PhD Thesis

Bio-Inspired Approaches to Adaptive Artificial Agents

A thesis submitted in compliance with the requirements for the degree of Doctor of Philosophy

by

Joachim Winther Pedersen

Supervisor:
Sebastian Risi

Date:
July 31st 2023

Department of Digital Design
IT University of Copenhagen

Abstract

Despite significant recent advances, artificial agents are still far behind biological agents in their abilities to adapt to novel and unexpected situations. This thesis contributes to the field of adaptive artificial agents, taking inspiration from biology to develop new methods that extend the capabilities of artificial agents controlled by neural networks. Several methods are introduced: (a) An algorithm, named *Evolve & Merge*, that progressively decreases the number of plasticity rules used to update the synapses of a neural network until the number of rules is orders of magnitude smaller than the number of rules. This is done without diminishing the agent's performance and without extending the overall training time; (b) A parameterization of neurons in a neural network that makes each neuron a tiny dynamical system. These neurons are shown to be expressive enough to solve several reinforcement learning (RL) tasks even when no synapses are optimized and only the parameters of the neurons are evolved in random neural networks; (c) A meta-learning framework that optimizes a network to provide a reward signal for an RL agent. The evolved reward signal is shown to enhance the training stability of the RL agent as well as enable the agent to maintain performance in novel circumstances through continued optimization with the evolved reward signal; (d) A demonstration of the minimal requirements for agents to become invariant to permutations of the input elements as well as the size of the input and output vectors; (e) A framework, named *Structurally Flexible Adaptive Neural Networks* (SFANN) that combines ideas from the earlier contributions of the thesis. SFANNs have a small number of plasticity rules, parameterized dynamic neurons, the ability to learn from rewards, and are flexible in the structure both when it comes to the input and output, and the hidden layers. This framework is put forward as a method that can be optimized in environments of different input and output dimensions to eventually allow a single set of parameters to serve as a general learner across many contexts.

Resumé

Trods betydelige fremskridt er kunstige agenters evner til at tilpasse sig nye og uventede situations stadig langt bagud i forhold til biologiske agenter. Denne afhandling henter inspiration fra biologi til at udvikle nye metoder der udvider evnerne hos kunstige agenter styret af neurale netværk. Denne afhandling bidrager med flere nye metoder: (a) En algoritme, kaldet *Evolve & Merge*, som gradvist reducerer antallet af plasticitetsregler, der bruges til at opdatere synapserne i et neuralt netværk, indtil antallet af regler er mange størrelsesordener mindre end antallet af synapser. Dette gøres uden at forringe agentens ydeevne og uden at forlænge den samlede træningstid; (b) En parameterisering af neuroner i et neuralt netværk, der gør hver neuron til et lille dynamisk system. Disse neuroner viser sig at være i stand til at løse flere *reinforcement learning* (RL) -opgaver, selv når ingen synapser er optimeret, og kun neuronerne udvikles i tilfældige neurale netværk; (c) En meta-læringsmetode, der optimerer et netværk til at give et belønningssignal til en RL-agent. Det optimerede belønningsignal forbedrer træningsstabiliteten for RL-agenten og giver mulighed for, at agenten kan opretholde sin præstationsevne under nye omstændigheder ved fortsat optimering med det optimerede belønningsignal; (d) Demonstration af de minimale krav for agenter for at gøre dem invariante over for permutering af inputelementerne såvel som størrelsen af input- og outputvektorerne; (e) En metode kaldet *Structurally Flexible Adaptive Neural Network* (SFANN), der kombinerer ideer fra afhandlingens tidligere bidrag. SFANN bruger et lille antal plasticitetsregler, parameteriserede dynamiske neuroner, belønnings-baseret læring og er fleksibel i strukturen, både når det kommer til input, output og de skjulte lag i netværket. Denne fremgangsmåde foreslås som en metode, der kan optimeres i miljøer med forskellige input- og outputdimensioner for med tiden at tillade, at et enkelt sæt af parametre fungerer som en generel læringsmekanisme på tværs af mange kontekster.

Denne afhandling er dedikeret til min familie, mine forældre, mine brødre Rasmus og Laurits og til min elskede Monica.

Acknowledgements

Working on this thesis has been my dream job and I have been privileged to have had the opportunity to work with many kind and interesting people. I would first and foremost like to thank my supervisor Professor Sebastian Risi who has given me extraordinary advice, inspiring ideas, and steady support throughout my time as a PhD student. His mentorship has played a pivotal role in shaping the outcome of this thesis. I am also indebted to the Robotics Evolution and Art Lab (REAL) and the Creative AI Lab at ITU for fostering a positive and conducive work environment.

I would like thank Andrea Soltoggio and his students, Eseoghene Ben-Iwhiwhu, Saptarshi Nath, and Christos Peridis for welcoming me into their group for my stay abroad. I am truly grateful for the opportunity to work with them.

Further, I would like to express my gratitude to everyone I had the chance to work with on different projects and from whom I have learned a lot. These include Weiyi Zou, Djordje Grbic, Rasmus Berg Palm, Worasuchad ”Happy” Haomachai, and Binggwong Leung.

I would also like to extend a special appreciation to Vibe Qvist Mathiasen and Julie Tweddell Jacobsen, who have worked as an outstanding PhD Support team. Their assistance and dedication have always been extremely helpful and it has been invaluable knowing that I could always come to PhD Support with any questions.

Lastly, I am grateful to the Independent Research Fund Denmark (Danmarks Frie Forskningsfond) for funding the research of this PhD thesis.

Table of Contents

Abstract iii
Resumé iv
Acknowledgements vi

1 Introduction 1

2 Background 7
2.1 Artificial Agents and Environments 7
2.2 Neural Networks 8
2.3 Reinforcement Learning 11
2.4 Bio-Inspired Optimization: Neuro-Evolution 13
2.5 Meta-Learning 18

3 Mapping Loss Landscapes of Meta-Learning 25
3.1 Optimizing a Starting Point 31
3.2 Optimizing Loss Landscapes 44
3.3 Optimizing an Optimizer 59
3.4 Putting it All Together 61

4 Evolving and Merging Hebbian Learning Rules 64
4.1 Approach 67
4.2 Results 71
4.3 Discussion 75
4.4 Conclusion 81

5 Learning to Act through Evolution of Neural Diversity in Random Neural Networks 83
5.1 Related Work 85
5.2 Evolving Diverse Neurons in Random Neural Networks 87

5.3 Experiments 89
5.4 Results 92
5.5 Discussion and Future Work 98

6 Evolution of an Internal Reward Function for Reinforcement Learning 101
6.1 Related Work 103
6.2 Approach: Evolving Internal Reward for Reinforcement Learning 105
6.3 Experiments 107
6.4 Results 111
6.5 Discussion 118
6.6 Conclusion 121

7 Minimal Neural Network Models for Permutation Invariant Agents 122
7.1 Related Work 123
7.2 Approach: A Minimal Neural Model for Permutation Invariance 126
7.3 Experiments 127
7.4 Results 134
7.5 Discussion 137
7.6 Conclusion 141

8 Evolution of Structurally Flexible Adaptive Neural Networks: Toward a Model for General Learners 143
8.1 Related Work 145
8.2 Structural Flexibility through Parameter Sharing Neuron and Synapse Classes 147
8.3 Experiments 155
8.4 Results 158
8.5 Discussion and Future Work 163

9 General Discussion and Future Research 169
9.1 General Discussion 169
9.2 Future Research 179

10 Conclusion 185

Bibliography 187

Appendix **213**
A Appendix Section 1 213

Chapter 1

Introduction

The ability to adapt to the environment is critical for any agent in the real world. Agents that can only function in circumstances that are exactly like those in an original training set are of limited use. The world is full of possible interactions between objects and agents in ever-expanding and unpredictable ways. Evolution has since the beginning of life created creatures of an enormous variety of shapes, sizes, and behavioral niches. With more complex creatures came increased needs for motor control and perception of the world, which was accompanied by a growing nervous system, nature's ultimate adaptation machine.

One of the long-standing challenges of artificial intelligence (AI) research is to create artificial agents that can cope with varying environments at the same level as animals. An early idea meant to bring these capabilities to machines is that of connectionism. From McCulloch-Pitts neurons from the 1940s to the deep artificial neural networks (ANNs) with billions of parameters of today, the idea of weighting and integrating information as a primary means to make machines more intelligent has come a long way.

The past decade has seen much progress in the research fields of deep learning (Schmidhuber, 2015; LeCun et al., 2015; Arulkumaran et al., 2017; Chauhan and Singh, 2018; Tay et al., 2022), and robotics (Pierson and Gashler, 2017). Using large data sets, neural networks have achieved state-of-the-art performance in domains such as computer vision (Voulodimos et al., 2018; Khan et al., 2022), natural language processing (Wolf et al., 2020), sequence modeling (Salehinejad et al., 2017), and speech recognition (Nassif et al., 2019). At the same time, detailed simulators of robots have enabled immense speed-ups in collecting trajectories of moving robots (Collins et al.,

2021; Choi et al., 2021).

However, a pervasive challenge for ANNs is their immense need for training data. Another is that ANNs after training can be vulnerable to failures if the incoming data comes from a different distribution than the training data. These challenges are especially limiting to the development of artificial agents meant to interact with complex environments. First, in a complex environment, out-of-distribution (OOD) events can occur at a frequent level, as it is not possible to plan for every single contingency and include it in the training data of the agent. Second, if the agent needs an extensive fine-tuning time every time a distributional shift occurs it will be at risk of always being behind and thus not able to exhibit useful behavior.

One framework that attempts to mitigate these challenges for artificial agents is called meta-learning, i.e., learning to learn. Several approaches exist to meta-learning. For some, the goal is to decrease the length of the adaptation phase when a distributional shift has occurred, or a novel task is encountered. For others, the goal is to let the network change a subset of its parameters continually so that the agent can adapt to the environment online. In either case, common for meta-learning approaches is that they consist of a double optimization loop. The idea is that the outer-loop optimization process only must be done once, allowing for inner-loop optimization to be run efficiently whenever needed in the future, or as a continual process for the rest of the agent's lifetime. Chapter 3 will revolve around the concept of meta-learning and how the loss landscapes of the two loops interact with each other during optimization. Using small experiments to visualize loss landscapes, the goal of Chapter 3 is to demonstrate that the parameters optimized in the outer-loop can affect the loss landscapes of the inner-loop parameters in one or more of three different manners. It is argued that categorizing meta-learning approaches in terms of how the outer-loop parameters affect the inner-loop loss landscape might be a useful alternative to the traditional focus on the type of optimizer used in the inner-loop.

Meta-learning is a broad framework that is not limited to approaches inspired by biology. While ANNs were originally inspired by the brain, they differ from biological neural networks on several key points. One crucial difference is that most ANNs have their connection strengths trained during a training phase, after which they stay fixed forever. Biological neural networks on the other hand undergo changes throughout their lifetimes. The framework

of Evolved Plastic Artificial Neural Networks (EPANNs) (Soltoggio et al., 2018) aims to incorporate the plasticity – and thereby the adaptability – of biological neural networks into artificial agents. The brain architectures found in biological agents have been shaped by evolution throughout millions of years (Breedlove and Watson, 2013). The specific wiring of the synapses within each individual brain is a result of learning to adapt to sensory input, faced within the lifetime of the individual (Power and Schlaggar, 2017; Stiles, 2000; Greenough and Black, 2013). Chapter 4 contributes to the EPANN framework by introducing the *Evolve and Merge* method. Here, a neural network is initialized with a unique parameterized synaptic learning rule for each of its synapses. These are optimized using evolution. However, these learning rules are during evolution clustered together with other similar rules, so that the number of unique rules gradually decreases, resulting in a parameter space that is just a fraction of the original size.

The work presented in this chapter was peer-reviewed and published as a conference paper: Pedersen, J. W., & Risi, S. (2021). Evolving and merging Hebbian Learning Rules: Increasing Generalization by Decreasing the Number of Rules. In Proceedings of the Genetic and Evolutionary Computation Conference (pp. 892-900).

Brains of animals are characterized by the presence of neurons of many different classes (Lillien, 1997; Soltesz et al., 2006). Each neural class has been shaped by evolution to contain unique properties that allow the processing of incoming signals in different manners. Further, biological neurons are dynamical systems, capable of integrating information over time and responding to inputs in a history-dependent manner (Sekirnjak and Du Lac, 2002; Izhikevich, 2003, 2007; Wasmuht et al., 2018). Whereas the evolutionary optimization of biological neural networks has resulted in networks having multiple classes of recurrent processors, artificial neural networks (ANNs) tend to contain homogeneous activation functions, and optimization is most often focused on the tuning of synaptic weights.

Information in neural networks spreads through many synapses and converges at neurons. Given that a single biological neuron is a sophisticated information processor in its own right (Marder et al., 1996; Beaulieu-Laroche et al., 2018; Beniaguev et al., 2021), it might be useful to reconsider the extreme abstraction of a neuron as being represented by a single scalar, and an activation function that is shared with all other neurons in the network. This is exactly the goal of Chapter 5. Here, a parameterization of neurons

in a neural network is introduced that turns neurons into tiny dynamical systems that are updated in parallel. The evolved neurons can respond in diverse manners to the same incoming inputs and this expressiveness lets the neural networks with optimized neurons solve several reinforcement learning tasks without optimizing the weights of the networks.

The work in this chapter was peer-reviewed and published as a conference paper: Pedersen, J. W., & Risi, S. (2023). Learning to Act through Evolution of Neural Diversity in Random Neural Networks. In Proceedings of the Genetic and Evolutionary Computation Conference (pp. 1248-1256).

One component of the ability to adapt to one's environment is to recognize the differences between useful and maladaptive behavior. Sub-systems of biological nervous systems have evolved mechanisms that convert sensory input to feedback signals used to guide future behavior. Internal reward systems are important as there exist no actions in complex environments that are objectively beneficial regardless of the circumstances. Agents thus need to not only have systems that can decide what to do next but also systems that can motivate behavioral changes going forward. Examples of this include the evolved taste perception of many species and the interaction different tastes have with dopaminergic networks that support the learning of food preferences (Sclafani et al., 2011; Waddell, 2013; Beauchamp, 2016). Further, some studies have shown that there are motivational neural circuits in the brain of animals located in the limbic system and striatum (Chau et al., 2004; Price and Drevets, 2012). These motivational neural circuits receive and integrate multiple types of information to generate reward and punishment signals and allow the animal brain to perform reinforcement learning processes based on these internal signals (Averbeck and Costa, 2017; Neftci and Averbeck, 2019). Chapter 6 of this thesis presents a series of experiments where a small recurrent neural network is evolved to provide a reward signal to a reinforcement learning agent. The results show that the evolved learning signal results in enhanced training stability and in some experiments a better recovery of performance after a distributional shift.

A shortened version of the work presented in this chapter was published and presented at GECCO'23: Zou, W., Pedersen, J. W., & Risi, S. (2023). Evolution of an Internal Reward Function for Reinforcement Learning. Companion to the Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO Companion 2023).

A different type of flexibility found in biological neural networks that is missing in most ANNs is *structural flexibility*. The fact that the parameters of an optimized ANN are strictly tied to their specific indices in the network architecture, also means that they are dependent on the elements of the external inputs always being presented in the same ordering. There are several downsides to this inflexibility. First, we might not always know beforehand the number of inputs that are ideal for our model. Ideally, we would be able to add a new sensor and the model would during deployment figure out how to integrate that information. However, most neural architectures do not easily allow this, and we would be forced to start all over if we wanted to incorporate the new input.

Additionally, there might be cases where we cannot guarantee that inputs to our model will always arrive in the same order as during the optimization phase. When parameters are tied to the indices of the ANN, any permutation of the input elements might be catastrophic. Recently, multiple methods have been proposed to mitigate the inflexibility of ANNs and make them invariant to permutations of inputs (Tang and Ha, 2021; Kirsch et al., 2021). In these studies, it was also found that the properties of invariance to permutations and changes in the size of the input vector were accompanied by extra robustness of the models to unseen perturbations after training. In Chapter 7, the minimal requirements that must be fulfilled in order to construct a neural network model that is invariant to permutations and the size of the inputs are identified. Such models are then constructed and tested along with different ablations to demonstrate the invariance properties of these minimal models.

The work in this chapter was peer-reviewed and published as a conference paper: Pedersen, J. W., & Risi, S. (2022). Minimal neural network models for permutation invariant agents. In Proceedings of the Genetic and Evolutionary Computation Conference (pp. 130-138).

An idea that underpins this thesis is that incorporating properties into our ANNs such that they become more flexible, both in terms of synaptic plasticity and network structure, will result in enhanced opportunities for optimizing agents with general learning capabilities. For this reason, the framework of Structural Flexible Adapting Neural Networks (SFANNs) is presented in Chapter 8 with a small series of experiments that serve as a proof-of-concept of the potential of the approach. Using ideas from earlier chapters and building on the Variable Sharing Meta-Learning framework by Kirsch and Schmid-

huber (Kirsch et al., 2019), networks are evolved that after evolution can be freely reconfigured and use a reward signal to organize into a functional network, even for environment variations that were never seen during evolution. This type of network has more properties in common with biological neural networks compared to normal ANNs. Namely, SFANNs consist of dynamical neurons and plastic synapses of different classes. By focusing on evolving the parameters of these building blocks rather than the parameters of a specific network architecture, the freedom is gained to put together a network of any size after evolution.

Chapter 9 contains a general discussion of the topics and approaches that have themed the thesis, and Chapter 10 contains the overall conclusion.
First, Chapter 2 will provide a brief introduction to the topics that are fundamental to this thesis.

Chapter 2

Background

This chapter will provide introductions to areas that the experiments in later chapters are based on. These areas include neural networks, reinforcement learning, evolution algorithms, and meta-learning. First, a small section will introduce the basic terminology of artificial agents and their environments.

2.1 Artificial Agents and Environments

An agent is an entity that interacts with its environment. It has the ability to perceive and comprehend information from the environment, subsequently taking actions that impact the state of its surroundings. As a result, these alterations in the environment reciprocally influence the agent's future perceptions.

In the case of an artificial agent, its actions are governed by a controlling model, typically a neural network, responsible for interpreting environmental inputs and generating appropriate responses. Throughout this thesis, artificial agents will always be controlled by various types of neural networks.

Environments can range from simple virtual ones to the real world. One crucial aspect is that the state of an environment depends on its past history. Alternatively, an environment may consist of static structures such as mazes or grids, with the only dynamic element being the agent's changing perspective over time. In either case, temporal consistency remains a significant characteristic of any environment. Additionally, environments are typically designed with specific tasks or objectives that agents must accomplish through their actions. Some environments are episodic, wherein

the environment is reset after a fixed number of time steps or following the agent's successful completion or failure of a task.

The lifetime of an agent spans the entire duration during which the agent operates and undergoes evaluation. This period may encompass multiple episodes and tasks within a particular environment, and the agent might traverse through various environments over its lifetime.

2.2 Neural Networks

Neural networks and deep learning are sub-fields of machine learning that draw inspiration from the structure and functioning of the human brain (McClelland et al., 1987). With enough data, artificial neural networks (ANNs) can be optimized, or trained, to solve complex tasks, surpassing traditional machine learning algorithms in various domains. This section will provide a brief introduction to feedforward neural networks, which is the simplest deep learning architecture, and which lays the foundation for all methods used in Chapters 3 to 8.

A feedforward neural network is a directed acyclic graph that consists of interconnected layers of artificial neurons, also known as nodes or units (Goodfellow et al., 2016). The flow of information within the network moves strictly in a forward direction, from the input layer to the output layer, without any feedback loops. When information reaches a layer of a neural network, we say that the neurons in the layer become activated. This simply means that a value is calculated for each neuron based on the incoming information, and this value corresponds to the level of activation of the neuron. Between each layer of neurons is a matrix, known as a weight matrix, that determines the strength of the connection between each of the sending neurons to each of the receiving neurons. The strength of a connection, also known as a weight, is represented by a single scalar in the matrix. The weights thus determine the communication between layers and shape the information as it is propagated forward through the network. Together, the weights in a neural network form a parameterized function, and it is the weights that are subject to optimization when training a neural network. In biological neural networks, connections between neurons are referred to as synapses. Along with the weights of a neural network, another set of parameters that are optimized during the training of a neural network are called the biases. A bias is a scalar, and each neuron in a feedforward neural network has a

bias associated with it. The bias contributes to the level of activation of the neuron in addition to the sum of incoming values coming from neurons of a previous layer. The input layer of a neural network is special in that the input neurons have no biases. The input simply passes the input features forward without any computation, and the output of the input layer is thus the input vector itself.

Practically, information is propagated from input to output using matrix multiplication. The weighted sum z_{l+1} of the inputs a_l to each neuron in layer $l+1$ is computed as follows:

$$z_{l+1} = W_{l+1} \cdot a_l + b_{l+1}$$

Here, "$\cdot$" represents matrix multiplication. The matrix multiplication between the weight matrix W_{l+1} and the input vector a_l performs the summation of weighted inputs from the previous layer. The bias term b_{l+1} is added element-wise. After information is propagated through a weight matrix, and the biases have been added, a non-linear activation function is applied element-wise to neurons of the receiving neurons. Activation functions facilitate the network's ability to approximate non-linear functions by introducing non-linearity in the network's computations. By making the transformation of information between layers non-linear they enable the neural network to capture and represent patterns and dependencies that are not linear within the data. Many different activation functions exist with commonly used ones being the ReLU function and the hyperbolic tangent function.

The choice of activation function also impacts the training of the neural network. Most commonly, neural networks are optimized in a process called backpropagation. This process involves the iterative adjustment of the network's weights to minimize a defined loss or error function. The error is computed by comparing the network's output with the desired output, for example a labels to inputs that need to be classified. The gradients of the loss function with respect to the network's weights are calculated using the chain rule and propagated backward through the network. Gradient descent can be used to update the weights and the biases of the network to minimize the loss and improve the network's performance. An alternative to backpropagation is the use of neuro-evolution, described in Section 2.4 below.

2.2.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of ANNs specifically designed to process sequential data by incorporating feedback connections (Salehinejad et al., 2017). In contrast to traditional feedforward neural networks, which treat input data as independent instances, RNNs possess an internal memory that allows them to retain and utilize information from previous time steps. This capability enables RNNs to effectively model temporal dependencies and learn from sequential patterns, making them highly valuable in domains such as natural language processing, speech recognition, and time series analysis.

In simple RNNs, the internal memory can simply be activations that are fed back into the input of the network or concatenated to one of its internal representations. Many architectures of RNNs exist (Yu et al., 2019) with varying levels of sophistication of how the internal memory is maintained and updated. Two commonly used architectures are called Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) and Gated Recurrent Unit (GRU). These are used in experiments in Chapter 8 and Chapter 7, respectively. These two differ slightly from each other in their architectures but are known to have similar performances.

Here are the equations that define the operations within an LSTM cell:

$$\mathbf{i}_t = \sigma(W_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i), \tag{2.1}$$

$$\mathbf{f}_t = \sigma(W_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f), \tag{2.2}$$

$$\mathbf{o}_t = \sigma(W_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o), \tag{2.3}$$

$$\mathbf{g}_t = \tanh(W_g[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_g), \tag{2.4}$$

$$\mathbf{c}_t = \mathbf{f}t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t, \tag{2.5}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t), \tag{2.6}$$

Here, σ represents the sigmoid activation function and tanh represents the hyperbolic tangent activation function. i_t is the input gate, f_t is the forget gate, and o_t is the output gate. g_t is used to update the cell state, c_t is the cell state, and h_t is the hidden state at time step t. W_i, W_f, W_o, and W_g are weight matrices. b_i, b_f, b_o, and b_g are bias vectors. h_{t-1} and x_t represent the previous hidden state and current input at time step t, respectively. $\odot$ denotes element-wise multiplication.

For the GRU, the hidden state and the outputs are determined as follows (Cho et al., 2014):

$$\mathbf{z}_t = \sigma(W_z[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_z), \tag{2.7}$$

$$\mathbf{r}_t = \sigma(W_r[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_r), \tag{2.8}$$

$$\mathbf{g}_t = \tanh(W_g[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_g), \tag{2.9}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \mathbf{g}_t, \tag{2.10}$$

2.3 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that focuses on the development of intelligent agents capable of learning optimal decision-making strategies through interactions with an environment (Sutton and Barto, 2018).

In contrast to supervised learning, where training data is typically provided with explicit labels or annotations, and unsupervised learning, which aims to discover hidden patterns or structures in unlabeled data, reinforcement learning takes a different approach. RL agents learn from the consequences of their actions rather than from pre-labeled examples. Through a process of trial and error, they explore the environment and gradually develop an understanding of which actions yield desirable outcomes and which do not.

In order to learn, an agent must receive feedback from the environment. At each step of interaction with the environment, the RL agent receives a numerical reward (or in cases of negative rewards, a numerical punishment) that serves as feedback, indicating the desirability of its recent action. In some environments, the reward value is simply zero the majority of the time, with meaningful feedback only being provided sparsely. The agent's objective is to maximize the long-term cumulative rewards it receives over time. By using this reward-based mechanism, RL agents learn to identify the actions that lead to higher rewards and, consequently, to more optimal decision-making.

The RL agent interacts with an environment that can be modeled as a Markov Decision Process (MDP). An MDP consists of a set of states representing the different configurations of the environment, a set of actions that the agent can choose from, and a transition function that determines the probability of transitioning from one state to another based on the chosen

action. Additionally, the environment provides rewards to the agent based on its actions and the resulting state transitions.

