim \mathcal{N}(\mu,\, \sigma^2)\ - c, c\big)$

This noisy action used for the Bellman update smoothens the value estimate by bootstrapping from similar state-action value estimates. It serves to make the policy smooth and addresses overfitting of the deterministic policy. The actor is trained using the sampled policy gradient:

$$\nabla_{\theta^\pi} J \sim \tfrac{1}{T} \textstyle\sum \nabla_a \mathcal{Q}(s, a|\theta^{\mathcal{Q}}_a)|_{s=s_i, a=a_i} \nabla_{\theta^\pi} \pi(s|\theta^\pi)|_{s=s_i}$$

The sampled policy gradient with respect to the actor's parameters θ^π is computed by backpropagation through the combined actor and critic network.

2.2.3 Evolutionary Algorithm

Evolutionary algorithms (EAs) are a class of search algorithms with three primary operators: new solution generation, solution alteration, and selection [62, 198]. These operations are applied on a population of candidate solutions to continually generate novel solutions while probabilistically retaining promising ones. The selection operation is generally probabilistic, where solutions with higher fitness values have a higher probability of being selected. Assuming higher fitness values are representative of good solution quality, the overall quality of solutions will improve with each passing generation. In this work, each individual in the evolutionary algorithm defines a deep neural network. Mutation represents random perturbations to the weights (genes) of these neural networks. The evolutionary framework used here is closely related to evolving neural networks, and is often referred to as neuroevolution [59, 148, 179, 202].

Algorithm 1 A simple Evolutionary Algorithm (EA)

```
1: Initialize $k$ random solutions
2: for Each generation do
3:     for Each solution do
4:         Probabilistically perturb solution through mutation and/or crossover
5:         Assign fitness score based on performance
6:         Probabilistically select for survival commensurate with its fitness score
```

2.2.4 Cooperative Coevolutionary Algorithm

Cooperative Coevolutionary Algorithms (CCEAs) are an extension of Evolutionary Algorithms (EA) [198, 62] that deal with multiple evolving sub-populations, and have been shown to perform well in cooperative multi-robot domains [58]. Multiple populations evolve in parallel, with each population developing a policy for one of the robots in the team. Policies from each population are drawn to form a team of robots, and the overall performance of this team is evaluated using a fitness function $F(z)$, where z is the joint state of all robots in the team. This fitness is then assigned to each policy in the team.

The key difference between a CCEA and an EA is the assignment of fitness to each evolving robot's policy. In a traditional EA, the fitness of a policy is simply the system performance attained by that policy. However, in a CCEA, all the robots in the team affect the overall system performance; this means that the fitness of a robot's policy is based on its interactions with its teammates. In practice, teams are constructed by randomly sampling across populations resulting in context-dependent and subjective fitness assignment [37].

Chapter 3: Modular Memory Units

The first thread of research uses memory to better identify and associate reward with its corresponding actions that was taken in the past. Remembering key features from past observations, and using it towards future decision making is a crucial part of most living organisms that exhibit intelligent adaptive behaviors. 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 [12, 75]. The neuroscience literature has increasingly emphasized the importance of explicit working memory as a pillar for learning intelligent behavior [95, 220].

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. GRUs and LSTMs, variations of RNNs with gated activations have been widely applied to a range of domains and represent the state-of-the-art in many sequence processing tasks [79, 208]. Please see Chapter 2.1 for a comprehensive description of these neural architectures.

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) [82] and Differentiable Neural Computers (DNCs) [83]. Although they have been successfully applied to solve some complex structured tasks [89], these networks are typically quite complex and unwieldy to train [107]. 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.

To address this issue, we 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.

3.1 Methodology

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 3.1 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

$$\begin{aligned} i^t &= \sigma\left(K_i x^t + R_i y^{t-1} + N_i m^{t-1} + b_i\right) && \textit{Input gate} \\ p^t &= \phi\left(K_p x^t + N_p m^{t-1} + b_p\right) && \textit{Block input} \end{aligned} \tag{3.1}$$

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

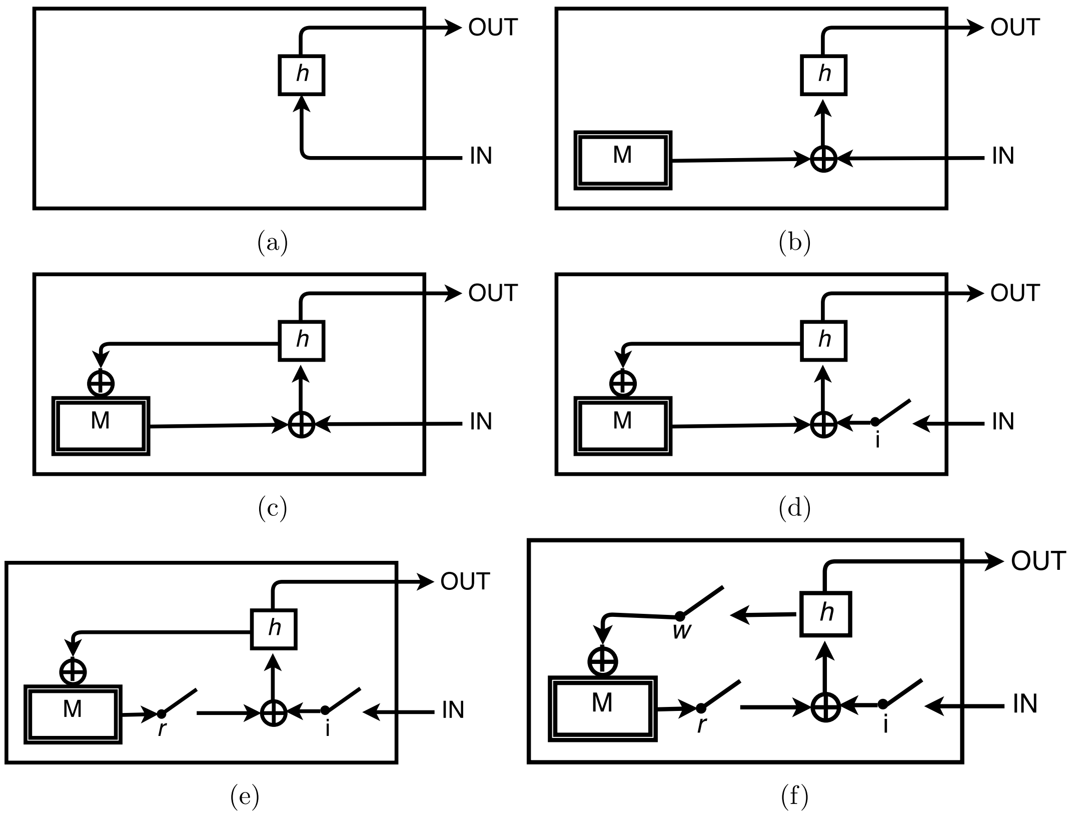


Figure 3.1: Illustration of a bottom up construction of a MMU: (a) We start with a simple feedforward neural network, whose hidden activations are marked as h. (b) We add a disconnected memory block, which now contributes information to the hidden activation. (c) We close the loop, and allow the hidden 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.

memory d^t is computed as:

$$\begin{aligned} r^t &= \sigma\left(K_r x^t + R_r y^{t-1} + N_r m^{t-1} + b_r\right) && \textit{Read gate} \\ d^t &= \theta\left(N_d m^{t-1} + b_d\right) && \textit{Memory decoder} \end{aligned} \tag{3.2}$$

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.

$$h^t = r^t \odot d^t + p^t \odot i^t \qquad \textit{Hidden activation} \tag{3.3}$$

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.

$$\begin{aligned} w^t &= \sigma\left(K_w x^t + R_w y^{t-1} + N_w m^{t-1} + b_w\right) && \textit{Write gate} \\ f^t &= \theta\left(Z_f h^t + b_f\right) && \textit{Memory encoder} \\ m^t &= (1-\alpha)\left(m^{t-1} + w^t \odot f^t\right) + \alpha\left(w^t \odot f^t + \left(1 - w^t\right) \odot m^{t-1}\right) && \textit{Memory update} \end{aligned} \tag{3.4}$$

The new output y^t is then computed as

$$y^t = \phi\left(Z_y h^t + b_y\right) \qquad \textit{New output} \tag{3.5}$$

Equations (3.1)-(3.5) 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. 3.1). 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

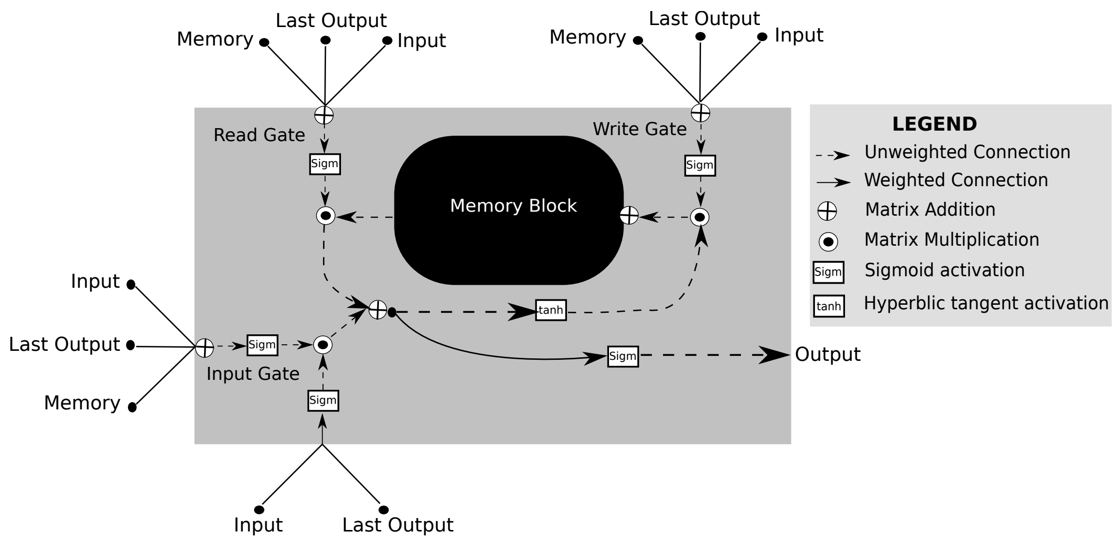


Figure 3.2: MMU neural architecture detailing the weighted connections and activations used in this paper. *Last Output* represents the recurrent connection from the previous network output. *Memory* represents the peephole connection from the memory block.

to challenge this axis of difficulty. 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 3.2 and Equation 3.4 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 3.2 shows the detailed schematic of the MMU instance.

Training We use neuroevolution to train our MMU network. However, memory-augmented architectures like the MMU, can be a challenge to evolve, particularly

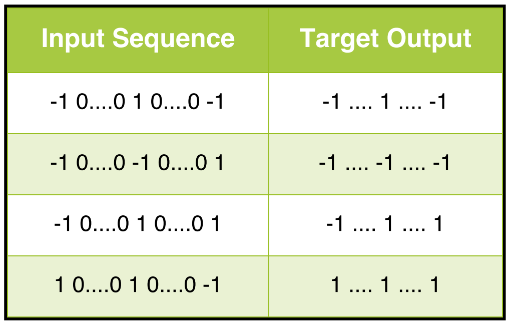
Algorithm 2 Neuroevolutionary algorithm used to evolve the MMU network

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

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 5 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.

3.2 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 [176]. 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



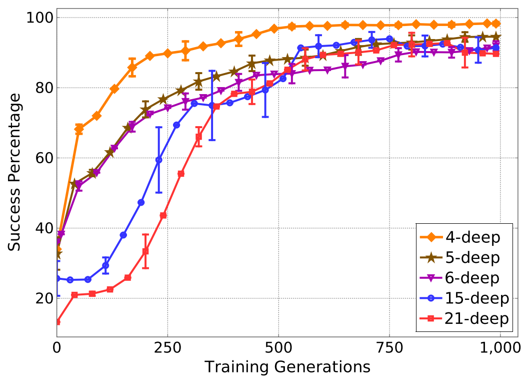
Input Sequence	Target Output
-1 0....0 1 0....0 -1	-1 .... 1 .... -1
-1 0....0 -1 0....0 1	-1 .... -1 .... -1
-1 0....0 1 0....0 1	-1 .... 1 1
1 0....0 1 0....0 -1	1 .... 1 1

Figure 3.3: 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].

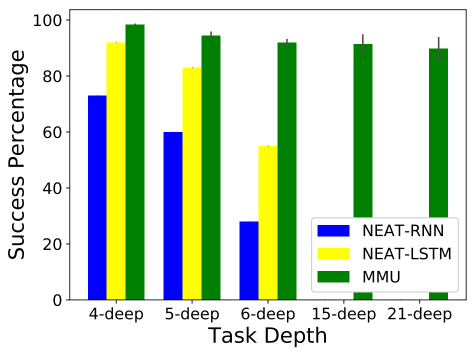
noise. The network is given a sequence of 1/-1 signals, interleaved with a variant number of 0's in between. After each introduction of a signal, the network needs to decide whether it has received more 1's or -1's. The number of signals (1/-1s) in the input sequence is termed the depth of the task. 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.

Further, the number of distractors (0's) is determined randomly following each



(a)



(b)

Figure 3.4: (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 [176]). 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.

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 3.3 describes the Sequence Classification task and the input-output setup to our MMU network.

3.2.1 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 3.4a 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 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.

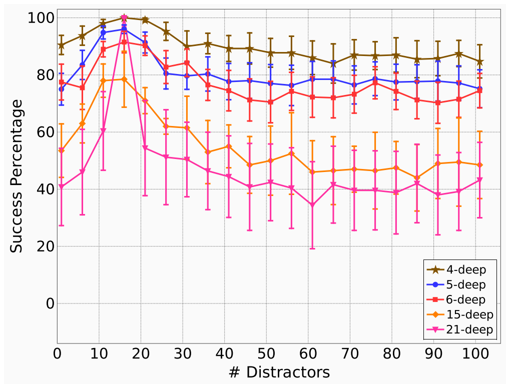
To situate the results obtained using the MMU architecture, we compare them against the results from [176] for NEAT-LSTM and NEAT-RNN (Fig. 3.4b). These results for NEAT-LSTM and NEAT-RNN represent the success percentage after 15,000 generations while the MMU results represent the success percentage after 1,000 generations. MMU demonstrates significantly improved success percentages across all task depths, and converges in 1/15th of the generations.

3.2.2 Generalization Performance

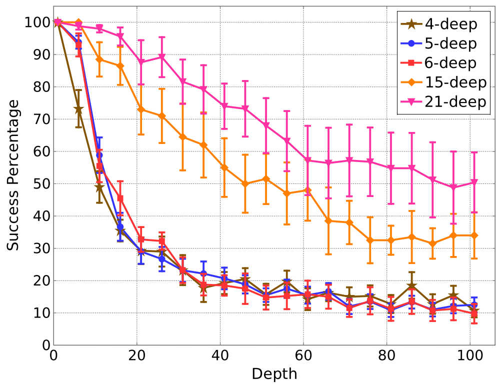
Section 3.2.1 tested the networks for the ability to generalize to new data using test sets. In this section, we take this notion a step further and test the network's ability to generalize, not only to new input sequences, but to tasks with longer depths or noise sequences than what they were trained for.

3.2.2.1 Generalization to Noise

Figure 3.5(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 between noise sequences of length between 10 and 20. This is expected as this is the length of noise sequences that



(a)



(b)

Figure 3.5: (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.

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.

3.2.2.2 Generalization to Depth

Figure 3.5(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% ± 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.

3.2.3 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.

3.2.3.1 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.

Table 3.1 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

#	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 3.1: 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).

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.

3.2.3.2 Role of Selective Memory Access

We investigate the role of selective memory access facilitated by the read/write gates. Figure 3.6 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

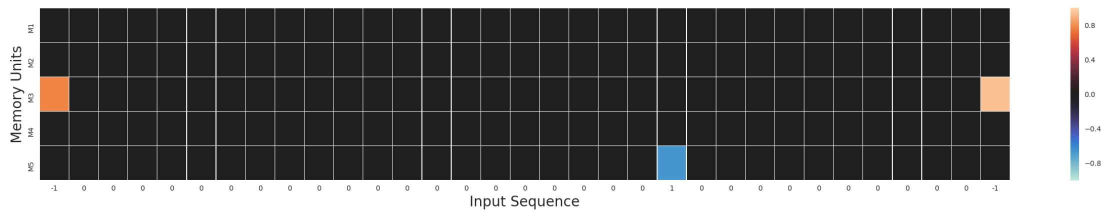


Figure 3.6: Illustration of the write interaction between memory and the central feedforward network in a 3-deep #3 permutation task from Table 3.1. The input sequence contains three signals [-1,1,-1] with 0s interleaved between them (x-axis of the heatmap). The color scheme corresponds to a scale of memory update magnitudes. Write activations in response to each individual input is shown for each of the five memory units.

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 3.7 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 3.6) 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

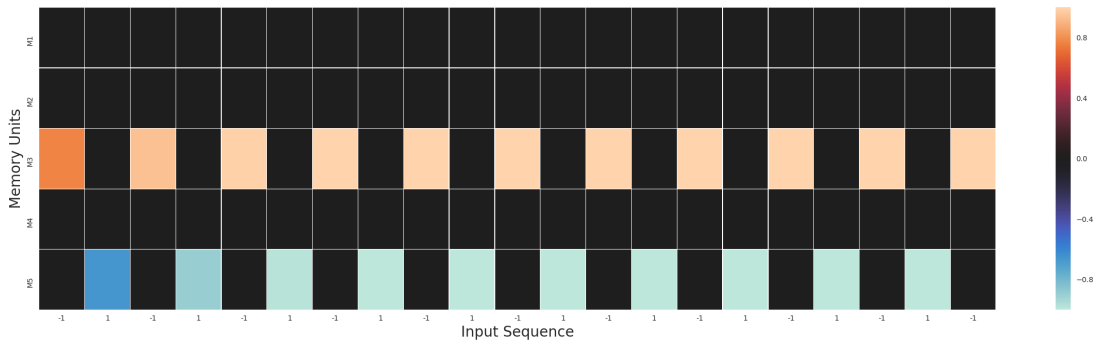


Figure 3.7: Illustration of the write interaction between memory and the central feedforward network in a 21-deep task with input sequence that followed the format [-1,1,-1,1,-1...] (flipping subsequent signal) interleaved with a variable number of 0s. The activations for the noise inputs (0s) are not shown but were all constantly 0s.

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.

3.2.3.3 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 3.2: 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.

3.2.3.4 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 3) 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.

Algorithm 3 Approximate pseudo-algorithm derived from the operations of the 21-champ MMU network

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

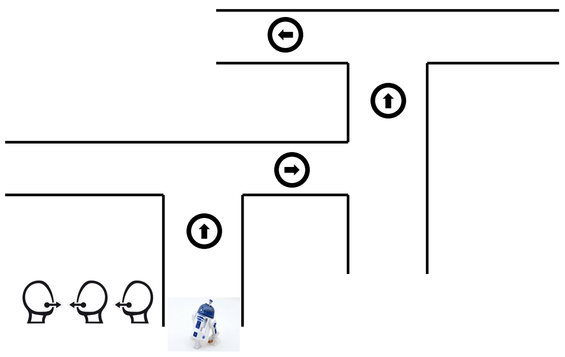


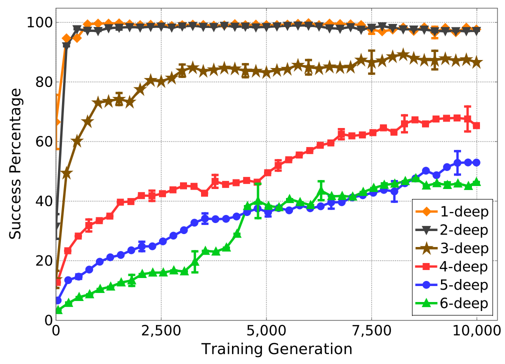
Figure 3.8: Sequence Recall task: An agent listens to a series of instructional stimuli (directions) at the start of the trial, then travels along multiple corridors and junctions. The agent needs to select a turn at each junction based on the set of directions it received at the beginning of the trial.

3.3 Experiment 2: Sequence Recall

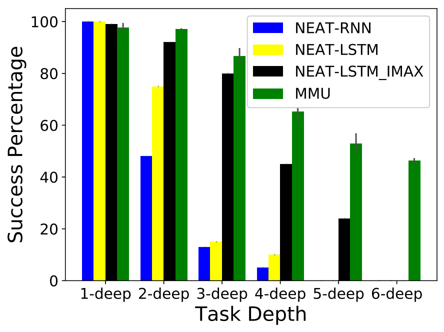
For our second experiment, we tested our MMU architecture in a Sequence Recall task (Fig. 3.8), which is also sometimes referred to as a T-Maze navigation task. This is a popular benchmark task used extensively to test agents with memory [14, 30, 176]. In a simple Sequence Recall task, an agent starts 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 [176], which consist of a sequence of independent junctions (Fig. 3.8). 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.

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



(a)



(b)

Figure 3.9: (a) Success rate for the Sequence Recall task of varying depth. (b) Comparison of MMU with NEAT-RNN and NEAT-LSTM (extracted from [176]). 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.

of noisy inputs.

3.3.1 Results

Figure 3.9a 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 3.9b 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 [176] 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

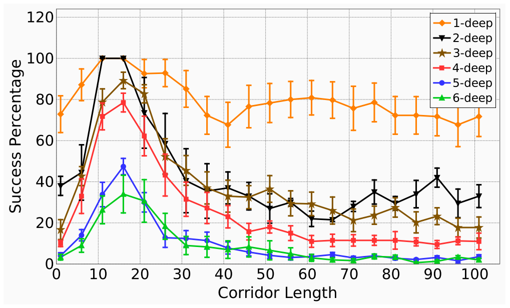


Figure 3.10: MMU network's generalization to arbitrary corridor lengths in the Sequence Recall task. All MMUs shown were trained with corridor lengths randomly sampled between 10 and 20. These were then tested for corridor lengths ranging from 0 to 101.

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. 3.9b, MMU achieves significantly higher success percentages for 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-Info-max. The influence of the unsupervised pre-training phase was shown in [176] to significantly improve performance over networks that do not undergo pre-training (also shown in Fig. 3.9b). 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.

3.3.2 Generalization To Noise

Figure 3.10 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.

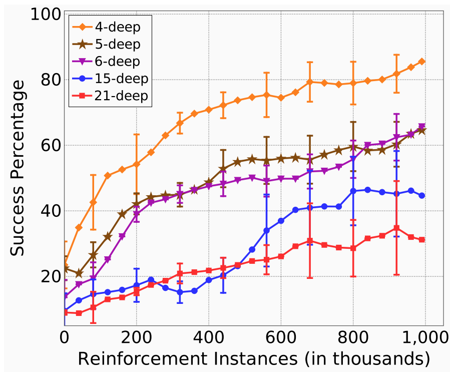
Overall, the networks demonstrate a modest ability for generalization towards varying corridor lengths. The peak performance across all the task depths is seen between noise sequences of length between 10 and 20. This is expected as this is the length of noise sequences that the networks were trained 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 3.2.2.1. This is reflective of the added complexity of the Sequence Recall task relative to the Sequence Classification task.

Similar to Section 3.2.2.1, 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.

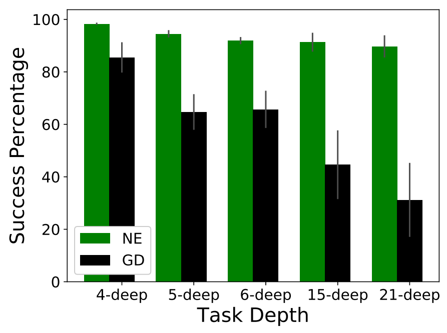
3.4 Differentiability

Experiments in Sections 3.2 and 3.3 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 3.2 and 3.3.

PyTorch [165], a Python based open source library for symbolic differentiation

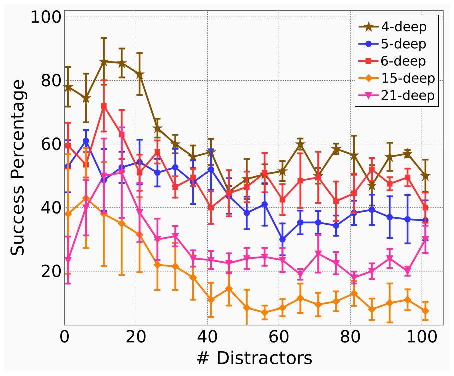


(a)

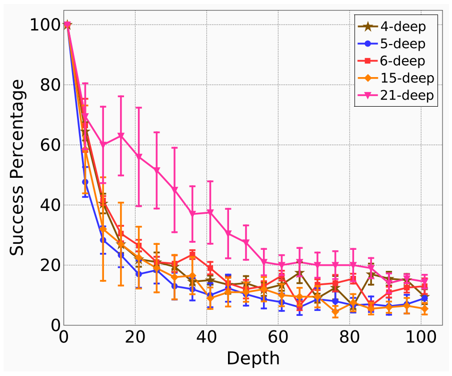


(b)

//



(c)



(d)

Figure 3.11: (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.

was used to implement the differentiable version of the MMU network. Adam [123] 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 [96]. 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.

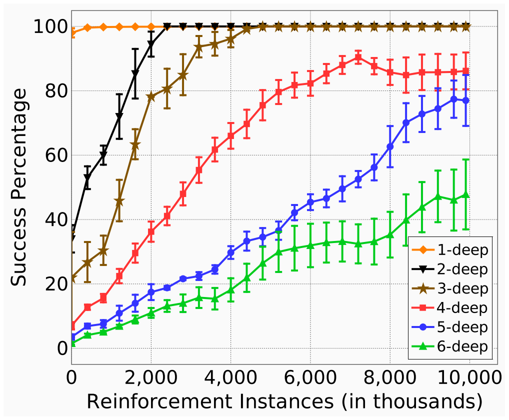
In this experiment, we perform these comparisons by controlling the number of **Reinforcement Instances** for each training algorithm. We define reinforcement instances as the number of times the method uses a training example to obtain reinforcement about its attempted solution. For neuroevolution,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

3.4.1 Sequence Classifier

Figure 3.11 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 3.4a in Section 3.2). 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 3.4a) 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 3.2), generalizability to larger numbers of interleaving noise, and depths is significantly reduced (shown in Figure 3.11c and 3.11d). A network trained using gradient descent in the 21-deep task achieved 14.5% ± 2.30% on a 101-deep task while a similar experiment for a network trained with neuroevolution yielded 50.4%±9.30% (Section 3.2.2.2). 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.

3.4.2 Sequence Recall



(a) (b)

Figure 3.12: (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 3.3.

Figure 3.13 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 3.3 Figure 3.10).

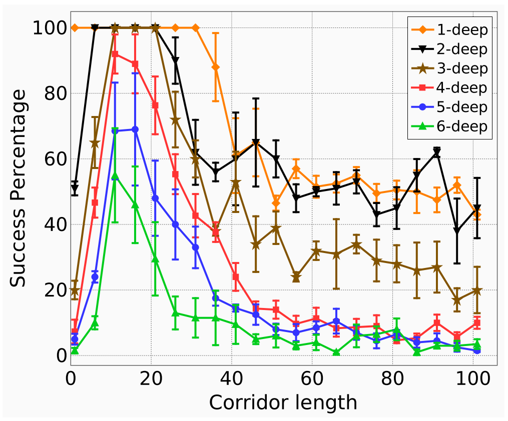


Figure 3.13: MMU network's generalization to arbitrary corridor lengths in the Sequence Recall task. All MMUs shown were trained for tasks of their respective depths with corridor lengths randomly sampled between 10 and 20.