The RL agent's decision-making strategy is captured by a policy, which defines the mapping between states and the actions the agent should take. The policy can be deterministic, where it directly maps states to actions, or stochastic, where it specifies a probability distribution over actions for each state. The goal of RL is to learn an optimal policy that maximizes the expected cumulative reward over time. A core challenge in reinforcement learning lies in finding an appropriate balance between exploration and exploitation (Norman and Clune, 2023). The agent needs to explore enough to discover better actions and continuously update its policy, while also exploiting the already learned knowledge to maximize its rewards. Striking this balance is crucial to avoid getting stuck in sub-optimal policies or missing out on discovering more rewarding actions. Reinforcement learning algorithms employ various techniques to learn optimal policies. These algorithms can be categorized into model-free (Çalışır and Pehlivanoğlu, 2019) and model-based approaches (Moerland et al., 2023). Model-free methods directly learn the policy or action-value functions without explicitly modeling the dynamics of the environment. A popular model-free RL algorithm, which is also used in Chapter 6, is called Proximal Policy Optimization (PPO) (Schulman et al., 2017). To update the PPO's policy, a surrogate objective function is used that is constructed using the ratio of the new policy's probability to the old policy's probability. The loss is then calculated as the minimum of the ratio weighted by a calculated advantage and the clipped ratio weighted by the advantage. This prevents large updates to the current policy and is meant to ensure training stability. The pseudo-code for the PPO can be found in Algorithm 1

Algorithm 1 Proximal Policy Optimization (PPO)

Require: Environment dynamics: $P(s'|s, a)$
Require: Initial policy parameters: θ
Require: Number of optimization epochs: K
Require: Number of mini-batches: M
Require: Learning rate: α
Require: Clipping parameter: ϵ_{clip}
1: **for** $k = 1$ to K **do**
2: Collect trajectories using the current policy: $\mathcal{D} = \{(s_i, a_i, r_i)\}$
3: Compute advantages: $\hat{A} = compute_advantages(\mathcal{D})$
4: Normalize advantages: $\hat{A} \leftarrow (\hat{A} - mean(\hat{A}))/std(\hat{A})$
5: **for** $m = 1$ to M **do**
6: Sample mini-batch: $\mathcal{B} \sim \mathcal{D}$
7: Compute old policy probabilities: $\pi_{old}(a|s)$
8: Compute ratio of new and old policy probabilities: $r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{old}(a|s)}$
9: Compute clipped surrogate objective:

$$L(\theta) = mean\left(\min\left(r(\theta) \cdot \hat{A}, clip\left(r(\theta), 1 - \epsilon_{clip}, 1 + \epsilon_{clip}\right) \cdot \hat{A}\right)\right)$$

10: Compute policy gradient: $\nabla_\theta L(\theta)$
11: Update policy using gradient ascent: $\theta \leftarrow \theta + \alpha \nabla_\theta L(\theta)$
12: **end for**
13: **end for**

Like other with other deep learning methods, RL methods based on neural networks have been known to come with the risk of overfitting to specific conditions of the training set (Zhang et al., 2018b). This can then challenge the trained model's ability to perform if conditions later change even just slightly (Wang et al., 2022; Zhou et al., 2022).

2.4 Bio-Inspired Optimization: Neuro-Evolution

The experiments presented in this thesis all made use of evolution algorithms for the optimization of artificial agents. This section will provide an introduction to evolution algorithms as optimization algorithms for deep neural networks that control the behaviors of agents.

The theory of evolution is arguably the most consequential scientific theory of modern biology (Dennett, 1995; Kuhn, 2012). More specifically, it provides a theory of the mechanisms that made all life on the planet come to be in their various forms. One of the accomplishments of the theory of evolution is that it can be used to explain the emergence of creatures of any complexity from single to billions of cells using the same simple processes of sexual and natural selection. The famous phrase "survival of the fittest" is a short, intuitive explanation of why lifeforms look, act, and spread the way they do all around the planet.

All living beings are situated in an environment, and the lifeforms most well-adapted to their surroundings will multiply and spread their genes more readily than maladjusted organisms. Small genetic mutations, which occur during the process of gene inheritance, lead to random variations in the expressed traits, i.e., the genotypes, of organisms.

The competition for finite resources acts like a sorting mechanism of these random mutations; some will lead to well-adapted organisms that will flourish and become more plentiful, while others will perish (Lewontin, 1970). This sorting mechanism is what is referred to as evolution. As demonstrated by the Tree of Life (Pace, 2009), natural selection has resulted in the vast diversity of lifeforms we see today, all originating from a common ancestor. Natural evolution has inspired a branch of algorithms for black-box optimization of parameterized functions. While it is debatable whether natural evolution itself can be characterized as an optimization algorithm, evolution algorithms (EAs) have a long and successful history within the field of computer science (Bäck and Schwefel, 1993).

The domain of natural life is full of examples of agents that are equipped by their genome with the ability to learn and adapt their behaviors, i.e., their phenotype during their lifetime (West-Eberhard, 1989). Optimization algorithms that take inspiration from this natural process might be well-positioned to achieve artificial agents with such adaptable capabilities as well.

Evolution algorithms first and foremost have the great advantage that it is always possible to optimize for exactly what we want, with no need to restrict our models to be differentiable (Such et al., 2017). This is something that evolution algorithms have in common with all algorithms that belong to the broader category of black-box optimization algorithms. It is, on the other hand, a major difference compared to RL algorithms. This provides two major advantages for EAs. First the possibility of introducing discrete

interventions to the genomes that are being optimized. An example of this could be a change in the size of the genome, e.g., adding or removing neurons or whole layers from the neural network being optimized. Further, the non-linear activation functions of the network could be subject to change and are not restricted to differentiable functions. Second, not relying on gradients provides the freedom to always optimize for the exact objectives that we are interested in. This plays a role in environments with sparse or deceiving reward structures as mentioned earlier (Such et al., 2017). Here we are usually not interested in the outcomes of specific actions but in the ability of the network to generate the optimal sequence of action. The culminated lifetime reward is a measure of exactly this. Perhaps even more significantly, EAs allow for optimizing directly for generalization across different episodes within an environment or even across different environments altogether. This only requires expanding the lifetime of the individuals in the population to include multiple episodes and aggregating performances in each episode to a single scalar. Being able to optimize directly for generalization is significant as RL algorithms are notoriously brittle.

The following will describe specific examples of evolution algorithms. The algorithms described below were all used in one or more of the experiments presented throughout this thesis.

2.4.1 Genetic Algorithms

Genetic algorithms (GA) are arguably the simplest of the evolution algorithms. At the same time, the simple genetic algorithm is perhaps the optimization algorithm that is most directly inspired by natural evolution. The algorithm works in the following manner: A population of random neural networks is initialized. In this case, the parameters of the neural networks constitute the genome of the individuals. They are all evaluated on the given task. Individuals are then sorted according to their scores. The top-performing individuals, often referred to as "elites", are then used to determine the parameters of the next generation using both crossover and random mutation to create variation. For crossover, elites are paired up to generate an off-spring genome, where each of its parameters is copied from one of the parents at random. For mutation, Gaussian noise is added to the parameters of the newly spawned genomes (Ha, 2017b). This is the most basic implementation of a genetic algorithm, but many more advanced types exist. An advantage shared across genetic algorithms is their ability to maintain a

diversity of solutions within the population. This decreases the risk of the optimization getting stuck forever in a local optimum. Diversity of solutions can be further encouraged in multiple different manners. One of these is the use of speciation. Speciation is the process of grouping together the most similar genomes within a population. Genomes then only compete directly with genomes belonging to the same species as themselves (Li et al., 2002).

A prominent example of a more advanced genetic algorithm that uses speciation is the Neuro-Evolution of Augmented Topologies (NEAT) algorithm (Stanley and Miikkulainen, 2002; Papavasileiou et al., 2021). NEAT is also an example of an evolution algorithm that allows for genomes of different sizes within the population. Beyond using crossover and random noise as mutation operators, the NEAT algorithm also contains operators that allow for adding neurons and connections to the neural network. Speciation helps ensure that genomes with novel innovations that might not be immediately useful, get the chance to integrate the newly added neuron into the solution in an adaptive manner (Stanley and Miikkulainen, 2002).

2.4.2 Natural Evolution Strategies

Algorithms referred to as Natural Evolution Strategies (NES), evaluate a population of individuals across several generations (Wierstra et al., 2014). However, unlike GA, with NES the scores of individuals in each generation are used to approximate a gradient used for updating the mean of the population. Instead of preserving a number of elite individuals and letting them recombine to form the next generation, all members of a new generation are generated by adding noisy perturbations drawn from a Normal distribution to the current mean of the population. After evaluation of each individual, we know which perturbations improved the mean of the population and which worsened it. Using this information, all the tested perturbations can be combined to form a pseudo-gradient that can then be used to update the mean of the population, such that the next generation has a new and improved starting point. Since the updates form a pseudo-gradient, they can be passed to any optimizer traditionally used with backpropagation, such as Adam or RMSprop, which then determines what the exact parameter updates should be (Ha, 2017a). Since all the tested perturbations are used for calculating the pseudo-gradient, the updates to the population contain information about both favorable and unfavorable directions to move the mean toward. NES thus directly uses more information than GA, which simply

discards all but the best individuals. This is an advantage when, e.g., fine-tuning parameters of a neural network since the updates are more directed than the random mutations of GA; NES updates are able to find and follow a smooth gradient and converge towards optima more easily than GA updates, which remain noisy throughout the optimization process. On the other hand, natural evolution strategies are limited in their types of mutation operators, as these algorithms assume that all genomes have the same size. Similar to GAs, NES has a long history. The surge of interest in deep neural networks with many parameters also resulted in a renewed interest in NES and their ability to optimize a large parameter space. Salimans et al. (2017) showed that NES can be a scalable alternative to RL algorithms due to the great potential of parallelization of the individuals within a population.

2.4.3 Covariance Matrix Adaptation Evolution Strategy

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is an evolution strategy that uses even more information from its evaluated population to not only update its mean but also to guide its search when generating the subsequent generation. As NES described above, CMA-ES iteratively updates the mean of its population to move it toward a better and better solution. However, NES usually draws perturbations to the mean from a distribution with a fixed standard deviation (or standard deviation with a scheduled decay) that is the same across all parameters. CMA-ES, on the other hand, uses elite individuals to calculate the covariance matrix of the parameters to be optimized and uses it to adapt the specific standard deviation of each parameter. This means that depending on the evaluations of the perturbations, some parameters might be perturbed much more than others. The dynamic adaptation of the search area allows the algorithm to widen the search if the best solutions have parameters far from the mean and narrow the search as the mean gets closer to the best solutions. For the technical details of the algorithm, see (Hansen, 2016). The sophisticated search of CMA-ES allows it to function well with small population sizes relative to other evolution strategies. However, the computational requirements for the calculation of the covariance matrix increase quadratically with the number of parameters to be optimized, which means that large parameter spaces can be prohibitively expensive to optimize with CMA-ES (Müller and

Glasmachers, 2018). Tang et al. (2020) used CMA-ES to optimize their self-interpretable agents for RL tasks. In the study by Ha and Schmidhuber (Ha and Schmidhuber, 2018), CMA-ES was used to optimize a small controller using latent representations of a recurrent world model as inputs.

2.5 Meta-Learning

This section will cover the concept of meta-learning. The goal of meta-learning approaches is to optimize for learning capabilities rather than solely immediate performance. Standard RL algorithms typically assume that the agent operates in a fixed environment with stationary dynamics, which limits their ability to adapt to new and changing environments. To address this limitation, meta-reinforcement learning (meta-RL) has emerged as an approach to enable agents to learn to adapt quickly and effectively to novel and unseen environments (Beck et al., 2023). In meta-RL, the agent is trained on a distribution of tasks and learns to quickly generalize to new tasks by leveraging its prior experience. With the terminology introduced in Section 2.1, we say that meta-reinforcement learning produces agents that can learn during their lifetimes.

2.5.1 Meta-learning in nature: The Genomic Bottleneck, Phenotypic Plasticity, and the Baldwin Effect

This section provides examples of research in meta-learning in biology. Although the principles of evolution apply to all organisms, including plants and fungi, animals are intuitively closer to the definition of agents laid out in Section 2.1. The actions performed by animals during their lifetime determine their fitness in their environment. The behavior of animals is shaped by many factors on multiple timescales, some of which are genetically evolved and others that are learned (Breedlove and Watson, 2013).

The phenotype of animals can be characterized as either fixed or labile (Scheiner, 1993). Labile traits have the ability to change during an animal's lifetime, adapting at least as fast as the changes in the environment. On the other hand, fixed traits remain constant after the developmental phase.

All biological entities possess a genome, which serves as their underlying code. The genome contains a growth algorithm that provides instructions

during the developmental phase for cells to assemble into various structures such as the skeleton, organs, muscle tissue, and nervous system. The resulting morphology, determined by the genetic code, inherently constrains the types of behavior available to an animal throughout its lifetime. For example, a turtle will never learn how to fly, and a giraffe will never learn how to breathe underwater due to the constraints imposed by their respective body structures dictated by their genomes.

However, animals that can learn from their environment and adapt their behavior within the constraints of their morphology have a competitive advantage over those that repeat the same mistakes (West-Eberhard, 1989). Learning during an individual's lifetime relieves the genome from the burden of containing all the necessary information for survival and reproduction. Instead, learning allows information to be stored externally and incorporated into an animal's behavior throughout its lifetime. The information required to describe the detailed configurations of animals with larger brains is too extensive to be contained within the genome alone, leading to what is known as the genomic bottleneck (Zador, 2019). By containing information on how to learn, more complex and adaptive behavior can emerge while keeping the genome relatively compact.

The selection for learning capabilities, rather than explicit behavior, can actually accelerate evolution towards optimal points, known as the Baldwin effect. Lifetime learning modifies the loss landscape of evolutionary search, transforming optimal areas into stronger attractors. This phenomenon was famously demonstrated by Hinton and Nowlan (1987), although the exact mechanisms may differ from those observed in animals.

However, learning capabilities come at a cost. Firstly, they require an intricate structure to support the learning process (Turney, 2002). An animal must possess an adequate sensory system to recognize relevant information from the environment and an internal reward system to guide learning. Additionally, the morphology and motor control system of the animal must be sophisticated enough to enable a range of behaviors that can be learned. These requirements necessitate an elaborate nervous system, which is metabolically costly to maintain.

The second cost of learning is the risk of outsourcing the information needed for certain skills to the environment, rather than having the skills innately developed. It is not guaranteed that the environment will always provide the correct learning opportunities for an individual, and there is a possibility that an individual may encounter a situation where a particular

skill is necessary for survival before having learned that skill.

For these reasons, learning is most relevant when the same genome can find itself in diverse environments. In a completely static environment, innate skills are likely to be more beneficial than innate learning capabilities. However, the capacity for learning even in static environments may facilitate the evolution of certain traits (Scheiner, 1993). Over time, these traits can transition from being learned to innate, a process known as accommodation.

The same is true in the case of optimizing artificial agents: if we have full information about the environment that the artificial agent is to be deployed in, and we do not expect this to ever change, a regular reinforcement learning algorithm for learning a static neural network model is likely sufficient and will allow the agent to perform will as soon as it is deployed. On the other hand, if there is uncertainty about the environment and the tasks that the agent will need to solve during deployment, meta-reinforcement learning might be necessary (Wang, 2021).

Put in machine learning terms, any learning exhibited by animals is the result of meta-learning, as the learning must be supported by a genome that was in turn discovered through evolution.

2.5.2 Meta-Learning in Machine Learning

In the field of machine learning, meta-learning is a sub-field that focuses on developing algorithms and models that can learn to adapt quickly and efficiently to new tasks or environments (Tian et al., 2022). In traditional machine learning, the model is typically trained on a fixed dataset and then deployed to make predictions on new inputs. However, in meta-learning, the goal is to train the model to learn how to learn, such that it can quickly adapt to new tasks with minimal additional training.

Meta-learning approaches address these challenges by leveraging the concept of ”meta-knowledge” or ”prior knowledge” about a task distribution. Instead of training models for a specific task, meta-learning algorithms train models to learn how to learn from a set of related tasks. This training process equips the model with the ability to quickly adapt and generalize its knowledge to new tasks.

Meta-learning algorithms aim to capture the underlying patterns and regularities across tasks and use them to improve generalization and adaptation. By observing multiple tasks, the meta-learner gains insights into the commonalities, differences, and relationships between tasks. This acquired

meta-knowledge can then guide the model's learning process.

The learning-to-learn process involves two levels of learning: the meta-level and the task-level. At the meta-level, the algorithm learns how to learn by adjusting the model's parameters to optimize performance across a set of training tasks. The goal is to find an initialization of the model's parameters that allows for efficient adaptation to new tasks (Hospedales et al., 2021). In more general terms, meta-learning involves a double optimization loop: the meta-level is the outer-loop and the task-level is the inner-loop.

At the task-level, the model is presented with individual tasks and is expected to learn the optimal solution using limited data. The meta-learned knowledge influences the model's learning process by providing useful initialization or regularization. By utilizing the insights gained from meta-learning, the model can adapt more quickly, effectively, and accurately to new tasks, even with limited data.

In order to demonstrate the meta-learning algorithm's ability for quick adaptation, support, and query sets are used (Beck et al., 2023). A support set refers to a small subset of examples that are sampled from a specific task. This set is used during the adaptation phase to fine-tune the task-specific learner's parameters. The support set provides the necessary context and information for the learner to update its parameters based on the characteristics and requirements of the task at hand. By exposing the learner to task-specific examples, the support set allows the learner to specialize and adapt its knowledge to the specific task, enabling improved performance.

On the other hand, a query set represents another subset of examples sampled from the same task, distinct from the support set. The query set is used to evaluate the performance of the adapted task-specific learner after the adaptation phase. It serves as a test set to assess how well the learner has generalized and learned from the support set. By evaluating the learner's predictions on the query set, we can measure its ability to generalize to new, unseen examples from the same task. The query set provides a means to estimate the learner's performance and assess the effectiveness of the adaptation process. In the few-shot learning setting, support sets and query sets are utilized together to simulate the scenario of encountering new tasks with limited labeled data. The support sets enable the learner to adapt quickly to new tasks by leveraging a small number of task-specific examples, while the query sets provide a measure of the learner's ability to generalize to unseen examples from the same tasks.

Meta-learning approaches are usually categorized as either a gradient-

based approach or a black-box approach. The following will provide brief descriptions of some of the most influential approaches within each category.

2.5.2.1 Model Agnostic Meta-Learning:

A popular meta-learning algorithm in the field of deep learning is called Model Agnostic Meta-Learning (MAML) (Finn et al., 2017). Since it was introduced many variations of the algorithm have been proposed(Abbas et al., 2022; Jeong and Kim, 2020; Behl et al., 2019; Kayaalp et al., 2022; Javed and White, 2019; Beaulieu et al., 2020; Nguyen et al., 2021), but MAML is, in essence, an approach to meta-learning that aims to learn an initialization of a model that can quickly adapt to new tasks with minimal data. MAML achieves this by training a model on a set of related tasks. In the inner-loop, the initialization is only optimized with a few gradient descent steps on a small set of data points from a specific task. When this has been done for a number of tasks, the original initialization is then updated via gradients through the inner-loop gradients, such that the initialization over time becomes maximally sensitive to task losses. This means that after the meta-optimization has concluded, the MAML model can now improve extremely rapidly on tasks within the task distribution compared to a random initialization. Any neural network architecture that can be trained with gradient descent can be used within the framework of MAML.

2.5.2.2 Black-Box Meta-Learning:

A different meta-learning approach has more specific requirements for the neural network architecture used. Black-box meta-learning seeks to optimize networks with memory capabilities to learn to self-correct. This has typically been done using RNNs. An RNN can be optimized to correct itself in supervised learning if the input to the network is extended with the loss of its previous output as well as the previous output itself(Hochreiter et al., 2001). When the RNN is optimized in this manner across multiple tasks it can over time learn to incorporate self-correcting updates within the dynamics of its internal memory. This has also been shown to be the case for RL tasks (Wang et al., 2017; Duan et al., 2016).

2.5.3 Plastic Neural Networks

A bio-inspired approach that can also be considered a meta-learning approach is that of plastic neural networks. A neural network can be considered to be plastic if some function updates the connection strengths of the network during the network's lifetime (Mouret and Tonelli, 2014). In other words, the goal is to learn learning functions. One type of plasticity function is known as Hebbian learning rules. In neuroscience, Hebbian theory proposes that if a group of neurons is activated close to each other in time, the neurons can increase their efficacy in activating each other at a later point (Holscher, 2008). In this way, the full group could thus become active, even if just a part of the group was initially activated by external stimuli (Lansner, 2009). This type of plasticity constitutes a local learning rule, where the change in connection only depends on information local to the two connected neurons, such as their activation. In this way, recurrent connections within a layer of neurons endow a network with the ability to "complete patterns"(Hunsaker and Kesner, 2013): if a stimulus at an earlier time has activated a certain group of neurons, also called a cell assembly, the network might be able to respond with the activation of this same cell assembly if it is faced with an incomplete or noisy presentation of this stimulus (Hoffmann, 2009). These cell assemblies have been proposed to be the basic units for thoughts and cognition (Buzsáki, 2010; Pruszynski and Zylberberg, 2019; Saxena and Cunningham, 2019), and their existence was first suggested by Hebb (Pulvermüller et al., 2014).

Several past studies have focused on finding ways to use plasticity inspired by Hebbian plasticity in ANNs. The motivation for this has for some studies been to study network plasticity in computational models (Song et al., 2000; Abbott and Nelson, 2000), and for other studies the aim has been to better enable models to generalize. For example, Soltoggio et al. (2008) evolve modulatory neurons that when activated will alter the connection strengths between other neurons. With this approach, they are able to solve the T-Maze problem, where rewards are not stationary. In a study by Orchard and Wang (2016), linear and non-linear learning rules are evolved to adapt to a simple foraging task. Yaman et al. (2019) use genetic algorithms to optimize delayed synaptic plasticity that can learn from distal rewards. Common to these examples is that learning rules that update neural connections have access to a reward signal during the lifetime of the agent.

2.5.3.1 Indirect Encoding with Plasticity

Optimization of plasticity rules can also be linked to the area of indirect encodings of ANNs. One of the goals of indirect encoding approaches is to be able to represent a full solution, like a large neural network, in a compressed manner (Bentley and Kumar, 1999; Gruau et al., 1996; Stanley and Miikkulainen, 2003). Indirect encoding schemes come in many variations. Examples include letting smaller networks determine the connection strengths of a larger network (Ha et al., 2016; Risi and Stanley, 2012a, 2011; Carvelli et al., 2020), or representing the connections of a neural network by Fourier coefficients (Koutnik et al., 2010; Gomez et al., 2012). Of particular relevance to our work are methods that use plasticity rules to make indirect encodings.

Early approaches include that of Chalmers (1991), where a single parameterized learning rule with 10 parameters was evolved to allow a feedforward network to do simple input-output associations. A bit more recently, approaches such as *adaptive HyperNEAT* (Risi and Stanley, 2010) have been deployed to exploit the geometry of a neural network to produce patterns of learning rules. The *HyperNEAT* approach has previously been used for improving a controller's ability to control robot morphologies outside of what was experienced during training (Risi and Stanley, 2013). Indirectly encoding plasticity has been shown to improve a network's learning abilities, with networks that are more regular showing improved performance (Tonelli and Mouret, 2013). These earlier results point to a deep connection between plasticity and indirect encodings, which has so far received little attention.

With these introductions, we are ready to delve deeper into the double optimization loop of meta-learning in the next chapter, which explores how the two optimization loops interact with each other.

Chapter 3

Mapping Loss Landscapes of Meta-Learning

Most of the results presented in later chapters involve meta-learning. As meta-reinforcement learning is an important part of this thesis, and a growing field in general, this chapter is dedicated to exploring the dynamics of the double-loop employed in meta-learning. An introduction to meta-learning can be found in Section 2.5 of the previous chapter. As described, meta-learning approaches are usually divided into two main categories: gradient-based methods and black-box meta-learning. In this chapter, a different delineation is proposed based on how the outer-loop optimization affects the inner-loop optimization. In other words, how does the ”meta” affect the ”learning” in different meta-learning setups? The standard way of categorizing meta-learning approaches separates them based on what optimization methods are used in the inner-loop: is gradient descent used, or is inner-loop optimization a learned function (Beck et al., 2023)? Distinguishing between meta-learning approaches in this manner has roots in the history of how early meta-learning approaches were developed. However, with an increasing number of meta-learning approaches being developed in recent years, a more principled way of categorizing these might be beneficial. First of all, the gradient-based versus black-box categorization has no room for approaches that use other types of optimization processes in the inner-loop, such as evolution algorithms, swarm optimization, or Bayesian optimization (see, e.g., (Song et al., 2019a)). Note, that there is also a bit of confusion of the terms here, as evolution algorithms (Bäck and Schwefel, 1993), swarm optimization (Kennedy and Eberhart, 1995; Poli et al., 2007), or Bayesian optimization

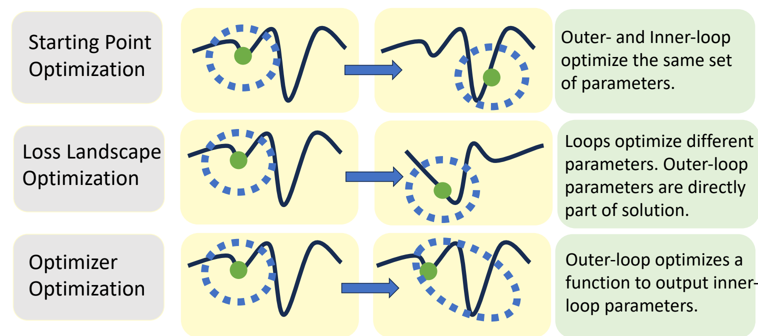


Figure 3.1: **Model of Meta-Learning Categories** This figure shows the three different ways a meta-learning outer-loop can improve the inner-loop loss landscape. The black curves depict loss landscapes of a single parameter optimized by the inner-loop. The green dots are the starting point of the inner-loop optimization. The dotted circle is the ”reach” of the inner-loop optimization, i.e., the area of the loss landscape from which the inner-loop optimization process is able to select parameters. Outer-loop optimization can aid inner-loop optimization through one or more of 1) Moving the starting point of the inner-loop optimization process to a more advantageous point; 2) Modifying the inner-loop loss landscape to be easier to navigate or; 3) Changing the reach of the inner-loop optimizer to make certain areas of the loss landscape more searchable.

(Frazier, 2018) are usually referred to as black-box optimizers - or gradient-free optimization - while in the terminology of meta-learning, "black-box" refers to an optimizer that is itself learned, usually represented by a neural network. Second, and more importantly, within the established categories of gradient-based or black-box meta-learning, different setups exist that can have major consequences for the overall optimization process regardless of the optimizer used. The perspective laid forward in this chapter is that it is not the particular type of optimizer used in any of the loops that is the most interesting, but rather how these two optimizers interact. More specifically, how can we expect the outer-loop optimization to aid the optimization of the inner-loop such that it can result in learning that is both fast and meaningful? Crucially for this argument is the simple observation that a particular optimizer and its particular hyperparameters only affect the ability of the optimizer to navigate in the loss landscape of the parameters it is optimizing, not the loss landscape itself. However, the outer- and inner-loop in a double-loop each have different loss landscapes associated with them, and depending on which parameters their respective optimizers operate on, these loss landscapes can be either static or dynamic throughout the optimization process. Categorizing double-loop optimization, or meta-learning algorithms, based on how they interact helps elucidate what a specific double-loop optimization algorithm is capable of, and the purpose of its individual optimizers.

There are some interactions that are always true in any double-loop optimization. The loss landscape of the outer-loop is from the onset of the optimization always shaped by the inner-loop optimizer, but it is also always static throughout the optimization process. Inner-loop optimization is always reset and restarted at the beginning of each new iteration of the outer-loop, and it is thus impossible for the inner-loop to dynamically change, which parameters are the most optimal of the outer-loop. Put differently, after we have made a choice of 1) the set of parameters that will be optimized in the outer-loop, and 2) the optimizer to be used in the inner-loop, the loss landscape of the outer-loop will be set in stone throughout its optimization process. The only time we might see any changes to the outer-loop loss landscape during the optimization process is when multiple different tasks are sampled in the inner-loop. Then the loss landscape of the outer-loop changes with the specific batch of tasks being sampled. This is just like any other optimization setting where the training set is defined by the current batch that it is optimized on. In the limit of having an infinite number of

tasks from the task distribution in the batch, there would be no changes to the outer-loop loss landscape.

There are three main ways in which the outer-loop can improve the efficiency of the inner-loop: 1) Starting point optimization: the outer-loop places the initial parameters of the inner-loop optimization in an area of the loss landscape that can be easily found; 2) Loss landscape optimization: the outer-loop shapes the inner-loop loss landscape itself, and makes is easier to navigate for the inner-loop optimizer; 3) Optimizer optimization: the outer-loop shapes how the inner-loop optimizer optimizes, and thus what points of the inner-loop loss landscape can be reached. See Figure 3.1 for an overview of this approach to categorizing meta-learning settings.

All of these ways of improving the inner-loop search have the same goal, but achieve them in different manners, providing meta-learning researchers with different opportunities and challenges. Despite this, loss landscape dynamics are usually not paid attention to in surveys and reviews of meta-learning of deep neural networks.

Throughout the chapter, I demonstrate the interaction between the two optimization loops in different meta-learning scenarios using a very simple environment that is solved using only at most two parameters for each loop. In this way, it is possible to derive and plot an accurate map of the loss landscapes associated with each loop through a grid search of the parameters.

Simple Environment:
To map loss landscapes of different meta-learning scenarios, an environment that is simple, fast to compute, and that can easily generate different tasks from a distribution, is needed. Here, I use the possibly most simple version of a point navigation task. In this environment, the agent navigates a 2-dimensional space. The agent begins at the origin of a graph and must move to a goal location. The input to the agent is its current location. An action in this environment is a 2-dimensional vector that is added to the agent's current location to determine the next location. The agent outputs actions until it has reached the goal location or it has run out of time. The reward at each time step is the negative of the square root of the distance between the agent's current location and the goal location. A single task in this environment is defined by the specific goal location. Goal locations can be randomly generated to constitute a task distribution. A neural network can easily be optimized to solve a single task in this environment. With the use of meta-learning, it is likewise easy to optimize a network that in a few attempts can

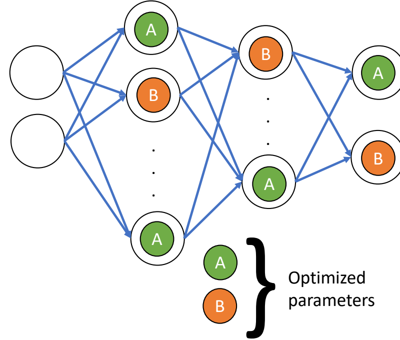


Figure 3.2: **Model with two trainable parameters.** A neural network where the weights are not trained. Rather, two scalars that are multiplied with the neurons' pre-action values are subject to optimization.

find any given goal location. However, due to the many parameters of neural networks, their loss landscapes are non-trivial to visualize. Some methods have been devised to make approximations of loss landscapes of neural networks. It is, however, a sad fact of the world that it is impossible to visualize a high-dimensional space in 2- or 3D without loss of information. Therefore, to show the loss landscapes as clearly as possible, I solve the task with just two parameters. Obviously, any task within this simple environment could be solved by simply optimizing a fixed action directly, which is always taken regardless of the input. Any fixed action that is equivalent to the goal location or scaled by a scalar between zero and one would indeed solve this task. However, in order to make the solution closer to the reinforcement learning situation, as well as to create a more interesting loss landscape, I construct a special neural network to solve this task. An illustration of this can be seen in Figure 3.2.

In this network, all weights and biases are fixed and never optimized. When the network is constructed, each neuron in the network is randomly assigned one of two scalars. These are depicted as A and B in Figure 3.2. The scalars are the parameters to be optimized. Computation of a layer in this neural network is:

$$\hat{x}_l^t = tanh((\hat{\mathbf{x}}_{l-1} \cdot \mathbf{W}_l + \mathbf{b}_l) \odot \alpha_l), \tag{3.1}$$

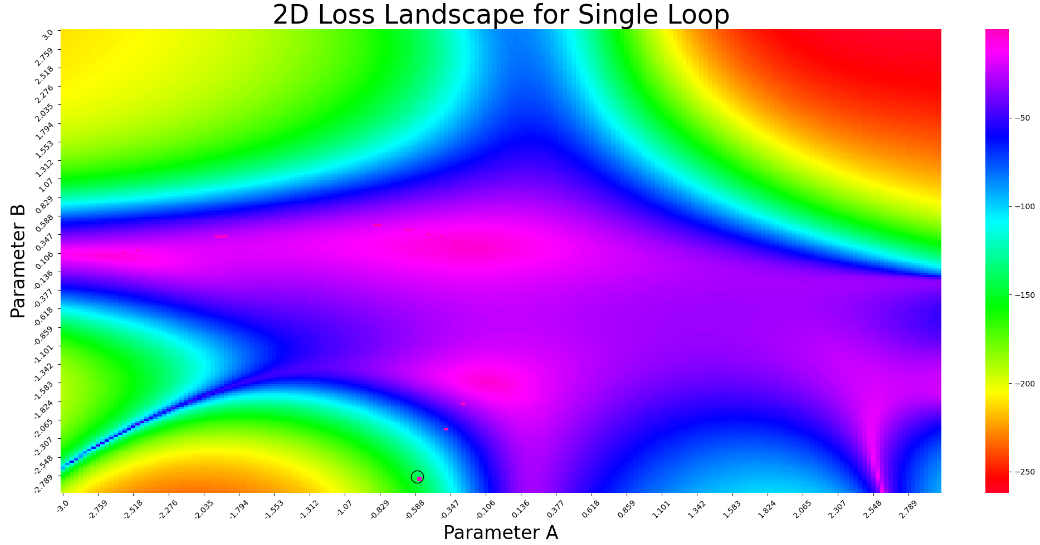


Figure 3.3: **Single-Loop Loss Landscapes.** The loss landscape contains 40,000 evaluations of the two parameters in the range (-3,3). The colors of the shortest wavelengths correspond to the highest fitness values. The black circle marks the most optimal point found through the grid search.

where α is a vector containing the scalars as they were assigned to the neurons. The same fixed network is used across all meta-learning scenarios below.

The fixed weights of the network can be seen as part of the task, in that they shape the loss landscape in a way that cannot be directly optimized. Using a fixed network as part of the task also makes it so that some goal directions are harder to reach, rather than having all tasks have the same level of difficulty. Throughout this chapter, the evolution algorithm used in each scenario is NES as described in Chapter 2, Section 2.4. For simplicity, this was chosen as both the outer- and inner-loop optimizer.

Before looking at different meta-loop scenarios, the loss landscape for a single optimization loop that solves a single task is shown in Figure 3.3. In this task, the agent needs to move to the point (0.25, 0.45).

This loss landscape was found through a grid search of both parameters in the interval [-3,3]. For each parameter, the interval was divided evenly by 200 points. These 200 points were used as the values for each parameter, resulting

in 40,000 evaluations. The shorter wavelengths of the rainbow spectrum signify a better performance. The loss landscape appears smooth, except for the most purple spots in the landscape that clearly stand out from their surroundings. This loss landscape will be used as a reference for comparison to different double-loop loss landscapes throughout the chapter.

Instead of a grid search, it is also possible to optimize the parameters with NES. This results in the training curve shown in Figure 3.5 in the next section.

3.1 Optimizing a Starting Point

3.1.1 Common Parameters - Single Task

The first meta-learning setting is characterized by two traits. First, there is still only a single task to solve, i.e., not a distribution of tasks. The single-task setting was also used in the seminal paper of Hinton and Nowlan (1987) to demonstrate the interaction between evolution and learning. Having just a single task is not a common situation for the use of meta-learning. However, interesting characteristics of different meta-learning settings can still be observed with just a single task. The single-task case is used here to enable an accurate depiction of the outer-loop loss landscape, and due to it being less computationally expensive than the multi-task setting.

Second, the inner- and outer-loop both optimize the same parameters. NES is used as the optimizer in both loops. The double-loop optimization works as follows: the outer-loop optimizer generates a population of candidates. Each of these candidates serves as a starting point from which a different optimizer searches. Specifically, since both optimizers are NES optimizers, each individual in the outer-loop population becomes the initial population means used to generate new populations around. Using a similar grid search as in the section above, it is possible to map the loss landscapes of both inner- and outer-loop optimization. There are some notable points to consider in the scenario. First, when using the same task as in the above section, the inner-loop loss landscape looks identical to the case of a single-loop optimizer. The outer-loop optimizer does not affect the inner-loop loss landscape in any way. Further, the loss landscapes of both optimizers remain static throughout the process of optimization. As a general note, it is also important to keep in mind that the hyperparameters of an optimizer do not

Neural Network:	
ANN Layers	[2, 256, 128, 2]
Activation function	tanh
Inner-Loop Optimizer:	
Pop. Size	64
Generations	10
Init. Sigma	Varying
Sigma Decay	0.99
Learning Rate	0.1
Outer-Loop Optimizer:	
Pop. Size	350
Generations	300
Init. Sigma	0.1
Sigma Decay	0.99
Learning Rate	0.01

Table 3.1: **Optimization Hyperparameters.** This is the standard setting of the two optimizers of their respective loops throughout this chapter. These hyperparameters are used in all experiments unless stated otherwise.

change the loss landscape associated with the parameters to be optimized. The optimizer's hyperparameters can be extremely important in how well the optimizer can navigate the loss landscape, but they do not modify the landscape itself. However, as seen below, any changes in the hyperparameters of an inner-loop optimizer might have a great impact on the loss landscape of the outer-loop. Since the inner-loop loss landscape is identical to the single-loop case, the following focuses on mapping outer-loop loss landscapes. For the first loss landscape, an optimizer identical to the one used for the single-loop optimization in the section above was used. The hyperparameters of the inner-loop optimizer can be seen in Table 3.1.

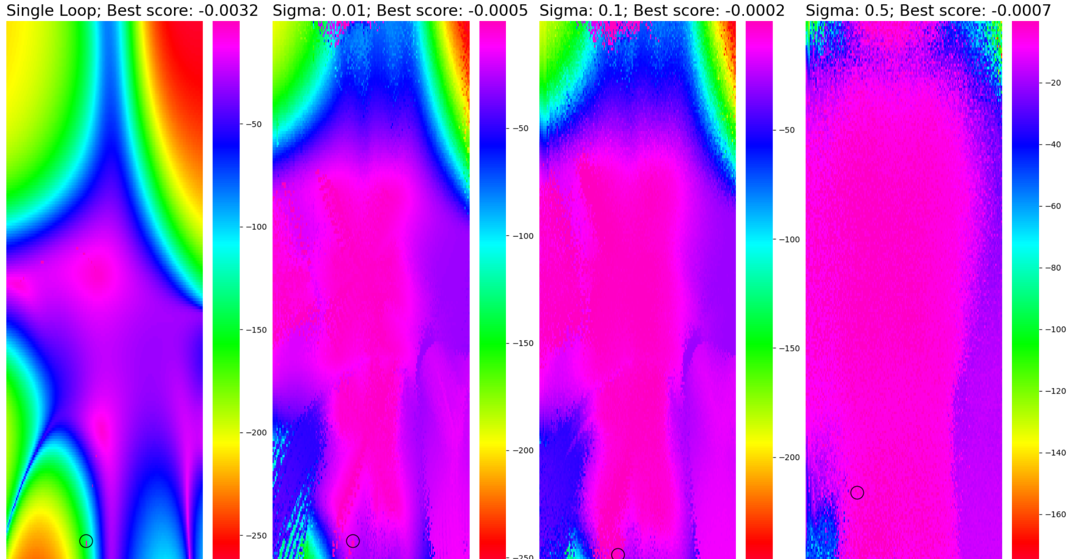


Figure 3.4: **Outer-Loop Loss Landscapes.** Loss landscapes that the outer-loop optimizer has to navigate. For comparison, the single loop loss landscape is included to the left. The outer-loop loss landscape changes as the sigma of the inner-loop NES optimizer increases. Smaller inner-loop sigmas result in outer-loop loss landscapes closer to the single-loop loss landscape. The larger sigma in the inner-loop means that the inner-loop optimizer searches through a wider range of solutions. As the inner-loop sigma increases, the areas of good performance in the outer-loop loss landscape expand and become more connected. At the same time, a larger inner-loop sigma makes the outer-loop loss landscape more noisy. Black circles indicate the optimal point of the outer-loop loss landscape as found through grid search.

As stated above, any changes to the parameters of the inner-loop optimizer have the potential to have an impact on the outer-loop loss landscape. Not because it changes anything in the inner-loop loss landscape, but because it changes how the inner-loop optimizer navigates the loss landscape associated with its parameters. This in turn changes how much it can improve upon the starting point provided by the outer-loop, and that in turn modifies what the outer-loop loss landscape looks like.

For each point in Figure 3.4, the inner-loop optimizer evaluated a pop-

ulation of 64 individuals and optimized it for 10 generations, resulting in a total of 25,600,000 evaluations in the environment to produce the landscapes of the outer-loop. Two things are noteworthy when comparing this plot with the single-loop loss landscape. First, the most well-performing areas have increased, while the areas of worst performance have decreased. This is what is meant when it is stated that learning can accelerate evolution and move the problem from finding a needle in a haystack to only having to find the area of the haystack that contains the needle (Hinton and Nowlan, 1987). Each point sampled in the outer-loop has a "reach" within the single-loop loss landscape that allows the sampled outer-loop point to end up with a better score than had its performance been recorded directly. If we conceptualize the outer-loop as evolution and the inner-loop as learning, we can say that adding the ability to learn on top of the evolved parameters makes advantageous traits more evolvable.

The second noteworthy point is that the loss landscapes look a bit less smooth. This is due to the noise introduced by the random sampling of individuals by the inner-loop optimizer. Note, that even though NES introduces noise in a specific manner, any inner-loop optimizer, including gradient descent-based RL algorithms, introduces noise in one way or another due to the use of random exploration.

The hyperparameter that most directly modifies the reach of the NES algorithm used in the inner-loop is the standard deviation of the distribution that it samples perturbations from, i.e., the sigma of the optimizer. Figure 3.4 demonstrates this well: the larger the sigma, the more pronounced the changes to the outer-loop loss landscape. The loss landscape with the largest inner-loop sigma (0.5), has the largest well-performing area, which is also fully connected. However, the most optimal score was found when using an inner-loop sigma of 0.1. This is likely because the smaller sigma allows for more tuning of the parameters in a more fine-tuned manner within the allowed number of generations.

Having a depiction of the loss landscape associated with the outer-loop, it is also interesting to examine the actual optimization of the outer-loop parameters, to see the impact of the inner-loop and different settings. Figure 3.5 shows training curves of the outer-loop for 300 generations of NES.

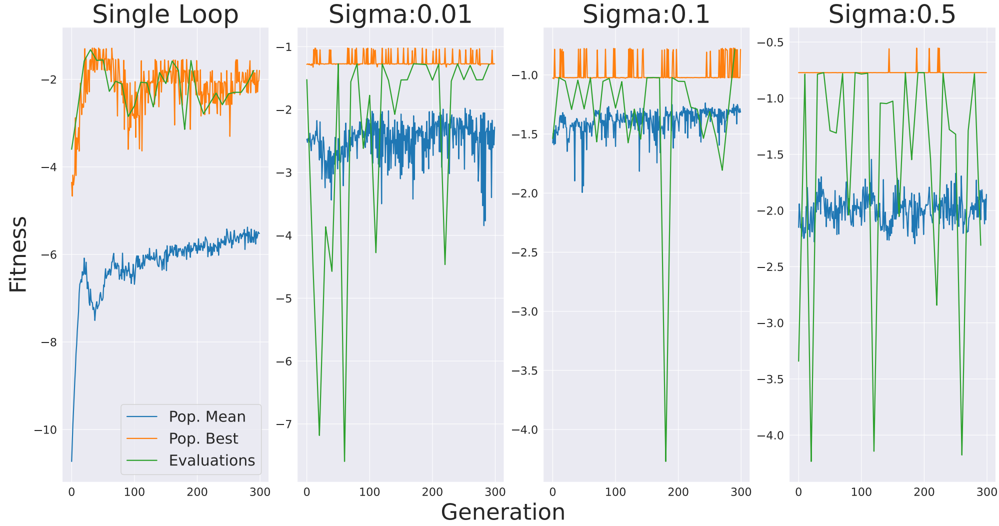


Figure 3.5: **Training curves of the outer-loop**. Different inner-loop sigmas affect outer-loop optimization. With a larger inner-loop sigma, good solutions tend to be found faster. However, the evaluations become less reliable, and the outer-loop optimization becomes increasingly difficult, as seen by the mostly flat population mean training curve in the right-most plot.

In Figure 3.5, it can be seen that the best individual in the population is already well-performing starting from the very first generation of the outer-loop optimization. It should be remembered, however, that this comes at the price of 640 times more evaluations per generation when we compare the double-loop optimization with the single-loop. Further, there does not seem to be much optimization going on, in that the curves of the outer-loop optimization runs are mostly flat. This is despite the fact that none of the runs have found solutions that are as good as the best solutions in Figure 3.4. While it is possible that better solutions would be found if given more generations to optimize, some trends are clear. First, the higher the inner-loop sigma, the better the initially found solution. At the same time, this also results in a smaller deviation from the initially found solution. A larger inner-loop sigma also results in a more flat population mean curve and more volatile evaluations.

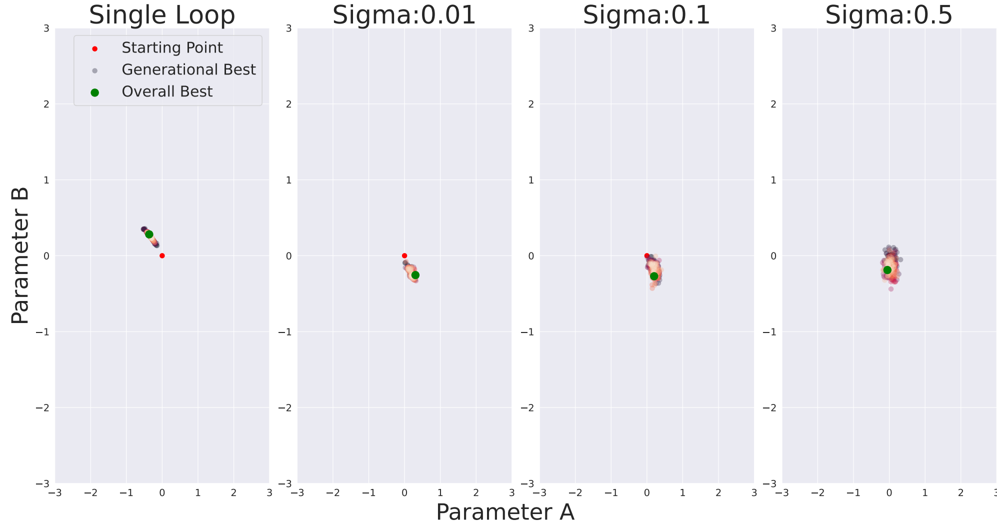


Figure 3.6: **Best outer-loop solutions in each generation.** For each generation, the best parameters are plotted as points in the graph. The brighter points were found later in the evolution run and the darker found earlier. The green point marks the best parameters found during the run. A larger inner-loop sigma results in a wider exploration of the outer-loop optimizer. However, in all runs, points are mostly concentrated in the same area, and in none of the runs did the outer-loop optimizer find solutions close to the optimal solutions found through grid search in Figure 3.4.

To shed more light on the optimization processes, Figure 3.6 shows the best solutions found in each generation in the evolution runs of Figure 3.5. The solutions do not venture for from the starting point in any of the double-loop runs. The larger the sigma, the less clear the direction away from the starting parameters of (0,0) there is. This highlights a negative effect that the inner-loop optimization risks having on the ability of the overall optimization process to find a solution. While learning (or any other conceptualization of inner-loop optimization) turns the optimal point into a stronger, and wider attractor in the outer-loop loss landscape, the same is also true for local optima. Two aspects of a larger inner-loop sigma make outer-loop optimization more difficult. First, if the inner-loop optimizer has a wider reach than the outer-loop optimizer (as is the case in this small experiment), there is a potential for all of the starting points sampled by the outer-loop to end

up in the exact same local optima. This limits the ability of the outer-loop optimizer to differentiate between any of the sampled points, and thus its ability to calculate a meaningful gradient. At the same time, a wider sampling in the inner-loop decreases the reliability of the outer-loop evaluations, simply by introducing more noise. This also makes meaningful gradients harder to obtain and increases the risk that the outer-loop parameters will never improve. The aspect of noise could be mitigated by either increasing the population size or the number of generations of the inner-loop optimizer. However, these are both parameters that in turn drastically increase the number of evaluations needed. Alternatively, the fitness score of each set of outer-loop parameters could be calculated as an average of the outcome of multiple inner-loop optimizations. Again, such a measure would also be computationally expensive.

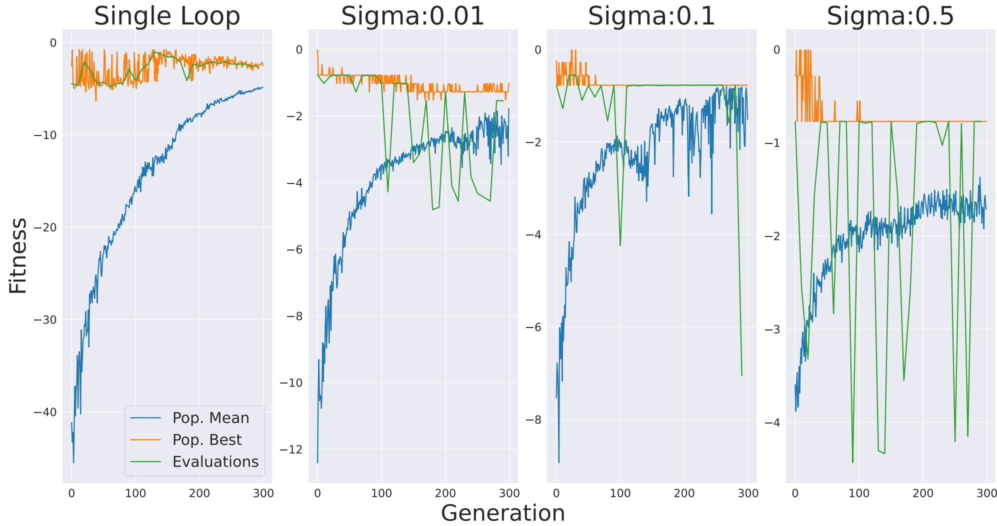


Figure 3.7: **Training curves of the outer-loop.** These training curves show the same small experiments as in Figure 3.5, except that the outer-loop sigma here was 1 instead of 0.1. Even though the hyperparameters of the outer-loop do not modify the outer-loop loss landscape, they can obviously still have a great effect on the optimization process. The same trends as in Figure 3.5 are visible here. However, here, there is a trend of early populations containing the optimal solutions.

While Figure 3.5 and 3.6 show the impact of inner-loop hyperparameters

on the optimization of the outer-loop, this optimization is obviously also affected by the hyperparameters of the outer-loop optimizer. Figure 3.7 shows training curves for the same experiment as in Figure 3.5 with the only difference being that the outer-loop optimizer had a sigma of 1 instead of 0.1. Strikingly, while the population means gradually becomes better, the best solution found in each generation decreases in performance over time. All the double-loop runs find solutions close to the global optima at the beginning of the evolution, but then seem to converge toward a less optimal point.

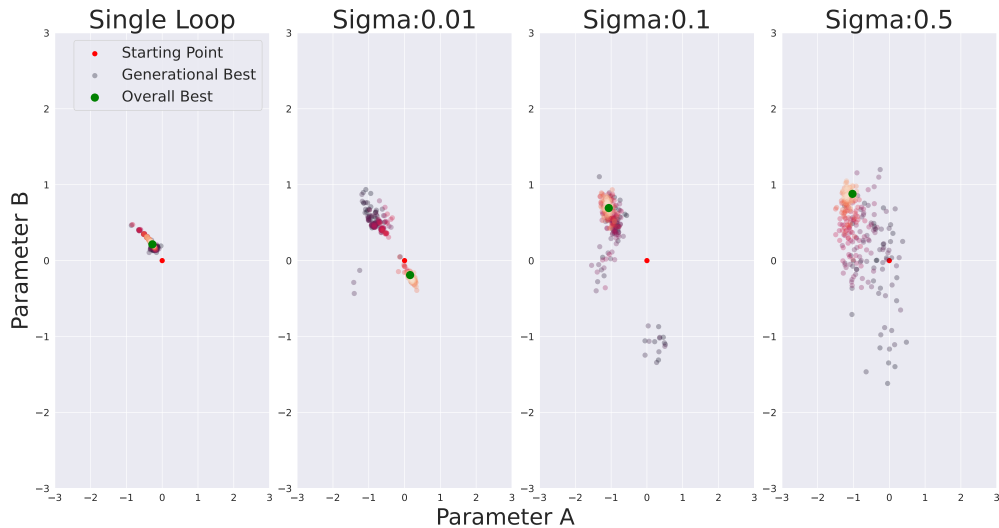


Figure 3.8: **Best outer-loop solutions in each generation.** A larger sigma in both outer- and inner-loop optimizers results in a larger area of search. Only the runs with double-loop optimization have found good solutions clustered in different areas of the loss landscape. However, the runs with larger inner-loop sigmas were unable to converge to the most optimal solution found.