Figure 3.12 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 3.9a in Section 3.3). 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, gradient descent is thus able to propagate errors backwards and successfully solve the Sequence Recall task.

3.5 Conclusion and Future Work

Credit assignment is extremely difficult when the associated reward for an action is only observed following after a period of time. Memory enables and agent to make these associations, and is an integral component of learning . 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.

MMU, by virtue of its decoupled topology 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 activation. Results in two benchmark deep memory tasks demonstrate that MMU significantly outperforms traditional memory-based methods. Further, MMU also exhibit robust generalization properties and can be trained readily using neuroevolution and gradient-based techniques.

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.

Chapter 4: Memory-Based Distributed Multiagent Coordination

In the previous chapter, we demonstrated how memory could be used to improve temporal credit assignment - making better associations between reward and actions taken in the past. However, it is unclear whether these augmented capabilities can be leveraged in multiagent settings where tight coordination between a number of agents is required. This chapter explores two distinct research applications dedicated to tackling this goal. Section 4.1 details the first application where memory is used to augment each agent policies in order to solve dynamic tasks that require adaptive behavior. Section 4.2 details another application where an external memory is used as a shared knowledge base among a team of agents for distributed one-shot decision making.

4.1 Multiagent Coordination with Memory-Augmented Teams

Autonomous multi-robot teams can accomplish complex tasks in highly dynamic and stochastic environments improving on both speed and effectiveness over single robot approaches. However, multi-robot coordination is a complex control problem especially when the task requires adaptive behaviors from the group of coordinating robots.

The requirement for adaptive behavior imposed by the task definition institutes an added layer of complexity in learning. This complexity is orthogonal to the one imposed by the need for adaptation to each other, inherent in a team of robots acting concurrently. It is important to disambiguate these two requirements for adaptive behavior, and consider each independently. The first statement of adaptation is imposed by the environment itself. In well-defined, static and stable environments, fixed and reactive robot behavior is generally sufficient. However, when the environment is ever changing, highly dynamic, or is responsive to the robot's own policies, adaptive behaviors provide a huge selective advantage.

An example would be a mobile robot operating in a factory, that is tasked with moving a package from Point A to Point B every hour. The robot takes route Z which is the most efficient way to travel from A to B. However, sometimes this path is obstructed due to construction or unforeseen disturbances. An adaptive robot here would take path Z and observe the obstruction a couple of times, but then would alter its policy to take another path Y for the next couple of deliveries as it adapts to the obstruction.

Extending these adaptive tasks to a multi-robot system adds another layer of complexity imposed by the concurrent actions of multiple robots operating within the same environment. The individual robots could be learning or executing static policies, and can differ in capability or objective. The robots may even differ by versions, where a new iteration of robots are added to the team for the same task.

For example, a new and more capable generation of package delivery robots may be introduced to the factory in our previous example. Instead of entirely decommissioning or reprogramming the previous team, it is cheaper and much more effective to build the new team of robots to work alongside the team that is already operational in the factory. Adapting to the previous team would allow the new team of robots to augment its capability, and leverage the resources already present in the factory towards a more effective strategy for package delivery. As robots are increasingly deployed in the real world, such adaptive behaviors will become increasingly important. This is particularly true in commercial applications like cleaning robots, mowing robots or factory robots where heterogeneity within the same task can be frequently borne out of iterations through the years, and adaptive behavior will be crucial for effective reconciliation.

Features relevant to adaptive decision making are often distributed over a robot's current as well as its past states. Remembering past events can allow a robot to more effectively adjust its behavior and adapt to local changes in the environment [50, 39]. Memory can also allow robots to better model other robots' capabilities, limitations and policies, empowering them to form better joint strategies and more complex coordination protocols. Adaptive behaviors for single robot systems have been explored in the past largely with the tools of memory [15, 17, 200]. However, the addition of further complexity in the form of other robots acting concurrently is not widely explored for this class of adaptive tasks.

A body of work has leveraged memory to train adaptive behaviors in single agent systems. Grabowski et al.[74] looked at the evolution of memory usage in environments where information about past experience is required for optimal decision making. Bakker used reinforcement learning in conjunction with a Long Short Term Memory (LSTM) to solve a single T-Maze task [15]. Bayer et al.

evolved varying LSTM cell structures [17] while Greve et al. recently evolved Neural Turing Machines [88] to solve a more complicated version of the T-Maze. However, all of these work deal with single agent systems that learn adaptive behavior as exacted by the environment. The addition of further complexity in the form of other agents acting concurrently is not widely explored for this class of adaptive tasks.

Extending single agent approaches to multiagent settings presents challenges in ensuring each agent learns to act in a way that benefits the entire team. A viable decomposition of the team goal is rarely available or easily computable, which necessitates for agents that learn to adapt to each other. Stone et al. [203] highlights this need for collaboration of autonomous agents without preordained protocols for coordination. Distributed policy learning is well suited to these kinds of problems and has received much attention in the field of multiagent coordination. Colby et al. [38] utilized a cooperative coevolutionary algorithm to train multi-robot teams exploring an unknown space environment with humans in the loop. Knudson and Tumer [126] co-evolved multi-robot teams to solve a tightly coupled task requiring multiple robot observations and extended it to heterogeneous multi-robot teams with varying capabilities. Hsieh et al. [104] developed a framework for deployment of an adaptive heterogeneous team consisting of aerial and ground robots, for an urban surveillance task.

We leverage the developments towards adaptive behavior facilitated through the integration of memory, and extend it to tasks that require, or benefit from, a team of robots learning to adapt and coordinate with each other. However, to benchmark out approach, we first introduce an extended T-Maze domain which serves to marry the difficulties of a multi-robot system with the complication of adaptive behavior prescribed by the task itself.

4.1.1 Extended T-Maze Domain

The T-Maze is a popular benchmark domain used to test for memory based adaptive behaviors in single robot applications [15][88]. We extend the T-Maze domain to a multi-robot paradigm where multiple robots traverse the maze in search for the reward. The location of the reward is changed periodically, and multiple robots are required to coordinate tightly in exploring and observing the reward. The corridors of the maze are identical and indistinguishable which prevents the robots from learning explicit value functions for the states. The robots are instead forced to coordinate tightly with their teammates and learn joint strategies that adapt to the changing reward location.

Figure 4.1 illustrates a 2-deep extended T-Maze domain with 2 robots. The number of junctions that a robot has to navigate past to reach the endpoint is termed the depth of the T-Maze. The 2-deep T-Maze with two binary decision points (junctions) consists of four separate endpoints marked A, B, C and D. Each robot starts off as shown, and navigates to one of the endpoints. This process is termed a **trial**. During each trial, the reward is present at one of the endpoints, and this location changes 2 times over the course of one **evaluation**. One full evaluation consists of n trials where n is the number of unique endpoints multiplied

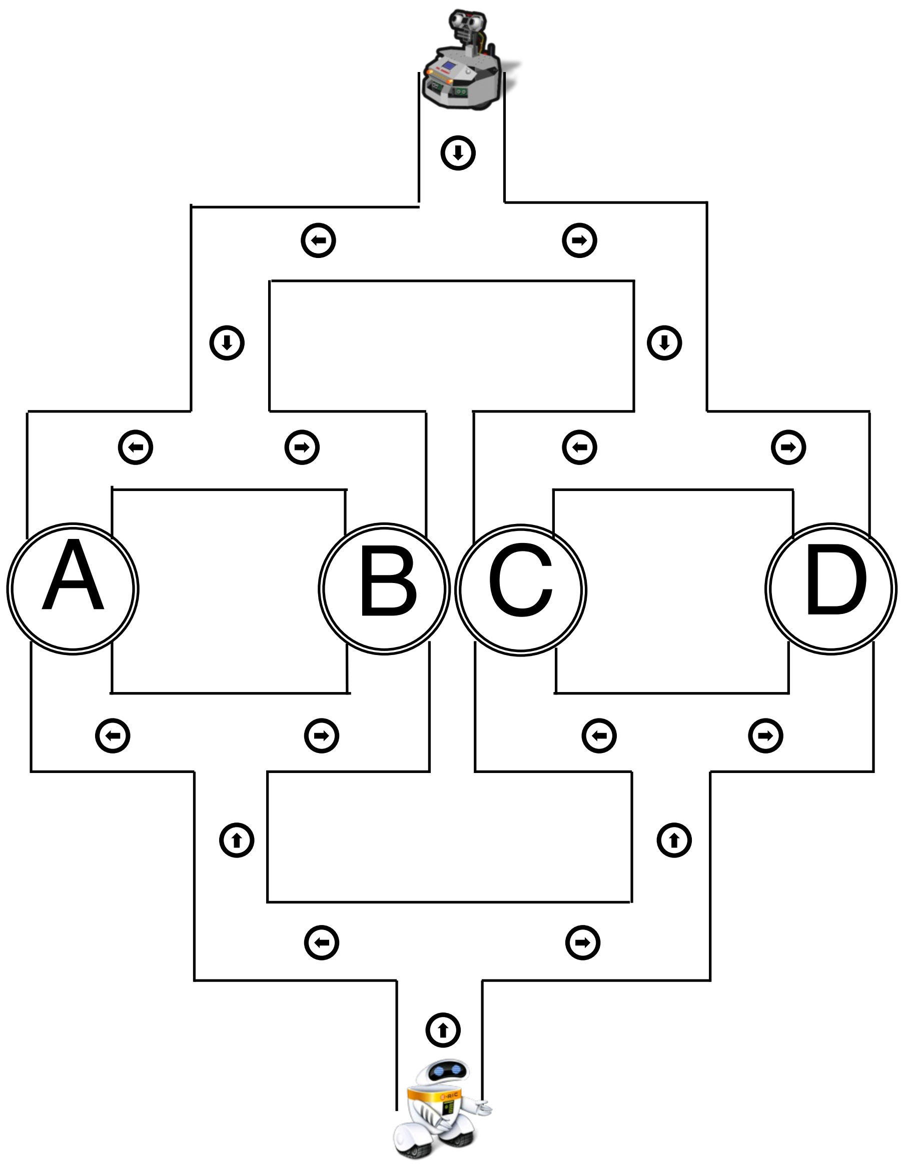


Figure 4.1: Illustration of a 2-deep Extended T-Maze with 2 robots. Each robot travels through the corridor and goes left or right at each junction, ending up in one of the endpoints A, B, C or D. The number of robots can be scaled arbitrarily by adding orthogonal planes sharing the endpoints.

by 3. For example, a 2-deep T-Maze has 4 unique endpoints leading to 12 trials per evaluation.

The robot records three values as its input. The first input measures the distance to the next junction. The second and third inputs are binary measurements that specify whether other robots were present and whether reward was observed, respectively, in the robot's last trial. The goal of the robot is to maximize the number of trials in which it observes the reward in coordination with the other robots. In a single robot T-Maze case, a policy would be to explore the endpoints in some defined sequence until the reward is observed. The robot then revisits the endpoint where the reward was observed until the location of the reward changes. At this point, the robot then resumes exploration of the endpoints to find the new location. This policy is optimal in the single robot case, and will be termed the **static policy**.

The extended T-Maze domain marries two distinct requirements for adaptive behaviors in one general framework. The first requirement for adaptation is imposed by the task itself where the robot needs to alter its strategy in response to the change of reward location. We will refer to this as **task adaptation**. The difficulty level of task adaptation can be altered by modulating the depth of the task. The second requirement of adaptive behavior is imposed by concurrent actions of multiple robots acting within the same environment and will be referred to as **multi-robot adaptation**. Figure 4.1 illustrates two robots acting together, but the number of robots can be scaled arbitrarily. The nature of multi-robot adaptation can be specified by the *objective function* that maps the joint action of the multi-robot team to a system performance value. In this work, we specify two such distinct objective function definitions:

Converge This objective function definition requires the reward and all the robots to be at the same endpoint for the reward to be considered observed in that trial. This is a tightly coupled objective definition which represents tasks like picking up heavy objects that might not be solvable by one robot, and explicitly require the co-presence of multiple robots.

Spread This is a loosely coupled objective function definition which requires the reward and a minimum of one robot to be at the same endpoint for the reward to be considered observed in that trial. This objective function represents tasks like exploring or information collection which can be completed by one robot, but is able to benefit from larger team sizes.

4.1.2 Multi-Robot Policy Training with Memory

We use a MMU network as our robot control policy leveraging its unique architecture that decouples memory from its feedforward computation. This detachment is crucial as our robots must be able to effectively filter noise and capture the relevant information over the course of a traversal of the Extended T-Maze. For example, a successful robot control policy should be able to ignore the noisy inputs as it travels the corridors of the T-Maze. Simultaneously however, in order to effectively localize itself, the robot should be able to remember the number of junctions it has passed, as each corridor of the Extended T-Maze looks virtually identical. Additionally, the robot also has to memorize relevant information about its last exploration endpoint so that it can explore a new one for its next trial.

We use CCEA modified for neuroevolution [71][59] to train our neural network control policies. However, memory-augmented architectures like the MMU, can be a challenge to evolve, particularly due to the large number of weights that parameterize highly non-linear and correlated operations. To alleviate this problem, we use independent mutational operators for each weight structure within our neural architectures, and use probabilistic extinction to retain diversity in the population.

Algorithm 5 details the CCEA used to evolve our multi-robot teams. The population k was set to 100, and the number of elites to 4% of k. M represents the size of the team and varies between $2-4$ for varying experiments. The number of randomly initialized simulations conducted to compute an evaluation, z was set to 7. The number of different teams an individual robot was drafted to, in order to compute its fitness, ϕ was set to 5. The mutation probability mut_{prob}, extinction probability $extinct_{prob}$ and extinction magnitude $extinct_{mag}$ were set to 0.9, 0.004, and 0.5 respectively.

4.1.3 Experimentation

We design a variety of experimental setups to evaluate out neural network controllers in the Extended T-Maze domain. The axes of variations that characterize our experiments are summarized below:

Algorithm 4 CCEA used to evolve our multi-robot teams

```
1: Initialize $M$ populations of $k$ neural network policies
2: Define a random number generator $r() \in [0, 1)$
3: for each $Generation$ do
4:     for $i = 1 : k \times \phi$ do
5:         Select a network from each population randomly
6:         Add networks to team $T_i$
7:         Run $z$ independent simulations for team $T_i$
8:         Assign reward to each members of $T_i$
9:     for each $Population$ do
10:        Rank networks based on fitness scores
11:        Select the first $e$ networks as elites
12:        Probabilistically select $(k - e)$ networks from the entire pool based on
           fitness, to form Set $S$
13:        for iteration $(k - e)$ do
14:            Randomly extract network $N_i$ from Set $S$
15:            for each weight matrix $W \in N_i$ do
16:                if $r() < mut_{prob}$ then
17:                    Randomly sample individual weights in $W$ and mutate them
                       by adding 10% Gaussian noise
18:        if $r() < extinct_{prob}$ then
19:            for each Network $N_i$ not in elites do
20:                if $r() < extinct_{mag}$ then
21:                    Reinitialize $N_i$'s weights
```

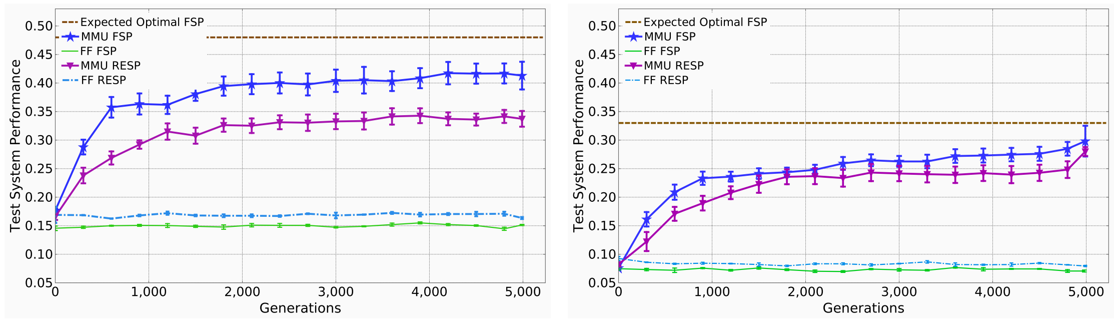


Figure 4.2: MMU and FF control policies are compared for a 2-deep (a) and 3-deep (b) tightly coupled Extended T-Maze, where both the static and learning robot have to be at the same endpoint alongside the reward to observe it.

Control Policy Architecture We test two distinct types of neural architectures as control policies that learn. The first kind is a **MMU network** that has memory, while the second type is a standard **feedforward neural network (FF)** without any memory. The same training approach described in Section 4.1.2 was used to evolve the FF control policies. This axis of variation evaluates the role of memory.

Robot Type We have two categories of robots working together in a team. The first type of robots are **learning robots** that use either a MMU or feedforward neural network as their policy, and are trained. The secondary category of robots are **static robots** that use the static policy as described in Section 4.1.1. The robot under the static policy explores the endpoints in a specific order, returns to an endpoint if it finds a reward there, and resumes exploring as the reward location is switched. This strategy is the optimal strategy for the T-Maze in a single robot case, and represents a hard-coded optimal policy that is already operational on site in a real world setting. The learning robot's goal is to augment the static robot by adapting to work alongside it.

Static Robot Type We test two types of static robots based on their exploration policy. The first type termed **fixed static policy (FSP)** will be a static robot that has a fixed endpoint it starts with and a fixed routine of exploration. The second type termed the **randomly exploring static policy (RESP)** will vary the endpoint it starts with, and the pattern in which it explores. The RESP represents an added challenge to the learning robots as they have to additionally infer the type of static robot they are teamed with, by observing them during the first couple of trials.

The protocol for computing the results were kept consistent across all the experimental variations. During each generation of training, the individual robot with the highest fitness from its population was selected to be part of a champion team. The champion team was then tested on a separate test set of 15 experiments, generated randomly for each generation. The average reward achieved by the champion team normalized by the number of trials, on this test set was then logged and reported as the **Test System Performance** for that generation. This protocol of a test set was implemented to shield the reported metrics from any bias of the distribution of random reward locations, depth of the task, or the size of the population. Ten independent statistical runs were conducted, and the average with error bars reporting the standard error in the mean were logged.

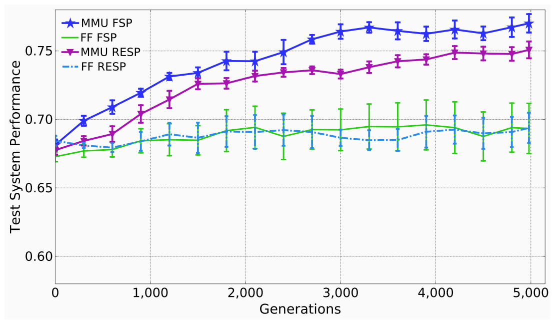
4.1.4 Experiment 1: Converge

This set of experiments test the multi-robot teams consisting of one static robot and one learning robot, under the converge objective function as explained in Section 4.1.1. This is a tightly coupled task definition where all the robots are required to be alongside the reward to observe it.

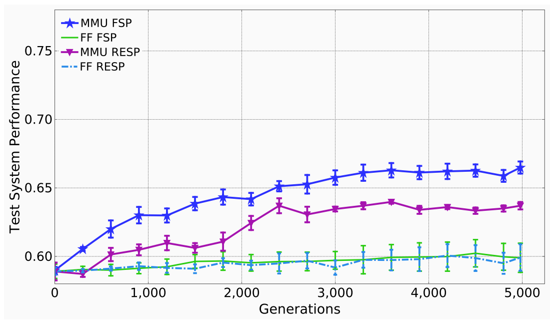
Figure 4.2(a) shows the results for a 2-deep Extended T-Maze domain tested against robots with both fixed (FSP) and randomly exploring static policies (RESP). The brown dashed line represents the expected system performance for an optimal joint policy between the learning robot and the FSP robot. The expected value for optimal joint policy alongside the RESP is not easily computable, but is provably lower than the FSP case. This is apparent as FSP is simply a special case of RESP where the type of robot exploration routine is fixed. The robot using the MMU network significantly outperforms the robot using the feedforward network as its policy across both types of static robot types. The MMU control policy achieves a system performance of $41.3 \pm 2.4\%$ and $33.7 \pm 1.4\%$ while the FF control policy achieves a system performance of $15.0 \pm 0.1\%$ and $16.5 \pm 0.2\%$ for the FSP and RESP variations respectively.

Figure 4.2(b) shows the results for a 3-deep Extended T-Maze domain tested against robots with both fixed (FSP) and randomly exploring static policies (RESP). The brown dotted line represents the expected system performance for an optimal joint policy between the learning robot and the FSP. The robot using the MMU network significantly outperforms the robot using the feedforward network as its control policy across both types of static robot types. The MMU control policy achieves a system performance of $29.8 \pm 2.7\%$ and $28.0 \pm 1.5\%$ while the FF control policy achieves a system performance of $7.1 \pm 0.2\%$ and $8.0 \pm 0.1\%$ for the FSP and RESP variations respectively. The MMU robots' performance alongside the FSP static robot variation has also not fully converged and seems to be approaching the expected optimal joint policy value of 33%.

The results in Figure 4.2 demonstrate that the robot using the MMU network is able to learn effective policies that let it adapt to the varying types of static robots. The robot using the FF network however fails to learn any intelligent joint policy at all. The critical factor behind this disparity is memory, which is the primary point of variation between the MMU and FF network architecture. The FF robots do not have enough input information to form an effective strategy that can learn to exploit the reward when its location is known, and explore when it is unknown. Additionally, the FF robots also do not have enough information to model their teammates' policies and built a strategy around it. An immediate mapping between their current input measurements and an effective exploration strategy or joint action policy is not existent. The MMU robots on the other hand have access to the same information (input) as the FF robots, but have the additional capability of memory. This ability to memorize past events and process temporal dependencies in their input channels forms a type of information bridge that can be leveraged to form effective exploration strategies that can adapt to the change in reward location. Additionally, the availability of memory also allows the MMU robot to model its teammates' policy and adapt alongside it to maximize



(a)



(b)

Figure 4.3: MMU and FF control policies are compared for a 2-deep (a) and 3-deep (b) Extended T-Maze domain, using the Spread objective where any one robot is sufficient in observing the reward.

system performance. This is especially apparent in the RESP case where in order to succeed, the MMU robot is required to first samples its colleague static robots' actions, identify its exploration strategy and adapts its behavior to best supplement that strategy. This behavior highlights the crucial role memory plays in learning adaptive behaviors that facilitate tight multi-robot coordination.

4.1.5 Experiment 2: Spread

This set of experiments test the multi-robot teams under the spread objective function as explained in Section 4.1.1. This is a loosely coupled task definition where a minimum of one robot is required to be alongside the reward to observe it.

Figure 4.3(a) shows the results for a 2-deep Extended T-Maze tested against both FSP and RESP robots. The expectation for the lower bound of performance (not plotted) is 48% and represents the worst possible system performance achievable by the learning robot. This is the expected system performance achieved simply by the hard coded policy of the static robot. The robot using MMU network outperforms the robot using the feedforward network as its policy across both types of static robot types. The MMU control policy achieves a system performance of $77.0 \pm 0.7\%$ and $75.1 \pm 0.6\%$ while the FF control policy achieves a system performance of $69.3 \pm 2.0\%$ and $69.4 \pm 1.2\%$ for the FSP and RESP variations respectively.

Figure 4.3(b) shows the results for a 3-deep Extended T-Maze domain tested against both FSP and RESP robots. The expectation for the lower bound of performance (not plotted) is 33%. The robot using the MMU network outperforms the robot using the feedforward network as its policy across both types of static robot policies. The MMU control policy achieves a system performance of $67.5 \pm 0.4\%$ and $63.7 \pm 0.3\%$ while the FF control policy achieves a system performance of $60.0 \pm 1.0\%$ and $60.0 \pm 0.8\%$ for the FSP and RESP variations respectively.

Similar to the results in Section 4.1.4, the results in Figure 4.3 demonstrate the MMU robot learning to form effective joint policies alongside both types of static robots. The FF robots however fail to learn and simply picks an endpoint to stick to. It fails to adapt to the task where the reward location changes periodically, and to the nature of the static robot's exploration strategy. The MMU robot however is able to leverage its memory and adapt to both the task, and the varying exploration strategies of its teammates, effectively augmenting the static robots in the task.

4.1.6 Multiple Learning Robots

To further probe our approach on the axis of **multi-robot adaptation**, we ablate the robot with static policy from our multi-robot team, and experiment with learning robots of varying team sizes.

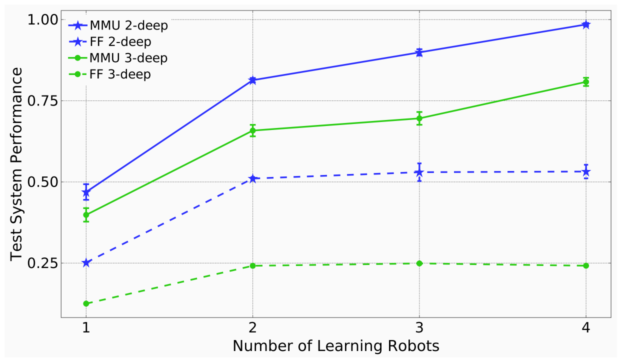


Figure 4.4: MMU and FF control policies are compared for a 2-deep and 3-deep Extended T-Maze, with Spread objective tested for varying number of learning robots.

Figure 4.4 shows the results for a 2-deep and 3-deep Extended T-Maze tested for varying number of learning robots. The robots using MMU network significantly outperform the robots using the feedforward network as its control policy across all values for team size and task depth. The system performance for the MMU robots improves with increasing number of robots for both variations of depth. This demonstrates that the MMU team is able to leverage memory and form coordinated exploration strategies that explore different sections of the Extended T-Maze, benefiting from larger team sizes. The FF team however fails to form effective joint exploration strategies and thus fails to fully benefit from increasing team sizes. This result highlights the critical role memory plays in facilitating the learning of effective coordination policies in adaptive tasks like the Extended T-Maze.

4.1.7 Conclusion and Future Work

We designed a diverse set of experiments across multiple axes of difficulty in the flexible Extended T-Maze domain. Our results demonstrated that the multi-robot teams trained with the MMU network perform significantly better in all tasks as compared with multi-robot teams trained with a traditional feedforward neural network. This is particularly significant considering that they both act with the same set of information. These results highlight the critical role memory plays in developing effective multi-robot coordination strategies that can adapt to changing environmental factors and teammates' policies.

The Extended T-Maze domain formulated in the paper is designed to be a general framework for testing adaptive tasks alongside the difficulties of a multi-robot approach. The framework is also devised to be modular and flexible such that each axis of difficulty can be regulated somewhat independently of the other. Future work will look to retain these properties while transcribing the domain into a ROS environment. This is prescribed with the objective of integrating the noise and constraints of a robot hardware within the domain, and facilitating benchmarks for performance with real robots. We will start with the MMU based controllers developed in this paper.

Additionally, in this work we used the system performance obtained for each multi-robot team as an evaluation for each of its member. Future work will look to integrate reward shaping techniques like the difference reward [2] that can better compute each robot's marginal utility and provide a cleaner signal to learn on.

4.2 Distributed One-Shot Decision Making

In the previous section, we used memory to augment each agent policies for improved performance in dynamic tasks that required adaptive behavior from a multiagent team. The memory was internal to each agent and the enhanced multiagent coordination observed largely stemmed from the increased capability of each agent to integrate information across time on their own series of observations. The 'expanded state' that each agent was able to perceive based on its memory enabled it to learn coordination strategies required to solve the task. An alternate pathway not addressed in this approach was whether memory could be used to share information across agents for adaptive multiagent coordination. This is particularly relevant in adversarial multiagent settings where a distributed team of agents have to rapidly adapt their behavior based on information observed by a single agent.

An agent that is dynamically able to adapt its policy based on a singular observation is an open challenge in Artificial Intelligence. This ability for rapid adaptation is a hallmark of human cognition, and is most closely related to one shot learning. While much progress has been recently made in realizing one shot learning [224], these have been limited to single agent systems. This is in contrast to most real world tasks where decision making is often distributed among multiple agents interacting across time and space. One shot learning on such systems is particularly challenging as local changes in agent policies can have unpredictable consequences in emergent system behavior.

In essence, distributed approaches parallelize the mechanism of processing information across a set of agents. These agents can be distributed in space, cybserspace, or simply have varying abilities for perception and action. In most real world problems, adaptive decision making based on features distributed across both space and time is required. For instance, real time traffic updates through an autonomous car network requires both inference on the sequence of sensor data from each car, and consolidation between multiple data streams acquired concurrently from numerous cars distributed in space. If an autonomous car observes an unanticipated blockage in a specific location, this observation has to be quickly identified among a plethora of normative information. It has to be processed, and dynamically retained for a period of time, so that the network can quickly adapt navigation routes for other cars approaching this location.

Distributed one shot learning amplifies the inherent difficulties of single agent one shot learning. A defining difficulty of one shot learning is the need for rapid characterization and association of incoming stimuli. For example, in the single agent season task [149], a variety of food items - some poisonous while others nutritious are presented to the agent during its lifetime. The goal of the agent is to consume the nutritious ones while avoiding the poisonous ones. The difficulty is that the designation of whether a food item is poisonous or nutritious is randomly assigned at every instance of the task. Simply learning to pick good actions (food item choice) is thus not an option. Instead the agent has to sample all the food items, identify whether they are poisonous, and adapt its behavior instantaneously to selectively consume the nutritious ones.

One approach to facilitate one shot learning is the incorporation of explicit memory which can be used to remember salient observations and recall them during future decision making [200]. For example, in the season task the agent can incorporate a working memory to remember whether a food is poisonous/nutritious after sampling it once. The agent can then consider these alongside its input to make decisions [149]. Adaptive one shot decision making for single agent systems have been explored in the past largely with the tools of memory [15, 17]. However, the increased complexity from multiple agents acting concurrently is widely unexplored for this class of tasks.

We introduce Distributed Modular Memory Unit (DMMU), a distributed learning framework that uses an external shared memory to rapidly consolidate information across multiple actors, acting asynchronously and in parallel. We exploit the of the MMU architecture introduced in 3 to enable selective interactions between the shared external memory and a variable number of agents. DMMU combines the ability of distributed learning in exploiting the spatial distribution of multiple agents in solving a task, with memory-based learning's ability to stitch together information across time. This enables DMMU to rapidly assimilate useful features from a group of agents acting in parallel, consolidate these into a reconfigurable external memory and use it for distributed one shot learning.

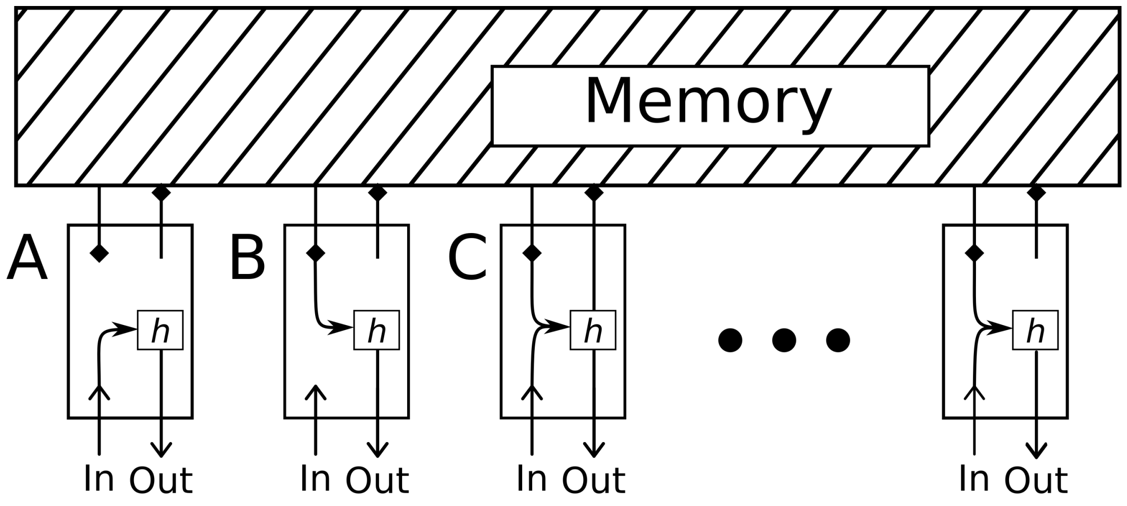


Figure 4.5: High level schematic of the DMMU framework. Each agent is comprised of a neural network with connections to the world (input/output) and memory (read/write) connections. Agents A, B, and C highlight the modularity of the framework. At this time, agent A ignores memory, and acts reactively based on its input. Agent B ignores its input and acts exclusively from memory. Agent C leverages all available information, combining the contents within memory and its immediate input in making decisions. Agent C also updates memory based on its decision. An agent can choose to perform any subset of these actions at any time.

4.2.1 Distributed Modular Memory Unit

Figure 4.5 depicts the organization of the DMMU framework. DMMU is designed to process streams of sequential observations from multiple agents concurrently. The principal feature behind the DMMU framework is its flexibility in processing sequential information across a variable number of agents while using an external memory block to capture and consolidate dynamic features across them.

Each information stream originates from an actor with individual agency. Each actor can be considered as an agent with its own unique policy, observing the environment and acting within it independently. Each of these agents is defined by a standard feedforward neural network with three major learnable components.

- **Input Gate:** The input gate filters the flow of information that comes from the environment to the agent. This serves to shield the agent from the noisy portions of each incoming observation and allows it to focus its attention on relevant features within its observation set.

- **Curated Memory Feed:** The agent's input is augmented with content that the agent selectively reads from an external memory. The agent has an independently learnable read gate which filters the content read from external memory. This serves to shape the contents of memory as per the needs of the agent, protecting it from being overwhelmed with extraneous information it might not need at a particular time.

- **Selective Memory Update:** The agent is augmented with a learnable write gate that allows it to selectively update the contents of the external memory. This gate allows the agent 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 agent.

The principal component of the DMMU framework is its modular and flexible integration between the external memory and the agents that interact with it. DMMU is an open system where an agent can join or leave dynamically during execution. This allows for a high degree of reconfigurability such that a variable number of agents (possibly heterogeneous), acting on their own unique set of observations asynchronously, can interact with the shared external memory in parallel. Each individual memory-agent interaction is modulated by the agent's read and write gates which serve to filter out noisy or extraneous content, refining information flow within the channel. This setup allows for the external memory to find stable representations for retaining dynamic features across multiple agents, yet be highly amenable to rapid updates. This enables effective consolidation of information and provides a framework for distributed one shot learning.

Training We use neuroevolution to train the DMMU framework using a weak learning signal. Each individual in our population defines a full DMMU network alongside all its agents. Distributed neural architectures like the DMMU, 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 use independent mutational operators for each weight sub-structure (matrix) within the DMMU framework. Additionally, we extend crossover operation to function at two levels of abstraction. The first crossover operation implemented with probability $crossover_{prob}$ operates by switching vector rows between matrices of weights parameterizing our individuals. Given two corresponding weight matrices from two individuals, an index is randomly chosen, and the vector of weights at that index is swapped. This is akin to exchanging a randomly sampled neuron. The second crossover operation implemented with probability $agentcrossover_{prob}$ operates by swapping agents across two individuals. Given two individuals, an agent is chosen randomly from each individual and swapped. Lastly, we use probabilistic extinc-

Algorithm 5 Neuroevolutionary algorithm used to evolve the DMMU

```
1: Initialize a population of $k$ DMMU networks
2: Define a random number generator $r() \in [0, 1)$
3: for Generation do
4:     for Individual do
5:         Compute average fitness based on $\phi$ runs
6:     Rank the population based on fitness scores
7:     Select the first $e$ candidates as elites
8:     Probabilistically select $(k - e)$ candidates based on fitness, to form Set $S$
9:     for iteration $(k - e)$ do
10:        Randomly select individual $N_i$ from Set $S$
11:        for Agent $A \in N_i$ do
12:            for weight matrix W $\in A$ do
13:                if $r() < mut_{prob}$ then
14:                    Probabilistically perturb weights in $W$ with 10% Gaussian noise
15:        Randomly select individual $N_j$ and $N_k$ from $S$
16:        for Agent $A_j \in N_j$ and $A_k \in N_k$ do
17:            for weight matrix W $\in A_j$ and $A_k$ do
18:                if $r() < crossover_{prob}$ then
19:                    Perform one point crossover
20:        Randomly select individual $N_l$ and $N_m$from $S$
21:        for Agent $A_l \in N_l$ and $A_m \in N_m$ do
22:            if $r() < agentcrossover_{prob}$ then
23:                Set $A_l = A_m$
[1]
```

tion (wiping off a randomly chosen portion of the population) to retain diversity within our population.

Algorithm 5 details the neuroevolutionary algorithm used to evolve the DMMU network. The size of the population k was set to 100, and the number of elites to 4% of k. The number of randomly initialized simulations conducted to compute a fitness evaluation for each individual, ϕ was set to 10. The mutation probability mut_{prob}, crossover probability $crossover_{prob}$ and agent crossover probability $agentcrossover_{prob}$ were set to 0.9, 0.05 and 0.05, respectively. The probability for an extinction event $extinction_{prob}$ and magnitude of extinction $extinction_{mag}$ were set to 0.004 and 0.5 respectively.

4.2.2 Cybersecurity Task

We introduce the cybersecurity task (depicted in Figure 4.6) where multiple agents are required to instantaneously adapt their behaviors based on their joint interactions within the environment. This task is based on real world cybersecurity considerations on effective responses to distributed denial of service (DDoS) attacks. A website provider often uses many resources (proxy servers) which can respond to user requests without exposing the enterprise firewall to incoming requests directly. In a DDoS attack, many fake requests are sent to the target website [154]. However, many of these attacks are executed with a botnet composed of conscripted devices without explicit intent or knowledge from the user owning the device. These devices may send genuine user requests at one moment,

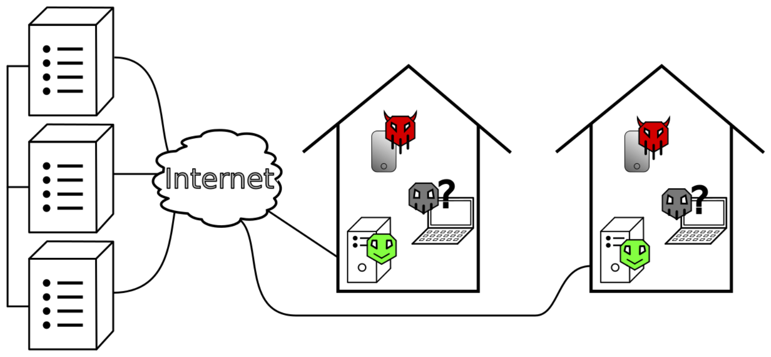


Figure 4.6: Simulated Cyber-Security Task: A server fleet handles requests from multiple devices, some of which may be conscripted by a botnet at a given time, and are carrying out a DDoS attack. For any episode, the server fleet has to figure out whether each type of device is nefarious or genuine based on a single interaction, and adapt its behavior to selectively serve requests from the genuine ones. In this example, many phones have been compromised by a newly found exploit, and have been conscripted into a botnet. The fleet of servers has identified this, and now ignores the phones' requests. The desktop computers are currently known to be genuine, and are thus served by the server fleet. The laptop devices have not been evaluated yet, and thus unknown.

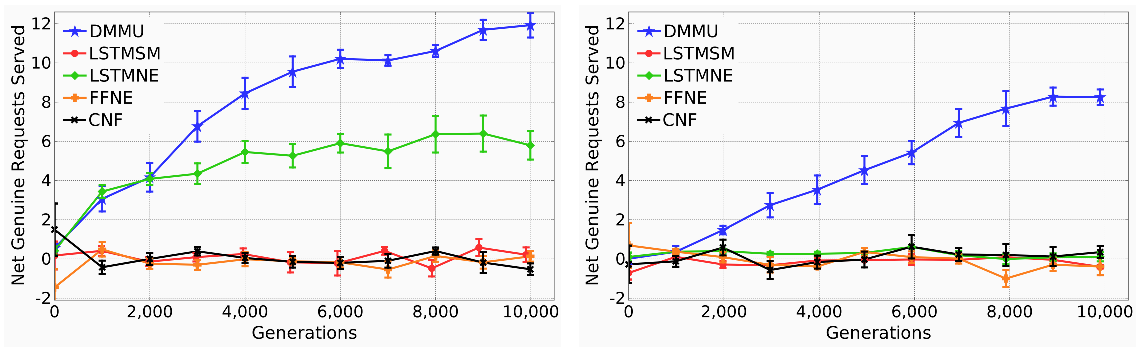
yet could be unwilling participants of an attacking botnet the next. Permanently blocking a device based on some malicious activity that once originated there is not a viable strategy as it indiscriminately penalizes all users, a majority of whom might be genuine customers of the website. It is more desirable to temporarily block requests from a device once malicious activity is detected, allowing time for the user to remedy their infection.

In our formulation of the cybersecurity task, the website server has a collection of proxy servers (agents) spread across the world that serve requests from multiple devices. For simplicity, we will refer to the entire system of our website and its proxy servers as the **server fleet** henceforth. For each episode of the simulation, our server fleet will receive a number of requests from multiple devices. A portion of these devices are nefarious, while others would be genuine. This distribution of nefarious/genuine devices will be reset at each episode of the task so our server fleet cannot simply remember which devices are nefarious and which are genuine (akin to the permanent blocking scenario described above). This prevents our server fleet from simply remembering action-value functions and forces it to dynamically determine the nefarious/genuine classification of each device for each new instance of the task.

In order to successfully withstand the attack, our server fleet must sample its incoming requests in parallel, and determine which type of devices are genuine and which are nefarious based on a singular interaction with them. The goal of the fleet will then be to selectively serve requests from genuine devices while avoiding interaction with the nefarious ones for that episode. Each fulfillment of a request may also take varying amount of time. For instance, when a proxy server decides to serve a request, it will take a variable number of time steps to complete that action during which the proxy server will remain static. This is termed the **busy period** and adds an extra layer of difficulty originating from asynchronicity between the proxy servers' operation.

Each nefarious request served leads to a negative reward of -1 while each genuine request served leads to a positive reward of $+1$. As an input, each proxy server in our server fleet records the number of requests pending for each device, and the reward associated with the last request served by itself ($+1$ and -1 for nefarious and genuine, respectively). Each proxy server (agent) will map its state to an output vector spanning the total number of devices that it can serve. The output vector represents the probability distribution over an agent's possible actions (devices the agent can choose between to serve a request from). The agent's action is then the index with the highest activation.

The ratio of nefarious/genuine devices is dynamically set at the start of each task instance. A random portion of the devices are nefarious while the rest are genuine. This adds an extra layer of stochasticity to the task. For each episode the environment can dynamically vary from being favorable (having mostly genuine users), to being adverse (having mostly nefarious users). The fitness of each fleet of servers is computed as the net number of genuine requests served. Since the designation of nefarious/genuine is randomly and uniformly distributed, the fleet of servers have to initially sample the requests to identify their category.



(a) (b)

Figure 4.7: Performance of DMMU alongside other baselines in the cybersecurity task. The server fleet consists of 10 proxy servers handling a request volume of 100 from 20 distinct devices. A random number of devices between {7,13} out of 20 chosen at the start of each task instance are nefarious while the rest are genuine. (a) A synchronous setup with a constant busy period of 1 timestep (b) Asynchronous memory access by agents with the busy period following each action randomly set between {0,2} timesteps.

4.2.3 Results

We compared DMMU in the cybersecurity task with four baselines spread across the centralized/decentralized and memoried/reactive axes. Feedforward Neural Ensemble **(FFNE)** and LSTM Neural Ensemble **(LSTMNE)** represents each proxy server as a feedforward neural network and a Long Short Term Memory (LSTM), respectively. Centralized Neural Framework **(CNF)** represents the entire server fleet as a single feedforward neural network that has direct centralized access to all the information and makes all decisions. LSTM with Shared memory **(LSTMSM)** represents the server fleet as a group of LSTM network with access to an external shared memory similar to the DMMU setup. However, since the output of a LSTM is strictly a function of its memory (cell) the LSTMSM setup is constrained to be centralized unlike the DMMU network.

The protocol for computing results was kept consistent across all the experimental variations. During each training generation, the individual network with the highest net genuine requests served was selected as the champion. The champion was then tested on a separate test set of 10 experiment instances, generated randomly for each generation. The average net genuine requests served by the champion on this test set was reported as the score for that generation. The protocol was implemented to shield the reported metrics from any bias in the distribution of nefarious users, nefarious/genuine ratios, or the population size. Ten independent statistical runs were conducted, and the average with error bars reporting the standard error in the mean were logged.

Figure 4.7 shows the comparative performance of DMMU with other baselines in controlling the server fleets where 10 proxy servers handle a request volume of 100 from 20 distinct devices. DMMU significantly outperforms all other baselines achieving a net of 11.93 ± 0.63 and 8.25 ± 0.39 genuine requests served for synchronous and asynchronous task variants, respectively. Both approaches without memory (CNF and FFNE) fail to learn in either of the tasks, and converge to an equilibrium state with the net genuine request served centered at 0. This is unsurprising as these methods lack memory and cannot associate actions with rewards over time. Even with centralized access to information and actions (CNF), the lack of memory is a principal limitation.

Surprisingly, LSTMSM fails to learn in both variants of the task despite sharing an external memory similar to DMMU. However, unlike DMMU where agents can selectively use memory's contents, LSTMSM's agents are constricted to condition their actions as a strict function of the shared external memory. This greatly limits their flexibility and leads them to fail in the task. This highlights the importance of DMMU's modularity which allows agents to read from, add to, or ignore the external memory at will. This enables DMMU to employ multiple agents with diverse policies working together in sampling devices concurrently, and encoding associations onto the shared external memory.

LSTMNE is able to learn positively and achieve a net genuine requests served of 5.8 ± 0.73 in the synchronous task (Figure 4.7a), but fails to learn in the asynchronous variant (Figure 4.7b) altogether. Unlike DMMU, LSTMNE has no external shared memory and lacks the ability to directly consolidate observations between agents. When the agent actions are synchronized, this limitation seems to curtail LSTMNE's performance. However, when the gurantees of synchronization is broken down, the limitation inhibits LSTMNE's ability to learn entirely.

DMMU, by virtue of its flexibility and modularity is able to learn positively, significantly outperforming all other baselines on both variants of the task (Figure 4.7). DMMU's learnable read and write gates provide a flexible framework for selective agent-memory interaction. This enables DMMU to discover protocols for retrieving and updating the contents of memory such that the shared memory is rapidly amenable, yet interpretable across all the agents. The external memory can be thought of as a "collective knowledge base". Training jointly with learned gating mechanisms introduce an inductive bias that favors representations amenable to sporadic agent access of this knowledge base. Collectively, DMMU's modular framework enables multiple agents to concurrently update this shared knowledge base based on one observation, while simultaneously allowing individual agents to independently incorporate it in rapidly adapting their behaviors.

4.2.4 Insight into Shared Memory Access

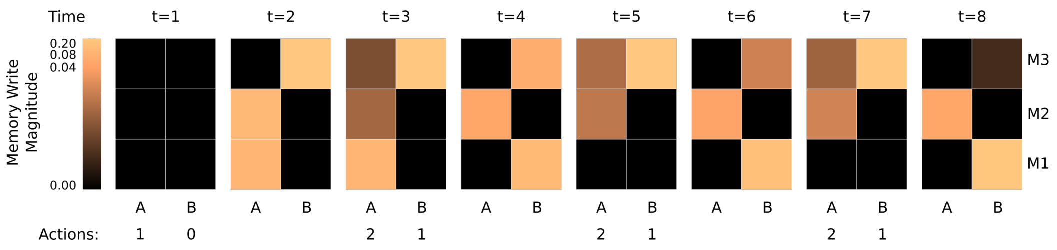


Figure 4.8: Illustration of the write interaction between a 3 cell long external memory and each agent (A1 and A2). In this task configuration, action 1 and 2 corresponding to serving the two genuine devices is desirable. The color scheme corresponds to a logarithmic scale of memory update magnitudes (what's written by each agent to the external memory). After each action taken, a busy period lasting the next timestep is observed where the agents are unable to act (even timesteps).

The shared external memory plays a crucial role in DMMU's success in learning distributed ones shot decision making. To establish the role played by the shared external memory, we perform a case study and probe the content and protocols for reading/writing between agents and the external memory. Since these interactions are distributed, interpreting them is very difficult. To allow easy interpretation and analysis, we use a smaller task size where 2 proxy servers handle a request volume of 12 from 4 distinct devices (each device sends 3 requests). We also set the busy period to a constant value of 1. This removes asynchronocity from this task and makes the agent-external memory interaction deterministic given the same initialization of nefarious/genuine device distribution. We train a DMMU framework until convergence, and probe its operation as detailed below.

Figure 4.8 illustrates the write interaction between memory and agents for the course of an experiment. This experiment started with the following ordered distribution of nefarious/genuine devices: first device is nefarious, second and third are genuine, and the fourth is nefarious. The **action** for each agent here is to pick a device (index of the list ranging from [0,3]) to serve a request from. The optimal joint action for the task configuration shown is action 1 and 2 which serves the two genuine devices.

In the first timestep, our agents take action 1 and 0 which serves a genuine and nefarious device, respectively. Note that nothing is written to memory by either of the agents during this time step (color intensities of the memory update). This makes sense as agents have simply taken their first action without getting a reinforcement at this stage. The write activations are generally lower across all timesteps where an action is taken following a busy period with no reinforcement (odd time steps). This shows that the agents primarily write information to the external memory when they receive a reinforcement (that they can associate with their earlier action). After observing the first reinforcement in timestep $t = 2$, the agents demonstrate instantaneous learning, adapting their behavior to change their joint action in service of Device 1 and 2. In this task configuration this joint action is desirable as it represents serving the genuine devices. The agents encode this positive reinforcement in their next timestep and then continue serving them.

Additionally, the first agent (A) primarily writes to the memory cell $M2$ while the second agent (B) writes to memory cells $M1$ and $M3$, leaving cell $M2$ virtually untouched. This seems to be a protocol for information consolidation across multiple agents writing to the same external memory block, without interfering with the information already encoded there.

#	Configuration	Reward	Final Memory State	Signature
1	$[G, G, N, N]$	6.0	$[0.72, 0.91, 0.78]$	$[0, 2, 1]$
2	$[N, N, G, G]$	4.0	$[2.09, 6.7\text{e}{-3}, 2.1\text{e}{-4}]$	$[2, 1, 0]$
3	$[N, G, G, N]$	5.0	$[1.58, 0.05, 0.38]$	$[1, 2, 0]$
4	$[G, N, N, G]$	4.0	$[1.77, 2.76, 5.98]$	$[0, 1, 2]$
5	$[G, N, G, N]$	5.0	$[1.77, 3.85, 9.8\text{e}{-5}]$	$[2, 0, 1]$
6	$[N, G, N, G]$	4.0	$[2.47, 0.81, 6.16]$	$[1, 0, 2]$

Table 4.1: Final memory states for running a fully trained DMMU fleet across all unique configurations of the case study task.

Table 4.1 shows the results for running our fully trained fleet servers across all 6 possible configurations of the task (each row is a different task instance with varying initialization configuration). The maximum possible reward in this task is 6 while the minimum possible is -6 corresponding to serving all the genuine and nefarious devices, respectively. For the first time step, our agents haven't had any interaction with the environment, and thus start with a fixed action for all task configurations. Agent A and B serve requests from devices 1 and 0, respectively (also depicted in Figure 4.8). In configuration #1 (first row of Table 4.1) where the two genuine device ordering match our agent's initial action, our agents are able to get the maximum possible reward of 6 for the task instance. Conversely, for configuration

#2 (second row) both of our agents' initial action serve nefarious requests leading to a -1 reward each within the first time step. However, our agents are able to use the external memory to instantaneously associate this negative reward with their actions. They use this information to then rapidly adapt their behavior, switching their action choices in the next timestep. Subsequently, this leads them to only serve genuine requests leading to a score of 4. All other configurations involve one of the agents starting out serving a genuine device and sticking with it, while the other agent starts out serving the nefarious device and then switches its action until it can find a genuine one to serve.

Additionally, Table 4.1 also shows the final memory state recorded for each configuration after our server fleet had run for 8 timesteps. While comparing these final memory states explicitly against each other is extremely difficult, we observed an interesting pattern. We sorted each final memory state in ascending order and recorded the sorted index as its **signature**. For each unique configuration of nefarious/genuine initialization, we get a unique memory signature. The memory signature is a coarse record of the cumulative agent-memory interactions within that run. The unique mapping from task configuration to final memory signature suggests that this coarse representation is predictive of the interactions that are necessary for solving a specific task configuration. The agents seem to be interpreting memory as a ternary digit (signature) that encodes the type of adaptive behavior necessary to solve a specific configuration.

4.2.5 Conclusion and Future Work

Distributed one shot learning in a multiagent system where agents can dynamically change their behavior based on an observation by one of the agents is a complex problem. However, as most real world systems increasingly move towards decentralization and become more distributed [161, 212, 221], it is an important challenge to tackle.

We build on the MMU topology described in Chapter 3 and introduced the DMMU framework that creates an external shared memory to effectively handle distributed one shot learning in a multiagent setting. DMMU is an open framework where a variable number of agents processing sequential observations concurrently and asynchronously, can interact with an external memory to consolidate dynamic features across them. This facilitates rapid assimilation of information within the external memory, and enables adaptive decision making based on singular observations. Results in the cybersecurity task demonstrated DMMU's efficacy in learning adaptive behaviors based on a singular observation, significantly outperforming other baselines operating with access to the same set of information.

In this work, the information observed by each agent was deterministic. The only noise within the system was introduced by the concurrent action of multiple agents. Future work will integrate environmental noise in each agent's observations and test the DMMU framework's robustness in dealing with this additional source of noise. Additionally, the volume of information required to be consolidated and retained by the shared external memory was relatively small (nefarious/genuine categorization). Future work will expand the complexity and volume of information that needs to be consolidated and retained by the shared external memory.

Chapter 5: Evolutionary Reinforcement Learning

In the previous chapter, we introduced new architectures that leveraged memory's capacity for temporal credit assignment to improve multiagent coordination. Memory facilitated our agents to expand their perception, enabling them to integrate information across time. Agents could then leverage this expanded capability to associate reward with actions even if they were taken in the distant past. However, as agents expand their perceptive capabilities, training them effectively becomes increasingly more challenging.

To elucidate this point, consider Agent R, a memory-less agent whose policy reactively map its observations to actions. Meanwhile, Agent M is an agent with memory whose policy maps a series of observations (history) to an action. Both agents operate in the environment observing the same information and mapping it to actions for k timesteps. If we analyze this scenario in isolation, the total number of observations here is k. Since our agents are parameterized by their policies - a function that maps observation to action, the space of observations is the domain of their function. Given a domain of k observations, the range spanned by Agent R is also k. However, for Agent M, the range spanned for the same k observation is k!. This is because the action mapped to by the agent with memory is sensitive to the permutation of the observations while the reactive agent R is not. This means that given the same set of observations, Agent M can access a much richer range of possible actions relative to the reactive agent R. However, this same expressivity also means that finding a desired mapping (training for a certain behavior) is more difficult as the search space of possible behaviors (mapping) is considerably larger.