Once again, we can track where the best solutions were found in each generation in Figure 3.8. First of all, we see that the double-loop optimization runs include more diverse solutions than the single-loop case. Even with the much larger standard deviation of the sampled solutions, the single-loop optimizer still mostly finds a solution within the same area as with the smaller standard deviation. For the double-loop runs, the effect of increasing the reach of the outer-loop optimizer had more pronounced effects. The

run with the largest inner-loop sigma includes solutions that are the furthest from the starting point. However, for the runs with an inner-loop sigma of 0.1 and 0.5, Figure 3.8 also shows that while the best solutions sampled in early generations were close to the optimal points of the respective loss landscapes of Figure 3.4, the later solutions were concentrated elsewhere, in a less optimal area. This is likely due to where the most optimal point in the inner-loop loss landscape is located. The black circle in Figure 3.3 indicates that the most optimal point is isolated within an area of bad solutions. This means that missing the optimal point slightly results in a very poor performance, and that it is impossible to reach the point through gradient descent unless one has a starting point that is already really close to it. In dynamical systems terms, this optimal solution acts more like a fixed point repeller than an attractor in the loss landscape.

The main points in this section that are relevant for the next sections are that parameters of the inner-loop optimizer can have large impacts on the outer-loop loss landscape, but when the outer-loop optimizes the starting point for the inner-loop optimization, the inner-loop landscape remains the same as in the single-loop case throughout the whole evolution. Further, one should keep in mind that while ”learning” (or inner-loop optimization) can accelerate the ”evolution” (outer-loop optimization), by making areas of high-performance into stronger attractors, this is true for both local and global optima. The inner-loop can thus increase the risk of the overall solution getting stuck in a local optimum that is impossible to escape for the outer-loop.

3.1.2 Common Parameters - Multiple Tasks

In the second meta-learning scenario, both loops still optimize the same parameters, but tasks are now drawn from a distribution. This is the scenario the MAML approach fits into. What does the sampling of tasks mean for the loss landscapes? For the inner-loop, there is a separate loss landscape for each task. Regardless of how good a starting point the outer-loop might provide, these separate loss landscapes remain the same from the beginning to the end of the optimization process. It is obviously true that a better starting point will make it easier for the inner-loop optimizer to find good parameters within its loss landscape, but the inner-loop loss landscape that can be mapped via grid search does not change.

The sampling of multiple tasks also affects the outer-loop loss landscape

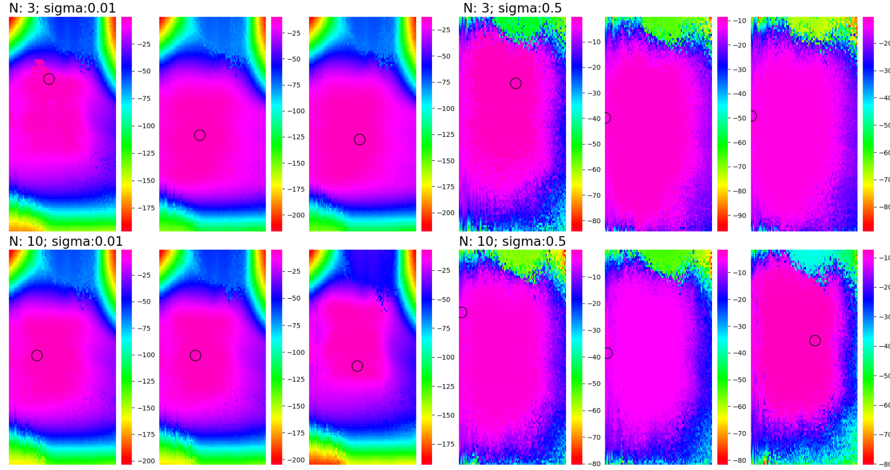


Figure 3.9: **Outer-loop loss landscapes for multiple inner-loop tasks.** Altogether, this figure contains six different task sets: three sets with three tasks, and three sets with 10 tasks. The outer-loop loss landscape is shown twice for each set, one with an inner-loop sigma of 0.01 and one with 0.5. Each set of tasks results in slightly different outer-loop loss landscapes. For each point in the loss landscape, the score was assigned as the mean of the scores found by the inner-loop optimizer in each of the tasks associated with the run. The differences between larger and smaller inner-loop sigmas are similar to the case of a single task. The areas of well-performing solutions are generally smaller in the loss landscapes with 10 tasks compared to the ones with only three tasks. However, the best solutions are of the same magnitude across the size of task sets.

in the same way as different batches do in, e.g., supervised learning. In the extreme case that only a single task is sampled per outer-loop generation, the outer-loop loss landscape changes just as rapidly as the inner-loop; every new task constitutes a new loss landscape. In the other extreme case, the limit of sampling an infinite number of tasks for each generation, the outer-loop loss landscape should remain fixed from generation to generation. A fixed loss landscape should be easier to navigate for the outer-loop optimizer, but more tasks per generation also make the optimization process more expensive.

Figure 3.9 shows the outer-loop loss landscapes for different sets of tasks and with different inner-loop sigmas. They all have a very similar overall

structure, which is expected for tasks coming from the same distributions. Some solutions will perform poorly for any goal location, for example, solutions that involve taking large steps in some direction, overshooting the goal, and continuing away from it. While the task sets all result in outer-loop landscapes that share a coarse-grained structure, the specific location of the most optimal points can vary a lot from across task sets, and can even be in opposite directions. The main difference between the setting with more tasks in the task set seems to be that the areas of high performance shrink when there are more tasks, meaning that there are fewer good solutions to be found. The optimal points are of a similar score, whether there are 10 or only three tasks. The intuitive explanation for this is that even though more tasks are sampled, there is the same probability of getting easy or hard tasks. The mean scores should thus be similar across the numbers of tasks in the set, with a higher variability in the means across task sets when the sets are small.

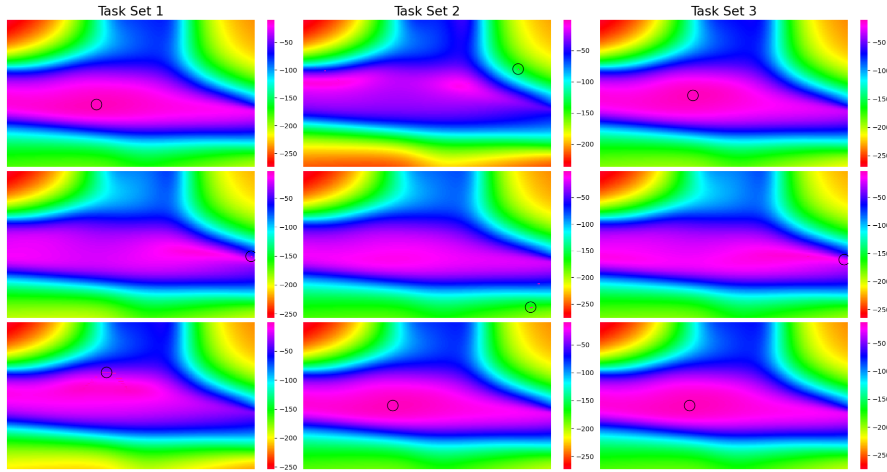


Figure 3.10: **Inner-loop loss landscapes of tasks from task sets.** The individual loss landscapes of each of the tasks that make up the task sets of three tasks in Figure 3.9. They all have a similar structure, but different optimal points.

For comparison, the inner-loop loss landscapes of the three task sets consisting of three tasks are shown in Figure 3.10. On an individual basis, the loss landscapes are also very similar, although the optimal points have a ten-

dency to be further from the origin. For some of the tasks, the optimal score within the searched area is much worse than in others. It might be the case that for some goal points, very good scores are impossible to obtain given the fixed neural network. It could also be that better optimal points could be found through a grid search of a wider area. Like the single task used throughout the previous section, some of these tasks have optimal points that are isolated within areas of poor performance.

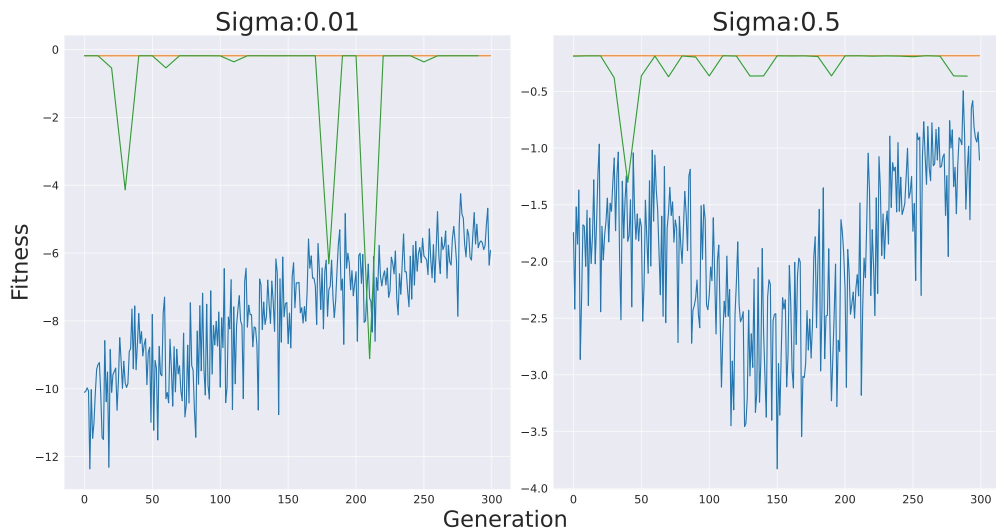


Figure 3.11: **Outer-loop training curves for multiple tasks.** Shown are the training curves for one of the task sets of three tasks from Figure 3.10. In both cases, the best solution in the population in the first generation found the optimal solution, and not much actual optimization was done.

Outer-loop optimization on one of the task sets with three tasks is shown in Figure 3.11. The outer-loop loss landscape of this task set can be seen in the first row and first column of Figure 3.9 for an inner-loop sigma of 0.01 and the first row and fourth column for an inner-loop sigma of 0.5. The outer-loop optimizer had a sigma of 1 in these runs. As can be seen, the best solutions for both runs were already found in the very first generation of evolution. It is worth noting that none of the outer-loop loss landscapes of Figure 3.9 have optimal points that are isolated from the main area of high-performing solutions.

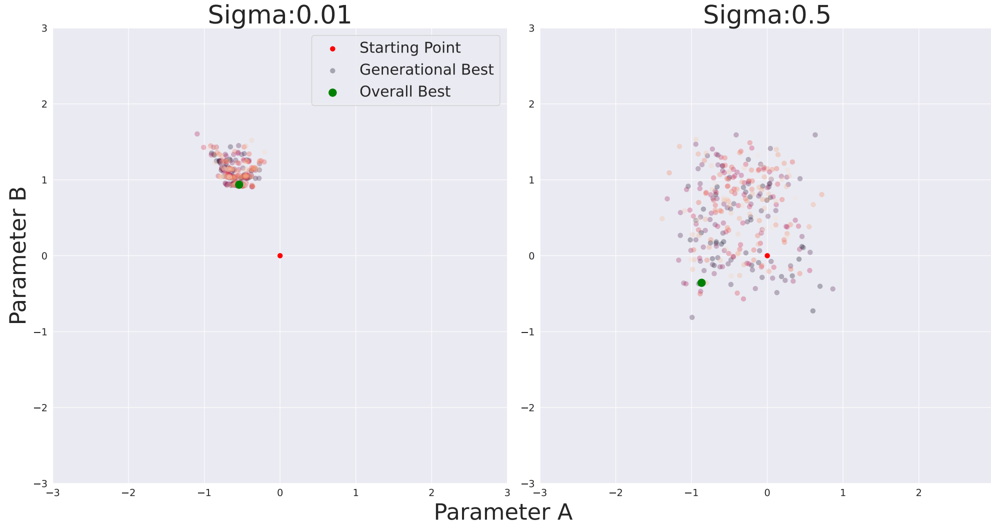


Figure 3.12: **Best outer-loop solutions in each generation.** The solutions reflect that the initial best individual in the population remained the best throughout. The run with a larger inner-loop sigma did not converge to any specific point, because an optimal solution could be reached by the inner-loop optimizer from a large area of starting points.

Looking at the best solutions of each generation in Figure 3.8, it is clear that in the run with a small inner-loop sigma, all of the best outer-loop solutions are concentrated close to the optimal point of the loss landscape in Figure 3.9. This is true for all generations, and while these plots do not tell us how easy it was for the inner-loop optimization to reach any of the optimal points in the individual tasks, it is reasonable to assume that the starting point with the best mean score is a very strong attractor, since it could be found so early in the outer-loop optimization. The best points of the run with a larger inner-loop sigma are more diffuse throughout, with the most recent solutions being concentrated closer to the optimal point of the loss landscape in Figure 3.9. However, since the scores of the best solutions are virtually identical for all these points (as seen in 3.11), it means that the whole area is close to being optimal. This is likely because there are multiple ways of getting a high score on average. Different starting points will be closer to the optimal points of different inner-loop loss landscapes, but yield close to the same average result.

In terms of loss landscape dynamics, the small experiments in this section demonstrated the same situation as the MAML approach (Finn et al., 2017): a starting point for inner-loop optimization is optimized in an outer-loop across multiple different tasks. The optimization runs showed that the optimal points of outer-loop loss landscapes can actually be easier to find when the loss landscape is calculated over multiple different tasks in the inner-loop.

3.2 Optimizing Loss Landscapes

3.2.1 Different Parameters - Single Task

In the third scenario, the outer-loop and inner-loop no longer optimize the same parameters, but only a single task is used, like in Scenario 1. This means that altogether four scalars are optimized in the fixed neural network instead of only two. The same fixed neural network weights and biases as in the previous scenarios are used in this section. However, each neuron is instead assigned one of the four different scalars at random to be multiplied by. The outer-loop optimizes two of these, and the inner-loop optimizes the other two. The two output neurons are assigned the two inner-loop parameters, respectively. The outer-loop no longer changes the starting point of the inner-loop optimization, which means that the inner-loop starting point is always (0,0).

The optimization of different parameters has a profound impact on the dynamics of the loss landscapes. In particular, the loss landscape of the inner-loop no longer remains the same throughout the evolution but changes every time the outer-loop parameters change. From the point of view of the inner-loop, the outer-loop parameters create a new, fixed network that the inner-loop parameters have to adapt to.

The hyperparameters of an optimizer generally do not affect the loss landscape that the optimizer navigates in. However, in this scenario, things are a bit more complicated. As we have seen in multiple different settings above, hyperparameters of the inner-loop optimizer impact the loss landscape of the outer-loop. Since the outer-loop parameters now affect the inner-loop loss landscape, a feedback mechanism from the hyperparameters of the inner-loop optimizer, through the outer-loop loss landscape, and back to the inner-loop loss landscape can now occur during optimization.

Figure 3.13 shows an example of an inner-loop loss landscape that changes

with the parameters of the outer-loop. The same goal location of (0.25, 0.45) as in Figure 3.3 was used for this demonstration. The loss landscapes differ from Figure 3.3 since there are now more different parameters in different random locations.

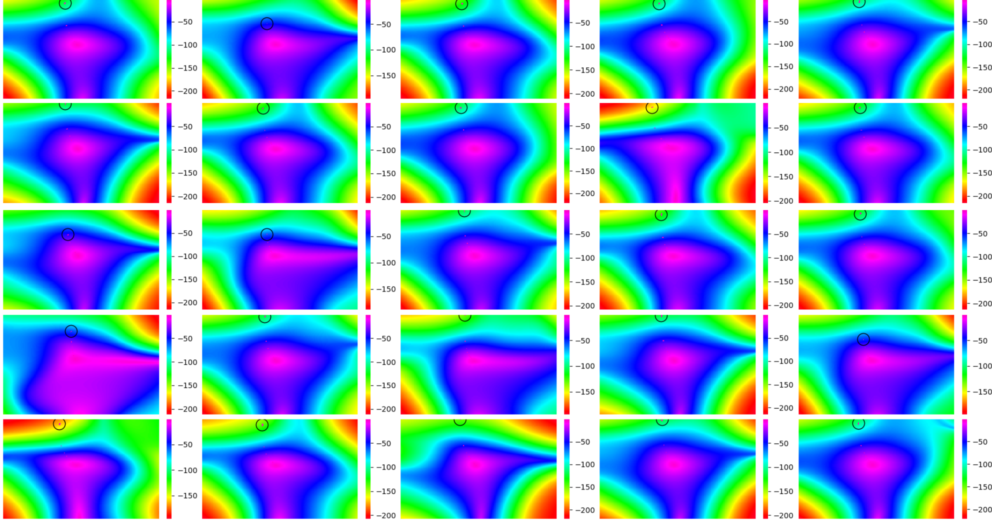


Figure 3.13: **Inner-loop loss landscape changing with outer-loop optimization.** When the outer- and inner-loop optimizers optimize different parameters that are all part of the solution, the outer-loop parameters modify the inner-loop loss landscape. Shown here are grid searches on the inner-loop parameters at different points in the evolution run with an inner-loop sigma of 0.01 from Figure 3.14. The top left loss landscape was recorded at the first generation of the evolution, and the subsequent recordings can be seen following the reading direction. Each of the loss landscapes are slightly different even though evaluation is always on the same task. However, as seen in Figure 3.15, most solutions found by the outer-loop optimizer were very similar, and the changes to the inner-loop loss landscapes are thus subtle. The optimal points in the loss landscapes remain in the same area throughout evolution.

The changes to the loss landscapes in Figure 3.13 are mostly minor. These small changes make sense in the context of the outer-loop optimization process, from which they were recorded, and which are visualized in Figures 3.14 and 3.15. Here, the optimization curves are largely flat, and the best points

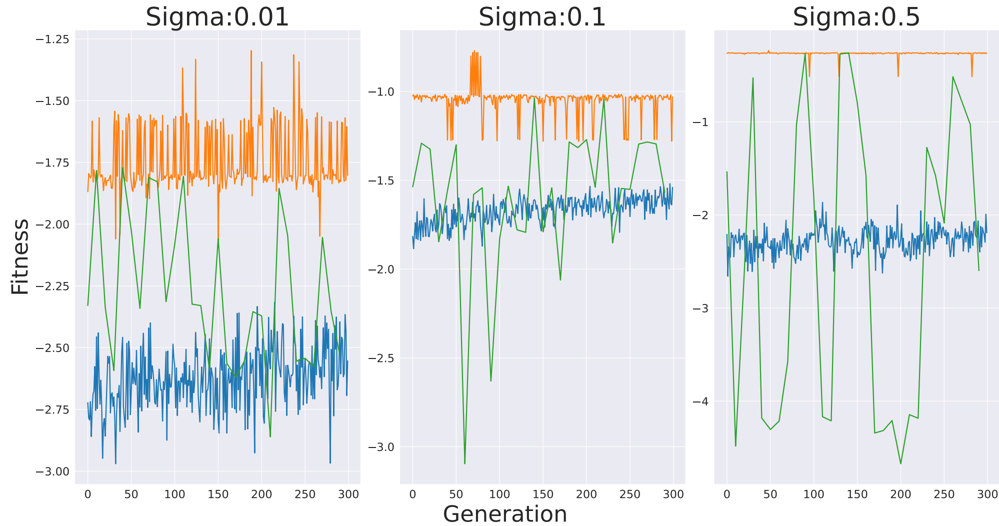


Figure 3.14: **Training curves of the outer-loop.** These runs only show minor improvements in the population mean over time. The run with an inner-loop sigma of 0.5 finds the best solution out of the three runs, but evaluations are highly volatile.

in each generation do not seem to converge toward a specific point, which means that the inner-loop loss landscape is also unlikely to change in a specific direction. There are some notable differences between these outer-loop loss landscapes and those of Figure 3.4. First, the overall structure of the loss landscapes changes less with an increasing inner-loop sigma. Second, the landscapes are all more noisy than their counterparts in Figure 3.4. Lastly, the optimal points are orders of magnitude worse than when only two parameters were optimized. Granted, a one-to-one comparison to Figure 3.4 cannot be made, since the scalars are not placed at the same locations in the fixed neural network. The differences in performance are, however, pretty striking.

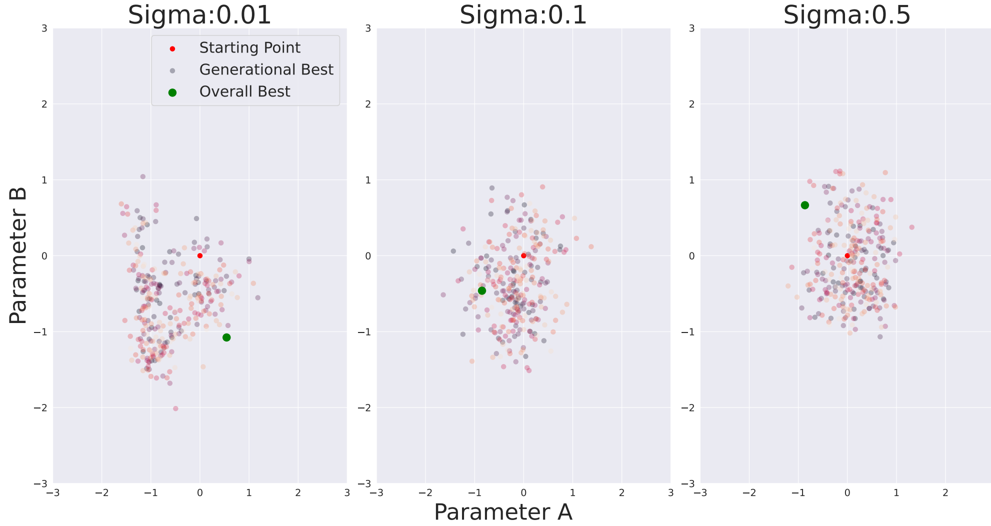


Figure 3.15: **Best outer-loop solutions in each generation.** Lighter points signify solutions found later in the evolution. The plotted solutions over time show that there is no strong trend of converging toward a specific solution in any of the three runs. This is especially true for the run with the largest inner-loop sigma of 0.5.

An explanation for the relatively poor performance with double the optimized parameters can come from the fact that each set of parameters in this double-loop setting has a smaller reach than the two parameters that were optimized in both loops in Figure 3.5. Since the inner-loop optimization now always has the same starting point, it relies on a high-performing point being within its reach during the relatively short inner-loop optimization. However, as seen in Figure 3.13, the optimal points tend to be located far from the origin and surrounded by low-performing solutions. This makes it extremely difficult for the inner-loop optimizer to make progress toward the optimal point. With this in mind, it makes sense that the best solution was found in the loss landscape with the largest inner-loop sigma.

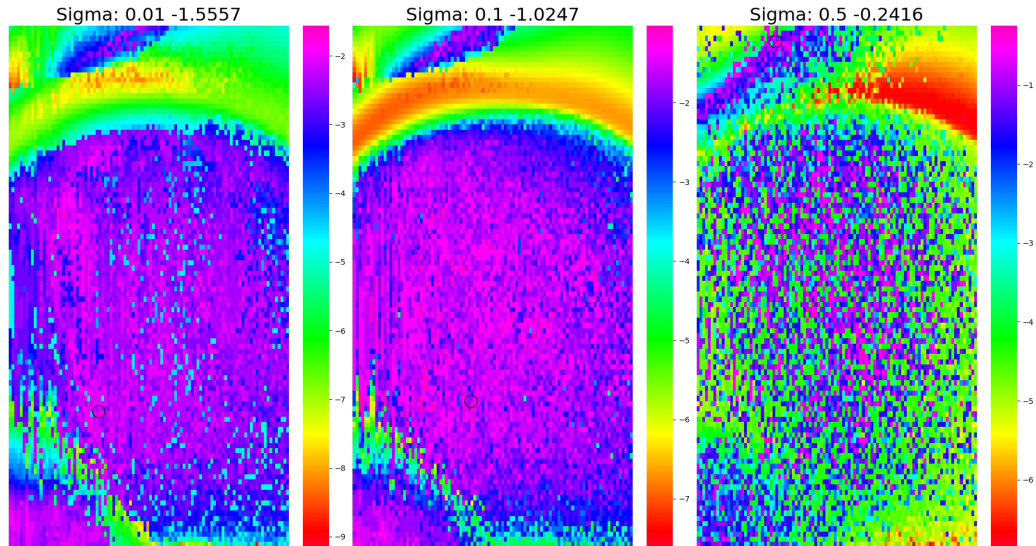


Figure 3.16: **Outer-Loop Loss Landscapes.** The loss landscapes of the outer-loop when outer- and inner-loop parameters are different, and only a single inner-loop task is sampled. The loss landscapes are much noisier than in Figure 3.4, even though the task is the same. When the outer-loop does not influence the starting point of the inner-loop parameters, finding a good solution depends more on the initial random exploration in the inner-loop, which makes the results more volatile. A noisier loss landscape is harder for the outer-loop optimizer to navigate.

It is further evident from Figure 3.13, that even though the outer-loop parameters have the potential to modify the inner-loop loss landscape, the inner-loop loss landscape does not seem to become easier to navigate over time. An easily testable hypothesis could be that the outer-loop parameters are not expressive enough to make a meaningful impact on the inner-loop loss landscape.

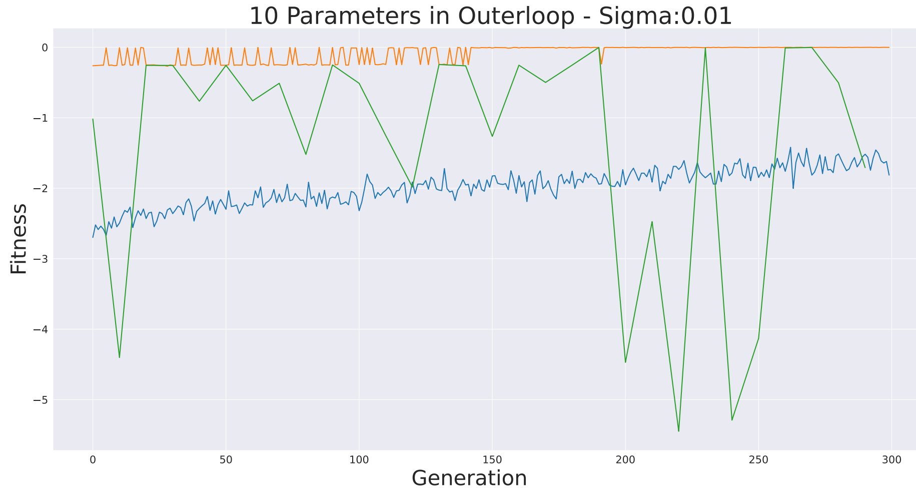


Figure 3.17: **Training curve of an outer-loop with more parameters.** The run shows minor improvements in the population mean over time.