In order to reap the benefits of the expanded capability we facilitated onto our memory-augmented agents, we need augmentations in our learning algorithms that can train these agents effectively. Towards this goal, this chapter introduces Evolutionary Reinforcement Learning (ERL) [118], a multilevel optimization framework that enables improved credit assignment for training agents to solve reinforcement learning problems. The core idea behind ERL is the hybridization of two distinct optimizers to achieve an emergent learner that inherits the best of both worlds. Each optimizer operates over varying purview, improving the policy to maximize returns over different time horizons. ERL leverages this dual-pronged optimization approach to enable better temporal credit assignment for the emergent learner. Further, by virtue of its joint optimization, ERL also addresses some core limitations within deep reinforcement learning such as the lack of effective exploration that lead to diverse behaviors and brittle convergence properties.

5.1 Motivation

Reinforcement learning (RL) algorithms have been successfully applied in a number of challenging domains, ranging from arcade games [157, 156], board games [195] to robotic control tasks [5, 141]. A primary driving force behind the explosion of RL in these domains is its integration with powerful non-linear function approximators like deep neural networks. This partnership with deep learning, often referred to as Deep Reinforcement Learning (DRL) has enabled RL to successfully extend to tasks with high-dimensional input and action spaces. However, widespread adoption of these techniques to real-world problems is still limited by three major challenges: temporal credit assignment with long time horizons and sparse rewards, lack of diverse exploration, and brittle convergence properties.

First, temporal credit assignment in reinforcement learning is challenging particularly when the reward is sparse (only observed after a series of actions). Temporal Difference methods [211] in RL use bootstrapping to address this issue but often struggle when the time horizons are long and the reward is sparse. Multi-step returns address this issue but are mostly effective in on-policy scenarios [42, 186, 187]. Off-policy multi-step learning [150, 192] have been demonstrated to be stable in recent works but require complementary correction mechanisms like importance sampling, Retrace [159, 229] and V-trace [51] which can be computationally expensive and limiting.

Secondly, RL relies on exploration to find good policies and avoid converging prematurely to local optima. Effective exploration remains a key challenge for DRL operating on high dimensional action and state spaces [171]. Many methods have been proposed to address this issue ranging from count-based exploration [163, 213], intrinsic motivation [19], curiosity [167] and variational information maximization [103]. A separate class of techniques emphasize exploration by adding noise directly to the parameter space of agents [63, 171]. However, each of these techniques either rely on complex supplementary structures or introduce sensitive parameters that are task-specific. A general strategy for exploration that is applicable across domains and learning algorithms is an active area of research.

Finally, DRL methods are notoriously sensitive to the choice of their hyper-parameters [98, 105] and often have brittle convergence properties [92]. This is particularly true for off-policy DRL that utilize a replay buffer to store and reuse past experiences [22]. The replay buffer is a vital component in enabling sample-efficient learning but pairing it with a deep non-linear function approximator leads to extremely brittle convergence properties [48, 92].

One approach well suited to address these challenges in theory is evolutionary algorithms (EA) [62, 198]. The use of a fitness metric that consolidates returns across an entire episode makes EAs indifferent to the sparsity of reward distribution and robust to long time horizons [181, 204]. EA's population-based approach also has the advantage of enabling diverse exploration, particularly when combined with explicit diversity maintenance techniques [41, 138]. Additionally, the redundancy inherent in a population also promotes robustness and stable convergence properties particularly when combined with elitism [4]. A number of recent work have used EA as an alternative to DRL with some success [40, 67, 181, 204]. However, EAs typically suffer with high sample complexity and often struggle to solve high dimensional problems that require optimization of a large number of parameters. The primary reason behind this is EA's inability to leverage powerful gradient descent methods which are at the core of the more sample-efficient DRL approaches.

To address this issue we introduce ERL, a hybrid algorithm that incorporates

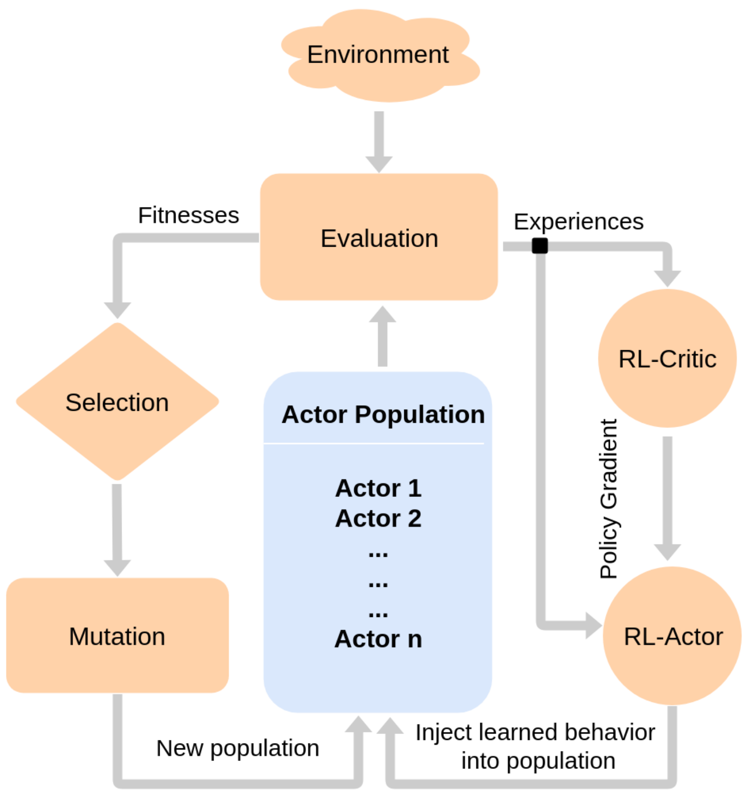


Figure 5.1: High level schematic of ERL highlighting the incorporation of EA's population-based learning with DRL's gradient-based optimization.

EA's population-based approach to generate diverse experiences to train an RL agent, and transfers the RL agent into the EA population periodically to inject gradient information into the EA. The key insight here is that an EA can be used to address the core challenges within DRL without losing out on the ability to leverage gradients for higher sample efficiency. ERL inherits EA's ability to address temporal credit assignment by its use of a fitness metric that consolidates the return of an entire episode. ERL's selection operator which operates based on this fitness exerts a selection pressure towards regions of the policy space that lead to higher episode-wide return. This process biases the state distribution towards regions that have higher long term returns. This is a form of implicit prioritization that is effective for domains with long time horizons and sparse rewards. Additionally, ERL inherits EA's population-based approach leading to redundancies that serve to stabilize the convergence properties and make the learning process more robust. ERL also uses the population to combine exploration in the parameter space with exploration in the action space which lead to diverse policies that explore the domain effectively.

Figure 5.1 illustrates ERL's double layered learning approach where the same set of data (experiences) generated by the evolutionary population is used by the reinforcement learner. The recycling of the same data enables maximal information extraction from individual experiences leading to improved sample efficiency.

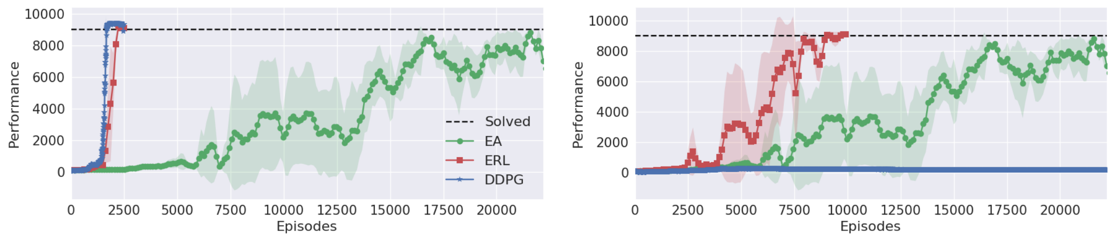


Figure 5.2: Comparative performance of DDPG, EA and ERL in a (left) standard and (right) hard Inverted Double Pendulum Task. DDPG solves the standard task easily but fails at the hard task. Both tasks are equivalent for the EA. ERL is able to inherit the best of DDPG and EA, successfully solving both tasks similar to EA while leveraging gradients for greater sample efficiency similar to DDPG.

5.2 Motivating Example

Consider the **standard Inverted Double Pendulum** task from OpenAI gym [24], a classic continuous control benchmark. Here, an inverted double pendulum starts in a random position, and the goal of the controller is to keep it upright. The task has a state space $\mathcal{S} = 11$ and action space $\mathcal{A} = 1$ and is a fairly easy problem to solve for most modern algorithms. Figure 7.1 (left) shows the comparative performance of DDPG, EA and our proposed approach: Evolutionary Reinforcement Learning (ERL), which combines the mechanisms within EA and DDPG. Unsurprisingly, both ERL and DDPG solve the task under 3000 episodes. EA solves the task eventually but is much less sample efficient, requiring approximately 22000 episodes. ERL and DDPG are able to leverage gradients that enable faster learning while EA without access to gradients is slower.

We introduce the **hard Inverted Double Pendulum** by modifying the original task such that the reward is disbursed to the controller only at the end of the episode. During an episode which can consist of up to 1000 timesteps, the controller gets a reward of 0 at each step except for the last one where the cumulative reward is given to the agent. Since the agent does not get feedback regularly on its actions but has to wait a long time to get feedback, the task poses an extremely difficult temporal credit assignment challenge.

Figure 7.1 (right) shows the comparative performance of the three algorithms in the **hard** Inverted Double Pendulum Task. Since EA does not use intra-episode interactions and compute fitness only based on the cumulative reward of the episode, the hard Inverted Double pendulum task is equivalent to its standard instance for an EA learner. EA retains its performance from the standard task and solves the task after 22000 episodes. DDPG on the other hand fails to solve the task entirely. The deceptiveness and sparsity of the reward where the agent has to wait up to 1000 steps to receive useful feedback signal creates a difficult temporal credit assignment problem that DDPG is unable to effectively deal with. In contrast, ERL which inherits the temporal credit assignment benefits of an encompassing fitness metric from EA is able to successfully solve the task. Even though the reward is sparse and deceptive, ERL's selection operator provides a selection pressure for policies with high episode-wide return (fitness). This biases the distribution of states stored in the buffer towards states with higher long term payoff enabling ERL to successfully solve the task. Additionally, ERL is able to leverage gradients which allows it to solve the task within 10000 episodes, much faster than the 22000 episodes required by EA. This result highlights the key capability of ERL: combining mechanisms within EA and DDPG to achieve the best of both approaches.

5.3 Methodology

The principal idea behind Evolutionary Reinforcement Learning (ERL) is to incorporate EA's population-based approach to generate a diverse set of experiences while leveraging powerful gradient-based methods from DRL to learn from them. In this work, we instantiate ERL by combining a standard EA with DDPG. Alternatively, any off-policy reinforcement learner that utilizes an actor-critic architecture can be used.

A general flow of the ERL algorithm proceeds as follow: a population of actor networks is initialized with random weights. In addition to the population, one additional actor network (referred to as rl_{actor} henceforth) is initialized alongside a critic network. The population of actors (rl_{actor} excluded) are then *evaluated* in an episode of interaction with the environment. The fitness for each actor is computed as the cumulative sum of the reward that they receive over the timesteps in that episode. A *selection* operator then selects a portion of the population for survival with probability commensurate on their relative fitness scores. The actors in the population are then probabilistically *perturbed* through mutation and crossover operations to create the next generation of actors. A select portion of actors with the highest relative fitness are preserved as elites and are shielded from the mutation step.

EA $\rightarrow$ RL: The procedure up till now is reminiscent of a standard EA. However, unlike EA which only learns between episodes using a coarse feedback signal (fitness score), ERL additionally learns from the experiences within episodes.

ERL stores each actor's experiences defined by the tuple *(current state, action, next state, reward)* in its replay buffer. This is done for every interaction, at every timestep, for every episode, and for each of its actors. The critic samples a random minibatch from this replay buffer and uses it to update its parameters using gradient descent. The critic, alongside the minibatch is then used to train the rl_{actor} using the sampled policy gradient. This is similar to the learning procedure for DDPG, except that the replay buffer has access to the experiences from the entire evolutionary population.

Data Reuse: The replay buffer is the central mechanism that enables the flow of information from the evolutionary population to the RL learner. In contrast to a standard EA which would extract the fitness metric from these experiences and disregard them immediately, ERL retains them in the buffer and engages the rl_{actor} and critic (see Section 2.2.2) to learn from them repeatedly using powerful gradient-based methods. This mechanism allows for maximal information extraction from each individual experiences leading to improved sample efficiency.

Temporal Credit Assignment: Since fitness scores capture episode-wide return of an individual, the selection operator exerts a strong pressure to favor individuals with higher episode-wide returns. As the buffer is populated by the experiences collected by these individuals, this process biases the state distribution towards regions that have higher episode-wide return. This serves as a form of implicit prioritization that favors experiences leading to higher long term payoffs and is effective for domains with long time horizons and sparse rewards. A RL learner that learns from this state distribution (replay buffer) is biased towards learning policies that optimizes for higher episode-wide return.

Diverse Exploration: A noisy version of the rl_{actor} using Ornstein-Uhlenbeck [223] process is used to generate additional experiences for the replay buffer. In contrast to the population of actors which explore by noise in their *parameter space* (neural weights), the rl_{actor} explores through noise in its *action space*. The two processes complement each other and collectively lead to an effective exploration strategy that is able to better explore the policy space.

RL $\rightarrow$ EA: Periodically, the rl_{actor} network's weights are copied into the evolving population of actors, referred to as *synchronization*. The frequency of synchronization controls the flow of information from the RL learner to the evolutionary population. This is the core mechanism that enables the evolutionary framework to directly leverage the information learned through gradient descent. The process of infusing policy learned by the rl_{actor} into the population also serves to stabilize learning and make it more robust to deception. If the policy learned by the rl_{actor} is good, it will be selected to survive and extend its influence to the population over subsequent generations. However, if the rl_{actor} is bad, it will simply be selected against and discarded. This mechanism ensures that the flow of information from the rl_{actor} to the evolutionary population is constructive, and not disruptive. This is particularly relevant for domains with sparse rewards and deceptive local minima which gradient-based methods can be highly susceptible to.

Algorithm 6 Evolutionary Reinforcement Learning

```
1: Initialize actor $\pi_{rl}$ and critic $\mathcal{Q}_{rl}$ with weights $\theta^{\pi}$ and $\theta^{\mathcal{Q}}$, respectively
2: Initialize target actor $\pi'_{rl}$ and critic $\mathcal{Q}'_{rl}$ with weights $\theta^{\pi'}$ and $\theta^{\mathcal{Q}'}$, respectively
3: Initialize a population of $k$ actors $pop_{\pi}$ and an empty cyclic replay buffer R
4: Define a a Ornstein-Uhlenbeck noise generator $O$ and a random number generator $r() \in [0, 1)$
5: for generation = 1, $\infty$ do
6:     for actor $\pi \in pop_{\pi}$ do
7:         fitness, R = Evaluate($\pi$, R, noise=None, $\xi$)
8:     Rank the population based on fitness scores
9:     Select the first $e$ actors $\pi \in pop_{\pi}$ as elites where $e$ = int($\psi$*k)
10:    Select $(k - e)$ actors $\pi$ from $pop_{\pi}$, to form Set $S$ using tournament selection with replacement
11:    while $|S|$ ¡ $(k - e)$ do
12:        Use crossover between a randomly sampled $\pi \in e$ and $\pi \in S$ and append to $S$
13:    for Actor $\pi \in$ Set $S$ do
14:        if $r() < mut_{prob}$ then
15:            Mutate($\theta^{\pi}$)
16:    _, R = Evaluate($\pi_{rl}$,R, $noise = O, \xi = 1$)
17:    Sample a random minibatch of T transitions $(s_i, a_i, r_i, s_{i+1})$ from R
18:    Compute $y_i = r_i + \gamma Q'_{rl}(s_{i+1}, \pi'_{rl}(s_{i+1}|\theta^{\pi'})|\theta^{\mathcal{Q}'})$
19:    Update $\mathcal{Q}_{rl}$ by minimizing the loss: $L = \frac{1}{T}\sum_i (y_i - \mathcal{Q}_{rl}(s_i, a_i|\theta^{\mathcal{Q}})^2$
20:    Update $\pi_{rl}$ using the sampled policy gradient

           $\nabla_{\theta^{\pi}} J \sim \frac{1}{T}\sum \nabla_a \mathcal{Q}_{rl}(s, a|\theta^{\mathcal{Q}})|_{s=s_i,a=a_i} \nabla_{\theta^{\pi}} \pi(s|\theta^{\pi})|_{s=s_i}$
21:    Soft update target networks: $\theta^{\pi'} \Leftarrow \tau\theta^{\pi} + (1-\tau)\theta^{\pi'}$ and $\theta^{\mathcal{Q}'} \Leftarrow \tau\theta^{\mathcal{Q}} + (1-\tau)\theta^{\mathcal{Q}'}$
22:    if generation mod $\omega = 0$ then
23:        Copy the RL actor into the population: for weakest $\pi \in pop_{\pi} : \theta^{\pi} \Leftarrow \theta^{\pi_{rl}}$
```

Algorithm 7 Function Evaluate

```
1: procedure EVALUATE($\pi$, R, noise, $\xi$)
2:     $fitness = 0$
3:     for i = 1:$\xi$ do
4:         Reset environment and get initial state $s_0$
5:         while env is not done do
6:             Select action $a_t = \pi(s_t|\theta^\pi) + noise_t$
7:             Execute action $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
8:             Append transition $(s_t, a_t, r_t, s_{t+1})$ to $R$
9:             $fitness \leftarrow fitness + r_t$ and $s = s_{t+1}$
10:    Return $\frac{fitness}{\xi}$, R
11: end procedure
```

Algorithm 8 Function Mutate

```
1: procedure MUTATE($\theta^\pi$)
2:     for Weight Matrix $\mathcal{M} \in \theta^\pi$ do
3:         for iteration = 1, $mut_{frac} * |\mathcal{M}|$ do
4:             Randomly sample indices $i$ and $j$ from $\mathcal{M}'s$ first and second axis,
   respectively
5:             if $r() < supermut_{prob}$ then
6:                 $\mathcal{M}[i, j] = \mathcal{M}[i, j]$ * $\mathcal{N}(0,\ 100 * mut_{strength})$
7:             else if $r() < reset_{prob}$ then
8:                 $\mathcal{M}[i, j] = \mathcal{N}(0,\ 1)$
9:             else
10:                $\mathcal{M}[i, j] = \mathcal{M}[i, j]$ * $\mathcal{N}(0,\ mut_{strength})$
11: end procedure
```

5.4 Experimental Details

Algorithm 6, 11 and 12 provide a detailed pseudocode of the ERL algorithm using DDPG as its policy gradient component. Adam [123] optimizer with gradient clipping at 10 and a learning rate of $5e^{-5}$ and $5e^{-4}$ was used for the rl_{actor} and rl_{critic}, respectively. The size of the population k was set to 10, while the elite fraction ψ varied from 0.1 to 0.3 across tasks. The number of trials conducted to compute a fitness score, ξ ranged from 1 to 5 across tasks. The size of the replay buffer and batch size were set to $1e^{6}$ and 128, respectively. The discount rate γ and target weight τ were set to 0.99 and $1e^{-3}$, respectively. The mutation probability mut_{prob} was set to 0.9 while the synchronization period ω ranged from 1 to 10 across tasks. The mutation strength $mut_{strength}$ was set to 0.1 corresponding to a 10% Gaussian noise. Finally, the mutation fraction mut_{frac} was set to 0.1 while the probability from super mutation $supermut_{prob}$ and reset $resetmut_{prob}$ were set to 0.05.

The following paragraphs detail the hyperparameters used for Evolutionary Reinforcement Learning (ERL) across all benchmarks. The hyperparameters that were kept consistent across all tasks are listed below.

Population size k = 10 This parameter controls the number of different individuals (actors) that are present in the evolutionary population at any given time. This parameter modulates the proportion of exploration carried out through noise in the actor's *parameter* space and its *action* space. For example, with a population size of 10, for every generation, 10 actors explore through noise in its parameters space (mutation) while 1 actor explores through noise in its action space (rl_{actor}).

Target weight $\tau = 1e^{-3}$ This parameter controls the magnitude of the soft update between the rl_{actor} / $critic$ networks and their target counterparts.

Actor Learning Rate = $5e^{-5}$ This parameter controls the learning rate of the actor network.

Critic Learning Rate = $5e^{-4}$ This parameter controls the learning rate of the critic network.

Discount Rate = 0.99 This parameters controls the discount rate used to compute the TD-error.

Replay Buffer Size = $1e^{6}$ This parameter controls the size of the replay buffer. After the buffer is filled, the oldest experiences are deleted in order to make room for new ones.

Batch Size = 128 This parameters controls the batch size used to compute the gradients.

Actor Neural Architecture = [128, 128] The actor network consists of two hidden layers, each with 128 nodes. Hyperbolic tangent was used as the activation function. Layer normalization [9] was used before layer.

Critic Neural Architecture = [200 + 200, 300] The critic network consists of two hidden layers, with 400 and 300 nodes each. However, the first hidden layer is not fully connected to the entirety of the network input. Unlike the actor, the critic takes in both state and action as input. The state and action vectors are each fully connected to a sub-hidden layer of 200 nodes. The two sub-hidden layers are

Parameter	HalfCheetah	Swimmer	Reacher	Ant	Hopper	Walker2D
ψ	0.1	0.1	0.2	0.3	0.3	0.2
ξ	1	1	5	1	5	3
ω	10	10	10	1	1	10

Table 5.1: Hyperparameters for ERL that were varied across tasks.

then concatenated to form the first hidden layer of 400 nodes. Layer normalization [9] was used before each layer. Exponential Linear Units (elu) [35] activation was used as the activation function. Table 5.1 details the hyperparameters that were varied across tasks.

Elite Fraction ψ The elite fraction controls the fraction of the population that are categorized as elites. Since an elite individual (actor) is shielded from the mutation step and preserved as it is, the elite fraction modulates the degree of exploration/exploitation within the evolutionary population. In general, tasks with more stochastic dynamics (correlating with more contact points) have a higher variance in fitness values. A higher elite fraction in these tasks helps in reducing the probability of losing good actors due to high variance in fitness, promoting stable learning.

Number of Trials ξ The number of trials (full episodes) conducted in an environment to compute a fitness score is given by ξ. For example, if ξ is 5, each individual is tested on 5 full episodes of a task, and its cumulative score is averaged across the episodes to compute its fitness score. This is a mechanism to reduce the variance of the fitness score assigned to each individual (actor). In general, tasks with higher stochasticity is assigned a higher ξ to reduce variance. Note that all steps taken during each episode is cumulative when determining the agent's total steps (the x-axix of the comparative results shown in the paper) for a fair comparison.

Synchronization Period ω This parameter controls the frequency of information flow from the rl_{actor} to the evolutionary population. A higher ω generally allows more time for expansive exploration by the evolutionary population while a lower ω can allow for a more narrower search. The parameter controls how frequently the exploration in action space (rl_{actor}) shares information with the exploration in the parameter space (actors in the evolutionary population).

5.5 Experiments

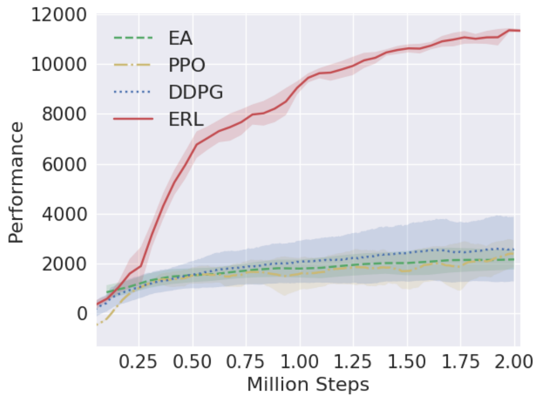
Domain: We evaluated the performance of ERL agents on 6 continuous control tasks simulated using Mujoco [217]. These are benchmarks used widely in the field [48, 98, 204, 188] and are hosted through the OpenAI gym [24].

Compared Baselines: We compare the performance of ERL with a standard neuroevolutionary algorithm (EA), DDPG [141] and Proximal Policy Optimization (PPO) [188]. DDPG and PPO are state of the art deep reinforcement learning algorithms of the off-policy and and on-policy variety, respectively. PPO builds on the Trust Region Policy Optimization (TRPO) algorithm [186]. ERL is implemented using PyTorch [166] while OpenAI Baselines [45] was used to implement PPO and DDPG. The hyperparameters for both algorithms were set to match the original papers except that a larger batch size of 128 was used for DDPG which was shown to improve performance in [105].

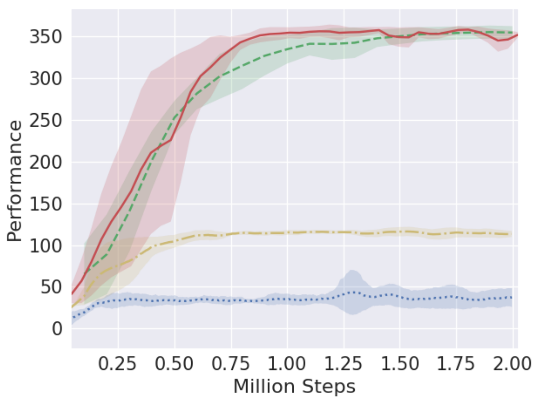
Methodology for Reported Metrics: For DDPG and PPO, the actor network was periodically tested on 5 task instances without any exploratory noise. The average score was then logged as its performance. For ERL, during each training generation, the actor network with the highest fitness was selected as the champion. The champion was then tested on 5 task instances, and the average score was logged. This protocol was implemented to shield the reported metrics from any bias of the population size. Note that all scores are compared against the number of steps in the environment. Each step is defined as an instance where the agent takes an action and gets a reward back from the environment. To make the comparisons fair across single agent and population-based algorithms, all steps taken by all actors in the population are **cumulative**. For example, one episode of HalfCheetah consists of 1000 steps. For a population of 10 actors, each generation consists of evaluating the actors in an episode which would incur $10,000$ steps. We conduct five independent statistical runs with varying random seeds, and report the average with error bars logging the standard deviation.

Results:

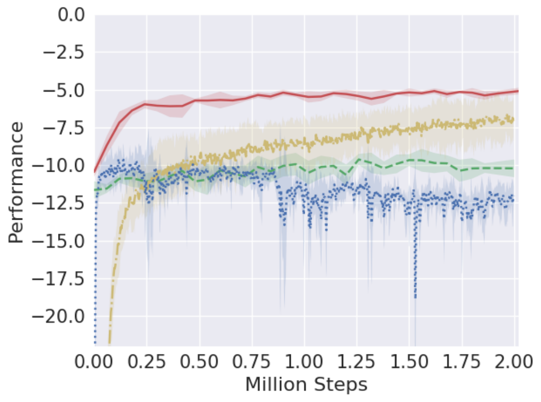
Figure 6.5 shows the comparative performance of ERL, EA, DDPG and PPO. The performances of DDPG and PPO were verified to have matched the ones reported in their original papers [141, 188]. ERL significantly outperforms DDPG



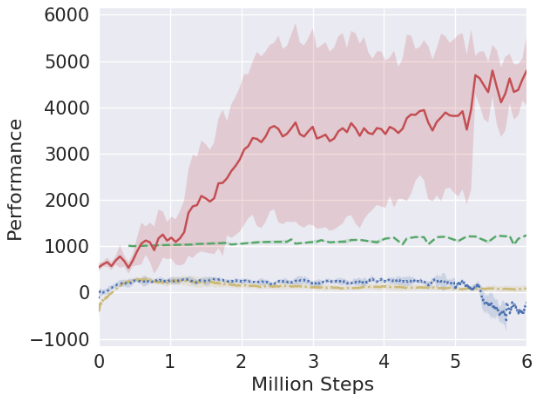
(a) HalfCheetah



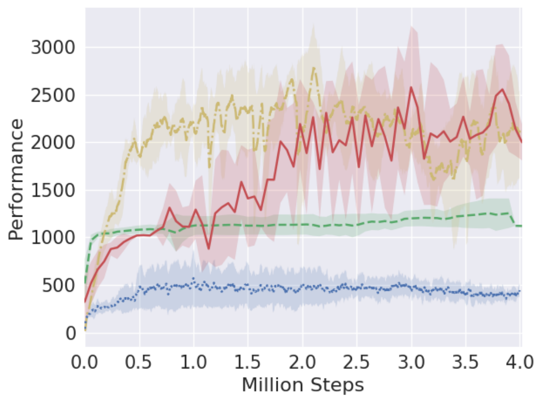
(b) Swimmer



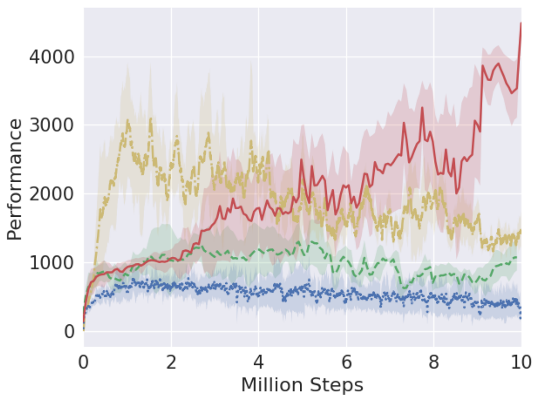
(c) Reacher



(d) Ant



(e) Hopper



(f) Walker2D

Figure 5.3: Learning curves on Mujoco-based continous control benchmarks comparing ERL (our proposed method) against PPO, DDPG and EA.

across all the benchmarks. Notably, ERL is able to learn on the 3D quadruped locomotion Ant benchmark where DDPG normally fails to make any learning progress [48, 90, 92]. ERL also consistently outperforms EA across all but the Swimmer environment, where the two algorithms perform approximately equivalently. Considering that ERL is built primarily using the sub-components of these two algorithms, this is an important result. Additionally, ERL significantly outperforms PPO in 4 out of the 6 benchmark environments[1].

The two exceptions are Hopper and Walker2D where ERL eventually matches and exceeds PPO's performance but is less sample efficient. A common theme in these two environments is early termination of an episode if the agent falls over. Both environments also disburse a constant small reward for each step of survival to encourage the agent to hold balance. Since EA selects for episode-wide return, this setup of reward creates a strong local minimum for a policy that simply survives by balancing while staying still. This is the exact behavior EA converges to for both environments. However, while ERL is initially confined by the local minima's strong basin of attraction, it eventually breaks free from it by virtue of its RL components: temporally correlated exploration in the action space and policy gradient-based on experience batches sampled randomly from the replay buffer. This highlights the core aspect of ERL: ***incorporating the mechanisms within EA and policy gradient methods to achieve the best of both approaches.***

Ablation Experiments: We use an ablation experiment to test the value

[1]Videos of learned policies available at https://tinyurl.com/erl-mujoco

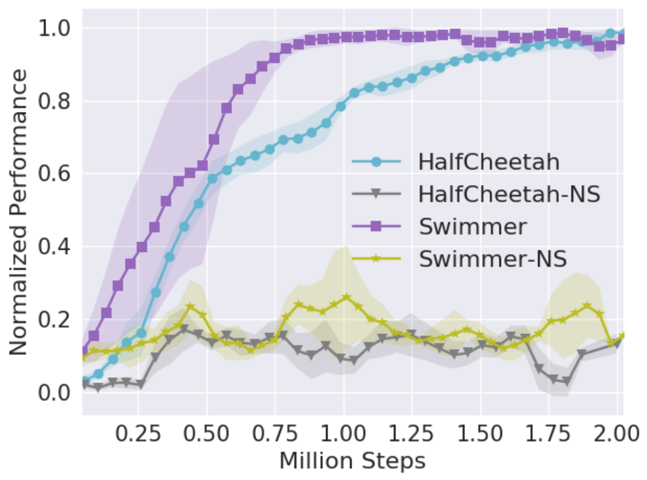


Figure 5.4: Ablation experiments with the selection operator removed. NS indicates ERL without the selection operator.

of the selection operator, which is the core mechanism for experience selection within ERL. Figure 5.4 shows the comparative results in HalfCheetah and Swimmer benchmarks. The performance for each benchmark was normalized by the best score achieved using the full ERL algorithm (Figure 6.5). Results demonstrate that the selection operator is a crucial part of ERL. Removing the selection operation (NS variants) lead to significant degradation in learning performance ($\sim$80%) across both benchmarks.

Interaction between RL and EA: To tease apart the system further, we ran some additional experiments logging whether the rl_{actor} synchronized periodically within the EA population was classified as an elite, just selected, or discarded during selection (see Table 5.2). The results vary across tasks with Half-Cheetah's and Swimmer standing at either extremes: rl_{actor} being the most and the least perfor-

	Elite	Selected	Discarded
Half-Cheetah	$83.8 \pm 9.3\%$	$14.3 \pm 9.1\%$	$2.3 \pm 2.5\%$
Swimmer	$4.0 \pm 2.8\%$	$20.3 \pm 18.1\%$	$76.0 \pm 20.4\%$
Reacher	$68.3 \pm 9.9\%$	$19.7 \pm 6.9\%$	$9.0 \pm 6.9\%$
Ant	$66.7 \pm 1.7\%$	$15.0 \pm 1.4\%$	$18.0 \pm 0.8\%$
Hopper	$28.7 \pm 8.5\%$	$33.7 \pm 4.1\%$	$37.7 \pm 4.5\%$
Walker-2d	$38.5 \pm 1.5\%$	$39.0 \pm 1.9\%$	$22.5 \pm 0.5\%$

Table 5.2: Selection rate for synchronized rl_{actor}

mant, respectively. The Swimmer's selection rate is consistent with the results in Figure 6.5b where EA matched ERL's performance while the RL approaches struggled. The overall distribution of selection rates suggest tight integration between the rl_{actor} and the evolutionary population as the driver for successful learning. Interestingly, even for HalfCheetah which favors the rl_{actor} most of the time, EA plays a critical role with 'critical interventions.' For instance, during the course of learning, the cheetah benefits from leaning forward to increase its speed which gives rise to a strong gradient in this direction. However, if the cheetah leans too much, it falls over. The gradient-based methods seem to often fall into this trap and then fail to recover as the gradient information from the new state has no guarantees of undoing the last gradient update. However, ERL with its population provides built in redundancies which selects against this deceptive trap, and eventually finds a direction for learning which avoids it. Once this deceptive trap is avoided, gradient descent can take over again in regions with better reward landscapes. These critical interventions seem to be crucial for ERL's robustness and success in the Half-Cheetah benchmark.

Note on runtime: On average, ERL took approximately 3% more time than DDPG to run. The majority of the added computation stem from the mutation operator, whose cost in comparison to gradient descent was minimal. Additionally, these comparisons are based on implementation of ERL without any parallelization. We anticipate a parallelized implementation of ERL to run significantly faster as corroborated by previous work in population-based approaches [40, 181, 204].

Using evolutionary algorithms to complement reinforcement learning, and vice-versa is not a new idea. Stafylopatis and Blekas combined the two using a Learning Classifier System for autonomous car control [199]. Whiteson and Stone used NEAT [202], an evolutionary algorithm that evolves both neural topology and weights to optimize function approximators representing the value function in Q-learning [233]. More recently, Colas et.al. used an evolutionary method (Goal Exploration Process) to generate diverse samples followed by a policy gradient method for fine-tuning the policy parameters [36]. From an evolutionary perspective, combining RL with EA is closely related to the idea of incorporating learning with evolution [1, 47, 222]. Fernando et al. leveraged a similar idea to tackle catastrophic forgetting in transfer learning [56] and constructing differentiable pattern producing networks capable of discovering CNN architecture automatically [57].

Recently, there has been a renewed push in the use of evolutionary algorithms to offer alternatives for (Deep) Reinforcement Learning [179]. Salimans et al. used a class of EAs called Evolutionary Strategies (ES) to achieve results competitive with DRL in Atari and robotic control tasks [181]. The authors were able to achieve significant improvements in clock time by using over a thousand parallel workers highlighting the scalability of ES approaches. Similar scalability and competitive results were demonstrated by Such et al. using a genetic algorithm with novelty search [204]. A companion paper applied novelty search [138] and Quality Diversity [41, 173] to ES to improve exploration [40]. EAs have also been widely used to optimize deep neural network architecture and hyperparmeters [106, 144]. Conversely, ideas within RL have also been used to improve EAs. Gangwani and Peng devised a genetic algorithm using imitation learning and policy gradients as crossover and mutation operator, respectively [67]. ERL provides a framework for combining these developments for potential further improved performance. For instance, the crossover and mutation operators from [67] can be readily incorporated within ERL's EA module while bias correction techniques such as [65] can be used to improve policy gradient operations within ERL.

5.6 Conclusion and Future Work

We introduced ERL, a hybrid algorithm that leverages the population of an EA to generate diverse experiences to train an RL agent, and reinserts the RL agent into the EA population sporadically to inject gradient information into the EA. ERL inherits EA's invariance to sparse rewards with long time horizons, ability for diverse exploration, and stability of a population-based approach and complements it with DRL's ability to leverage gradients for lower sample complexity. Additionally, ERL recycles the date generated by the evolutionary population and leverages the replay buffer to learn from them repeatedly, allowing maximal information extraction from each experience leading to improved sample efficiency. Results in a range of challenging continuous control benchmarks demonstrate that ERL outperforms state-of-the-art DRL algorithms including PPO and DDPG.

From a reinforcement learning perspective, ERL can be viewed as a form of 'population-driven guide' that biases exploration towards states with higher long-term returns, promotes diversity of explored policies, and introduces redundancies for stability. From an evolutionary perspective, ERL can be viewed as a Lamarckian mechanism that enables incorporation of powerful gradient-based methods to learn at the resolution of an agent's individual experiences. In general, RL methods learn from an agent's life (individual experience tuples collected by the agent) whereas EA methods learn from an agent's death (fitness metric accumulated over a full episode). The principal mechanism behind ERL is the capability to incorporate both modes of learning: learning directly from the high resolution of individual experiences while being aligned to maximize long term return by leveraging the low resolution fitness metric. This dual-pronged learning approach enables improved temporal credit assignment for the emergent learner and leads to faster learning.

In this paper, we used a standard EA as the evolutionary component of ERL. Incorporating more complex evolutionary sub-mechanisms is an exciting area of future work. Some examples include incorporating more informative crossover and mutation operators [67], adaptive exploration noise [63, 171], and explicit diversity maintenance techniques [40, 41, 138, 204]. Other areas of future work will incorporate implicit curriculum based techniques like Hindsight Experience Replay

[5] and information theoretic techniques [53, 92] to further improve exploration.

Chapter 6: Collaborative Evolutionary Reinforcement Learning

In the previous chapter, we introduced ERL and showed how it could be used to improve over policy gradient or EA methods in isolation. One component within ERL was its integration of population-based exploration in the parameter space to supplement policy-gradient's exploration in the action space. While the population-based parameter exploration helped generate diverse experiences, the policy gradient's exploration range was still limited by its operating policy. The is because a policy gradient algorithm employs a noisy version of its operating policy as its behavioral policy that is used for exploration. This puts the burden of both exploitation and exploration onto the same set of hyperparameters.

In this chapter, we introduce Collaborative Evolutionary Reinforcement Learning (CERL) [116], a scalable framework that leverages a portfolio of learners that learn with different time-horizons to explore different parts of the solution space while remaining loyal to the task. This process is directed by a resource manager that dynamically re-distributes computational resources amongst the learners - favoring the best as a form of online algorithm selection. The diverse set of experiences generated by this adaptive process are stored in a shared replay buffer for collective exploration enabling better sample efficiency.

Figure 6.1 illustrates CERL's multi-layered learning approach where each learner exploits the data generated by a diversity of "behavioral policies" stemming from

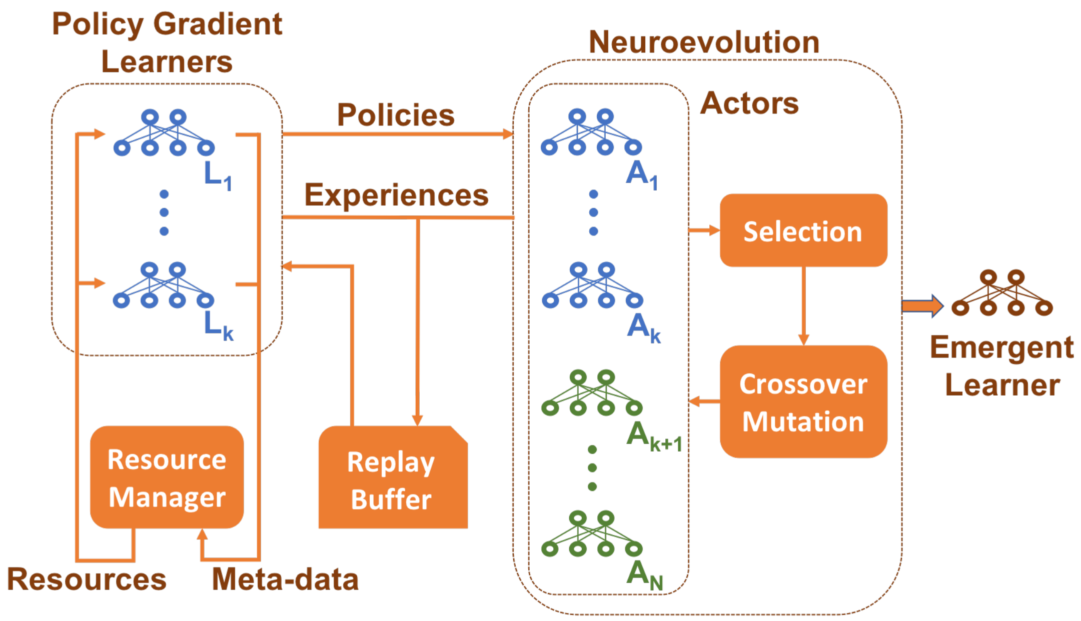


Figure 6.1: High level schematic of CERL. A portfolio of policy gradient learners operate in parallel to neuroevolution for collective exploration, while a shared replay buffer enables collective exploitation. Resource Manager drives this process by dynamically allocating computational resources amongst the learners.

other learners in the portfolio. An evolutionary population operating in parallel augments this process by extending exploration to the parameter space of policies through mutation. Evolution also introduces redundancies in the population to stabilize learning, intermixes sub-components within policies through crossover, and binds the entire underlying process to generate an emergent learner that exceeds the sum of its parts. Experiments in a range of continuous control benchmarks demonstrate that CERL inherits the best of its composite learners while remaining overall more sample-efficient.

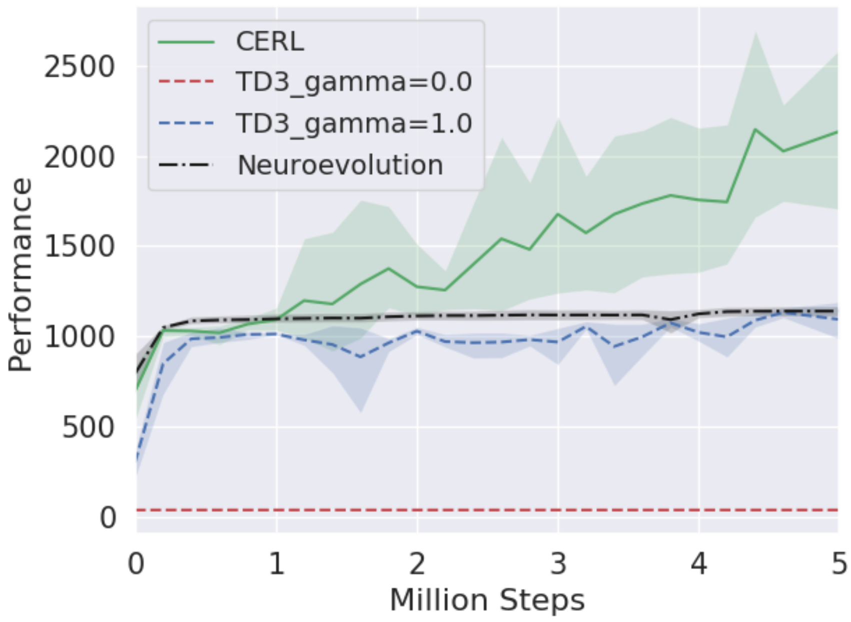


Figure 6.2: Comparative performance of Neuroevolution, TD3 ($\gamma = 0.0, 1.0$) and CERL (built using them) in the Hopper benchmark.

6.1 Motivating Example

Consider the Hopper task from OpenAI gym [24], a classic continuous control benchmark used widely in recent DRL literature [48, 105, 98, 186]. Here, the goal is to make a two-dimensional, one-legged robot hop as fast as possible without falling. The task has a state space dimension of $\mathcal{S} = 11$ and action space dimension of $\mathcal{A} = 3$. TD3 has been shown to solve this problem fairly easily [65] (also shown in Figure 6.5 in Section 6.3). However, TD3 solves this problem with a tuned discount rate ($\gamma = 0.99$). It is interesting how sensitive this performance would be to varying choices of a discount rate (γ), including ones that are clearly sub-optimal.

Figure 7.1 shows the comparative performance of TD3 ($\gamma = 0.0$), TD3 ($\gamma = 1.0$), neuroevolution and our proposed approach: CERL built using the two TD3 variations as its learners. TD3 ($\gamma = 0.0$) represents an extremely greedy learner whose optimization horizon is limited to its immediate reward. On the contrary, TD3 ($\gamma = 1.0$) represents a long-term learner whose optimization horizon is virtually infinite. However, since it seeks to optimize a return which is a function of all future action and states in the trajectory, it learns with significant amount of variance. Both learners represent the extreme ends of the spectrum and would not be expected to learn well. Figure 7.1 corroborates this expectation: TD3 ($\gamma = 0.0$) fails to entirely learn as the most greedy action with respect to the immediate reward is rarely aligned with the cumulative episode-wide return. TD3 ($\gamma = 1.0$), on the other hand, has a reward ceiling of 1000 - imposed by the variance of its computed return. Similarly, neuroevolution on its own also fails to solve the task within the 5 million steps tested. However, CERL, which is built directly on top of these learners, is able to continue learning beyond this - reaching a score of 2136 ± 512.

While each of the learners fails to solve the problem individually, they collaboratively succeed in solving it under the CERL framework. A key reason here is that each learner fails when required to simultaneously exploit well and produce good behavioral policies that explore the space well. Being able to do both is key to solving the problem and tuning the discount rate is akin to finding this trade-off. CERL provides an alternate approach to finding this trade-off - by employing both learners to explore the space while dynamically distributing the resources to the better performer for effective exploitation. Even when a learner is ill-suited for solving the task by itself, it can serve to be a key 'behavioral policy' that explores critical parts of the search space and generates experiences which are key to learning well on the task. CERL exploits these diversities to define an emergent learner that surpasses the sum of its parts.

6.2 Methodology

The principal idea behind Collaborative Evolutionary Reinforcement Learning (CERL) is to incorporate the strengths of multiple learners, each optimizing over varying time-horizons of the underlying task (MDP). While a specific learner is unlikely to be an optimal choice for the task throughout the learning process, a diverse collection of learners is significantly more likely to be so. This is particularly true for exploration, where different learners can contribute a diverse set of behavioral policies while remaining loyal to the task. A shared replay buffer ensures that all learners exploit this diverse data generated. A resource manager supervises this process by dynamically re-distributing computational resources to favor the better performing learners. Finally, this entire underlying apparatus is bound together by evolution which serves to integrate the best policies, explore in the parameter space and exploit any decomposition in the policy space with crossover operands. The emergent learner combines the best of its underlying composite processes, leading to a whole larger than the sum of its parts.

A general flow of the CERL algorithm proceeds as follows: a population of actor networks is initialized with random weights. The population is then *evaluated* in an episode of interaction with the environment (*roll-out*). The fitness for each actor is computed as the cumulative sum of the rewards received in the roll-out. A *selection* operator selects a portion of the population for survival with probability commensurate with their relative fitness scores. The weights of the actors in the population are then probabilistically *perturbed* through mutation and crossover operators to create the next **generation** of actors. A select portion of actors with the highest relative fitness are shielded from the mutation step and are preserved as elites.

Portfolio: The procedure described so far is reminiscent of a standard EA. However, in addition to the population of actors, CERL initializes a collection of *learners* (henceforth referred to as a *portfolio*). Each learner is initialized with its own actor, critic and has an associated learning algorithm defined with its own distinct hyperparameters. In this paper, the variation across learners is realized through varying discount rates (γ). However, in general, this can be any other variation in the hyperparameters, including a difference in the learning algorithm itself. The variation in discount rate used in this work can be interpreted as each learner optimizing over a distinct time-horizon of the underlying MDP. Learners with lower discount rates optimize a "greedier" objective than the ones with larger discount rates (long-term optimizers). The greedier objective has the benefit of being highly learnable but is not guaranteed to be aligned with the true learning goal. On the other hand, the long-term objective is more aligned to the true learning goal but is not as learnable - suffering from high variance due to its returns being conditioned on a longer time horizon. Thus, the portfolio represents a diverse set of learners, each with its own strengths and weaknesses.

Adaptive Resource Allocation: CERL is initialized with a computation resource budget of b workers dedicated to running roll-outs for its learner portfolio (separate from the resources used to evaluate the evolutionary population of actors). Allocation $\mathcal{A}$ defines the allotment of this resource budget amongst the learners within the portfolio for each generation of learning. This is initialized uniformly - each learner gets an equal number of dedicated workers to run roll-outs using its actor as the behavioral policy. Each learner stores statistics about the number of cumulative roll-outs it has run y, and a value metric v, defined as the discounted sum of the cumulative returns received from its own roll-outs. v is updated after every roll-out as:

$$v' \Leftarrow \alpha * return + (1 - \alpha) * v$$

After each generation, an upper confidence bound (UCB) [8] score $\mathcal{U}$ is computed for each learner based on its node statistics using Equation 6.1. This formulation is commonly used in solving multi-bandit problems [27, 110]. The UCB score is known to provide good trade-offs between exploitation and exploration and has been extensively used for reinforcement learning in the form of tree searches [6, 195] and algorithms selection [133].

$$\mathcal{U}_i = v_i^n + c * \sqrt{\frac{\log(\sum_{i=1}^{b} y_i)}{y_i}} \tag{6.1}$$

where, v^n is v normalized to be $\in (0, 1)$

The UCB scores are normalized to form a probability distribution, and allocation A is re-populated by iterative sampling from this distribution. The allocation describes the new allotment of resources (roll-out workers) amongst the learners for the next generation. The process can be seen as a meta-operation that adaptively distributes resources across the learners dynamically during the course of learning. The underlying UCB technique used to control this distribution ensures a systematic approach to balancing exploitation and exploration when allocating resources across learners.

Shared Experiences: The collective replay buffer is the principal mechanism that enables the sharing of information across the evolutionary population and amongst the learners in the portfolio. In contrast to a standard EA which would extract the fitness metric from each of its roll-outs and disregard them immediately, or ensemble methods that treat different learners separately, CERL pools all experiences defined by the tuple *(current state, action, next state, reward)* in its collective replay buffer. This is done for every interaction, at every time-step, for every episode and for each of its actors (including the evolutionary population and each roll-out conducted by the portfolio of learners). All learners are then able to sample experiences from this collective buffer and use it to update its parameters repeatedly using gradient descent. This mechanism allows for increased information extraction from each individual experiences leading to improved sample efficiency.

Diverse Exploration: In contrast to most methods where a learner learns based on data that its behavioral policy generates, CERL enables its portfolio of learners to leverage the data generated by a diverse set of actors. This includes the actors within the neuroevolutionary population and the actors stemming from other learners in the portfolio. Since each learner optimizes over varying time-horizons of the same underlying MDP, the associated actors lead to diverse behavioral policies exploring different regions of the solution space while remaining aligned with the task at hand. Additionally, in contrast to the learners which explore in their *action* space, the neuroevolutionary population explores in its *parameter* (neural weights) space using the mutation operator. The two processes complement each other and collectively lead to an effective strategy that is able to better explore the policy space.

Portfolio $\rightarrow$ EA: Periodically, each learner network is copied into the evolutionary population of actors, a process referred to as *Lamarckian transfer*. The frequency of *Lamarckian transfer* controls the flow of information from the gradient-based learners in the portfolio to the gradient-free evolutionary population. This is the core mechanism that enables the evolutionary framework to directly leverage the information learned through gradient-based optimization. The evolutionary process also acts as an amplifier in the realization of adaptive resource allocation. Good learner policies are selected to survive and reproduce - extending their influence in the population over subsequent generations. These policies and their descendants contribute increasingly more data experiences into the collective replay buffer and influence the learning of the all portfolio learners. Bad learner policies, on the other hand, are rejected to minimize their influence. Finally, crossover serves to exploit any decomposability in the policy space and combines good "sub-components of the policies" present in the diverse evolutionary population.

Algorithm 9 Object Learner

```
1: procedure INITIALIZE($\gamma$)
2:     Set discount rate=$\gamma$, $count$=0 and value $v$=0
3:     Initialize actor $\pi$ and critic $\mathcal{Q}$ with weights $\theta^{\pi}$ and $\theta^{\mathcal{Q}}$, respectively
4:     Initialize target actor $\pi'$ and critic $\mathcal{Q}'$ with weights $\theta^{\pi'}$ and $\theta^{\mathcal{Q}'}$, respectively
5: end procedure
```

Algorithm 9, 10, 11 and 12 provide a detailed pseudo-code of the CERL algorithm using a portfolio of TD3 learners. The choice of hyperparameters is explained in the Appendix. Additionally, our source code [1] is available online.

6.3 Results

Domain: CERL is evaluated on 5 continuous control tasks on Mujoco [217]. These benchmarks are used widely in the field [118, 204, 188] and are hosted on OpenAI gym [24].

Compared Baselines: For each benchmark, we compare the performance of CERL with its composite learners ran in isolation. While not constrained to this arrangement, CERL here is built using a combination of a neuroevolutionary algorithm (EA) and 4 policy gradient based learners. We use TD3 [64] as our policy gradient learner as it is the current state-of-the-art off-policy algorithm for

[1] https://github.com/intelai/cerlgithub.com/intelai/cerl

these benchmarks. The 4 TD3 learners are identical with each other apart from their discount rates which are 0.9, 0.99, 0.997, and 0.9995. These were not tuned for performance.

We also ran CERL with a single learner - picking the best TD3 learner for each task. This is equivalent to ERL [**?**] with the exception of the resource manager. However, the resource manager does not have any functional effect when there is only one learner.

Methodology for Reported Metrics: For TD3, the actor network was periodically tested on 10 task instances without any exploratory noise. The average score was logged as its performance. During each training generation, the actor network with the highest fitness was selected as the champion. The champion was then tested on 10 task instances, and the average score was logged. This protocol shielded the reported metrics from any bias of the population size. We conduct 5 statistically independent runs with random seeds from $\{2018, 2022\}$ and report the average with error bars showing a 95% confidence interval.