Figure 3.17 shows an optimization run where 10 parameters are optimized in the outer-loop instead of just two. Unfortunately, the extra parameters make it impossible to visualize an accurate depiction of the outer-loop loss landscape. However, this run finds a much better solution than the runs of Figure 3.14. Further, the best solution in the generation tends to always be close to perfect. With a more expressive set of outer-loop parameters, the changes to the inner-loop loss landscape also become more directed from the beginning of the optimization process towards the end, as can be seen in Figure 3.18. Importantly, the optimal point of the loss landscape is gradually moved from being surrounded by very low-performance solutions to becoming embedded within less poor solutions, and closer to the starting point of the inner-loop optimizer.

This section showed major differences in the loss landscape dynamics compared to the previous two sections. The main role of the outer-loop is no longer to find a good starting point that the inner-loop can navigate from, but the change the inner-loop landscape itself to be easier to navigate.

3.2.2 Different Parameters - Multiple Tasks

In this section, how the same loss landscape dynamics as in the previous section affect multiple inner-loop loss landscapes at once is examined. In Figure

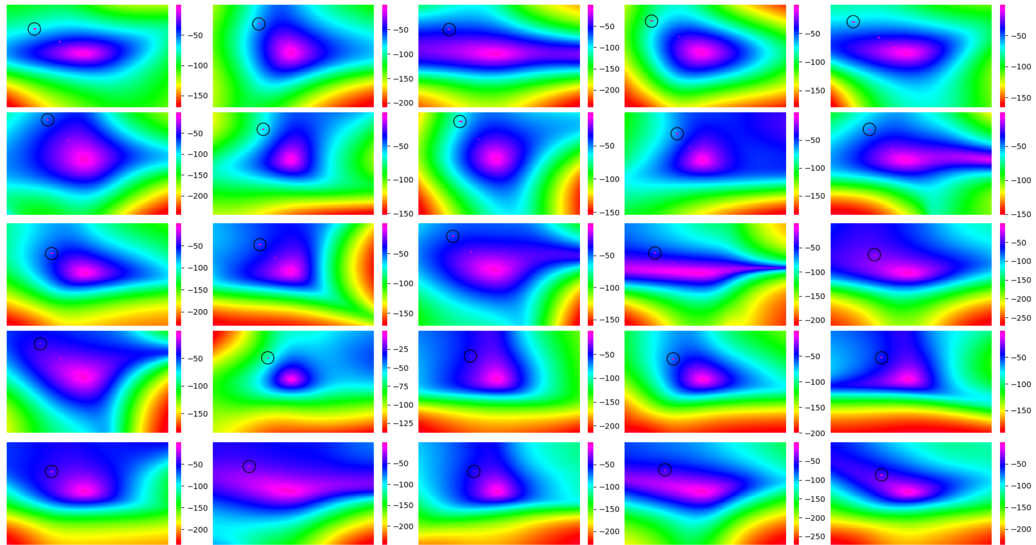


Figure 3.18: **Inner-loop loss landscape changing with outer-loop optimization.** Shown are the same loss landscapes as in Figure 3.13, but with more parameters optimized in the outer-loop. Here, the inner-loop loss landscapes change more throughout the evolution. The most important change that occurs throughout the evolution of the outer-loop parameters, is that the optimal point in the inner-loop loss landscape moves closer to the starting point of the inner-loop parameters (the origin) and that it is moved from areas of very low performance towards areas with higher performance. This means that the outer-loop parameters successfully have created a loss landscape for the inner-loop optimizer that is easier to navigate.

3.19, three outer-loop loss landscapes are shown for three different task sets. The two look very similar and also have similar optimal points. However, the left-most loss landscape in the figure looks qualitatively different from the others and has a much better optimal point. This is a difference compared to what we saw in Figure 3.9 with the same sets of tasks. In Figure 3.9 the optimal points differ across the task sets, but the overall structure remains similar. A possible explanation could be that with the outer-loop parameters' ability to alter the inner-loop loss landscapes comes the potential for higher sensitivity to different sets of tasks. With small sets of tasks, it could be the case that some sets of inner-loop loss landscapes are easier to modify in a synergistic direction than other sets. However, this hypothesis would have to be tested with many more experiments, which is outside the scope of this chapter.

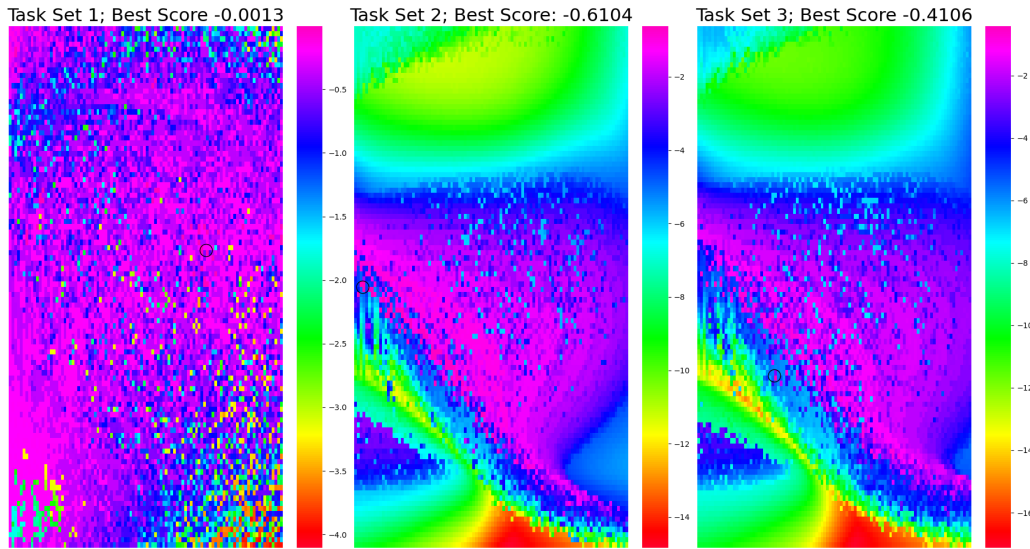


Figure 3.19: **Outer-loop loss landscapes for different task sets.** Three different task sets consisting of three tasks each. The inner-loop sigma was 0.5 in each case. The loss landscape to the left, which has by far the best scoring optimal point, looks quite different from the other two loss landscapes are similar in both structure and score of their optimal points.

Figure 3.20 shows the outer-loop loss landscapes for three different sets of 10 tasks. As expected, the loss landscapes are more similar to each other than

the loss landscapes in Figure 3.19. However, once again, the loss landscapes differ more from each other compared to the same task sets in Figure 3.9, supporting the above hypothesis. The magnitudes of the best points are similar to the loss landscapes of only three tasks.

Running optimization of the outer-loop for 300 generations using one of the task sets of three tasks results in the training curves found in Figure 3.21. In the previous section, it seems very difficult for the outer-loop optimization process to find better solutions than what was found in the initial generations. Only the run with an inner-loop sigma of 0.01 sees a small improvement in its best solutions over time.

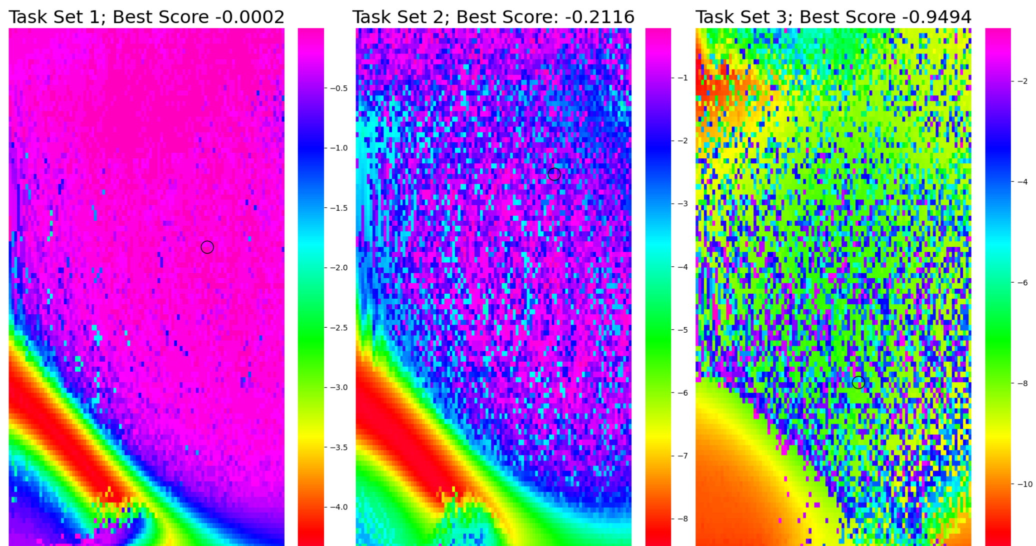


Figure 3.20: **Outer-loop loss landscapes for task sets with 10 tasks.** Shown are loss landscapes similar to those in Figure 3.19, but with each task set consisting of 10 different tasks instead of 3. Structurally, the landscapes are similar, however, the loss landscape to the left has a much better scoring optimal points than the other two. Note, that the tasks here are chosen independently from the tasks in Figure 3.19 and it is by coincidence that it is the loss landscape to the left that is scoring best in both cases.

We can record the inner-loop landscapes of each of the three tasks before and after the 300 generations of outer-loop optimization. This is done in Figures 3.22 and 3.23. From these figures, it is clear that there is the most

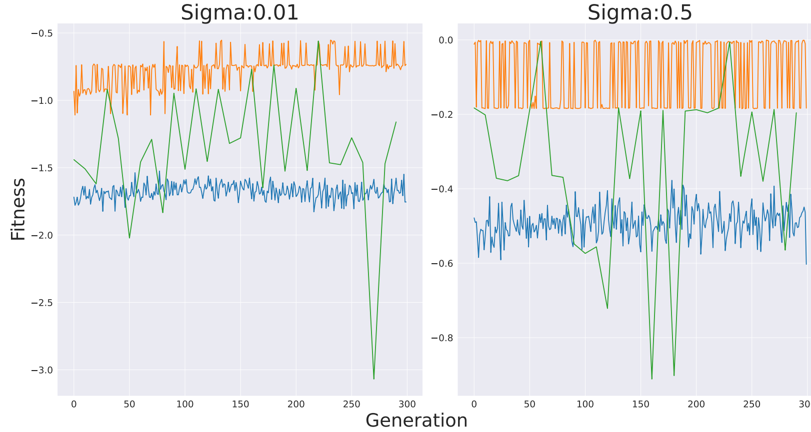


Figure 3.21: **Training curves for outer-loop optimization with three inner-loop tasks.** As was also the case in 3.14, the curve of the population mean remains flat throughout all generations.

change occurring to the inner-loop loss landscapes in the run with an inner-loop sigma of 0.01. There is a trend of optimal points in the loss landscapes becoming better after the outer-loop optimization, but this is not always the case. It is also not clear from visual inspection of the loss landscapes whether the optimal points are actually easier to reach after optimization in either of the two optimization runs.

The lack of clear improvements of the loss landscapes after outer-loop optimization makes sense when considering Figures 3.21 and 3.24. There does not seem to be any clear direction of the outer-loop optimization process, except for the switch in the clusters of best solutions occurring in the run with an inner-loop sigma of 0.01.

As in the previous section, we can try to improve the outer-loop optimization by adding more parameters for the outer-loop to optimize. This once again prevents the mapping of the outer-loop loss landscapes in 2D, but it provides the training curves found in Figure 3.25, as well as the changing inner-loop loss landscapes in Figures 3.26 and 3.27. In the plots of the changing loss landscapes, some of the same trends are present with fewer parameters in the outer-loop. The loss landscapes after outer-loop optimization tend to have better optimal points, than before optimization, but it is not always the case.

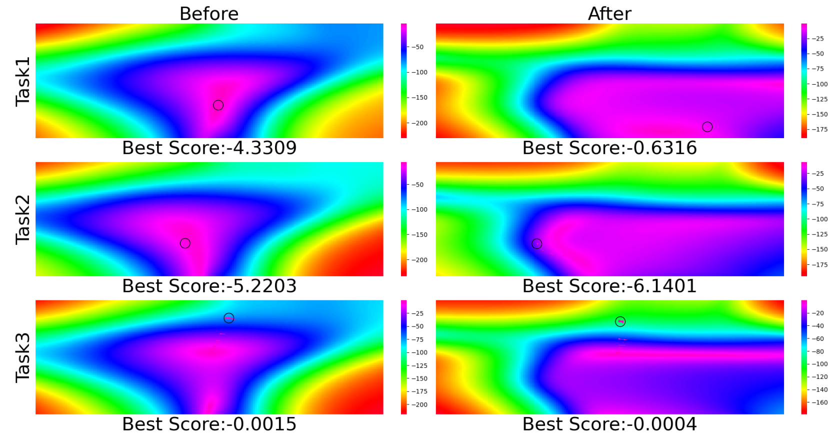


Figure 3.22: **Changing inner-loss landscapes with inner-loop sigma 0.01.** This figure shows the inner-loop loss landscapes of the three different tasks within the task set before and after the outer-loop parameters have been optimized for 300 generations. For tasks 1 and 3, the scores of the optimal points in the loss landscapes are better after optimization, whereas the score of the optimal point for task 2 is slightly worse.

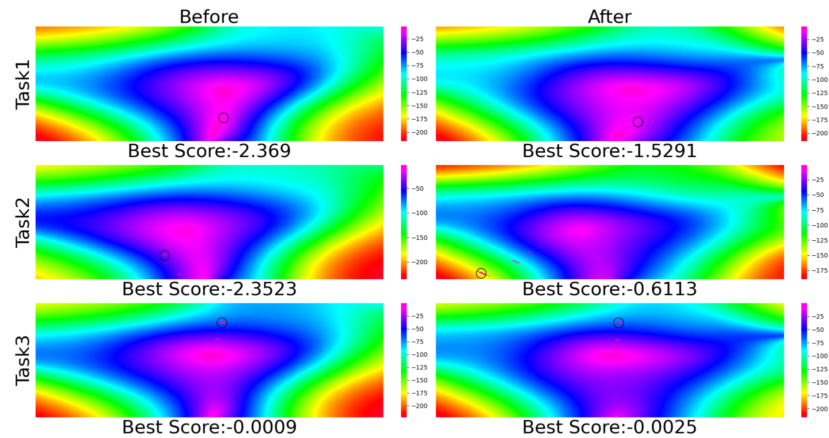


Figure 3.23: **Changing inner-loss landscapes with inner-loop sigma 0.5.** In this setting, the loss landscapes change less than in Figure 3.22. The optimal points in tasks 1 and 3 are similar before and after. Task 2 has a better optimal point in the loss landscape after outer-loop optimization.

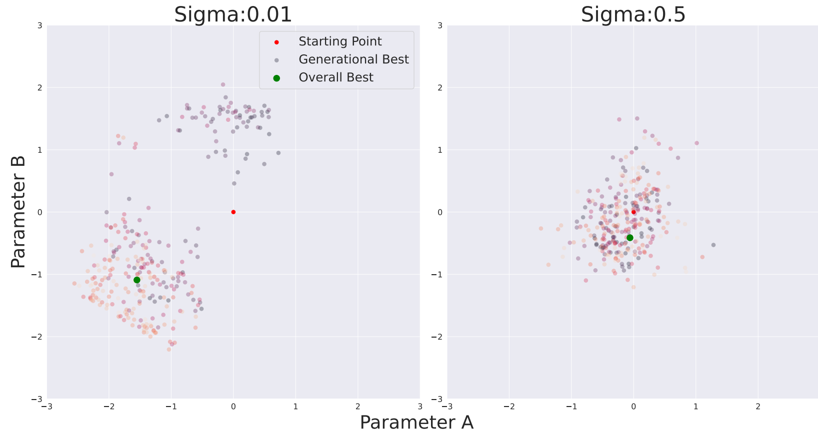


Figure 3.24: **Best solutions found in each generation.** For the inner-loop sigma of 0.5, the best solutions found are mostly diffusely placed in the parameter space. The run with a sigma of 0.01 has a change in where the best solutions are clustered at the beginning of evolution to where most of the best solutions are found later in evolution.

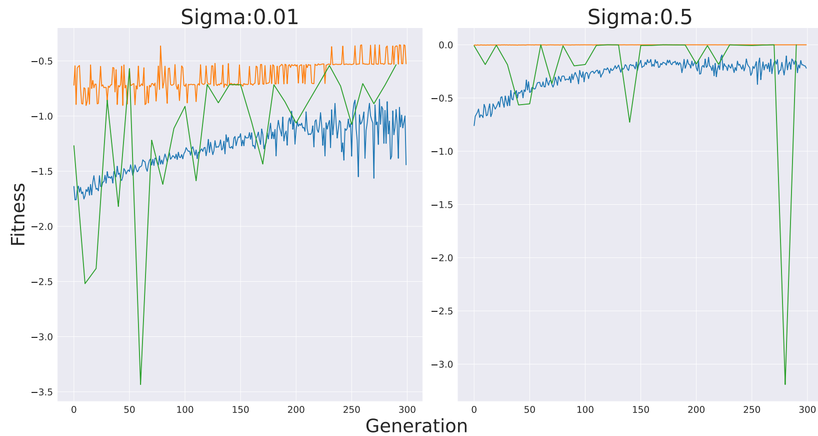


Figure 3.25: **Training curves for an outer-loop with 10 parameters.** When adding trainable parameters to the outer-loop, the optimization results are slightly better than in Figure 3.21 within the 300 generations.

We can observe some differences compared to Figures 3.22 and 3.23. The loss landscapes obviously already before optimization different when

the outer-loop optimizes more parameters. However, in Figures 3.26 and 3.27 with more outer-loop parameters, the inner-loops of the different runs are already more different from each other than the initial inner-loop loss landscapes when only two parameters were optimized in the outer-loop.

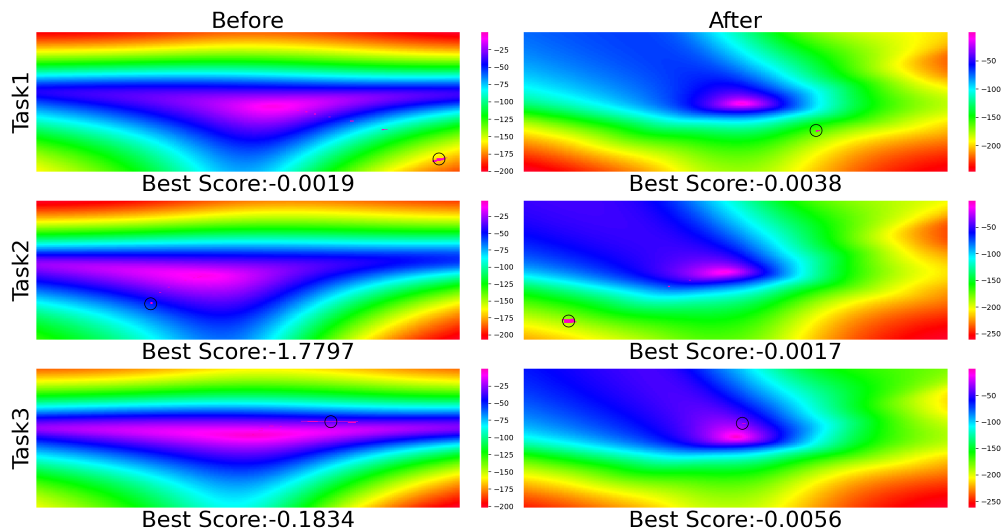


Figure 3.26: **Changing inner-loss landscapes with inner-loop sigma 0.01 and 10 outer-loop parameters.** The top solutions in the loss landscapes are improved in tasks 2 and 3. After optimization, the magnitudes of the top solutions are similar for each task.

Another difference when the outer-loop optimized more parameters can be seen in the loss landscapes after optimization. In the runs with 10 outer-loop parameters, the area of well-performing solutions in the inner-loop loss landscapes appears to be smaller after optimization. This is especially true for the run with an inner-loop sigma of 0.01.

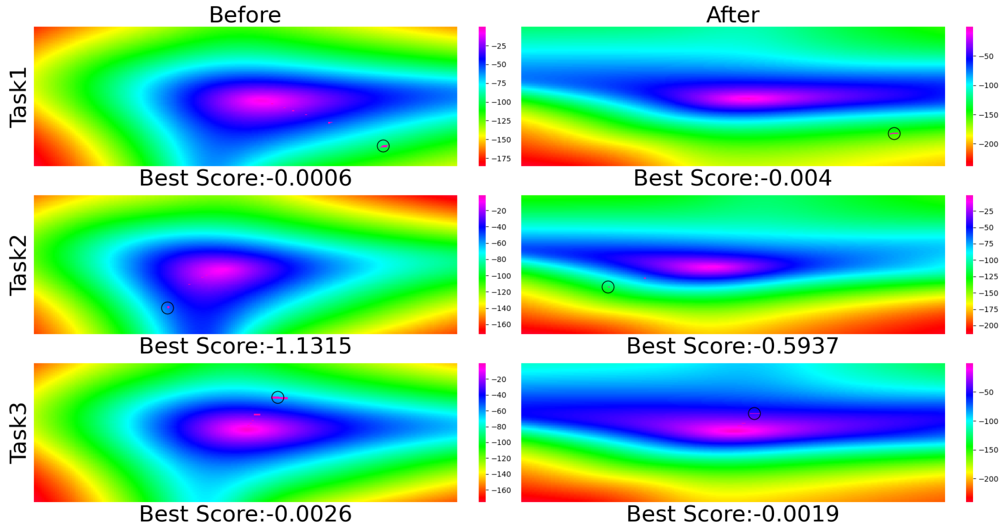


Figure 3.27: **Changing inner-loss landscapes with inner-loop sigma 0.5 and 10 outer-loop parameters.** Only the optimal point of the loss landscape of task 2 changed materially.

To emphasize the difference more outer-loop parameters make to the changes in the inner-loop throughout optimization, the same experiment can be repeated with even more outer-loop parameters to be optimized. Figures with the results of using 100 parameters in the outer-loop can be found in Appendix A With 100 trainable parameters in the outer-loop, the outer-loop training curves look similar to with 10 parameters. The same trends with the inner-loop loss landscapes also hold: most landscapes after outer-loop optimization have better optimal points, but this is not true for all of them.

Building on the previous section, this section showed that the outer-loop parameters have the ability to modify the loss landscapes of the inner-loop parameters in all the tasks that are involved in calculating the performance of the outer-loop parameters. The small experiments showed that after optimization most inner-loop loss landscapes had improved optimal points, but that this was not true for all inner-loop loss landscapes. That some inner-loop loss landscapes actually had worse optimal points after outer-loop optimization is likely the consequence of a compromise that the outer-loop parameters have to make to gain the highest possible average score across the different tasks. Since the outer-loop parameters do not have the ability to change the

starting point of the inner-loop optimization, nor do they have the ability to influence how the optimizer searches in terms of depth or breadth, the only option is to modify the loss landscape of the inner-loop parameters such that good solutions can be found consistently. With more outer-loop parameters, there is a greater potential to modify the inner-loop loss landscape, but it is likely unreasonable to assume that all optimal solutions across all tasks could be moved into the same place. After all, the optimal solutions are constrained by the objective realities of the task environment. For this reason, we might see sacrifices made to what scores are possible to reach in individual tasks in order to secure a consistently high average return.

An interesting dynamic between the outer- and inner-loop optimization is worth emphasizing here: changes to the inner-loop reach by changing, e.g., the inner-loop sigma has consequences for the outer-loop loss landscape, as we have seen in all experiments in this chapter. Since the outer-loop parameters in this meta-learning setting affect the inner-loop loss landscape, this means that changing the hyperparameters of the inner-loop optimizer has downstream effects on its own loss landscape. This is seen most clearly by comparing Figures A2 and A3 in the appendix. With a smaller inner-loop sigma, the inner-loop loss landscape changes more than with a larger one. Since an inner-loop sigma of 0.01 means that the inner-loop optimizer focuses its search in a smaller area, it makes sense that the outer-loop parameters need to adapt the space more to accommodate the narrower search compared to when the inner-loop optimizer has a broader reach. We can thus say that for this particular meta-learning setting, a larger inner-loop sigma has a greater effect on the outer-loop loss landscape, but a smaller inner-loop sigma results in greater changes to the inner-loop loss landscapes throughout the outer-loop optimization process.

Some interesting additional studies could be made with regard to the number of parameters it takes in the outer-loop to significantly modify the inner-loop loss landscape. Will the inner-loop loss landscape change more or less over time, if we add more inner-loop parameters relative to the outer-loop parameters? Or is the most important thing that the outer-loop actually has a chance of improving and does not have entirely flat training curves?

3.3 Optimizing an Optimizer

Black-box meta-learning is sometimes framed as learning a function that optimizes another set of parameters. Arguably the simplest and most used version of this type of meta-learning is to optimize the hyperparameters of a function that already has the characteristics of an optimizer. Figure 3.28 shows the outer-loop loss landscape where parameters optimized in the outer-loop are the learning rate and the sigma of the inner-loop optimizer. The inner-loop is the same as for 3.4 with the same fixed network and the same task, and the inner-loop loss landscape is thus the same as in Figure 3.3. Again, changing the hyperparameters of the inner-loop optimizer does not change the loss landscape of the inner-loop, only its ability to navigate the landscape. It is also important to keep in mind that even though changing the inner-loop sigma resulted in changes to the outer-loop loss landscapes in all of the examples in the previous sections of this chapter, it does not mean that the outer-loop loss landscape changes throughout the outer-loop optimization process in this case with the varying inner-loop sigma and learning rate. This is because the outer-loop loss landscape in Figure 3.28 directly shows the resulting scores of using different sets of sigma- and learning rate-values themselves, and not their effects on some other optimized parameters.