The "Steps" Metric: All scores reported are compared against the number of environment steps. A step is defined as an agent taking an action and receiving a reward back from the environment. To make the comparisons fair across single-agent and population-based algorithms, all steps taken by all actors in the population, and by all learners in the portfolio are counted cumulatively.

Hyperparameter Selection: The hyperparameters used for CERL were **not** tuned to generate the results, unless specifically stated. The parameters used for the TD3 learners were simply inherited from [64], while the evolutionary param-

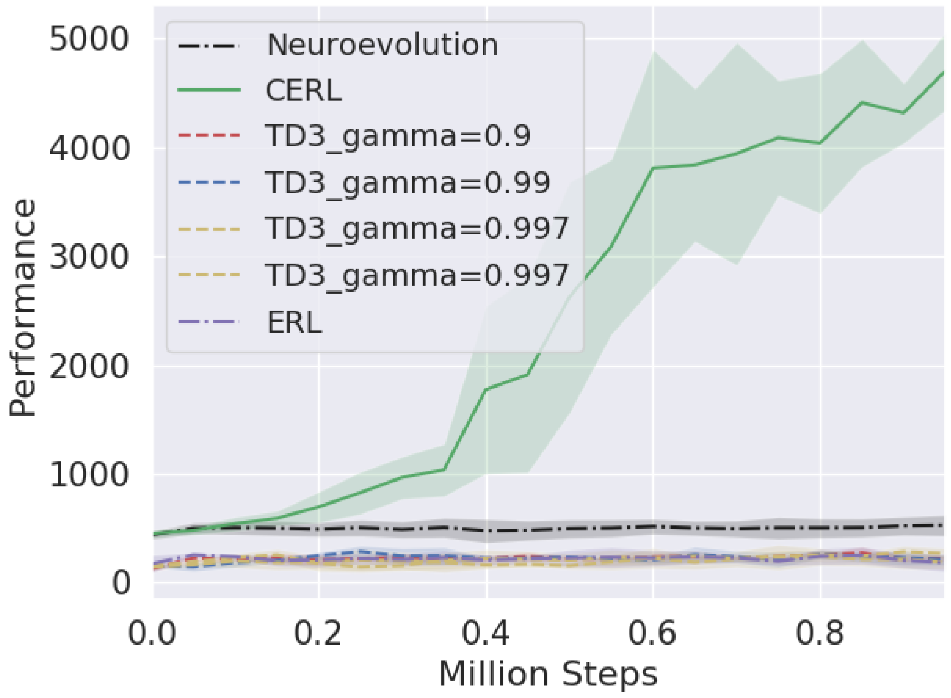


Figure 6.3: Comparative Results for CERL tested against its composite learners in the Humanoid benchmark.

eters were inherited from [118]. The computational budget of b workers was set to 10 to match the evolutionary population size. The UCB exploration coefficient was set to 0.9 which numerically makes the relative weight of exploration and exploitation terms in Equation 6.1 close to equilibrium at the start.

CERL significantly outperforms neuroevolution, as well as all versions of TD3 with varying discount rates. The TD3 learners fail to learn at all, which is consistent with reports in previous literature [92]. On the other hand, neuroevolution alone was shown to solve Humanoid, but required 62.5 millions roll-outs [137]. CERL is able to achieve a score of 4702.0 ± 356.5 within 1 million environment steps (approximately 4000 roll-outs). Considering that CERL only uses a combination of these learners, this is a significant result. Each learner in isolation fails to learn on the task entirely, while the same learners when incorporated under the CERL framework, are able to solve it jointly. This is because none of the learners are able to succeed when burdened with both exploring the solution space to generate an expansive set of data, and exploiting it aptly. However, when the learners collectively explore diverse regions of the solution space, and collectively exploit these experiences, they succeed. The single-learner ERL also fails to learn this task. Since the key difference between ERL and CERL is the use of multiple learners, this demonstrates that the performance gains of CERL come primarily from this collaboration.

Resource-manager's Sensitivity to Exploration: Figure 6.4 shows the comparative performance for CERL tested with varying c (exploration coefficient in Equation 6.1) for the Humanoid benchmark. CERL with $c = 0.9$ performs the best as it provides a good balance of exploration and exploitation for the resource-manager. However, CERL with $c = 0.0$ and $c = 5.0$ both are also able to learn well the benchmark, but are less sample-efficient. An important point to note is that $c = 0.0$ does not lead to the complete lack of exploration. As all learners start with random weights, the returns are close to random at the beginning of learning and serves to bootstrap exploration. On the other hand, a c of 10 does lead to extremely high exploration. As expected, this prolonged exploration leads to even lower sample efficiency. This highlights the role that the resource-manager plays in dynamically redistributing resource and finding the balance between exploration

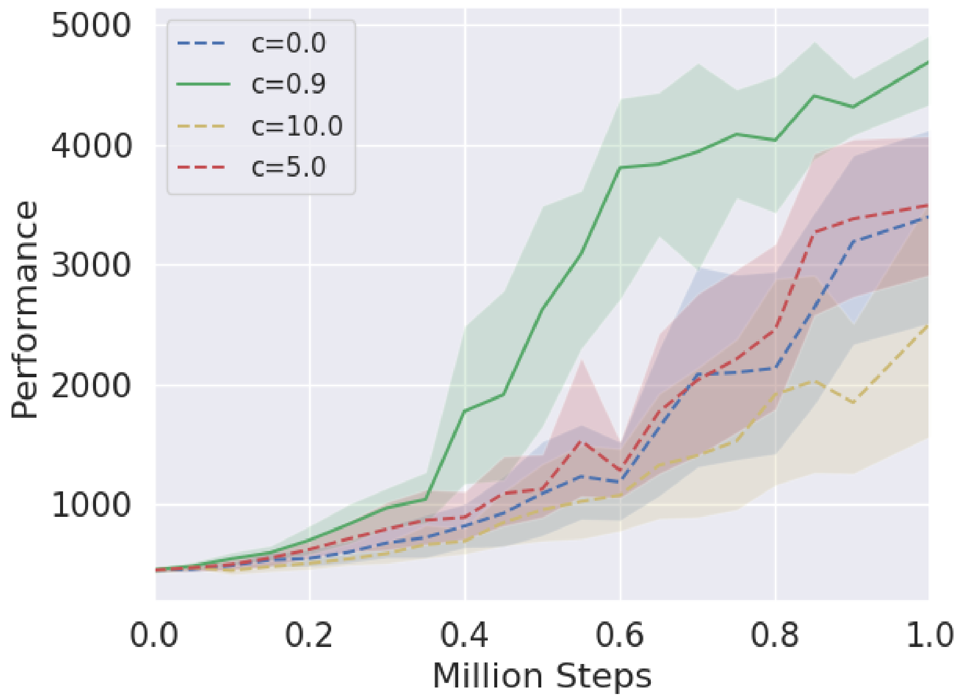
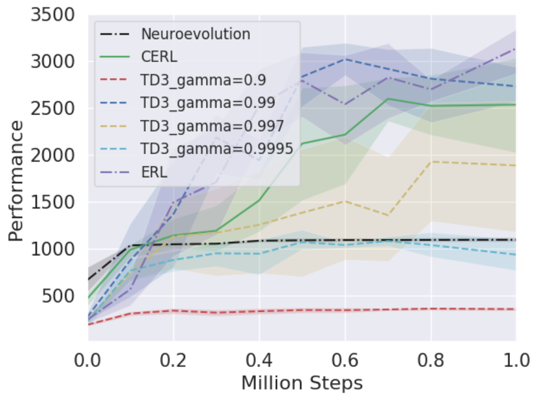


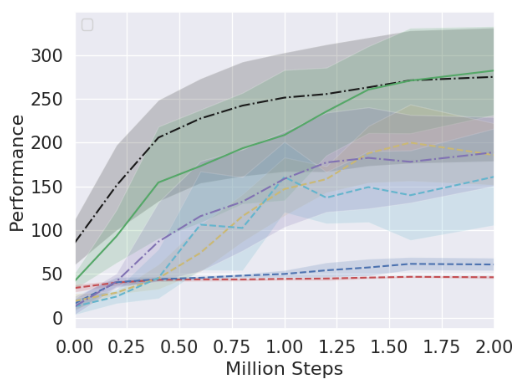
Figure 6.4: Sensitivity analysis for resource-manager exploration (c) in the Humanoid benchmark

and exploitation.

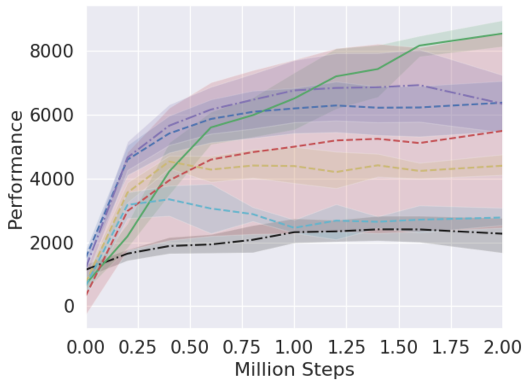
Additional Mujoco Experiments: Figure 6.5 shows the comparative performance of CERL, alongside its composite learners in 4 additional environments simulated using Mujoco. Unlike the 3D humanoid benchmark, these domains are 2D, have considerably smaller state and action spaces, and are relatively simpler to solve. One of the four TD3 learners: TD3 with a discount rate of 0.99 (TD3-0.99) is able to solve 3 out of the 4 benchmarks, with the exception of Swimmer. CERL is also able to solve these benchmarks but is less sample-efficient that TD3-0.99. However, on the Swimmer benchmark, while all of the TD3 learners fail to solve the



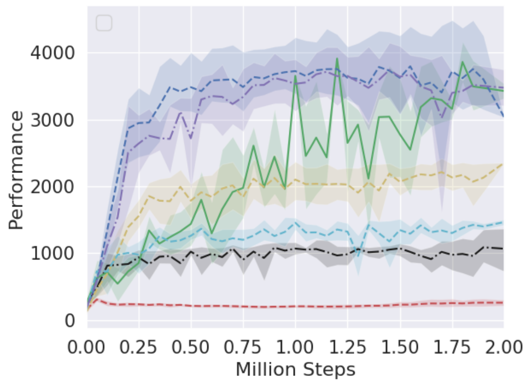
(a) Hopper



(b) Swimmer



(c) HalfCheetah



(d) Walker2D

Figure 6.5: Comparative results of CERL with 4 learners (TD3 with discount rates of 0.9, 0.99, 0.997 and 0.9995) against the learners in isolation, and neuroevolution.

task, CERL successfully solves it similar to neuroevolution. This emphasizes the key strength of CERL: the ability to inherit the best of its composite approaches.

While TD3-0.99 is more sample-efficient in 3 out of the 4 benchmarks, CERL is more sample-efficient than all the other TD3-based learners. This suggests that 0.99 is an ideal discount rate for these tasks. Any deviation from this value leads to considerable loss in performance for TD3. In other words, this is a sensitive hyperparameter that has to be rigorously tuned. CERL achieves this functionality online through its resource-manager, which adaptively re-distributes computational resources across the learners. While this invariably leads to the use of more samples when compared to an ideal hyperparameter that is known *a priori*, CERL is able to identify and exploit the best hyperparameters online via joint exploration. Additionally, as demonstrated in the cases of Swimmer and Humanoid (Figure 6.3), this exploration itself is critical to successful learning as there does not exist one hyperparameter that can solve the task all by itself. Overall, CERL enables an arguably simpler alternative to network design compared to complex hyperparameter tuning methodologies.

Allocation: Table 6.1 reports the final cumulative resource-allocation rate across the four learners for CERL in the five Mujoco benchmarks tested. L1, L2, L3 and L4 correspond to learners with $\gamma = 0.9, 0.99, 0.997$ and 0.9995, respectively. L2 seems to be the learner that is generally preferred across most tasks. This is not surprising as this value for γ is the hyperparameter used in [64] after tuning. However, in the Swimmer benchmark, this choice of hyperparameter is not ideal. Learners with higher γ perform significantly better on the task (Figure 6.5). CERL is able to identify this online and allocates more resources to L3 and L4 with higher γ. This flexibility for online algorithm selection, in combination with its evolutionary population, enables CERL to solve the Swimmer benchmark effectively.

A closely related work to CERL is Population-based Training (PBT) [106], which employs a population to jointly optimize models and its associated hyper-

Table 6.1: Average cumulative resource-allocation rate for CERL across benchmarks. (error intervals omitted as all were < 0.04)

Task	L1	L2	L3	L4
Humanoid	0.24	0.35	0.20	0.20
Hopper	0.14	0.27	0.32	0.27
Swimmer	0.17	0.20	0.36	0.27
HalfCheetah	0.29	0.32	0.24	0.15
Walker	0.14	0.28	0.33	0.25

parameters online. However, unlike CERL, PBT does not dynamically redistribute computational resources amongst its learners; instead, it relies entirely on its evolutionary process for learner selection. Additionally, learners in PBT are isolated and do not share experiences with each other for collective exploitation - a key mechanism in CERL for the retention of sample-efficiency. Collective exploitation of a diverse set of experiences is a popular idea, particularly in recent literature. Colas et al. used an evolutionary method (Goal Exploration Process) to generate diverse samples followed by a policy gradient method for fine-tuning the policy parameters [36] while Khadka and Tumer incorporated the two processes to run concurrently formulating a Lamarckian framework [118]. From an evolutionary perspective, this is closely related to the idea of incorporating learning with evolution [1, 47, 222].

Another facet of CERL is algorithm selection [66, 197, 178] - an idea that has been explored extensively in past literature. Lagoudakis and Littman formulated algorithm selection as an MDP and used Q-learning to solve classic order statistic selection and sorting problems [131]. Cauwet et al. addressed noisy optimization using a portfolio of online reinforcement learning algorithms [29]. Conversely, Laroche and Feraud introduced Epochal Stochastic Bandit Algorithm Selection (ESBAS), which tackled algorithm selection in reinforcement learning itself, formulating it as a K-armed stochastic bandit problem [133]. The resource-manager in CERL closely builds on this formulation to inherit its good exploration-exploitation trade-off properties. However, unlike ESBAS, CERL leads to soft algorithm selection - carried out through the allocation of computation resources rather than a hard binary selection.

6.4 Conclusion and Future Work

We presented CERL, a scalable algorithm that allows gradient-based learners to jointly explore and exploit solutions in a gradient-free evolutionary framework. Experiments in continuous control demonstrate that CERL's emergent learner can outperform its composite learners while remaining overall sample-efficient compared to traditional approaches.

Strengths: CERL is generally insensitive to its hyperparameters and to those of the individual learners. The Humanoid and Swimmer problems are examples where state-of-the-art algorithms show high sensitivity to their hyperparameters while CERL required no hyperparameter tuning. Significantly, the Humanoid problem demonstrates that CERL is able to find effective solutions using participating learners that fail completely on their own. This makes CERL a simpler design alternative to complex hyperparameter tuning and one that seems to generalize well across multiple tasks.

A practical consideration for CERL is the parallel operation of gradient-based and gradient-free methods. The former, involving backpropagation, is typically suited for GPUs. The latter, involving forward-propagation, is typically suited for CPUs and is highly scalable, leading to impressive wall-clock performances [181, 204]. By leveraging both modes simultaneously, CERL provides a principled way to parallelize learning and to cater one's learning algorithm to the available hardware.

Limitations: CERL can be less sample-efficient for simple tasks where the ideal hyperparameters are known *a priori*. This is apparent in the case of Walker2d (Fig 6.5) and can be attributed to the exploration involved in selecting learners. However, CERL does eventually match the performance shown by the learner with the known ideal hyperparameter. This weakness of CERL is contingent on the ability to derive the ideal parameters for a learner - a process which by itself generally consumes significant resources that are often not reported in literature.

Here, we explored homogeneous learners optimizing over varying time-horizons of a task. Future work will extend this to learners that are different algorithms themselves. Incorporating stochastic actors from SAC [92] with the deterministic TD3 actors is an exciting area. Another promising line of work would be to incorporate learning within the resource manager to augment the current UCB formulation.

Algorithm 10 CERL Algorithm

```
1: Initialize portfolio $\mathcal{P}$ with $q$ learners (Alg 9) - varying $\gamma$
2: Start allocation $\mathcal{A}$ uniformly, and set # roll-out $H = 0$
3: Initialize a population of $k$ actors $pop_\pi$ and an empty cyclic replay buffer $\mathcal{R}$
4: Define a Gaussian noise generator $O = \mathcal{N}(0, \sigma)$ and a random number generator $r() \in [0, 1)$
5: for generation = 1, $\infty$ do
6:     for actor $\pi \in pop_\pi$ do
7:         fitness, R = Evaluate($\pi$, R, noise=None)
8:     Rank the population based on fitness scores
9:     Select the first $e$ actors $\pi \in pop_\pi$ as elites
10:    Select $(k - e)$ actors $\pi$ from $pop_\pi$, to form Set $S$ using tournament selection with replacement
11:    while $|S| < (k - e)$ do
12:        Crossover between a random $\pi \in e$ and $\pi \in S$ and append to $S$
13:    for Actor $\pi \in$ Set $S$ do
14:        if $r() < mut_{prob}$ then
15:            Mutate($\theta^\pi$)
16:    for Learner $\mathcal{L} \in \mathcal{P}$ do
17:        for ii =1,$\mathcal{A}_i$ do
18:            $score$, R = Evaluate($L_\pi$,R,$noise = O$)
19:            $\mathcal{L}_v$ = $\alpha$ * score + (1 - $\alpha$) * $\mathcal{L}_v$
20:            $\mathcal{L}_{count}$ += 1
21:    ups = # of environment steps taken this generation
22:    for ii = 1, ups do
23:        for Learner $\mathcal{L} \in \mathcal{P}$ do
24:            Sample a random minibatch of T transitions $(s_i, a_i, r_i, s_{i+1})$ from $\mathcal{R}$
25:            Update the critic via a Bellman update using the min of $\mathcal{L}_{\mathcal{Q}'_j}(s_{i+1}$
26:            Update $L_\pi$ using the sampled policy gradient with noisy actions
27:            Soft update target networks:
28:            $L_{\theta^{\pi'}} \Leftarrow \tau L_{\theta^\pi} + (1 - \tau) L_{\theta^{\pi'}}$ and
29:            $L_{\theta^{\mathcal{Q}'}} \Leftarrow \tau L_{\theta^{\mathcal{Q}}} + (1 - \tau) L_{\theta^{\mathcal{Q}'}}$
30:    Compute the UCB scores $\mathcal{U}$ using
31:    for Learner $\mathcal{L} \in \mathcal{P}$ do
               $$\mathcal{U}_i = \mathcal{L}_v + c * \sqrt{\frac{\log_e H}{\mathcal{L}_{count}}}$$
32:    Normalize $U$ to be within $[0, 1)$ and set $\mathcal{A} = []$
33:    Sample from $U$ to fill up $\mathcal{A}$
34:    if generation mod $\omega = 0$ then
35:        for Learner $\mathcal{L} \in \mathcal{P}$ do
36:            Copy $L_\pi$ into the population: for weakest $\in pop_\pi : \theta^\pi \Leftarrow L_{\theta^\pi}$
```

Algorithm 11 Function Evaluate used in CERL

```
1: procedure EVALUATE($\pi$, R, noise)
2:     $fitness = 0$
3:     Reset environment and get initial state $s_0$
4:     while env is not done do
5:         Select action $a_t = \pi(s_t|\theta^\pi) + noise_t$
6:         Execute action $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
7:         Append transition $(s_t, a_t, r_t, s_{t+1})$ to $R$
8:         $fitness \leftarrow fitness + r_t$ and $s = s_{t+1}$
9:     Return $fitness$, R
10: end procedure
```

Algorithm 12 Function Mutate used in CERL

```
1: procedure MUTATE($\theta^\pi$)
2:     for Weight Matrix $\mathcal{M} \in \theta^\pi$ do
3:         for iteration = 1, $mut_{frac} * |\mathcal{M}|$ do
4:             Sample indices $i$ and $j$ from $\mathcal{M}'s$ first and second axis, respectively
5:             if $r() < supermut_{prob}$ then
6:                 $\mathcal{M}[i, j] = \mathcal{M}[i, j]$ * $\mathcal{N}(0,\ 100 * mut_{strength})$
7:             else if $r() < reset_{prob}$ then
8:                 $\mathcal{M}[i, j] = \mathcal{N}(0,\ 1)$
9:             else
10:                $\mathcal{M}[i, j] = \mathcal{M}[i, j]$ * $\mathcal{N}(0,\ mut_{strength})$
11: end procedure
```

Chapter 7: Multiagent Evolutionary Reinforcement Learning

In the last two chapters, we introduced ERL and CERL - both multilevel optimization frameworks that combined two optimizers inheriting the best of both approaches. ERL leveraged this dual-pronged optimization approach to define an emergent learner capable of improved temporal credit assignment. However, it is unclear whether this multilevel optimization framework can be leveraged to additionally address structural credit assignment in multiagent settings that require tight coordination between many agents. This chapter puts forward Multiagent Evolutionary Reinforcement Learning (MERL), a multilevel optimization framework to address this open research question. MERL builds on ERL with the goal of addressing structural credit assignment for multiagent coordination.

7.1 Motivation

Deep Reinforcement Learning (DRL) has been successfully applied to a range of challenging tasks such as Atari games [157], industrial data center cooling applications [52] and controlling humanoids [239]. Most of these tasks involve single agents, where the agent's local objective is identical to the global system objective.

However, many real world applications like air traffic control [221], multi-robot coordination [191, 240], communication and language [135, 158], and autonomous driving [190] involve multiple agents interacting with each other. Unfortunately, traditional DRL approaches are ill-suited to tackling multiagent problems due to a host of challenges including non-stationary environments [60, 147], structural credit assignment [3, 175], and the explosion of the search space with increasing number of agents [140].

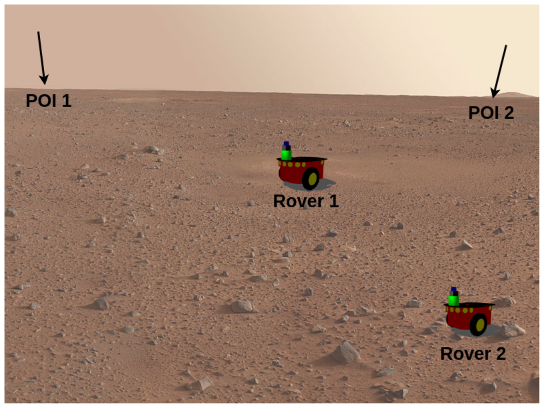
Consider soccer where a team of agents coordinate to achieve a global objective of winning. Directly optimizing this objective to train each agent is sub-optimal due to two reasons. First, it fails to encapsulate the contributions of individual agents to the final result. Second, it is usually very sparse - a single scalar capturing performance of an entire team operating across an extended period of time. This makes it a weak metric to learn on. Domain knowledge has been used to design agent-specific rewards [43, 237]. However, this is not very generalizable. For example, a team that is winning may benefit from protecting its lead by temporarily being more defensive. This objective now becomes misaligned with the local objectives of the strikers that prioritize scoring. This leads to sub-optimal coordination overall.

MERL is a hybrid algorithm that combines gradient-based and gradient-free learning to address sparse and noisy coordination objectives without the need to manually design agent-specific rewards to align with a global objective. MERL employs a two-level approach: a local optimizer (policy gradient) learns using local rewards computed directly over each agent's observation set. This has the advantage of being high-fidelity and dense - a perfect signal to learn non-coordination related aspects of the world such as perception and navigation. A global optimizer (evolutionary algorithm) learns to directly optimize the global reward which encodes the true system goal. The two processes operate concurrently and share information.

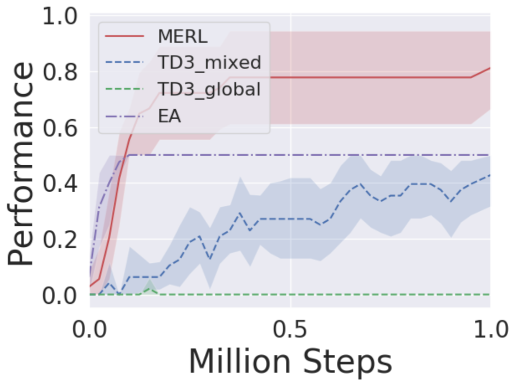
Our hypothesis is that the solution to the coordination task often exists on a smaller manifold than that for the related navigation and perception tasks. For instance in soccer, assume that each player has mastered their *self-oriented skills* such as perceiving the world, passing, dribbling, and running. Given these skills, the *coordination aspect* of the game can be roughly reduced to planning who/when to make passes and what spaces to occupy and when. The search space for learning the self-oriented skills is significantly larger than that for coordination. MERL leverages this split within the structure of the task: employing local rewards coupled with fast policy gradient methods to learn the self-oriented skills while employing the less powerful but more general global optimizer (neuroevolution) to learn coordination skills.

A key strength of MERL is that it optimizes the true learning goal (global reward) directly while leveraging local rewards as an auxiliary signal. This is in stark contrast to reward shaping techniques that construct a proxy reward to incentivize the attainment of the global reward [3, 44]. Apart from requiring domain knowledge and manual tuning, this approach also poses risks of changing the underlying problem itself [162]. MERL on the other hand is not susceptible to this mode of failure and is guaranteed to optimize the global reward. We test MERL in a multi-rover domain with increasingly complex coordination objectives. Results demonstrate that MERL significantly outperforms state-of-the-art multiagent reinforcement learning methods like MADDPG while using the same set of information and reward functions.

7.2 Motivating Example



(a) Rover domain



(b) MERL vs TD3 vs EA

Figure 7.1: (a) Rover domain with a clear misalignment between local and global reward functions (b) Comparative performance of MERL compared against TD3-mixed, TD3-global and an Evolutionary Algorithm (EA).

Consider the rover domain [3], a classic multiagent task where a team of rovers coordinate to explore a region. The global objective is to observe all POIs (Points of Interest) distributed in the area. Each robot also receives a local reward defined as the negative distance to the closest POI. In Figure 7.1(a), a team of two rovers R_1 and R_2 seek to explore and observe POIs P_1 and P_2. R_1 is closer to P_2 and has enough fuel to reach either of the POIs whereas R_2 can only reach P_2. There is no communication between the rovers.

If R_1 optimizes only locally by pursuing the closer POI P_2, then the global objective is not achieved since R_2 can only reach P_2. The globally optimal solution for R_1 is to spend more fuel and pursue P_1 - this is misaligned with its locally optimal solution. This is related to social sequential dilemmas [139, 169]. Figure 7.1(b) shows the comparative performance of four algorithms - namely TD3-mixed, TD3-global, EA and MERL on this coordination task.

TD3-mixed and TD3-global optimize a scalarized version of joint objective or just the global reward, respectively. Since the global reward is extremely sparse (only disbursed when a POI is observed) TD3-global fails to learn anything meaningful. In contrast, TD3-mixed, by virtue of its dense local reward component, successfully learns to perceive and navigate. However, the mixed reward is a static scalarization between local and global rewards that are not always aligned as described in the preceding paragraph. TD3-mixed converges to the greedy local policy of R_1 pursuing P_2.

EA relies on randomly stumbling onto a solution - e.g., a navigation sequence that takes the rovers to the correct POIs. The probability of one of the rovers stumbling onto the nearest POI is significantly higher. This is also the policy that EA converges to.

MERL combines the core strengths of TD3 and EA. It exploits the local reward to first learn perception and navigation skills - treating it as a dense, auxiliary reward even though it is not aligned with the global objective. The task is then reduced to its coordination component - picking the right POI to go to. This is effectively tackled by the EA engine within MERL and enables it to find the optimal solution. This ability to leverage reward functions across multiple levels even when they are misaligned is the core strength of MERL.