Thus, even though the parameters in the outer- and inner-loop are different from each other, the inner-loop loss landscape is not dynamic throughout the optimization process. This is because the outer-loop parameters are not part of the model that is ultimately evaluated, and the inner-loop parameters that are part of this model do not have to adapt to the outer-loop parameters. This case thus differs from the meta-learning settings presented in the sections above in that the parameter sets of the two loops are not the same, no starting point is learned for the inner-loop optimization, and the inner-loop landscape is not optimized for easier navigation. The way that the outer-loop improves the inner-loop optimization is by adapting the reach of the optimizer. The small experiment of Figure 3.28 is just a very simple example of increasing the reach through adapting the learning rate and sigma, which are obviously highly associated with the reach of the NES optimizer. However, it is easy to imagine much more intricate ways of evolving an optimizer to change the way it navigates the loss landscape associated with its parameters.

More advanced examples of using an outer-loop to optimize an inner-loop optimizer can be found in the field of plastic artificial neural networks. An

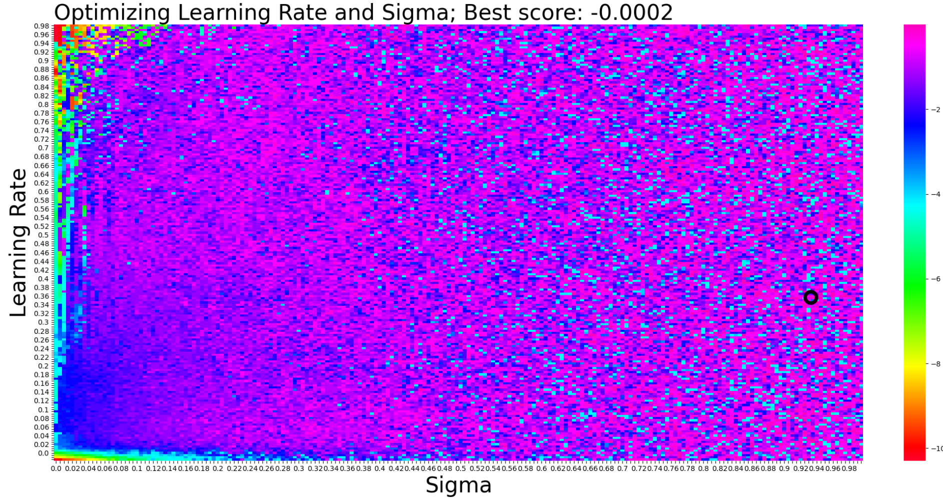


Figure 3.28: **Optimizing hyperparameters of an inner-loop optimizer.** Outer-loop loss landscape when the parameters optimized in the outer-loop are the learning rate and sigma of the inner-loop optimizer. The inner-loop loss landscape is the same as in Figure 3.3.

introduction to this field can be found in Section 2.5.3 of Chapter 2. Instead of optimizing the solution directly, several studies show that it is possible to optimize rules for how the solution should change over time. Thus, with such approaches, the parameters in the outer-loop do not participate directly in the solution but determine which solutions it is possible to reach. The same logic can be applied to any methods that involve meta-learning of a hypernetwork or an indirect encoding of a solution. As long as the role of the outer-loop parameters is to help find a different model, the particular approach belongs to the same meta-learning category as meta-learning hyperparameters of a solution. Meta-learning approaches within this category can, obviously, vary widely in how much the outer-parameters need to learn from scratch, where learning hyperparameters of an already established optimization function is at one extreme end of the spectrum.

With regards to plastic neural networks, it is, unfortunately, generally not possible to visualize the inner-loop loss landscapes in the same way as has been done throughout this chapter. This is because most plastic neural network approaches involve updating the weights of the network at every time step within the environment that the plastic neural network operates

in. Even if only two parameters were updated by the plasticity rules, we are dealing with optimal sequences of parameters instead of simply optimal parameters. Whether inner-loop parameters are updated continually with each time step in the environment makes a big difference in what kind of solutions can be found. It certainly creates challenges in visualizing the solutions of an episode as a whole. On the other hand, the frequency of updates can be seen as a hyperparameter of the inner-loop optimizer. Thus, from the perspective of meta-learning categories, updating inner-loop parameters more or less frequently should not change which meta-learning category a method belongs to.

3.4 Putting it All Together

Throughout this chapter, it has been demonstrated how the outer-loop can improve the inner-loop optimization in three different ways: finding a better starting point as seen in Section 3.1; shaping the inner-loop loss landscapes as seen in Section 3.2; and through optimizing the inner-loop optimizer itself as briefly seen and discussed in Section 3.3. Optimizing solutions of only two parameters allows for visualizing the results of a grid search that uncovers the loss landscape of the task at hand. This helps in explaining concepts and strengthening intuitions. However, most interesting problems often require a large number of parameters to solve. One should always be careful when extrapolating intuitions from low-dimensional spaces to dimensions with many dimensions. It has for example been shown, that local optima points are much more common in low-dimensional loss landscapes and that in higher dimensions, saddle-points are more frequent (Lipton, 2016). That being said, there is no reason to believe that the three mechanisms of the outer-loop to affect the inner-loop, described throughout this chapter, are not the same for meta-learning of large neural networks.

The three different mechanisms have so far only been shown and described separately from each other. However, it is certainly possible for meta-learning approaches to make use of more than one of these at a time. For example, when optimizing an LSTM to implement an RL algorithm in its dynamics (as described in Chapter 2, Section 2.5), the outer-loop optimizes the gates of the LSTM, which in turn optimizes the hidden state of the LSTM during interactions with the environment. Thus, the LSTM gates act as an optimizer (Wang et al., 2017; Duan et al., 2016). However, the gaits of the LSTM are

also part of the direct solution, as the hidden state has to interact with an output gate to produce the output of model.

As another example, the model of Beaulieu et al. (2020) has an outer-loop that optimizes both a starting point of a model, as well as a function that is used to modulate the inner-loop updates of the model. It could even be argued that the activity-independent term in the Hebbian ABCD-learning rule provides a starting point for plastic neural networks in the work of Najarro and Risi (2020) so that this variant of plastic neural networks learns both a starting point and an optimizer in the form of the set of learning rule. Table 3.2 lists a number of meta-learning approaches and labels which of the three outer-loop mechanisms for improved inner-loop optimization are involved with the methods.

Considering the possible mechanisms that can be at play in meta-learning can help inform our decisions when developing new algorithms. It can also be beneficial when analyzing cases where existing algorithms fail. Such considerations will obviously be specific to any concrete pairing of algorithm and problem, but an understanding of how meta-learning dynamics allows us to ask questions such as whether it is reasonable to assume that starting point optimization can find a good initial point for inner-loop optimization for all of the given task distribution. Or, as another example, whether the reason that a particular loss landscape optimizer fails is because the inner-loop optimizer cannot reach any optimal points from its starting point, or because the outer-loop parameters are not expressive enough to make meaningful modifications to the inner-loop loss landscape. Or, as a last example, if we have good reasons to believe that the inner-loop loss landscapes generally are easy to navigate, might it then make the most sense to focus on optimizing the inner-loop optimizer for faster inner-loop optimization?

The next chapter revolves around experiments using a particular form of meta-learning, specifically evolved Hebbian learning rules.

Paper Name	Citation	1	2	3
Model Agnostic Meta-Learning for Fast Adaptation of Deep Networks	(Finn et al., 2017)	X		
Practical Bayesian Optimization of Machine Learning Algorithms	(Snoek et al., 2012)			X
Designing Neural Network Architectures Using RL	(Baker et al., 2016)		X	
Learning to Reinforcement Learn	(Wang et al., 2017)		X	X
Learning to Continually Learn	(Beaulieu et al., 2020)	X		X
Meta-Learning through Hebbian Learning in Random Networks	(Najarro and Risi, 2020)			X
Introducing Symmetries to Black-Box Meta-RL	(Kirsch et al., 2021)		X	X
Meta Networks	(Munkhdalai and Yu, 2017)		X	X
ES-MAML Simple Hessian-Free Meta-Learning	(Song et al., 2019a)	X		

Table 3.2: **Meta-Learning Algorithms.** Examples of meta-learning algorithms and how the outer-loop influences the inner-loop loss landscape in one or more of the three possible ways. 1: Starting point optimization. 2: Loss landscape optimization. 3: Optimizer optimization.

Chapter 4

Evolving and Merging Hebbian Learning Rules

The inability to adapt to out-of-distribution (OOD) situations makes artificial agents less useful and despite many advances in the field of reinforcement learning and evolutionary strategies, making adaptable agents remains a challenge (Zhang et al., 2018a,b; Zhao et al., 2019; Justesen et al., 2018). If we want agents to be deployed in complex environments, we cannot expect to be able to train them in all possible situations beforehand. Incorporating synaptic plasticity rules, introduced in Chapter 2, Section 2.5.3, has been shown to endow agents with enhanced behavioral flexibility and robustness.

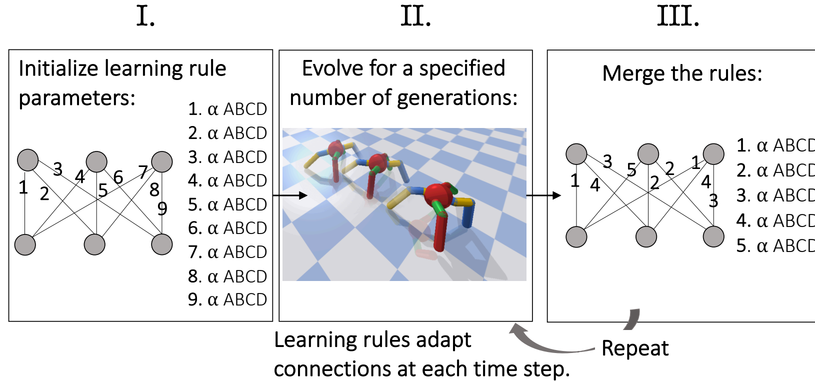


Figure 4.1: **Evolve and Merge Approach.** First, one learning rule is randomly initialized for each connection in the network. Then, the learning rule parameters are evolved for a predetermined number of generations. Subsequently, the rules that have similar parameters are merged into a single rule. The new reduced rule set is then evolved further. This process is repeated and continues until the maximum number of generations allowed has been reached.

Here, we build upon the approach introduced in Najarro and Risi (Najarro and Risi, 2020), where parameters of local plasticity rules - not the connections of the network - were evolved. With this approach, ANNs with plastic connections showed better performances than static ANNs when faced with changes in robot morphology not seen during training.

However, in contrast to nature, in which genomes encode an extremely compressed blueprint of a nervous system, the approach by Najarro and Risi (2020) required each connection in the network to have its own learning rule, significantly increasing the number of trainable parameters.

The amount of information it takes to specify the wiring of a sophisticated brain is far greater than the information stored in the genome (Breedlove and Watson, 2013). Instead of storing a specific configuration of synapses, the genome is thought to encode a much smaller number of rules that govern how the wiring should change throughout the lifetime of the individual (Zador, 2019). In this manner, learning and evolution are intertwined: evolution shapes the rules that in turn shape learning (Hinton and Nowlan, 1987; Price et al., 2003; Snell-Rood, 2013). If the rules encoded by the genome

do not allow the individual to learn useful behavior, then these rules will have to be adapted or go extinct. The phenomenon that a large number of synapses has to be controlled by a small number of rules has been called the ”genomic bottleneck”, and it has been hypothesized that this bottleneck acts as a regularizer that selects for generic rules likely to generalize well (Zador, 2019).

Here, we aim to mimic the genomic bottleneck by limiting the number of rules to be much smaller than the number of connections in the neural networks, in the hope that we can evolve more robust agents. The main insight in the novel approach introduced in this chapter (Figure 4.1) is that after having optimized one learning rule for each synapse in the network, it is possible with a simple clustering approach to drastically reduce the number of unique learning rules required to achieve good results. In fact, we show that starting from having one rule per connection in the ANN (12,288 rules) we can decrease this number of rules by 96.875 % to have only 384 rules controlling all 12,288 connections. We further show that as the number of learning rules decreases, the robustness to unfamiliar morphologies tends to increase. In addition to being inspired by the compressed representations in genomes, the method proposed in this chapter is related to the field of indirect encoding, which has a long history in artificial intelligence research (Gruau et al., 1996; Bentley and Kumar, 1999; Stanley and Miikkulainen, 2003; Tonelli and Mouret, 2013), and further introduced in Section 2.5.3.

As we gradually decrease the number of learning rules, we end up with a set of rules, which have a smaller number of trainable parameters than there are connections in the network. We operationalize robustness as being able to perform well across an array of settings not seen during training. We compare the robustness of plastic ANNs to that of different static ANNs (described further in section 4.1.4): a plain static network with the same architecture as our plastic networks, a smaller static network, and static networks where noise is applied to their inputs during optimization. The plastic networks outperform the plain static networks of different sizes in terms of robustness and perform at the same level as the best configurations of static networks with noisy inputs while requiring a significantly lower number of trainable parameters.

In the future it will be interesting to further increase the expressivity of the evolved rules, potentially allowing an even greater genomic compression with increased generalization to robots with drastically different morphologies.

4.1 Approach

The approach introduced in this chapter evolves a set of local learning rules, where the number of rules is ultimately much smaller than the number of connections. In contrast to other indirect encoding methods, instead of starting with a small rule set, the *Evolve & Merge* approach starts with a large number of trainable parameters compared to the number of trainable parameters in the ANN and only over the course of the evolution end up with a smaller number of trainable parameters by merging rules that have evolved to be very similar (Figure 4.1).

For learning rules, we use a parameterized abstraction of Hebbian learning. The so-called "ABCD" rule, which has been used several times in the past as a proxy for Hebbian learning (Najarro and Risi, 2020; Niv et al., 2002; Risi et al., 2010; Orchard and Wang, 2016), updates the connection between two neurons in the following manner:

$$\Delta w_{ij} = \alpha(Ao_i o_j + Bo_i + Co_j + D) \tag{4.1}$$

where w_{ij} is the connection strength between the neurons, o_i and o_j are the activity levels of the two connected neurons, α is a learned learning rate, and A, B, C, and D are learned constants.

Instead of directly optimizing connection strengths in the plastic network, we only optimize parameters of the ABCD learning rules, which in turn continually adapt the network's connections throughout the lifetime of the agent. The parameters of the learning rules are randomly initialized before evolution following a normal distribution, $\mathcal{N}(0, 0.1)$, and are optimized at the end of each generation. At the beginning of each new episode, the connection strengths of a plastic network are randomly initialized, drawn from a uniform distribution, $Unif(-0.1, 0.1)$. At each time step of the episode, after an action has been taken by the agent, each connection strength is changed according to the learning rule that it is assigned to. Below, we show the training and performance of models for which a unique learning rule is optimized for each connection in the network, as well as models where multiple connections share the same learning rule.

4.1.1 Environment

We train and evaluate our models on how well they can control a simulated robot in the AntBullet environment (Ellenberger, 2018). Here, the task is to

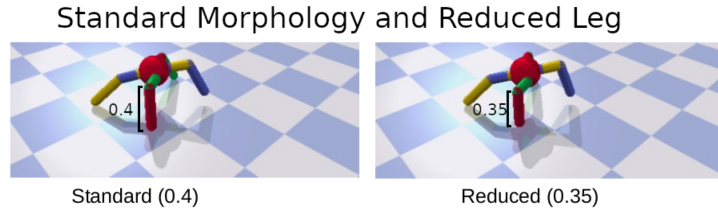


Figure 4.2: **Robot Environment.** Right: The standard morphology that the models were optimized for. Left: An example of the most severe leg reduction done to a leg to test robustness. The lower part of the leg was reduced from 0.4 to 0.35. In this figure, the reduction was done to the red part of the leg in the foreground of the image.

train a three-dimensional four-legged robot to walk as efficiently as possible in a certain direction.

The neural network controlling the robot receives as input a vector of length 28, in which the elements correspond to the position and velocity of the robot, as well as the angles and angular velocities of the robot's joints. To control the robot's movements, the output of the neural network is a torque for each of the eight joints of the robot, resulting in a vector of eight elements.

Previous research has mostly focused on a robot's ability to cope with catastrophic damages to a leg (Cully et al., 2015; Colas et al., 2020; Najarro and Risi, 2020). Here we take a different approach, where legs are modified, but still usable (Figure 4.2). For evaluation, we create slight variations on the standard morphology by shrinking the lower part of the robot's legs (the "ankles" in the robot's xml file). The standard length of an "ankle" is 0.4. We evaluate five different reductions, reducing the length of an "ankle" to 0.39, 0.38, 0.37, 0.36, and 0.35, respectively. One by one, each of the four legs has its ankle reduced to each of these five lengths. In addition, we evaluate in a setting where both front legs are reduced at the same time, as well as a setting where the front left leg and the back right leg are reduced at the same time. We thus have five different reductions on six different leg combinations, amounting to 30 different morphology variations. This way we can more precisely determine how much variation from the original setting the models are able to cope with. When evaluating with a varied morphology, we adapt the reward to solely reflect the distance traveled in the correct direction, so that no points are given for just "staying alive".

Table 4.1: Hyperparameters for ES

Parameter	Value
Population Size	500
Learning Rate	0.1
Learning Rate Decay	0.9999
Learning Rate Limit	0.001
Sigma	0.1
Sigma Decay	0.999
Sigma Limit	0.01
Weight Decay	0

4.1.2 Evolution Strategy

All models are optimized using Evolution Strategy (ES) (Salimans et al., 2017). In static networks connections are evolved directly and in plastic networks only the learning rule parameters are evolved. We use an off-the-shelf implementation of ES (Ha, 2017a) with its default hyperparameters, except that we set weight decay to zero (see Table 4.1 for complete hyperparameter configurations). This implementation uses mirrored sampling, fitness ranking, and the Adam optimizer for optimization. In all runs, a population size of 500 is used, and optimization spans 1,600 generations.

4.1.3 Merging of Rules

In order to produce rule sets with fewer rules than the number of connections in the network, we use the K-Means clustering algorithm (Pedregosa et al., 2011) to gradually merge the learning rules throughout training. We first initialize a network with one learning rule for each synapse (12,288 different learning rules) and optimize these learning rules for 600 generations. We then half the number of learning rules by using the K-Means algorithm to find 6,144 cluster centers among the 12,288 rules. The new learning rule of a given synapse will simply be the cluster center of the learning rule, to which it was previously assigned. The newly found smaller set of learning rules is then optimized for 200 generations before K-Means clustering is used to half the number of rules again. The number of generations between merging of

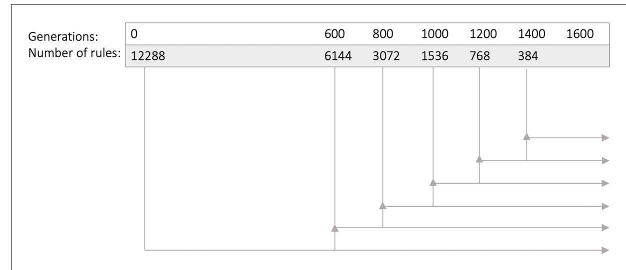


Figure 4.3: **Rule Merging.** The number of rules is iteratively halved using K-Means clustering. Every reduced rule set is optimized until the time limit of 1,600 generations so that fair comparisons between models can be made.

the rules as well as how many cluster centers to reduce the rule set to are hyperparamters that were determined to work well in preliminary experiments, and the choices mainly depend on how much total training time is permitted. This process is repeated until we have optimized for 1,600 generations altogether. In order to be able to make fair comparisons of the different reduced rule sets, we also optimize each individual reduced set until it has been optimized for 1,600 generations (see Figure 4.3).

4.1.4 Experiments

We compare a total of 12 different models in terms of their ability to learn and their robustness (see Table 4.2 and Figure 4.9a, 4.9b). Common to all of them is that they are feedforward networks, they have two hidden layers, they have no biases, and the hyperbolic tangent function is used for all activations.

All networks have 128 neurons in the first hidden layer and 64 neurons in the second. The **plain static model** is a static neural network without any Hebbian learning. We also optimize and evaluate a **smaller static model**, in which the hidden layers are sizes 32 and 16 (1,536 parameters). Two other static network models were trained in a setting where **noise** was applied to the input throughout the optimization. If the models can perform well despite noisy inputs, they might also be more robust to morphology changes despite being static, and therefore we create these models as additional baselines to compare our approach to. At each time step during optimization, a vector of the same size as the input with elements drawn from a normal

distribution was created and then added to the input element-wise. For one model, the distribution was $\mathcal{N}(0, 0.05)$, and the other had a distribution of $\mathcal{N}(0, 0.1)$. In the figures below, these models are named after the amount of noise their inputs received during optimization.

The **ABC** model type was trained with an incomplete ABCD rule: the learning rate and the D parameter were omitted, so only the activity-dependent terms were left. For all other plastic network models, the rules consisted of five parameters (A, B, C, D, and the learning rate). The model called α**ABCD** was optimized with a rule for each connection throughout evolution, and the number of rules was never reduced. This is the same method as was introduced by Najarro and Risi (Najarro and Risi, 2020). The model called "**500 rules from start**" was initialized with just 500 rules, and this number remained unchanged. Each connection of the ANN was randomly assigned to one of the 500 rules at initialization. In the figures summarizing the results (Figures 4.8,4.7, 4.9a, 4.9b), the rest of the plastic models are simply named after the number of rules they have in their rule sets.

4.2 Results

After optimizing for 1,600 generations, we see that the static networks tend to perform much better on the standard morphology that the models are optimized on. Figure 4.4 presents the training curves of each of the reduced rule sets and Figure 4.5 shows the training curves of the rest of the models. For comparisons with different reinforcement learning algorithms in the standard AntBullet environment see Pardo (Pardo, 2020) for baselines where, e.g., Proximal Policy Optimization (PPO) achieves a score of around 3100, Deep Deterministic Policy Gradient (DDPG) scores around 2500, and Advantage Actor-Critic (A2C) scores around 1800.

Performances after optimization are summarized in Table 4.2 and Figures 4.7,4.8, 4.9a, 4.9b. The reduced rule sets generally end up with a better performance when evaluated on the original settings compared with the case where the number of rules is equal to the number of connections throughout. Looking at the box plots in Figure 4.9a, we see that all models with reduced rule sets (named by the number of rules) have a higher mean performance across all novel settings than the model with a rule for each connection (named α**ABCD** in the figures). They also have a higher mean performance than the plain static model, which has a large variability in its performance

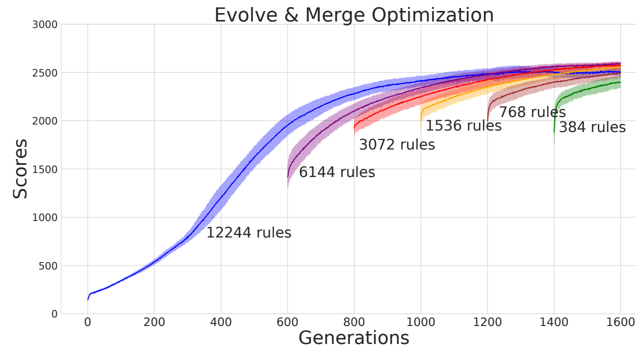


Figure 4.4: **Evolve & Merge Training Results.** Training curves of five independent evolution runs. The solid lines reflect the average population means for each run of each model. The filled areas are the standard deviations of the models' population means. Immediately after a merging of rules has occurred, the newly reduced rule set is set back a bit in its performance compared to before merging, but performance quickly recovers and improves. Without adding to the total optimization time of 1600 generations, a rule set of just 384 different rules reaches a comparable population mean score as a rule set with 12244 rules.

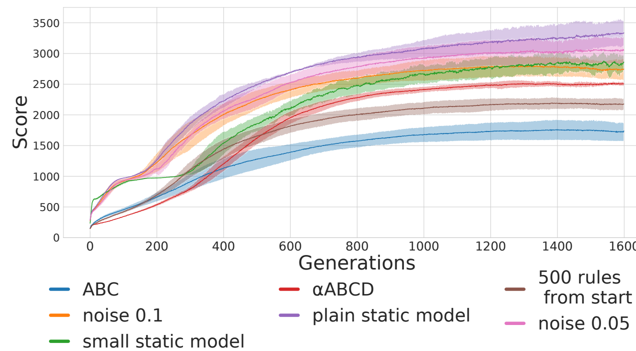


Figure 4.5: **Training Results.** For all models, we ran five independent evolutionary runs. The solid lines reflect the average population means for a given model throughout evolution. The filled areas are the standard deviations of the models' population means. While the static models end up with better scores, all models, except 'ABC', are able to obtain reasonably well-performing populations within 1600 generations.

Model Name	Num. Params.	Orig. Score	Novel Score
plain static model	12,288	**3,513**±221	2,095±442
small static model	1,536	3,040±170	923±387
noisy (0.05)	12,288	3,283±213	2,172±395
noisy (0.1)	12,288	3,020±126	2,256±226
500 from start	2,500	2,185±92	1,931±90
ABC	36,864	1,731±158	1,462±206
α ABCD	61,440	2,528±52	2,173±126
6,144 rules	30,720	2,649±13	2,259±100
3,072 rules	15,360	2,631±21	2,244±123
1,536 rules	7,680	2,631±27	**2,323**±90
768 rules	3,840	2,580±33	2,286±157
384 rules	**1,920**	2,516±44	2,267±186

Table 4.2: Parameter counts and scores after optimization. Orig. Scores are scores under original morphology settings, and Novel Scores are under the unseen altered settings. Scores reflect the means of 5 models of each model type. A model's score is its mean performance over 100 episodes. Standard deviations are provided next to each mean. The plain static network has the highest score in the original settings but has a massive decrease in scores in the novel settings. All reduced rule sets have higher mean scores with smaller standard deviations in the novel settings, than any static network.

Figure 4.6: Evolved Rules Examples. Strip plots of values of connection weight changes of the top 10 most followed rules in an optimized version of each of the reduced rule sets. The names of each of the strips seen along the x-axis are the indices of the rules, and in parentheses are the number of updates that the rule made during an episode with the original environment settings. A strip indicates what updates were made by the rules during this episode.

across its different evolution runs. The average performance tends to increase as the number of rules decreases. The **ABC** model and the model that had only 500 rules from the beginning achieved training performances considerably lower than the rest of the models. The static models optimized with noisy inputs obtain similar scores as the reduced rule sets. When the noise added to the input is drawn from $\mathcal{N}(0, 0.05)$, the best evolution run of the model is, like the static network, better than any of the runs with the reduced rule sets, but its worst run is far worse. When the noise is drawn from $\mathcal{N}(0, 0.1)$ the average performance is at the same level as the most reduced rule set and the top performance is a bit better. Overall, while the ceiling of the performances seems to be lower for the plastic networks, the floor tends to be much higher (except for the case of the ABC-only models). The latter point is especially visible if we look at the worst performances of the runs of each model, as is done in Figure 4.9b. Here we see that the model trained with 500 learning rules from the beginning of the training is the least likely to get catastrophically bad performances.

4.2.1 Discovered Rules

To get a visual intuition of what type of learning rules evolved, Figure 4.6 shows strip plots of the top 10 most followed rules in each of the reduced rule

sets. From these, we can see that several different types of rules are found by evolution. First, one type of rule that is found in the top 10 of all the rule sets, has all its updates closely concentrated around zero (e.g., the rule indexed 1020 in the rule set of 1536 rules, or the rule indexed 114 in the set of 384 rules). Second, multiple rules provide relatively strong updates, both positive and negative, but which have a gap around zero (e.g., index 104 in the 384 set, or 837 in the 6144 set). Another apparent type of rule provides both positive and negative updates but has no gap around zero (e.g., index 734 in the 768 set, or 248 in the 3072 set). Lastly, we see a type of rule, that has the vast majority of its updates to be of a specific sign (e.g., index 240 in set 768 (mostly negative), or 784 in set 3072 (mostly positive)). These observations suggest that many connections are destined to only receive very small updates throughout the episode, and remain largely unchanged compared to some of the stronger updates that other rules provide. Further, the fact that multiple of the rules that have the most connections assigned to them can provide both positive and negative updates, confirms that two connections following the same rules might end up being updated very differently from each other. Fewer rules do therefore not necessarily make the neural network less expressive.

4.3 Discussion

In this chapter, we build upon the results of Najarro and Risi (Najarro and Risi, 2020) which showed that increased robustness can be achieved by evolving plastic networks with local Hebbian learning rules instead of evolving ANN connections directly. We show that the robustness can be enhanced even further with an *Evolve & Merge* approach: throughout optimization we iteratively use a clustering algorithm to merge similar rules, resulting in a smaller rule set at the end of the optimization process. While the plastic networks are not able to get scores as high as the highest scores of the static networks, the plastic networks are less likely to get catastrophically bad scores, when the morphology of the robot is changed slightly. Note, that since the plastic networks are initialized randomly at the beginning of each new episode, a somewhat lower score for plastic networks under the original training setting is to be expected. This is because we cannot expect the initially random network to perform well in the first few time steps when the learning rules have only had a little time to adapt the connections. Static

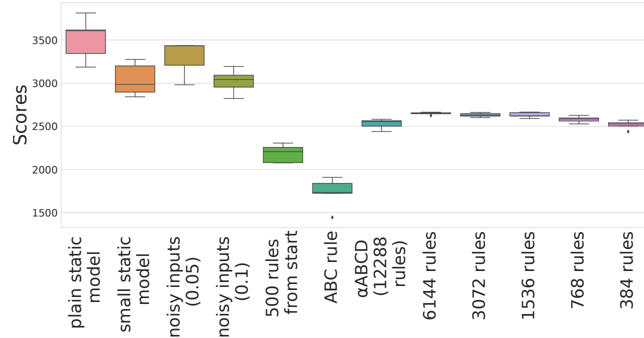


Figure 4.7: **Original Environment Scores.** Box plots for the scores of the optimized models in the original environment setting. For each model, the score is averaged over 100 independent episodes. As we optimized 5 models of each type, the box plots show the variation of the scores within a given model type. See Section 4.1.4 for model descriptions. All static models have better scores than any plastic model in the original setting. The reduced rule sets see no decrease in performance compared to the model with a rule for each connection.

networks, on the other hand, can be optimized to perform well under familiar settings from the very first time step, but suffer from a lack of robustness.

Our results indicate that the generalization capabilities tend to improve as the number of rules is reduced. At the end of the *Evolve & Merge* approach, we thus have a model, that has a smaller number of trainable parameters and at the same time generalizes better. Using a simple clustering algorithm as a way of merging the learning rules, we are able to go from initially having 12,288 rules (corresponding to 61,440 trainable parameters) to having just 384 learning rules (corresponding to 1,920 trainable parameters) while improving robustness, all without increasing the number of generations used for optimization.

In order to make fair comparisons between all the models, we did not allow for more generations for the reduced rule sets here. However, it is likely that if we were to permit more optimization time, we could decrease the number of learning rules even further. The observed robustness cannot just be attributed to a smaller number of trainable parameters, since the smaller static networks that performed at the same level as the plastic networks in the original settings, did not show robust performances in altered settings.

The static network optimized with noisy inputs, on the other hand, had

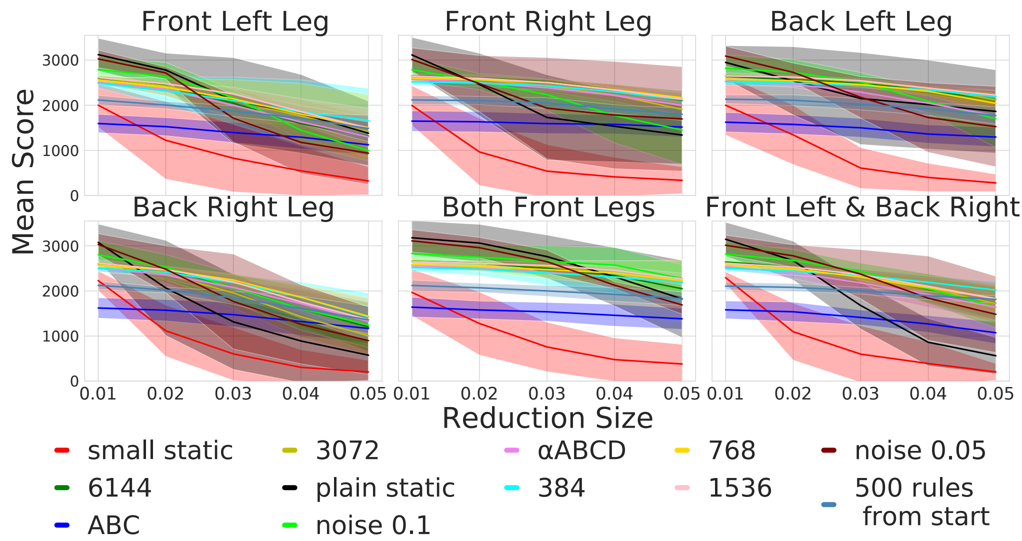
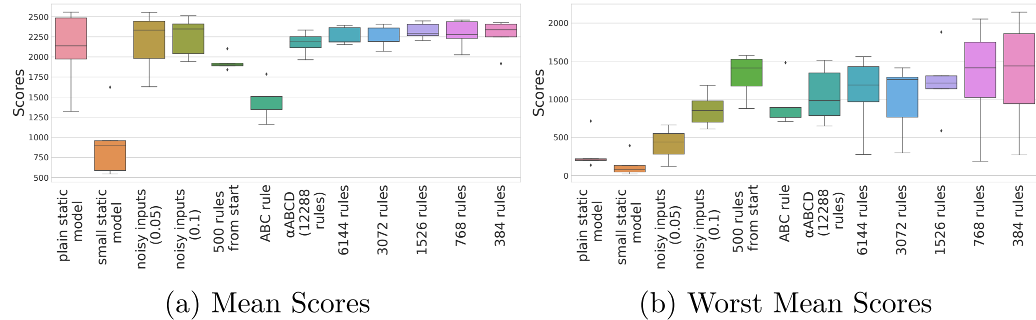


Figure 4.8: **Mean scores for all models in all novel environment settings.** As leg reductions increase (x-axis), the models tend to perform worse (y-axis). The mean scores are calculated over 100 episodes for each model. The static models have much more variability in their scores than the plastic ones. Further, static networks tend to start with a good score for the smallest reduction but decrease rapidly as reductions become larger. The plastic networks, on the other hand, have much flatter performance curves.



(a) Mean Scores (b) Worst Mean Scores

Figure 4.9: Generalisation Performance. Box plots for the (a) mean scores and (b) worst mean scores of the optimized models across all novel environment settings. For each model, the score is averaged over 100 independent episodes. The worst mean scores shown in (b) show how bad a model is at its worst across 100 episodes. The worst scores of the plastic networks tend to be much better than the worst scores of the static ones. This shows that plastic models are at less risk of getting a catastrophically bad score where the robot is barely able to move at all. As in Figure 4.7, the box plots show the variation of the scores within a given model type.

average performances across all novel settings and optimization runs, which were very similar to that of the best plastic network model. Noisy inputs have long been used for data augmentation in supervised tasks, such as speech recognition, to gain more robust models (Cui et al., 2015). Adding noise to inputs has also been used as a way to get robust representations in autoencoders (Vincent et al., 2008). However, using noise in training has been explored to a lesser extent in reinforcement learning-type frameworks (Igl et al., 2019). In this chapter, the static networks with noisy input provide a strong baseline to compare the reduced plastic networks in terms of their robustness, and as we can see, the plastic networks achieve similar performances. However, in order to achieve such results by applying noise to the inputs to static networks, we need to carefully pick the correct amount of noise to apply; too little noise and the results will be indistinguishable from the plain static approach, and too much noise is likely to hinder progress completely. Using the plastic approach, the parameters will regulate themselves, and with the *Evolve & Merge* method, we have the added benefit of ending up with a smaller number of trainable parameters. The similar performances between static networks with noisy inputs and plastic networks

are interesting and will have to be explored further in the future. The rather naive approach to using noise by simply adding it on top of the input is unlikely to be a promising method to improve upon to make even more robust models. It is, on the other hand, easier to imagine improvements to the learning rules (see section 4.3.1).

The evolved learning rules used here are inspired by Hebbian learning. However, the model that used only the activity-dependent terms of the parameterized rule (the ABC terms), failed to perform well, making it clear that the stability provided by the constant terms is necessary for the locomotion task used here.

The results also showed that starting from a complete rule set, and then merging the rules throughout optimization achieved superior results compared to starting with a small number of rules. This draws parallels to the recent ”Lottery Ticket Hypothesis” (Frankle and Carbin, 2018) that builds upon the notion that a trained neural network can most often be pruned drastically, resulting in a much smaller network that performs just as well as the unpruned network (Li et al., 2016). However, if one were to start training from scratch with a randomly initialized small network of the same size as the pruned network, the training is likely to be much more difficult, and one is unlikely to get the same performance. The hypothesis states that when initializing a large network, it is likely that one of the combinatorially many sub-networks inside the full network will be ”easily trainable” for the given problem; we are more likely to have a ”winning ticket” within the random initialization since we have so many of them. The number of sub-networks - or potential winning tickets - dwindles rapidly if we decrease the size of the full network. To the best of our knowledge, the methods used for finding winning tickets (Frankle and Carbin, 2018; Zhou et al., 2019) have not yet been explored in the case where the optimization method is ES, and much less in the context of indirect encoding. Our results hint that the Lottery Ticket Hypothesis might also hold in the indirect encoding setting that we employ here. Further investigations into when the Lottery Ticket Hypothesis asserts itself in indirect encoding schemes might provide valuable insights for future approaches.

Deciding how to evaluate one’s models on OOD circumstances can seem a bit arbitrary, as countless different changes to a simulated environment can be made. Several previous studies have focused on the ability of a robot to show robustness in the face of a severe leg injury (Cully et al., 2015; Colas et al., 2020; Najarro and Risi, 2020). Here we opted for slight variations on leg

lengths instead, partly to highlight how quickly neural network models can break completely. For many of the models, a severe injury is not required in order to put a well-performing model at risk of malfunctioning. Had we chosen a larger range of reductions, it is likely that the results would have been tilted to favor the plastic networks more, whereas a smaller range would have had the opposite effect. The issue of deciding how to evaluate the generalization capabilities of a model speaks to a larger discussion of overfitting in artificial agents (Zhao et al., 2019). In some sense, the static networks do what we ask them to more so than the plastic ones; when we optimize the networks we implicitly ask them to overfit as much as possible to the problem that we present them. In the approach presented in this chapter, we explicitly optimize for one setting but hope that our models will also be able to perform in different settings. This is different from the normal meta-learning framework described in Chapter 2, Section 2.5. However, meta-learning approaches require us to make somewhat arbitrary choices regarding which different tasks we should expose our models to during training. For this reason, it is still useful to develop methods that are intrinsically as robust as possible.

4.3.1 Future Directions

We showed that it is indeed possible to evolve relatively well-performing models with an increased robustness to morphology changes, while at the same time having fewer trainable parameters; this approach opens up interesting future research directions.

As mentioned in Section 2.5.3, an important concept within Hebbian theory is that of cell assemblies, which after repeated exposure to stimuli can become increasingly correlated and able to perform pattern completion. In this theoretical framework that has been supported by a wide array of evidence(See et al., 2018; Miller et al., 2014; Hampson and Deadwyler, 2009)(for a review, see (Saxena and Cunningham, 2019)), recurrent connections between the neurons are often assumed. In the current study, we have only used simple feedforward networks with two hidden layers. Applying the *Evolve & Merge* approach to local learning rules for more advanced neural networks with recurrent connections will be an interesting line of research in the future. More generally, the *Evolve & Merge* approach also lends itself well to be combined with methods that evolve the neural architecture of the network that the learning rules apply to. An example of this is the NEAT

algorithm (Stanley et al., 2003; Risi and Stanley, 2011, 2012a).

It will also be interesting to combine this indirect encoding method with a meta-learning framework such as MAML (Finn et al., 2017). Models with few trainable parameters, controlling a large, expressive ANN might intuitively be an ideal candidate for few-shot learning in the MAML framework.

Further, while local learning rules such as spike-time dependent plasticity are important drivers of change in synapses in the brain, synaptic plasticity is also affected by neuromodulators (Dayan, 2012; Feldman, 2012). Extending the evolved learning rules to be able to take into account reward signals could greatly improve the model's ability to respond to changes in the environment in an adaptive manner, and it is something we look forward to implementing in future studies. Something similar to this has been explored in other approaches to plastic networks (Soltoggio et al., 2008; Ellefsen et al., 2015; Bertens and Lee, 2020; Ben-Iwhiwhu et al., 2020), and we expect this to also be a beneficial addition to our approach.

While we have evolved a set of relatively simple learning rules, our approach could just as well be used in conjunction with more complicated rules, e.g., rules with more parameters (Chalmers, 1991), or rules that produce non-linear outputs of the inputs (Orchard and Wang, 2016; Bertens and Lee, 2020). With more expressive rules, it might be possible to limit the number of rules and trainable parameters even further.

On a practical note, having few trainable parameters opens up for the possibility of using more sophisticated optimization methods such as CMA-ES as described in 2, Section 2.4.3.

4.4 Conclusion

If the environment one wishes to deploy an artificial agent in is guaranteed to be identical to the training environment, one might be better off evolving a static network to control the agent. However, if the aim is to have artificial agents act in complex real-world environments, such guarantees cannot be made. The evolution of plastic networks that can better adapt to changes, is therefore an interesting prospect. The results shown in this chapter contribute to the development of robust plastic networks. We show that it is simple to achieve a fairly small rule set and that as the number of rules decreases, the robustness of the model increases. This can be achieved with no additional optimization time. We believe that there are several exciting ways

to continue this line of research. The next chapter explores a method that might be helpful for plastic networks in a more indirect manner by focusing on parameters associated with the activity of the neurons.

Chapter 5

Learning to Act through Evolution of Neural Diversity in Random Neural Networks

This chapter turns the attention away from connections of ANNs and focuses the neurons on the networks, another area with much room for extending existing methods with bio-inspired approaches. As the name suggests, ANNs originally took inspiration from biological networks found in the brains of animals (Fasel, 2003; Hassabis et al., 2017). The main analogous feature of ANNs to biological ones is that nodes in the network distribute information to each other and that the network learns to represent information through the gradual tuning of their interconnections. In ANNs, neuro-centric computation is abstracted to an activation function that is usually shared between all neurons within a layer or even the whole network.

By parameterizing neurons to a larger degree than current common practice, we might begin to approach some of the properties that biological neurons are characterized by as information processors.

Inspired by the neural diversity found in biological brains, we introduce a parameterized neural unit with a feedback mechanism (Figure 5.1). The idea is that evolving a unique set of parameters for each neuron has the potential to create a diverse set of neurons. While we are ultimately interested in potential synergies from evolving both weights and neural parameters, in order to investigate the expressive power of the proposed neural units in isolation, here we only evolve the parameters of neurons; the randomly initialized synaptic weights are fixed throughout the entire process.

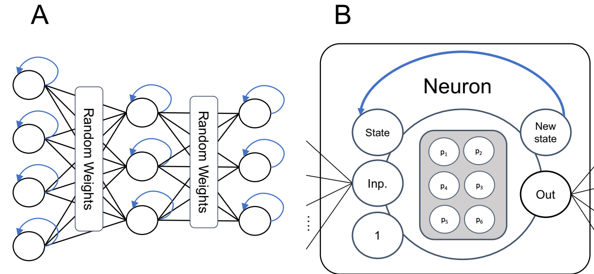


Figure 5.1: **Illustration of the proposed neural unit.** (A) Neural network with random weights and a layer of neural units. (B) Zoomed-in view of a neural unit. The parameters p_i are optimized in order to achieve an expressive function. These parameters are used to integrate the input with a neural state and a bias term through a vector-matrix multiplication. It outputs a value to be propagated to the next layer, as well as its own new state. For further details, see Section 5.2.

Our results show that even when never optimizing any synaptic parameters, the random networks with evolved neural units achieve performances competitive with simple baselines in different reinforcement learning tasks. For baseline comparisons, we evolve the weights of feedforward neural networks using a standard hyperbolic tangent function as non-linearities. We compare our approach both with (1) a small network with a similar number of weights as there are trainable parameters in the neural units of the main experiments, and (2) a network with the same number of neurons as in the main experiments and thus many more tunable synaptic weights. We also test a non-recurrent version of the neural units. The environments used are a variation of the classic CartPole environment (Gaier and Ha, 2019), the `BipedalWalker-v3` environment, and the `CarRacing-v0` environment (Brockman et al., 2016). In all cases, networks with evolved neurons performed on par with the weight-optimized networks.

By showcasing the potential of these more expressive and diverse neurons in fixed random networks, we hope to pave the way for future studies exploring potential synergies in the optimization of synaptic and neural parameters. More specifically, we believe that more expressive neural units like the ones proposed here could be useful in combination with online synaptic activity-dependent plasticity functions (van Ooyen, 1994; Abbott and Nelson, 2000;

Stiles, 2000; Soltoggio et al., 2018; Najarro and Risi, 2020).

5.1 Related Work

Neurocentric Optimization. Biases in neural networks are examples of neuro-centric parameters. When an ANN is optimized to solve any task, the values of the network's weights and biases are gradually tuned until a functional network has been achieved. The network most commonly has a one bias value for each neuron and the weight parameters thus greatly outnumber these neuro-centric bias parameters. It is well known that the function of biases is to translate the activations in the network (Benítez et al., 1997) and ease the optimization of the network.

Another well-known example of neuro-centric parameters is found in the PReLU activation functions (He et al., 2015) where a parameter is learned to determine the slope of the function in the case of negative inputs. Introducing this per-neuron customization of the activation functions was shown to improve the performance of networks with little extra computational cost.

Neuro-centric parameter optimization can also be found within the field of plastic neural networks (see Chapter 2, Section 2.5.3). In one of their experiments, Urzelai and Floreano (Urzelai and Floreano, 2001) optimized plasticity rules for each neuron (they referred to this as 'node encoding'), such that each incoming synapse to a node was adapted by a common plasticity rule. The idea of neuro-centric parameters is thus far from new. However, in contrast to earlier work, in this chapter, we explore the potential of solely optimizing neuro-centric parameters in a randomly initialized network without ever adapting the weights.

Activation Functions in Neuroevolution. How artificial neurons are activated has a major impact on the performance of ANNs (Nwankpa et al., 2018). Evolution has been used to discover activation functions to optimize performance in networks that are optimized on supervised classification problems using backpropagation (Basirat and Roth, 2018; Liu et al., 2020; Bingham et al., 2020; Bingham and Miikkulainen, 2022). These approaches aim to find a single optimal activation of neurons that all neurons within a layer or a whole network share.

Not all ANNs have a single activation function for all hidden neurons. Some versions of Neuro-Evolution of Augmented Topologies (NEAT) (Stanley and Miikkulainen, 2002; Papavasileiou et al., 2021) allow for different

activation functions on each neuron. The NEAT algorithm searches through networks with increasing complexity over the process of evolution. Starting from a simple network structure, each new network has a chance of adding a new neuron to the network. When a new neuron is added, it can be allocated a random activation function from a number of predetermined functions. In newer versions of NEAT, mutations allow activation functions of neurons to be changed even after it was initially added (Hagg et al., 2017). This resulted in more parsimonious networks.

In their *weight agnostic neural network* (WANN) work, Gaier and Ha (2019) used NEAT to find network structures that could perform well, even when all weights had the same value. Notably, hidden neurons could be evolved to have different activation functions from each other. This likely extended the expressiveness of the WANNs considerably. Our work is similar in spirit to that of Gaier and Ha, in that we are also exploring the capabilities of a component of neural networks in the absence of traditional weight optimization. However, in our work, all networks have a standard fully connected structure. Furthermore, we do not choose from a set of standard activation functions but introduce stateful neurons with several parameters to tune for each neuron.

Lottery Tickets & Supermasks. The Lottery Ticket Hypothesis (Frankle and Carbin, 2018; Frankle et al., 2019) was mentioned in Section 4.3 of the previous chapter. An even stronger take on the Lottery Ticket Hypothesis states that due to the sheer number of sub-networks that are present within a large network, it is possible to learn a binary mask on top of the weight matrices of a randomly initialized neural network, and in this manner get a network that can solve the task at hand (Malach et al., 2020; Wortsman et al., 2020; Ramanujan et al., 2020). This has even been shown to be possible at the level of neurons; with a large enough network initialization, a network can be optimized simply by masking out a portion of the neurons in the network (Wortsman et al., 2020; Malach et al., 2020).

Learning parameterized versions of neurons could be seen as learning a sophisticated mask on top of each neuron as opposed to a simple binary mask. The idea of learning a binary mask on top of a random network relies on the random network being sufficiently large (Malach et al., 2020) so that the chance of it containing a useful sub-network is high. Masking neurons is a less expressive masking method than masking weights: it is equivalent to masking full columns of the weight matrices instead of strategically singling out weights in the weight matrix. As such, the method of masking neurons

with binary masks requires larger random networks to be successful. In this chapter, we optimize neuro-centric parameters in relatively small networks. This is possible because the neurons themselves are much more expressive than a binary mask.

5.2 Evolving Diverse Neurons in Random Neural Networks

Typically, optimization of ANNs has been framed as the learning of distributed representations (Bengio et al., 2013) that can become progressively more abstract with the depth of the network. Optimization of weights is a process of fine-tuning the iterative transformation of one representation into another to end up with a new, more useful representation of the input. How the intermediate layers respond to a given input depends on the specific configuration of the weight matrix responsible for transforming the input as well as their activation function.

Randomly-initialized networks can already perform useful computations (Ulyanov et al., 2018; He et al., 2016; Hochreiter and Schmidhuber, 1997). When neural units are trained specifically to interpret signals from a fixed random matrix, as is the case in this chapter, the initially arbitrary transformations will become meaningful to the function, as long as there exist detectable patterns between the input the function receives and the output that the function passes on and is evaluated on. Whether a pattern is detectable depends on the expressiveness of the function. With this in mind, it is reasonable to assume that if neurons in the network are made more expressive, they can result in useful representations even when provided with arbitrary transformations of the input.

Motivated by the diversity of neuron types in animal brains (Soltesz et al., 2006), we aim to test how well a neural network-based agent can perform reinforcement learning tasks through optimization of its neuro-centric parameters alone without optimizing any of its neural network weights. An illustration of the neural model we optimize in this chapter is shown in Fig. 5.1. Each neural unit consists of a small three-by-three matrix of values to be optimized. Each neural unit in a layer is at each time step presented with a vector with three elements. The input value, propagated through the random connection from the previous layer, is concatenated with the current state of the neuron

and a bias term. Together, these form a vector. The output of a neuron is the vector-matrix multiplication. From the perspective of a single neuron, this can be written as:

$$[\hat{x}_{l,i}^t, h_{l,i}^t, o] = \tanh(\mathbf{n}_{l,i} \cdot [x_{l,i}^t, h_{l,i}^{t-1}, 1]^T). \tag{5.1}$$

Here, $x_{l,i}^t$ is the input value, $h_{l,i}^{t-1}$ is the state of the neuron, and $\mathbf{n_{l,i}}$ the matrix of neural parameters, with l denoting the current layer in the network, i the placement of the neuron in the layer, and t is the current time step. The hyperbolic tangent function is used for non-linearity and to restrict output values to be in [-1, 1]. $\hat{x}_{l+1,j}^t$ is the value that is propagated through weights connecting to the subsequent layer, and $h_{l,i}^t$ is the updated state of the neuron. o, the third value in the output of the neuron is discarded. As the three-by-three matrix has nine values in total, we need to optimize nine parameters for each neuron in the network.

Representing a neuron by a small matrix means that the neuron can take more than a single value as input, as well as output more than one value. Here, we utilize this to endow each neuron with a state. The state of the neuron is integrated with the input through the optimized neural parameters. Part of the neuron's output becomes the new state of the neuron, which is fed back to the neuron with the next input. This turns our neurons into small dynamical systems. Presented with the same input value at different points in the neuron's history can thus yield different outputs. We find that stateful neurons provide a convenient and efficient way of equipping a network with some memory capabilities. One can see a layer of such neurons as a set of tiny recurrent neural networks (RNNs) that are updated in parallel with local inputs, unique to each RNN. As such, a layer of this proposed neural unit differs from simple RNN architectures, such as Jordan Networks (Jordan, 1997) or Elman Networks (Elman, 1990) in that a state associated with a neuron only affects the next state and output of that particular neuron. These local recurrent states only rely on the small matrix of the neural unit, i.e., n times nine parameters, where n is the number of neurons in the layer. A recurrent layer of, e.g., an Elman Network requires an n-by-n sized matrix to feed its activations back into itself. Additionally, the calculation of the neural state and the output of the neuron are separated to a higher degree for the neural units, compared to the hidden state being a copy of the neural output.