7.3 Methodology

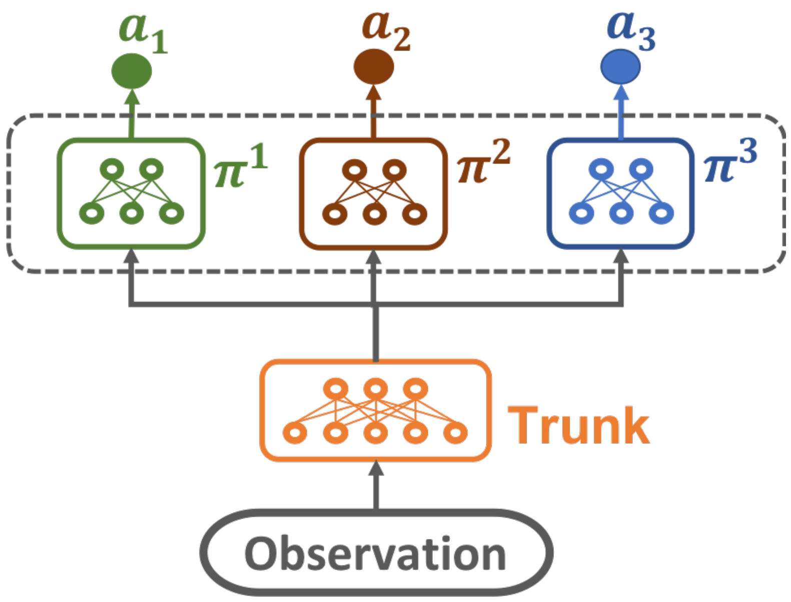


Figure 7.2: Multi-headed policy π

Policy Topology: We represent our multi-agent (*team*) policies using a multi-headed neural network π as illustrated in Figure 7.2. The head π^k represents the k-th agent in the team. Given an incoming observation for agent k, only the output of π^k is considered as agent k's response. In essence, all agents act independently based on their own observations while sharing weights (and by extension, the features) in the lower layers (*trunk*). This is commonly used to improve learning speed [196]. Further, each agent k also has its own replay buffer (R^k) which stores its *experience* defined by the tuple *(state, action, next state, local reward)* for each episode of interaction with the environment (*rollout*) involving that agent.

Global Reward Optimization: Figure 7.3 illustrates the MERL algorithm. A population of multi-headed teams, each with the same topology, is initialized with random weights. The replay buffer $\mathcal{R}^k$ is shared by the k-th agent across all teams in this population. The population is then *evaluated* for each rollout. The global reward for each team is disbursed at the end of the episode and is considered as its **fitness score**. A **selection** operator selects a portion of the population for survival with probability proportionate to their fitness scores. The weights of the teams in the population are probabilistically *perturbed* through mutation and crossover operators to create the next *generation* of teams. A portion of the teams with the highest relative fitness are preserved as elites.

Policy Gradient The procedure described so far resembles a standard EA except that each agent k stores each of its experiences in its associated replay buffer (R^k) instead of just discarding it. However, unlike EA, which only learns based on the low-fidelity global reward, MERL also learns from the experiences within episodes of a rollout using policy gradients. To enable this kind of "local learning", MERL initializes one multi-headed policy network π_{pg} and one critic $\mathcal{Q}$. A noisy version of π_{pg} is then used to conduct its own set of rollouts in the environment, storing each agent k's experiences in its corresponding buffer (R^k) similar to the evolutionary rollouts.

Local Reward Optimization: Crucially, each agent's replay buffer is kept separate from that of every other agent to ensure diversity amongst the agents.

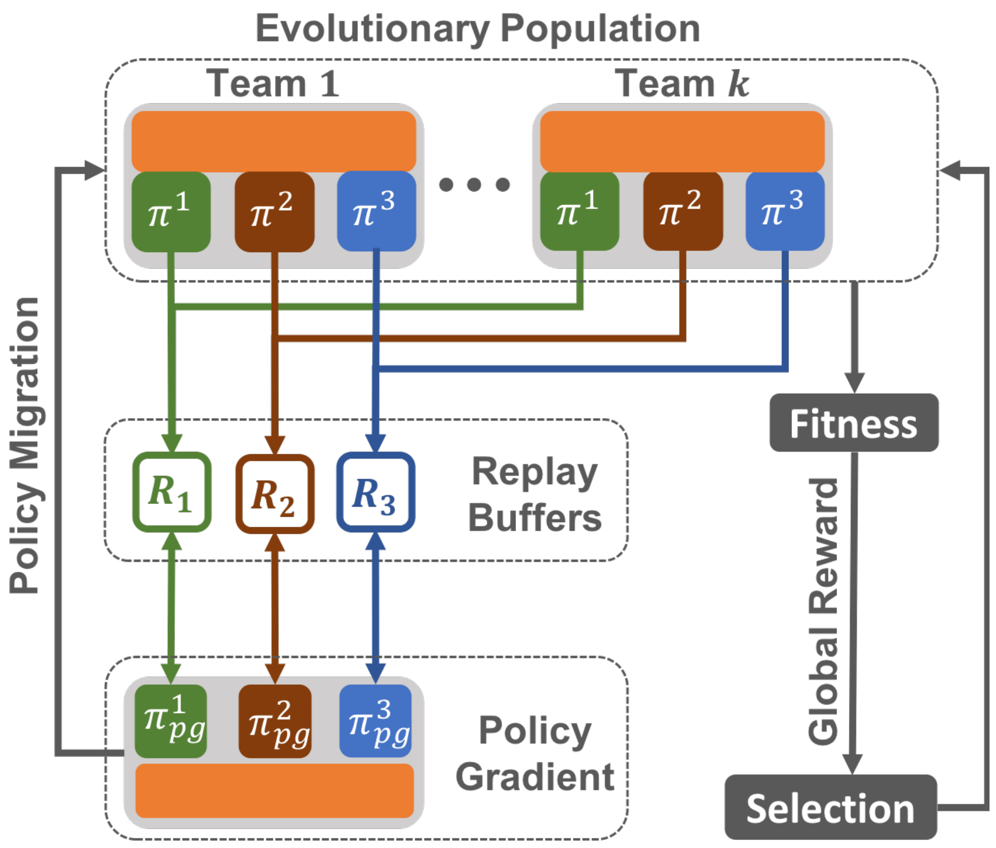


Figure 7.3: High level schematic of MERL highlighting its integrated optimization framework that leverages reward functions across multiple levels.

The shared critic samples a random mini-batch uniformly from each replay buffer and uses it to update its parameters using gradient descent. Each agent π^k_{pg} then draws a mini-batch of experiences from its corresponding buffer (R^k) and uses it to sample a policy gradient from the shared critic. Unlike the teams in the evolutionary population which directly seek to optimize the global reward, π_{pg} seeks to maximize the local reward per agent while exploiting the experiences collected via evolution.

Local → Global Migration: Periodically, the π_{pg} network is copied into the evolving population of teams and can propagate its features by participating in evolution. This is the core mechanism that combines policies learned via local and global rewards. Regardless of whether the two rewards are aligned, evolution ensures that only the performant derivatives of the migrated network are retained. This mechanism guarantees protection against destructive interference commonly seen when a direct scalarization between two reward functions is attempted. Further, the level of information exchange is automatically adjusted during the process of learning, in contrast to being manually tuned by an expert designer.

Algorithm 13 and 14 provides a detailed pseudo-code of the MERL algorithm. The choice of hyperparameters is explained in the Appendix. Additionally, our source code [1] is available online.

7.4 Rover Domain

The domain used in this paper is a variant of the rover domain used in [3, 175]. Here, a team of robots aim to observe Points of Interest (POIs) scattered across the environment. The robots start out in the center of the field, randomly distributed within an area 10% of the total field. The POIs are initialized randomly outside this area with a minimum distance of 2m from any robot. This is inspired by real-world scenarios of exploration of an unknown environment where the team of

[1]https://tinyurl.com/y6ercltshttps://tinyurl.com/y6erclts

Algorithm 13 Multiagent Evolutionary Reinforcement Learning

```
1: Initialize a population of $k$ multi-head teams $pop_\pi$, each with weights $\theta^\pi$ initialized randomly
2: Initialize a shared critic $\mathcal{Q}$ with weights $\theta^\mathcal{Q}$
3: Initialize an ensemble of $N$ empty cyclic replay buffers $\mathcal{R}^k$, one for each agent
4: Define a white Gaussian noise generator $\mathcal{W}_g$ random number generator $r() \in [0, 1)$
5: for generation = 1, $\infty$ do
6:     for team $\pi \in pop_\pi$ do
7:         $g$, $\mathcal{R}$ = Rollout ($\pi$, $\mathcal{R}$, noise=None, $\xi$)
8:         _, $\mathcal{R}$ = Rollout ($\pi$, $\mathcal{R}$, noise=$\mathcal{W}_g$, $\xi = 1$)
9:         Assign $g$ as $\pi$'s fitness
10:    Rank the population $pop_\pi$ based on fitness scores
11:    Select the first $e$ actors $\pi \in pop_\pi$ as elites
12:    Select the remaining $(k - e)$ actors $\pi$ from $pop_\pi$, to form Set $S$ using tournament selection with replacement
13:    while $|S| < (k - e)$ do
14:        Single-point crossover between a randomly sampled $\pi \in e$ and $\pi \in S$ and append to $S$
15:    for Agent $k$=1,$N$ do
16:        Randomly sample a minibatch of $T$ transitions $(s_i, a_i, l_i, s_{i+1})$ from $R^k$
17:        Compute $y_i = l_i + \gamma \min_{j=1,2} \mathcal{Q}'_j(s_{i+1}, a^\sim|\theta^{\mathcal{Q}'_j})$
18:        where $a^\sim = \pi'_{pg}(k, s_{i+1}|\theta^{\pi'_{pg}})$ [action sampled from the $k^{th}$ head of $\pi'_{pg}$] $+\epsilon$
19:        Update $\mathcal{Q}$ by minimizing the loss: $L = \frac{1}{T}\sum_i(y_i - \mathcal{Q}(s_i, a_i|\theta^\mathcal{Q})^2$
20:        Update $\pi^k_{pg}$ using the sampled policy gradient

               $\nabla_{\theta^\pi_{pg}} J \sim \frac{1}{T}\sum \nabla_a \mathcal{Q}(s, a|\theta^\mathcal{Q})|_{s=s_i,a=a_i} \nabla_{\theta^\pi_{pg}} \pi^k_{pg}(s|\theta^\pi_{pg})|_{s=s_i}$

21:        Soft update target networks: $\theta^{\pi'} \Leftarrow \tau\theta^\pi + (1 - \tau)\theta^{\pi'}$ and $\theta^{\mathcal{Q}'} \Leftarrow \tau\theta^\mathcal{Q} + (1 - \tau)\theta^{\mathcal{Q}'}$
22:        Migrate the policy gradient actor $pop_j$ : for weakest $\pi \in pop^j_\pi$ : $\theta^\pi \Leftarrow \theta^{\pi_{pg}}$
```

Algorithm 14 Function Rollout

```
1: procedure ROLLOUT($\pi$, $\mathcal{R}$, noise, $\xi$)
2:     $fitness = 0$
3:     for j = 1:$\xi$ do
4:         Reset environment and get initial joint state $js$
5:         while env is not done do
6:             Initialize an empty list of joint action $ja = []$
7:             for Each agent (actor head) $\pi^k \in \pi$ and $s_k\ in\ js$ do
8:                 $ja \Leftarrow ja \cup \pi^k(s_k|\theta^{\pi^k}) + noise_t$
9:             Execute $ja$ and observe joint local reward $jl$, global reward $g$ and joint next state $js'$
10:            for Each Replay Buffer $\mathcal{R}_k \in \mathcal{R}$ and $s_k$, $a_k$, $l_k$, $s'_k$ in $js$, $ja$, $jl$, $js'$ do
11:                Append transition $(s_k, a_k, l_k, s'_k)$ to $R_k$
12:            $js = js'$
13:            if env is done: then
14:                $fitness \leftarrow g$
15:     Return $\frac{fitness}{\xi}$, $\mathcal{R}$
16: end procedure
```

robots are air-dropped towards the center of the field.

Robot Capabilities: Each robot is loosely based on the characteristics of a Pioneer robot [215]. Its observation space consists of two channels dedicated to detecting POIs and rovers, respectively. Each channel receives intensity information over $10°$ resolution spanning the $360°$ around the robot's position. This is similar to a LIDAR. Since within each $10°$ bracket, it returns the closest reflector, occlusions make the problem partially-observable. Each robot outputs two continuous actions: δh and δd representing change in heading and drive, respectively. The maximum change in heading is capped at $180°$ per step while the maximum drive is capped at $1m/s$.

Reward Functions: The team's **global reward** is the percentage of POIs observed at the end of an episode. This is computed and broadcast to each robot at the end of an episode. It is sparse, low-fidelity, and noisy from each agent's point of view as it compiles the entire team's joint state-action history onto a single scalar value. However, it is an appropriate metric to evaluate the overall performance of the team without having to simultaneously account for each agent's navigation or perception skills.

Each robot also receives a **local reward** computed as the negative distance to the closest POI. In contrast to global reward, the local reward is dense, high-fidelity, and not noisy as it depends solely on the robot's own observations and actions. Critically, this local reward is not necessarily aligned with the global objective since it does not aim to maximize the total number of POIs observed by the group. This makes it a good training metric for each agent to learn local skills like navigation to a particular POI without having to simultaneously account for the global objective.

POI Observation Requirements: The coordination objective in the rover domain is expressed as the *coupling* requirement [175, 3]. A coupling requirement of n means that n robots are required to be within an activation distance d_a of a POI simultaneously in order to observe it. In the simplest case, a coupling of $n = 1$ defines a coordination problem where the robots need to spread out and explore the area on their own. This is similar to a set cover problem.

In contrast, a coupling of $n > 1$ defines a coordination problem where the robots need to form sub-teams of n and explore the area jointly. The presence of m robots within the activation distance of a POI, where $m < n$, generates no reward signal. This defines a tough exploration problem and is based on tasks like lifting a rock where multiple robots need to coordinate tightly to achieve any success at all.

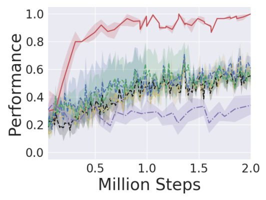
7.5 Results

Compared Baselines: We compare the performance of MERL with a standard neuroevolutionary algorithm (EA) [62], MADDPG [147] and MATD3, a variant of MADDPG that integrates the improvements described within TD3 [64] over DDPG. Internally, MERL uses EA and TD3 as its global and local optimizer, respectively. MADDPG on the other hand was chosen as it is the state-of-the-art multiagent RL algorithm. We implemented MATD3 ourselves to ensure that the differences between MADDPG and MERL do not originate from having the more stable TD3 over DDPG.

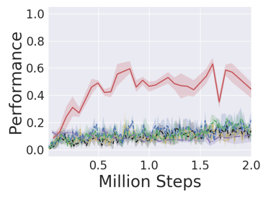
Further, MADDPG and MATD3 were tested with using either only global rewards or mixed (global + local) reward functions. The local reward function here is simply defined as the negative of the distance to the closest POI. MERL inherently leverages both reward functions while EA directly optimizes the global reward function. These variations for the baselines allow us to evaluate the efficacy of the differentiating features of MERL as opposed to improvements that might come from other ways of combining reward functions

Methodology for Reported Metrics: For MATD3 and MADDPG, the actor network was periodically tested on 10 task instances without any exploratory noise. The average score was logged as its performance. For population-based approaches (MERL and EA), we choose the actor network with the highest fitness as the champion for each generation. The champion was then tested on 10 task instances, and the average score was logged. This protocol shielded the reported metrics from any bias of the population size. We conduct 5 statistically independent runs with random seeds from $\{2019, 2023\}$ and report the average with error bars showing a 95% confidence interval.

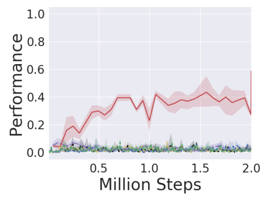
The Steps Metric: All scores reported are compared against the number of environment steps (frames). A step is defined as the multiagent team taking a joint action and receiving a feedback from the environment. To make the comparisons fair across single-team and population-based algorithms, all steps taken by all teams in the population are counted cumulatively.



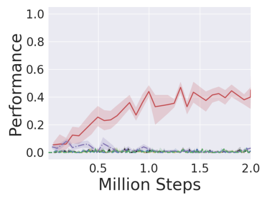
(a) Coupling=1



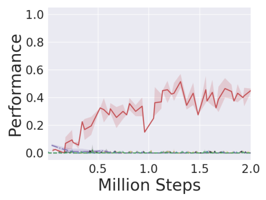
(b) Coupling=2



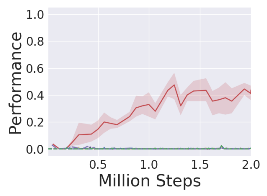
(c) Coupling=3



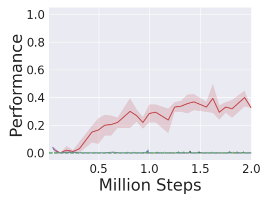
(d) Coupling=4



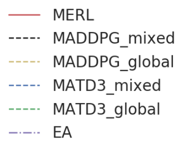
(e) Coupling=5



(f) Coupling=6



(g) Coupling=7



(h) Legend

Figure 7.4: Performance on the Rover Domain with coupling varied from 1 to 7. MERL significantly outperforms other baselines while being robust to increasing complexity of the coordination objective.

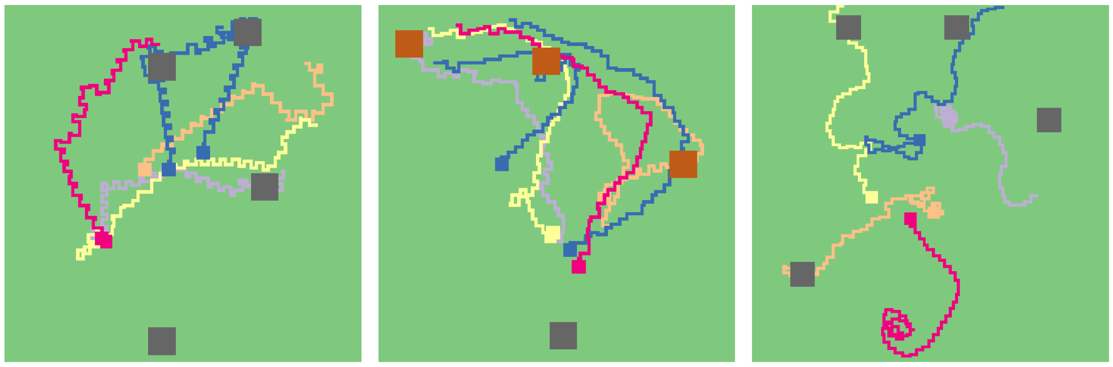
Rover Domain Setup: For each coupling requirement of n, $2n$ robots were initialized accompanied with 4 POIs spread out in the world. The coordination problem to be tackled was thus two-fold. First, the team of robots had to learn to form sub-teams of size n. Next, the sub-teams would now need to coordinate with each other to spread out and cover different POIs to ensure they get all 4 within the time allocated. Both the team formation, and coordinated spreading out has to be done autonomously and adaptively based on the distribution of the robots and POIs (varied randomly for each instance of the task). This is the core difficulty of the task.

Figure 7.4 shows the comparative performance of MERL, MADDPG (global and mixed), MATD3 (global and mixed), and EA tested in the rover domain with coupling requirements from 1 to 7. MERL significantly outperforms all baselines across all coupling requirements. The tested baselines clearly degrade quickly beyond a coupling of 2. The increasing coupling requirement is equivalent to increasing difficulty in joint-space exploration and entanglement in the global coordination objective. However, it does not increase the size of the state-space, complexity of perception, or navigation. This indicates that the degradation in performance is strictly due to the increase in complexity of the coordination objective.

Notably, MERL is able to learn on coupling greater than $n = 6$ where methods without explicit reward shaping have been shown to fail entirely [175]. This is consistent with the performances of the baselines tested here as none of them use explicit domain-specific reward shaping. MERL successfully completes the task using the same set of information and coarse, unshaped reward functions as the other algorithms. The primary mechanism that enables this is MERL's bi-level approach whereby it leverages the local reward function to solve navigation and perception while concurrently using the global reward function to learn team formation and effective coordination.

Team Behaviors: Figure 7.5 illustrates the trajectories generated for the rover domain with a coupling of $n = 3$. The trajectories for partially and fully trained MERL are shown in Figure 7.5(a) and (b), respectively. During training, when MERL has not discovered success in the global coordination objective (no POIs are successfully observed), MERL simply proceeds to optimize the local objective for each robot. This allows it to reach trajectories such as the ones shown in 7.5(a) where each robot learns to go towards a POI.

Given this joint behavior, the probability of having 3 robots congregate to the



(a) MERL (training) (b) MERL (trained) (c) MATD3 (trained)

Figure 7.5: Visualizations illustrating the trajectories generated with a coupling of 3. Red and black squares represent observed and unobserved POIs respectively

same POI is substantially improved in comparison to random undirected exploration by each robot. Once this scenario is stumbled upon, the global optimizer (EA) within MERL will explicitly select for agent policies that lead to such team-forming joint behaviors. Eventually it succeeds as shown in Figure 7.5(b). Here, team formation and collaborative pursuit of the POIs is immediately apparent. Two teams of 3 robots each form at the start of the episode. Further, the two teams also coordinate among each other to pursue different POIs in order to maximize the global team reward. While the POI allocation is not perfect, (the one in the bottom is left unattended) they do succeed in successfully observing 3 out of the 4 POIs.

In contrast, MATD3-mixed fails to successfully observe any POI. From the trajectories, it is apparent that the robots have successfully learned to perceive and navigate to reach POIs. However, they are unable to use this *sub-skill* towards fulfilling the coordination objective. Instead each robot is rather split on the objective that it is optimizing. Some robots seem to be in sole pursuit of POIs without any regard for team formation or collaboration while others seem to exhibit random movements.

The primary reason for this is the mixed reward function that directly combines the local and global reward function. Since the two reward functions have no guarantees of alignment across the state-space of the task, they invariably lead to learning these sub-optimal joint-behaviors that solve a certain form of scalarized mixed objective. In practice, this problem can be addressed by manually tuning the scalarization coefficients to achieve the required coordination behavior. However, without such manual reward shaping, MATD3-mixed fails to solve the task. In contrast, MERL is able to solve the task without any reward shaping or manual tuning.

7.6 Conclusion and Future Work

We introduced MERL, a hybrid algorithm that can combine global objectives with local objectives even when they are not aligned with each other. MERL achieves this by using a fast policy-gradient local optimizer to exploit dense local rewards while concurrently leveraging a global optimizer (EA) to tackle the coordination aspects of the task.

Results demonstrate that MERL significantly outperforms MADDPG, the state-of-the-art multiagent RL method . We also tested a modification of MADDPG to integrate TD3 - the state-of-the-art single-agent RL algorithm - as well as variations that utilized only global, or global and local rewards. These experiments demonstrated that the core improvements of MERL come from the combination of EA and policy gradient algorithm which enable MERL to combine multiple objectives without relying on alignment between those objectives. This differentiates MERL from other approaches like reward scalarization and reward shaping that either require extensive manual tuning or can detrimentally change the MDP [**?**] itself.

Here, we limited our focus to cooperative domains. Future work will explore MERL for adversarial settings such as Pommerman [177], StarCraft [108, 225] and RoboCup [124, 145]. Further, MERL can be considered a bi-level approach to combine local and global objectives. Extending MERL to generalized multilevel rewards is another promising area for future work.

Chapter 8: Conclusion

Multiagent coordination in the presence of noise, sparse objectives, and across long temporal periods of operation is a complex problem to solve. However, as most real world systems increasingly move towards decentralization and become more distributed [135, 158, 161, 190, 191, 212, 221, 240], it is an important challenge to tackle.

The core contribution of this dissertation is to tackle credit assignment for multiagent reinforcement learning: establishing associations between action and reward when they are separated by time and obfuscated by the inherent noise of multiple agents acting concurrently. Towards this end, we investigated four distinct research threads and introduced the following contributions:

1. Modular Memory Unit (MMU), a novel memory-augmented neural network topology that enables reliable retention and propagation of information over an extended period of time in the presence of noise [113, 115].

2. Distributed MMU (DMMU), which an external shared memory is collectively read and written by a team of agents to enable distributed one-shot decision making [114, 119, 120, 121].

3. Evolutionary Reinforcement Learning (ERL), a hybrid framework that combines the strengths of an Evolutionary Algorithm (EA) with fast gradient-

based algorithms for effective deep reinforcement learning [118]. Extended ERL to introduce Collaborative ERL (CERL), which employs a collection of policy gradient learners (portfolio), each optimizing over varying resolution of the same underlying task for improved exploration of the search space [116].

4. Multiagent ERL (MERL), which leverages the multilevel optimization framework of ERL to tackle sparse multiagent coordination problems by leveraging dense local reward even when there is no guarantee of alignment between the two [117].

The remainder of this chapter summarizes each of the preceding chapters and elaborates on their contributions.

Contribution 1: Modular Memory Units (MMUs): Chapter 3 leveraged memory as a tool in enabling better credit assignment by facilitating associations across actions and rewards separated in time. We introduced MMU, a novel memory-augmented neural network topology that leverages independent read and write gates which serve to decouple the memory from its central feedforward computation. This allows for regimented memory access and update, administering the ability to choose when to read from memory, update it, or simply ignore it. This enables 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. Results on deep memory benchmark tasks demonstrated that MMU significantly outperforms traditional memory-based methods while being robust to dramatic increase in the length of time between reward and its associated action.

Contribution 2: Distributed Modular Memory Units (DMMUs): Chapter 4 leveraged this capability for improved temporal credit assignment to build memory-augmented teams that learn adaptive emergent behaviors that require information aggregation across time and between agents. Further, we introduced DMMU where an external memory is collectively used by a team of agents as a shared knowledge base. Each agent can selectively and asynchronously access the external memory to accept and/or disseminate pertinent information as they are observed. This allows for rapid consolidation of salient information enabling distributed one-shot decision making. Results in a T-maze benchmark and a Cybersecurity domain for distributed one-shot decision making demonstrated that our MMU-based solution significantly outperform reactive agents and traditional memory-based agents.

Contribution 3: Evolutionary Reinforcement Learning (ERL): Chapter 5 introduced ERL, a multilevel optimization framework that leverages reward functions across multiple hierarchies to improve learning. ERL combines the strengths of a global optimizer (Evolutionary Algorithm - EAs) with a local optimizer (policy gradient). It inherits EA's invariance to the distribution and sparsity of rewards, diverse exploration, and stability of a population-based approach and complements it with policy gradient's ability to leverage gradients for higher sample efficiency and faster learning. Experiments in a range of challenging control benchmarks demonstrate that ERL significantly outperform prior policy gradient and EA methods.

Chapter 6 extended the ERL framework to introduce Collaborative ERL (CERL) which employs a collection of policy gradient learners (portfolio), each optimizing over varying resolution of the same underlying task. This leads to a diverse set of policies that are able to reach diverse regions within the solution space. A shared replay buffer pairs this improved exploration with collective exploitation for improved learning. Results in a range of continuous control benchmarks demonstrate that CERL significantly outperforms its composite learner while remaining overall more sample-efficient.

Contribution 4: Multiagent Evolutionary Reinforcement Learning (MERL): Finally, Chapter 7 introduced MERL, a hybrid algorithm that leverages the multi-level optimization framework of ERL to enable improved multiagent coordination without requiring an explicit alignment between local and global reward functions. MERL uses fast, policy-gradient based learning for each agent by utilizing their dense, local rewards. Concurrently, an evolutionary algorithm is used to recruit agents into a team by directly optimizing the sparser global objective. We explore problems that require coupling (a minimum number of agents required to coordinate for success), where the degree of coupling is not known to the agents. Results demonstrate that MERL's integrated approach is more sample-efficient and retains performance better with increasing coupling orders compared to the state-of-the-art multiagent policy-gradient algorithms.

Bibliography

[1] David Ackley and Michael Littman. Interactions between learning and evolution. *Artificial life II*, 10:487–509, 1991.

[2] Adrian Agogino and Kagan Tumer. Efficient evaluation functions for evolving coordination. *Evolutionary Computation*, 16(2):257–288, 2008.

[3] Adrian K Agogino and Kagan Tumer. Unifying temporal and structural credit assignment problems. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 980–987. IEEE Computer Society, 2004.

[4] Chang Wook Ahn and Rudrapatna S Ramakrishna. Elitism-based compact genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 7(4):367–385, 2003.

[5] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[6] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017.

[7] Jose A Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards. *arXiv preprint arXiv:1806.07857*, 2018.

[8] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[9] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[10] T. Back, D. B. Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. Oxford University Press, 1997.

[11] A Baddeley, M Eysenck, and M Anderson. Memory. *Psychology Press*, 2009.

[12] Alan D Baddeley and Graham Hitch. Working memory. *The Psychology of Learning and Motivation*, 8:47–89, 1974.

[13] Arash Bahrammirzaee. A comparative survey of artificial intelligence applications in finance: artificial neural networks, expert system and hybrid intelligent systems. *Neural Computing and Applications*, 19(8):1165–1195, 2010.

[14] Bram Bakker. 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, 2001.

[15] Bram Bakker. Reinforcement learning memory. *Neural Information Processing Systems 14*, pages 1475–1782, 2002.

[16] Pierre Baldi, Søren Brunak, Paolo Frasconi, Giovanni Soda, and Gianluca Pollastri. Exploiting the past and the future in protein secondary structure prediction. *Bioinformatics*, 15(11):937–946, 1999.

[17] Justin Bayer, Daan Wierstra, Julian Togelius, and Jürgen Schmidhuber. Evolving memory cell structures for sequence learning. *Artificial Neural Networks–ICANN 2009*, pages 755–764, 2009.

[18] Paul A Beardsley, Scott Frazier Watson, and Javier Alonso-Mora. Shared control of semi-autonomous vehicles including collision avoidance in multi-agent scenarios, December 22 2015. US Patent 9,216,745.

[19] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

[20] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[21] Jean Berger and Nassirou Lo. An innovative multi-agent search-and-rescue path planning approach. *Computers & Operations Research*, 53:24–31, 2015.

[22] Shalabh Bhatnagar, Doina Precup, David Silver, Richard S Sutton, Hamid R Maei, and Csaba Szepesvári. Convergent temporal-difference learning with arbitrary smooth function approximation. In *Advances in Neural Information Processing Systems*, pages 1204–1212, 2009.

[23] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[24] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[25] Arthur E Bryson. A gradient method for optimizing multi-stage allocation processes. In *Proc. Harvard Univ. Symposium on digital computers and their applications*, page 72, 1961.

[26] Arthur Earl Bryson. *Applied optimal control: optimization, estimation and control*. CRC Press, 1975.

[27] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[28] Wonmin Byeon, Thomas M Breuel, Federico Raue, and Marcus Liwicki. Scene labeling with lstm recurrent neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3547–3555, 2015.

[29] Marie-Liesse Cauwet, Jialin Liu, and Olivier Teytaud. Algorithm portfolios for noisy optimization: Compare solvers early. In *International Conference on Learning and Intelligent Optimization*, pages 1–15. Springer, 2014.

[30] Ollion Charles, Pinville Tony, and Doncieux Stéphane. With a little help from selection pressures: evolution of memory in robot controllers. *Artificial Life*, 13:407–414, 2012.

[31] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[32] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[33] Jen Jen Chung, Carrie Rebhuhn, Connor Yates, Geoffrey A Hollinger, and Kagan Tumer. A multiagent framework for learning dynamic traffic management strategies. *Autonomous Robots*, pages 1–17, 2018.

[34] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[35] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[36] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. Gep-pg: Decoupling exploration and exploitation in deep reinforcement learning algorithms. *arXiv preprint arXiv:1802.05054*, 2018.

[37] Mitchell Colby, Jen Jen Chung, and Kagan Tumer. Implicit adaptive multi-robot coordination in dynamic environments. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 5168–5173. IEEE, 2015.

[38] Mitchell Colby, Logan Yliniemi, and Kagan Tumer. Autonomous multiagent space exploration with high-level human feedback. *Journal of Aerospace Information Systems*, pages 1–15, 2016.

[39] Thomas S Collett, Paul Graham, and Virginie Durier. Route learning by insects. *Current Opinion in Neurobiology*, 13(6):718 – 725, 2003.

[40] Edoardo Conti, Vashisht Madhavan, Felipe Petroski Such, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. *arXiv preprint arXiv:1712.06560*, 2017.

[41] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503, 2015.

[42] Kristopher De Asis, J Fernando Hernandez-Garcia, G Zacharias Holland, and Richard S Sutton. Multi-step reinforcement learning: A unifying algorithm. *arXiv preprint arXiv:1703.01327*, 2017.

[43] Sam Devlin, Marek Grześ, and Daniel Kudenko. Multi-agent, reward shaping for robocup keepaway. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1227–1228. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[44] Sam Michael Devlin and Daniel Kudenko. Dynamic potential-based reward shaping. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, pages 433–440. IFAAMAS, 2012.

[45] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. https://github.com/openai/baselines, 2017.

[46] Stuart Dreyfus. The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45, 1962.

[47] Mădălina M Drugan. Reinforcement learning versus evolutionary computation: A survey on hybrid algorithms. *Swarm and Evolutionary Computation*, 2018.

[48] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[49] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[50] Reuven Dukas. Evolutionary biology of insect learning. *Annual Review of Entomology*, 2008.

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

[52] Richard Evans and Jim Gao. Deepmind ai reduces google data centre cooling bill by 40%. *DeepMind blog*, 20, 2016.

[53] Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

[54] Daniel J Fagnant and Kara Kockelman. Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice*, 77:167–181, 2015.

[55] Raul Fernandez, Asaf Rendel, Bhuvana Ramabhadran, and Ron Hoory. Prosody contour prediction with long short-term memory, bi-directional, deep recurrent neural networks. In *Interspeech*, pages 2268–2272, 2014.

[56] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[57] Chrisantha Fernando, Dylan Banarse, Malcolm Reynolds, Frederic Besse, David Pfau, Max Jaderberg, Marc Lanctot, and Daan Wierstra. Convolution by evolution: Differentiable pattern producing networks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 109–116. ACM, 2016.

[58] Sevan G Ficici, Ofer Melnik, and Jordan B Pollack. A game-theoretic and dynamical-systems analysis of selection methods in coevolution. *Evolutionary Computation, IEEE Transactions on*, 9(6):580–602, 2005.

[59] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

[60] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1146–1155. JMLR. org, 2017.

[61] D. Fogel. An introduction to simulated evolutionary optimization. In *IEEE Transactions on Neural Networks*. 1994.

[62] David B Fogel. *Evolutionary computation: toward a new philosophy of machine intelligence*, volume 1. John Wiley & Sons, 2006.

[63] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[64] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[65] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[66] Matteo Gagliolo and Jürgen Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3-4):295–328, 2006.

[67] Tanmay Gangwani and Jian Peng. Genetic policy optimization. *arXiv preprint arXiv:1711.01012*, 2017.

[68] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[69] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, pages 189–194. IEEE, 2000.

[70] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[71] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Efficient non-linear control through neuroevolution. In *European Conference on Machine Learning*, pages 654–662. Springer, 2006.

[72] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, 9(May):937–965, 2008.

[73] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[74] Laura Grabowski, David Bryson, Fred Dyer, Charles Ofria, and Robert Pennock. Early evolution of memory usage in digital organisms. *Procedings of the International Conference on Artificial Life*, 2010.

[75] Laura M Grabowski, David M Bryson, Fred C Dyer, Charles Ofria, and Robert T Pennock. Early evolution of memory usage in digital organisms. In *ALIFE*, pages 224–231, 2010.

[76] Alex Graves. Supervised sequence labelling. In *Supervised Sequence Labelling with Recurrent Neural Networks*, pages 5–13. Springer, 2012.

[77] Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376. ACM, 2006.

[78] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[79] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE, 2013.

[80] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.

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

[82] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines, 2014. Preprint at https://arxiv.org/abs/1410.5401.

[83] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[84] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[85] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 2016.

[86] Jason Gregory, Jonathan Fink, Ethan Stump, Jeffrey Twigg, John Rogers, David Baran, Nicholas Fung, and Stuart Young. Application of multi-robot systems to disaster-relief scenarios with limited communication. In *Field and Service Robotics*, pages 639–653. Springer, 2016.

[87] Rasmus Boll Greve, Emil Juul Jacobsen, and Sebastian Risi. Evolving neural turing machines. In *Neural Information Processing Systems: Reasoning, Attention, Memory Workshop*, 2015.

[88] Rasmus Boll Greve, Emil Juul Jacobsen, and Sebastian Risi. Evolving neural turing machines. In *Neural Information Processing Systems: Reasoning, Attention, Memory Workshop*, 2015.

[89] Rasmus Boll Greve, Emil Juul Jacobsen, and Sebastian Risi. Evolving neural turing machines for reward-based learning. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 117–124. ACM, 2016.

[90] Shixiang Gu, Tim Lillicrap, Richard E Turner, Zoubin Ghahramani, Bernhard Schölkopf, and Sergey Levine. Interpolated policy gradient: Merging

on-policy and off-policy gradient estimation for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 3849–3858, 2017.

[91] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems*, pages 66–83. Springer, 2017.

[92] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[93] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[94] Jack Hanson, Yuedong Yang, Kuldip Paliwal, and Yaoqi Zhou. Improving protein disorder prediction by deep bidirectional long short-term memory recurrent neural networks. *Bioinformatics*, page btw678, 2016.

[95] Demis Hassabis, Dharshan Kumaran, Christopher Summerfield, and Matthew Botvinick. Neuroscience-inspired artificial intelligence. *Neuron*, 95(2):245–258, 2017.

[96] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[97] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[98] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *arXiv preprint arXiv:1709.06560*, 2017.

[99] Sepp Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen*. PhD thesis, diploma thesis, institut für informatik, lehrstuhl prof. brauer, technische universität münchen, 1991.

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

[101] Geoffrey A Hollinger, Sunav Choudhary, Parastoo Qarabaqi, Christopher Murphy, Urbashi Mitra, Gaurav S Sukhatme, Milica Stojanovic, Hanumant Singh, and Franz Hover. Underwater data collection using robotic sensor networks. *IEEE Journal on Selected Areas in Communications*, 30(5):899–911, 2012.

[102] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[103] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1109–1117, 2016.

[104] M Ani Hsieh, Anthony Cowley, James F Keller, Luiz Chaimowicz, Ben Grocholsky, Vijay Kumar, Camillo J Taylor, Yoichiro Endo, Ronald C Arkin, Boyoon Jung, et al. Adaptive teams of autonomous aerial and ground robots for situational awareness. *Journal of Field Robotics*, 24(11-12):991–1014, 2007.

[105] Riashat Islam, Peter Henderson, Maziar Gomrokchi, and Doina Precup. Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. *arXiv preprint arXiv:1708.04133*, 2017.

[106] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

[107] Herbert Jaeger. Artificial intelligence: Deep neural reasoning. *Nature*, 538(7626):467–468, 2016.

[108] Niels Justesen and Sebastian Risi. Learning macromanagement in starcraft from replays using deep learning. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 162–169. IEEE, 2017.

[109] Athanasios Ch Kapoutsis, Savvas A Chatzichristofis, Lefteris Doitsidis, Joao Borges de Sousa, Jose Pinto, Jose Braga, and Elias B Kosmatopoulos.

Real-time adaptive multi-robot exploration with application to underwater map construction. *Autonomous robots*, 40(6):987–1015, 2016.

[110] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246, 2013.

[111] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[112] Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.

[113] Shauharda Khadka, Jen Jen Chung, and Kagan Tumer. Evolving memory-augmented neural architecture for deep memory problems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 441–448. ACM, 2017.

[114] Shauharda Khadka, Jen Jen Chung, and Kagan Tumer. Memory-augmented multi-robot teams that learn to adapt. In *Multi-Robot and Multi-Agent Systems (MRS), 2017*, pages 128–134. IEEE, 2017.

[115] Shauharda Khadka, Jen Jen Chung, and Kagan Tumer. Neuroevolution of a modular memory-augmented neural network for deep memory problems. *Evolutionary computation*, pages 1–26, 2018.

[116] Shauharda Khadka, Somdeb Majumdar, Santiago Miret, Evren Tumer, Tarek Nassar, Zach Dwiel, Yinyin Liu, and Kagan Tumer. Collaborative evolutionary reinforcement learning. In *International Conference on Machine Learning*, 2019.

[117] Shauharda Khadka, Somdeb Majumdar, and Kagan Tumer. Evolutionary reinforcement learning for sample-efficient multiagent coordination. In *Advances in Neural Information Processing Systems*, page In Review, 2019.

[118] Shauharda Khadka and Kagan Tumer. Evolution-guided policy gradient in reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1196–1208, 2018.

[119] Shauharda Khadka, Connor Yates, and Kagan Tumer. A memory-based multiagent framework for adaptive decision making. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1977–1979. International Foundation for Autonomous Agents and Multiagent Systems, 2018.

[120] Shauharda Khadka, Connor Yates, and Kagan Tumer. Distributed one shot decision making. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2019.

[121] Shauharda Khadka, Connor Yates, and Kagan Tumer. Distributed one shot decision making. In *Multi-Robot and Multi-Agent Systems (MRS), 2019*, page In Review. IEEE, 2019.

[122] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

[124] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. Robocup: The robot world cup initiative. 1995.

[125] Alexander Kleiner, Alessandro Farinelli, Sarvapali Ramchurn, Bing Shi, Fabio Maffioletti, and Riccardo Reffato. Rmasbench: benchmarking dynamic multi-agent coordination in urban search and rescue. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 1195–1196. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[126] Matt Knudson and Kagan Tumer. Coevolution of heterogeneous multi-robot teams. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 127–134. ACM, 2010.

[127] AN Kolmogoro. On the representation of continuous functions of several variables as superpositions of functions of smaller number of variables. In *Soviet. Math. Dokl*, volume 108, pages 179–182, 1956.

[128] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[129] Ankit Kumar, Ozan Irsoy, Jonathan Su, James Bradbury, Robert English, Brian Pierce, Peter Ondruska, Ishaan Gulrajani, and Richard Socher. Ask me anything: Dynamic memory networks for natural language processing. *CoRR, abs/1506.07285*, 2015.

[130] Felix Kunz, Dominik Nuss, Jürgen Wiest, Hendrik Deusch, Stephan Reuter, Franz Gritschneder, Alexander Scheel, Manuel Stübler, Martin Bach, Patrick Hatzelmann, et al. Autonomous driving at ulm university: A modular, robust, and sensor-independent fusion approach. In *2015 IEEE intelligent vehicles symposium (IV)*, pages 666–673. IEEE, 2015.

[131] Michail G Lagoudakis and Michael L Littman. Algorithm selection using reinforcement learning. In *ICML*, pages 511–518. Citeseer, 2000.

[132] Kevin J Lang, Alex H Waibel, and Geoffrey E Hinton. A time-delay neural network architecture for isolated word recognition. *Neural networks*, 3(1):23–43, 1990.

[133] Romain Laroche and Raphael Feraud. Reinforcement learning algorithm selection. *arXiv preprint arXiv:1701.08810*, 2017.

[134] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[135] Angeliki Lazaridou, Alexander Peysakhovich, and Marco Baroni. Multi-agent cooperation and the emergence of (natural) language. *arXiv preprint arXiv:1612.07182*, 2016.

[136] Yann Le Cun, D Touresky, G Hinton, and T Sejnowski. A theoretical framework for back-propagation. In *The Connectionist Models Summer School*, volume 1, pages 21–28, 1988.

[137] Joel Lehman, Jay Chen, Jeff Clune, and Kenneth O Stanley. Es is more than just a traditional finite-difference approximator. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 450–457. ACM, 2018.

[138] Joel Lehman and Kenneth O Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336, 2008.

[139] Joel Z Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Marecki, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 464–473. International Foundation for Autonomous Agents and Multiagent Systems, 2017.

[140] Fu-Dong Li, Min Wu, Yong He, and Xin Chen. Optimal control in microgrid using multi-agent reinforcement learning. *ISA transactions*, 51(6):743–751, 2012.

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

[142] Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7, 1970.

[143] Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

[144] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

[145] Siqi Liu, Guy Lever, Josh Merel, Saran Tunyasuvunakool, Nicolas Heess, and Thore Graepel. Emergent coordination through competition. *arXiv preprint arXiv:1902.07151*, 2019.

[146] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6379–6390, 2017.

[147] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*, pages 6379–6390, 2017.

[148] Benno Lüders, Mikkel Schläger, Aleksandra Korach, and Sebastian Risi. 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, 2017.

[149] Benno Lüders, Mikkel Schläger, and Sebastian Risi. Continual learning through evolvable neural turing machines. In *NIPS 2016 Workshop on Continual Learning and Deep Networks (CLDL 2016)*, 2016.

[150] Ashique Rupam Mahmood, Huizhen Yu, and Richard S Sutton. Multi-step off-policy learning without importance sampling ratios. *arXiv preprint arXiv:1702.03006*, 2017.

[151] James Martens and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1033–1040, 2011.

[152] Grégoire Mesnil, Yann Dauphin, Kaisheng Yao, Yoshua Bengio, Li Deng, Dilek Hakkani-Tur, Xiaodong He, Larry Heck, Gokhan Tur, Dong Yu, et al. Using recurrent neural networks for slot filling in spoken language understanding. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 23(3):530–539, 2015.

[153] Grégoire Mesnil, Xiaodong He, Li Deng, and Yoshua Bengio. Investigation of recurrent-neural-network architectures and learning methods for spoken language understanding. In *Interspeech*, pages 3771–3775, 2013.

[154] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, April 2004.

[155] Janardan Misra and Indranil Saha. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, 74(1):239–255, 2010.

[156] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

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

[158] Igor Mordatch and Pieter Abbeel. Emergence of grounded compositional language in multi-agent populations. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[159] Rémi Munos. Q (λ) with off-policy corrections. In *Algorithmic Learning Theory: 27th International Conference, ALT 2016, Bari, Italy, October 19-21, 2016, Proceedings*, volume 9925, page 305. Springer, 2016.

[160] Keiji Nagatani, Seiga Kiribayashi, Yoshito Okada, Kazuki Otake, Kazuya Yoshida, Satoshi Tadokoro, Takeshi Nishimura, Tomoaki Yoshida, Eiji Koyanagi, Mineo Fukushima, et al. Emergency response to the nuclear accident at the fukushima daiichi nuclear power plants using mobile rescue robots. *Journal of Field Robotics*, 30(1):44–63, 2013.

[161] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction.* Princeton University Press, 2016.

[162] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[163] Georg Ostrovski, Marc G Bellemare, Aaron van den Oord, and Rémi Munos. Count-based exploration with neural density models. *arXiv preprint arXiv:1703.01310*, 2017.

[164] Giambattista Parascandolo, Heikki Huttunen, and Tuomas Virtanen. Recurrent neural networks for polyphonic sound event detection in real life recordings. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 6440–6444. IEEE, 2016.

[165] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch, 2017.

[166] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[167] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)*, volume 2017, 2017.

[168] Barak A Pearlmutter. Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269, 1989.

[169] Julien Perolat, Joel Z Leibo, Vinicius Zambaldi, Charles Beattie, Karl Tuyls, and Thore Graepel. A multi-agent reinforcement learning model of common-pool resource appropriation. In *Advances in Neural Information Processing Systems*, pages 3643–3652, 2017.

[170] Fernando J Pineda. Generalization of back-propagation to recurrent neural networks. *Physical review letters*, 59(19):2229, 1987.

[171] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

[172] Mariya Popova, Olexandr Isayev, and Alexander Tropsha. Deep reinforcement learning for de novo drug design. *Science advances*, 4(7):eaap7885, 2018.

[173] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40, 2016.

[174] Gintaras V Puskorius and Lee A Feldkamp. Neurocontrol of nonlinear dynamical systems with kalman filter trained recurrent networks. *IEEE Transactions on neural networks*, 5(2):279–297, 1994.

[175] Aida Rahmattalabi, Jen Jen Chung, Mitchell Colby, and Kagan Tumer. D++: Structural credit assignment in tightly coupled multiagent domains. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4424–4429. IEEE, 2016.

[176] Aditya Rawal and Risto Miikkulainen. 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, 2016.

[177] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. Pommerman: A multi-agent playground. *arXiv preprint arXiv:1809.07124*, 2018.

[178] John R Rice. The algorithm selection problem. In *Advances in computers*, volume 15, pages 65–118. Elsevier, 1976.

[179] Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(1):25–41, 2017.

[180] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[181] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[182] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76, 2017.

[183] Jürgen Schmidhuber. A fixed size storage o (n3) time complexity learning algorithm for fully recurrent continually running networks. *Neural Computation*, 4(2):243–248, 1992.

[184] Jürgen Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992.

[185] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[186] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[187] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

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

[189] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[190] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295*, 2016.

[191] Weihua Sheng, Qingyan Yang, Jindong Tan, and Ning Xi. Distributed multi-robot coordination in area exploration. *Robotics and Autonomous Systems*, 54(12):945–955, 2006.

[192] Craig Sherstan, Brendan Bennett, Kenny Young, Dylan R Ashley, Adam White, Martha White, and Richard S Sutton. Directly estimating the variance of the {

\

lambda}-return using temporal-difference methods. *arXiv preprint arXiv:1801.08287*, 2018.

[193] Hava T Siegelmann and Eduardo D Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

[194] Hava T Siegelmann and Eduardo D Sontag. On the computational power of neural nets. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 440–449. ACM, 1992.

[195] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[196] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a

general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[197] Kate A Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41(1):6, 2009.

[198] William M Spears, Kenneth A De Jong, Thomas Bäck, David B Fogel, and Hugo De Garis. An overview of evolutionary computation. In *European Conference on Machine Learning*, pages 442–459. Springer, 1993.

[199] Andreas Stafylopatis and Konstantinos Blekas. Autonomous vehicle navigation using evolutionary reinforcement learning. *European Journal of Operational Research*, 108(2):306–318, 1998.

[200] Kenneth O Stanley, Bobby D Bryant, and Risto Miikkulainen. Evolving adaptive neural networks with and without adaptive synapses. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 4, pages 2557–2564. IEEE, 2003.

[201] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[202] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[203] Peter Stone, Gal A Kaminka, Sarit Kraus, and Jeffrey S Rosenschein. Ad hoc autonomous agent teams: Collaboration without pre-coordination. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[204] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[205] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in neural information processing systems*, pages 2440–2448, 2015.

[206] Lifa Sun, Shiyin Kang, Kun Li, and Helen Meng. Voice conversion using deep bidirectional long short-term memory based recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 4869–4873. IEEE, 2015.

[207] Martin Sundermeyer, Tamer Alkhouli, Joern Wuebker, and Hermann Ney. Translation modeling with bidirectional recurrent neural networks. In *EMNLP*, pages 14–25, 2014.

[208] Martin Sundermeyer, Tamer Alkhouli, Joern Wuebker, and Hermann Ney. Translation modeling with bidirectional recurrent neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 14–25, 2014.

[209] Donny Sutantyo, Paul Levi, Christoph Möslinger, and Mark Read. Collective-adaptive lévy flight for underwater multi-robot exploration. In *2013 IEEE International Conference on Mechatronics and Automation*, pages 456–462. IEEE, 2013.

[210] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[211] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[212] Melanie Swan. *Blockchain: Blueprint for a new economy.* " O'Reilly Media, Inc.", 2015.

[213] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, OpenAI Xi Chen, Yan Duan, John Schulman, Filip DeTurck, and Pieter Abbeel. # exploration: A study of count-based exploration for deep reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2750–2759, 2017.

[214] Trias Thireou and Martin Reczko. Bidirectional long short-term memory networks for predicting the subcellular localization of eukaryotic proteins. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(3), 2007.

[215] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. A real-time algorithm for mobile robot mapping with applications to multi-robot and 3d mapping. In *ICRA*, volume 1, pages 321–328, 2000.

[216] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2), 2012.

[217] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[218] Walter F Truszkowski, Michael G Hinchey, James L Rash, and Christopher A Rouff. Autonomous and autonomic systems: A paradigm for future space exploration missions. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 36(3):279–291, 2006.

[219] Chih-Fong Tsai and Jhen-Wei Wu. Using neural network ensembles for bankruptcy prediction and credit scoring. *Expert systems with applications*, 34(4):2639–2649, 2008.

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

[221] Kagan Tumer and Adrian Agogino. Distributed agent-based air traffic flow management. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 255. ACM, 2007.

[222] Peter Turney, Darrell Whitley, and Russell W Anderson. Evolution, learning, and instinct: 100 years of the baldwin effect. *Evolutionary Computation*, 4(3):iv–viii, 1996.

[223] George E Uhlenbeck and Leonard S Ornstein. On the theory of the brownian motion. *Physical review*, 36(5):823, 1930.

[224] Oriol Vinyals, Charles Blundell, Tim Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *Advances in Neural Information Processing Systems*, pages 3630–3638, 2016.

[225] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler,

John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[226] Viola Volpato, Badr Alshomrani, and Gianluca Pollastri. Accurate ab initio and template-based prediction of short intrinsically-disordered regions by bidirectional recurrent neural networks trained on large-scale datasets. *International journal of molecular sciences*, 16(8):19868–19885, 2015.

[227] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*, 37(3):328–339, 1989.

[228] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and K Lang. Phoneme recognition: neural networks vs. hidden markov models vs. hidden markov models. In *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*, pages 107–110. IEEE, 1988.

[229] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

[230] Paul J Werbos. Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer, 1982.

[231] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[232] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

[233] Shimon Whiteson and Peter Stone. Evolutionary function approximation for reinforcement learning. *Journal of Machine Learning Research*, 7(May):877–917, 2006.

[234] Ronald J Williams. Complexity of exact gradient computation algorithms for recurrent neural networks. Technical report, Technical Report Technical Report NU-CCS-89-27, Boston: Northeastern University, College of Computer Science, 1989.

[235] Ronald J Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural computation*, 2(4):490–501, 1990.

[236] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[237] Simon A Williamson, Enrico H Gerding, and Nicholas R Jennings. Reward shaping for valuing communications during multi-agent coordination. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 641–648. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[238] Kyle Hollins Wray, Luis Pineda, and Shlomo Zilberstein. Hierarchical approach to transfer of control in semi-autonomous systems. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 1285–1286. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

[239] Zhaoming Xie, Patrick Clary, Jeremy Dao, Pedro Morais, Jonathan Hurst, and Michiel van de Panne. Iterative reinforcement learning based design of dynamic locomotion skills for cassie. *arXiv preprint arXiv:1903.09537*, 2019.

[240] Logan Yliniemi, Adrian K Agogino, and Kagan Tumer. Multirobot coordination for space exploration. *AI Magazine*, 35(4):61–74, 2014.

[241] Logan Yliniemi, Adrian K Agogino, and Kagan Tumer. Simulation of the introduction of new technologies in air traffic management. *Connection Science*, 27(3):269–287, 2015.

[242] Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 362, 2015.

[243] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.