5.3 Experiments

We optimize neural units in otherwise standard fully connected feedforward neural networks. All networks in our experiments have two hidden layers, containing 128 and 64 neurons, respectively. We use learned neural units for all neurons, including in the input and output layers. The sizes of the input and output layers vary with the environments described below. The fixed weight values are sampled from $\mathcal{N}(0, 0.5)$. We ran each experiment three times on different seeds, except for the weight-optimized models in the Car Racing environment, which we ran only twice due to its longer training times.

While neurons with recurrent states are common in the field of spiking neural networks that emphasizes biological realism (Gerstner, 1990; Izhikevich, 2003, 2007; Garaffa et al., 2021), it is a departure from the simple neurons found in most ANNs. As a control, we also optimize neuro-centric parameters for neurons without a recurrent state (**Simple Neuron**). The setup for optimizing these is very similar to that of the stateful neurons, but no part of the output of the vector-matrix multiplication is fed back to the neuron's input at the next time step. Instead of being represented by three-by-three matrices, these simple neurons are represented by two-by-two matrices and thus have fewer parameters to optimize.

As baselines, we optimize the weights of standard feedforward networks. For these, we run two different settings: one has a similar number of adjustable parameters as the number of parameters in the neural unit approach (**Small FFNN**). To get the number of weights to be similar to neural parameters, the widths, and depths of these networks have to be considerably smaller than the random networks used in the main experiments. In the second baseline setting (**Same FFNN**), we train weights of networks that have the same widths and depths as the networks in the main experiments, and thus many more adjustable parameters. Unless stated otherwise, the activation function for all baseline experiments is the hyperbolic tangent function for all neurons.

5.3.1 Environments

We test the effectiveness of evolving a diverse set of neurons in randomly initialized networks in three diverse continuous control tasks: the CartPoleSwingUp environment (Gaier and Ha, 2019), the Bipedal Walker en-

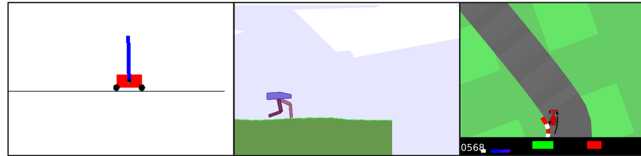


Figure 5.2: **Environments used for experiments.** Left: `CartPoleSwingUp`. Middle: `BipedalWalker-v3`. Right: `CarRacing-v0`.

vironment, and the Car Racing environment (Brockman et al., 2016), which are described below:

CartPoleSwingUp. This environment is a variation of the classic control task (Barto et al., 1983), where a cart is rewarded for balancing a pole for as long as possible (Fig. 5.2; left). In the `CartPoleSwingUp` variation, an episode starts with the pole hanging downwards, and to score points, the agent must move the cart such that the pole gets to an upright position. From there, the task is to keep balancing the pole. We use the implementation of Gaier and Ha (Gaier and Ha, 2019). The agent gets five values as input and must output a single value in $[-1, 1]$ in order to move the cart left and right. With these input- and output layers, the total number of neurons in the network becomes 198, and we optimize 1,792 parameters. For a feed-forward network with a similar number of weights, we optimize a network with two hidden layers of size 48 and 32, respectively.

Bipedal Walker. To get a maximum score in the `BipedalWalker-v3` environment (Brockman et al., 2016), a two-legged robot needs to learn to walk as efficiently and robustly as possible (Fig. 6.3; middle). The terrain is procedurally generated with small bumps that can cause the robot to trip if its gait is too brittle. Falling over results in a large penalty to the overall score of the episode. Observations in this environment consist of 24 values, including LIDAR detectors and information about the robot's joint positions and speed. For actions, four continuous values in $[-1, 1]$ are needed. This results in a network with 220 neurons altogether. To get a network with a similar number of weights as there are parameters in the main experiment, we train a feedforward network with two hidden layers, both containing 32 neurons.

Car Racing. In the `CarRacing-v0` domain (Brockman et al., 2016), a car must learn to navigate through procedurally generated tracks (Fig. 5.2; right). The car is rewarded for getting as far as possible while keeping inside

Table 5.1: Convolutional Layer Parameters.

	Layer 1	Layer 2
Input Channels	3	6
Output Channels	6	8
Kernel Size	3	5
Stride	1	2
Activation Function	tanh	tanh
Bias	Not used	Not used

the track at all times. To control the car, actions consisting of three continuous values are needed. One of these is in $[-1, 1]$ (for steering left and right), while the other two are in $[0, 1]$, one for controlling the gas, and the other for controlling the break. The input is a top-down image of the car and its surroundings. For the experiments in this chapter, we wanted to focus on the effectiveness of the neural units in fully connected networks. We, therefore, followed a strategy closely mimicking that used by Najarro and Risi (2020) to get a flat representation of the input image. We normalize the input values and resize the image into the shape of $3x84x84$. The image is then sent through two convolutional layers, with the hyperparameters of these specified in Table 5.1. After both layers, a two-dimensional max pooling with kernel size 2 and stride 2 was used to gradually reduce the number of pixels.

The output of the convolutional layers is flattened, resulting in a vector containing 648 values. This vector is then used as input to a fully connected feedforward network with our proposed neural units. Importantly, the parameters of the convolutional layers stay fixed after initialization and are never optimized. The output layer has three neurons. With the much larger input layer, this network has 844 neurons and 7,596 adjustable parameters. Since two of the action values should be in $[0, 1]$, a sigmoid function is used for these two output neurons in place of the hyperbolic tangent function as shown in Equation 5.1.

For the baseline experiments, we use the same strategy of convolutional layers with fixed parameters to get a flat input for the feedforward networks, the weights of which we are optimizing. Since the input is so large, the feedforward network can only have a single hidden layer of size 12 to get a similar number of adjustable parameters as in the main approach. This

results in a network with 7,827 parameters, including weights and biases. The activation function for all neurons in the network is the hyperbolic tangent function, except for two of the output neurons, which are activated by the sigmoid function.

5.3.2 Optimization Details

For parameter optimization, we use a combination of a Genetic Algorithm (GA) and CMA-ES (Hansen, 2006). More specifically, GA is used for the first 100 generations of the optimization. The GA used here searches the parameter space broadly with a larger population size than CMA-ES. The best solution found by the GA is then used as a good starting point for the CMA-ES algorithm to continue evolution. Starting with a GA search with a large sigma, was in preliminary experiments found to help the CMA-ES avoid getting stuck early at a local optimum. For both algorithms, we use off-the-shelf implementations provided by Ha (2017a) and Hansen (2006). For all experiments, the GA uses a population size of 512, and its mutations are drawn from a normal distribution $\mathcal{N}(0, 1)$. All other hyperparameters are the default parameters of the implementation. The large sigma means that the GA can cover a large area but in a coarse manner. For the CMA-ES algorithm, we use a population size of 128, and set weight decay to zero. Other than that, hyperparameters are the default parameters of the implementation. The total number of generations is 1,400 for the `Car Racing` environment and 4,000 for the `BipedalWalker` and `CartPoleSwingUp` environments.

For the large weight-optimized networks, CMA-ES becomes impractical to use due to its use of the covariance matrix of size N^2 with N being the number of parameters to optimize. For this reason, we instead use Evolution Strategy (sometimes referred to as "OpenES" (Ha, 2017b)) as described by Salimans et al. (2017) and. We use a population size of 128 and otherwise use the default parameters of the implementation (Ha, 2017b).

5.4 Results

Evaluations of the most successful runs of each experimental setting are summarized in Table 5.2. All experimental settings achieved good scores on the `CartPoleSwingUp` task. Only the weight-optimized network with hidden lay-

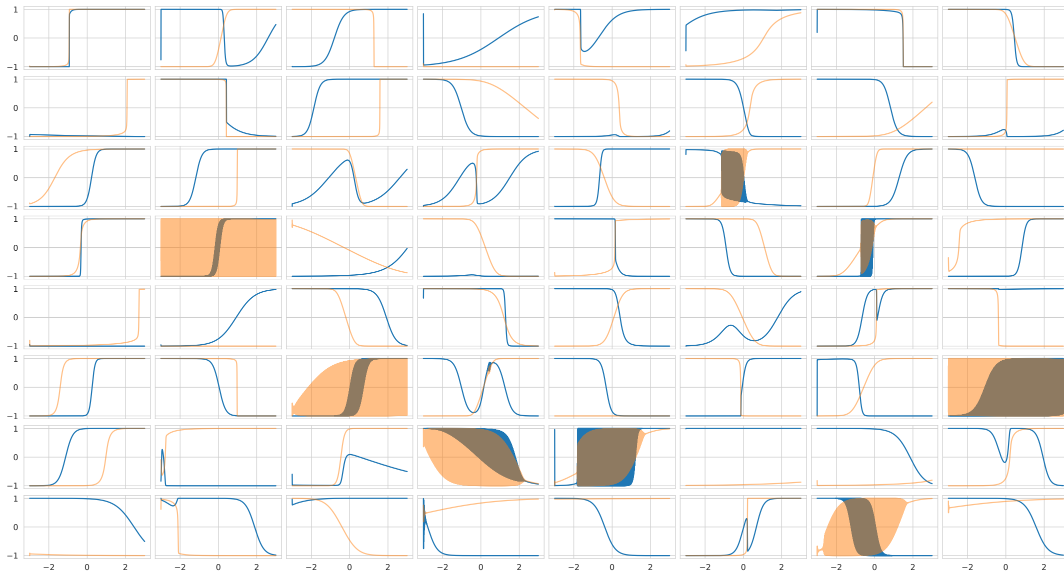


Figure 5.3: **Evolved Neural Diversity.** Displayed are each of the 64 activated neurons of the second hidden layer optimized to solve the `CarRacing` task. In blue are the activations that are passed on to the next layer. In orange are the neural states. One thousand inputs are given from -3 to 3 in an ordered manner, from most negative to most positive. Given the updatable neural state, the ordering of the inputs matter, and a different ordering would have yielded different plots. For all plots, the neural state is initialized as zero before the first input. Several of the found activations look like we would expect the hyperbolic tangent function with a bias and/or the possibility of a negative sign to look. Few functions seem unresponsive to the input. However, many functions are clearly both responsive and different from the standard form of the hyperbolic tangent function. We find functions that have oscillatory behavior in some or all of the input space. Other functions are non-monotonic and have peaks and valleys in particular areas.

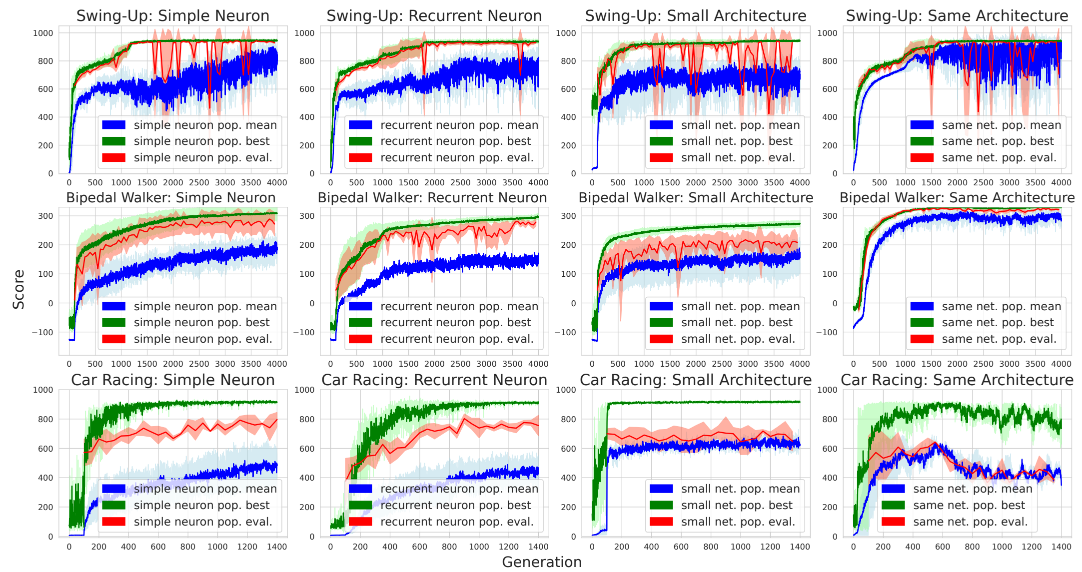


Figure 5.4: **Training curves for all experimental settings.** Means and standard deviations for populations of runs on different seeds. Each setting was run three times except for the weight-optimized models in the Car Racing environment, which were run twice. Every 50th generation, the current solution was evaluated 64 times (red).

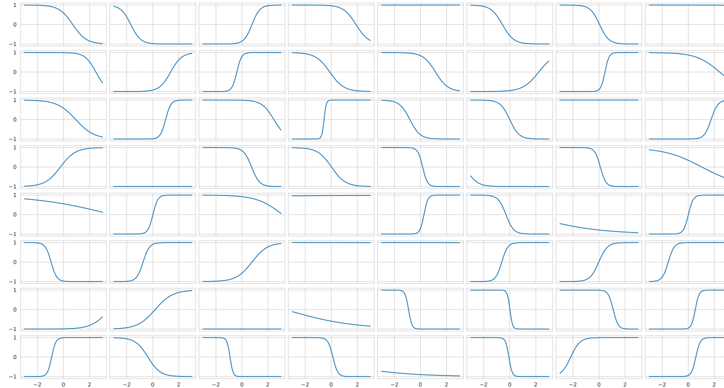


Figure 5.5: **Activations of Non-Recurrent Neurons.** Displayed are each of the 64 activations of the second hidden layer optimized to solve the `CarRacing` task. One thousand inputs are given from -3 to 3 in an ordered manner, from most negative to most positive. Neural activations are either monotonically increasing or decreasing, or unresponsive to the input.

ers of size 128 and 64 managed to average a score above 300 points over 100 episodes in the `BipedalWalker-v3` environment. However, the optimized recurrent neurons in a random network came close with just a fraction of the number of optimized parameters. The smaller weight-optimized network and the simple neurons achieved similar scores of 235 and 239, respectively. This is indicative of the agent having learned to walk to the end of the level in most cases but in an inefficient manner.

In the `CarRacing-v3` environment, the agent based on recurrent neurons scored the highest, though none of the approaches reached an average score above 900 over 100 episodes, which is needed for the task to be considered solved. However, with a mean score above 800, the agent was able to successfully complete the majority of the procedurally generated test episodes. Training curves for all experimental settings can be found in Figure 5.4.

5.4.1 Investigating Evolved Neurons

To gain a better idea of how a neural network with random weights but optimized neuro-centric parameters is solving the task, we plotted all activated neurons of a layer of the champion network (Fig. 5.3) of the `CarRacing` environment. The figure shows that while several of the found activations look

Table 5.2: Table of Results. Means and standard deviations over 100 episodes, and the number of parameters optimized for each experimental setting. Scores are evaluated with the most successful run in each setting. For context, results from Weight Agnostic Neural Networks (WANN) (Gaier and Ha, 2019) are included as another method that does not optimize weights. The number of parameters listed for WANN is the final number of connections in the evolved structure.

	Model	Score	# Param.
`CartPoleSwingUp`	Simple Neurons	892 ± 177	792
	Rec. Neurons	916 ± 79	1,782
	Small FFNN	**927** ± 83	1,889
	Same FFNN	922 ± 73	9,089
	WANN	732 ± 16	52
`BipedalWalker`	Simple Neurons	239 ± 52	880
	Rec. Neurons	295 ± 63	1,980
	Small FFNN	235 ± 13	1,988
	Same FFNN	**318** ± 46	11,716
	WANN	261 ± 58	210
`CarRacing`	Simple Neurons	820 ± 118	3,372
	Rec. Neurons	**822** ± 74	7,587
	Small FFNN	770 ± 167	7,827
	Same FFNN	752 ± 171	91,523
	WANN	608 ± 161	245

like the standard form we would expect from a hyperbolic tangent function with a bias, many of the other types of functions also emerged after optimization. We see functions with strong oscillatory behaviors, some in the whole input space, and some only in smaller sections. Other functions have extra peaks and valleys compared to the standard hyperbolic tangent. We hypothesize that this property allows each neuron to respond with more nuance to its input. Additionally, given the same inputs, this collection of neurons responds diversely. As seen in Table 5.2, this evolved a diversity of neural computations within a layer allowed the agents to perform well, even though information between layers is projected randomly. For a similar depiction of the activations of the simple, non-recurrent neurons, see Figure 5.5.

5.4.2 Comparison to Weight Agnostic Neural Networks

An approach similar in spirit to ours is the weight agnostic neural network (WANN) approach by Gaier and Ha (2019). As detailed in Section 5.1, in WANNs, only the architecture of the neural network is learned (including choosing an activation function from a predefined set for each neuron) while avoiding weight training. While an apples-to-apples comparison is not possible (due to different optimization algorithms), it is nevertheless interesting to see how these two methods compare in terms of performance. Since connections are added to the WANN models during optimization, we cannot directly compare the number of parameters that were optimized in these models to that of the neural units. In Table 5.2, we simply list the final number of synapses in the evolved network structures reported by Gaier and Ha, to give an idea of the network sizes. The optimized neurons tend to score better in all three environments. These results suggest that it might be easier to optimize customizable neural units for each position in a fully connected network than it is to learn a network structure from scratch.

In the future, these approaches could be complementary. We imagine that extending the WANN approach with more expressive neurons could allow their evolved neural architectures to become significantly more compact and higher performing.

5.5 Discussion and Future Work

In this chapter, we introduced an approach to optimize parameterized, stateful neurons. Training these alone yielded neural networks that can control agents in the `CartPoleSwingUp`, `CarRacing`, and `BipedalWalker` environments, even when the weights of the network were never optimized. While optimizing small neural units alone is unlikely to beat state-of-the-art methods on complicated tasks, the neuro-centric optimization alone did enable meaningful behavioral changes in the agents. We find these results encouraging, as they pave the way for interesting future studies.

The weight-optimized networks achieved superior scores compared to the neural units in the `CartPoleSwingUp` and `BipedalWalker` environments. This is not surprising; weight optimization of ANNs now has a long history of success in a plethora of domains. When using random transformations, there is a risk of getting a degraded signal, something that can be compensated for easily by tuning the weights of the transformations. Surprisingly, the optimized neural units achieved the best score of the experimental settings in the `CarRacing` task. The failure of the larger weight-optimized network to perform well here might be explained by the relatively low population size compared to the number of parameters being optimized (128, and 91,523, respectively). This population size was the same for all experimental settings to ensure that all models were evaluated the same number of times in the environments during optimization. The advantage of having a smaller number of adjustable parameters also came into display in that the larger models could not be optimized by the more powerful CMA-ES method.

As part of the proposed parameterized neurons, we included a persistent neural state that is fed back to the input of the neuron at the subsequent time step. This endows the network with a memory mechanism. Memory as local neural states is unusual in ANNs but is much more common in more the biologically inspired Spiking Neural Networks (SNNs) (Tavanaei et al., 2019; Pfeiffer and Pfeil, 2018; Izhikevich, 2006). Such a neural state is most useful for data with a temporal element, such as agents acting in an environment. It is reasonable to assume that the same approach would have limited use in tasks with unordered data. However, for RL tasks, stateful neurons provide a relatively inexpensive way of allowing the network to have some memory capacity. Setting up more common recurrent neural networks (RNNs), like LSTMs (Hochreiter and Schmidhuber, 1997) or GRUs (Cho et al., 2014), for the tasks used in this chapter, would result in the need for many more

adjustable parameters than the number of parameters in the neural units optimized here. Combining stateful neurons with more commonplace RNNs could result in interesting memory dynamics on different timescales.

A simpler version of the parameterized neurons with no recurrent state was also tested. Examples of activations of these neurons can be found in 5.5. While a variety of activation curves are displayed, they are all limited to monotonically increasing or decreasing along the x-axis. However, even the more simple representation of neurons was able to be optimized within a randomly connected network to get relatively high scores - though lower than the recurrent neurons - in all tasks. Especially surprising was the performance in the `CarRacing` environment, which was close to the score achieved by the recurrent neurons. While memory capacity might be an advantage, it does not seem necessary to perform relatively well in the chosen RL tasks. Note, that it is straightforward to incorporate different or more information into the neural units. Interesting examples of additional information could be reward information from the previous time step or the average activation value of the layer at the previous time step in order to add some lateral information to the neural activation.

The ability to improve performance while leaving weights random opens up the possibility for future work of combining neural units like the ones proposed here with the approach of masking weights (Frankle and Carbin, 2018) mentioned in Section 5.1. An advantage of using masks is that one can train masks for different tasks on the same random network (Wortsman et al., 2020). With pre-trained neural representations, it should be possible to bias the random network to perform actions that are generally useful within a specific task distribution. One could then train masks on top of the untouched weights to perform well on specific tasks, conceivably more efficiently than with generic activation functions.

Another potentially interesting avenue for future work is to combine the optimization of neurons with synaptic plasticity functions (Soltoggio et al., 2018). A lot of work has been done in the area of learning useful Hebbian-like learning rules (Chalmers, 1991; Miconi, 2016; Mouret and Tonelli, 2014; Floreano and Mattiussi, 2008; Najarro and Risi, 2020; Risi and Stanley, 2012b; Soltoggio et al., 2008; Tonelli and Mouret, 2013; Wang et al., 2019; Chalvidal et al., 2022; Pedersen and Risi, 2021). Less work in the field has explored the interaction between learning rules and neural activation in ANNs, despite the fact that most learning rules take neural activations as inputs. It seems likely that more expressive neural units would in turn result in more

expressive updates of weights via activity-dependent learning, and thus more powerful plastic neural networks.

Having shown that the proposed neural units can achieve well-performing networks when optimized alone, future experiments will explore the co-evolution of neural and synaptic parameters. It will be interesting to see whether a synergistic effect arises between these two sets of parameters. If both weights and neurons are optimized together, will there be as much diversity in the resulting set of neural units, or will the need for diversity decrease?

The work presented in this chapter demonstrates yet another way to extend ANNs, which are still far from their biological counterparts in countless aspects. While the work presented here does not claim to have presented a biologically plausible approach, we do believe that inspiration from biological intelligence still offers great opportunities to explore new variations of ANNs that can ultimately lead to interesting and useful results. This perspective is discussed in more detail in Chapter 9. Before that, a method for evolving a reward signal for a reinforcement learning agent is presented in the next chapter.

Chapter 6

Evolution of an Internal Reward Function for Reinforcement Learning

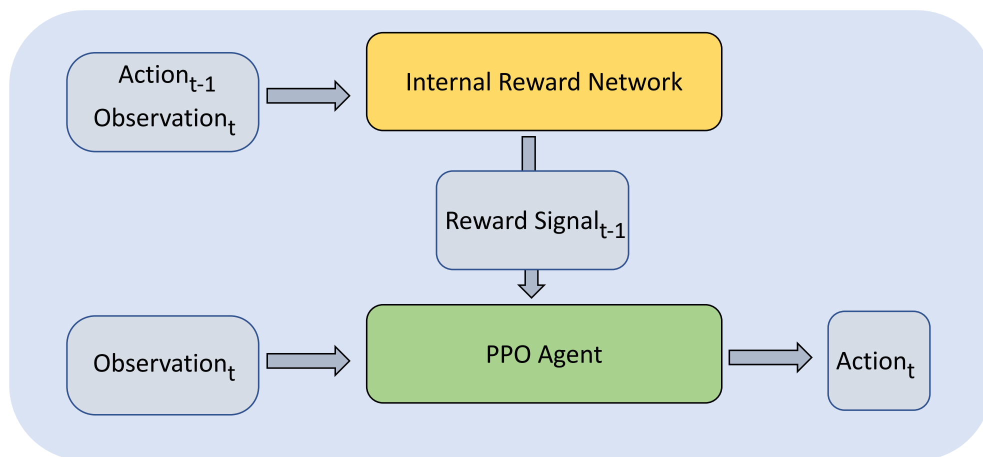


Figure 6.1: **Overview of EIR-RL.** The Internal Reward Network takes inputs that are readily available to the PPO agent and outputs a reward signal that is used to update the parameters of the networks of the PPO.

Whereas the previous two chapters focused on experiments concerning mechanisms internal to the policy networks, the main focus of this chapter is on evolving a network that can support the learning process of a separate policy network.

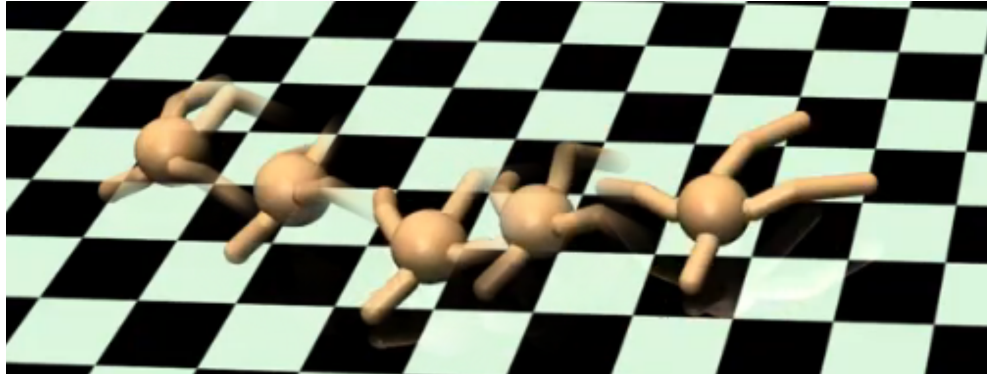


Figure 6.2: **Quadruped robot** having learned to walk with a severe shortening of its legs using rewards from an internal reward network.

Research has shown that ANNs can be trained to control robots using reinforcement learning (RL) (Kober et al., 2013). Training ANNs with RL has two requirements: lots of example trajectories to train the ANNs on and a useful reward signal that leads to desired behavior when maximized. Since these requirements can be met through several modern simulators, most training of ANNs to control robots happens in simulations (although see (Wu et al., 2022) and (Smith et al., 2022)).

However, in dynamic environments, there is a risk that environmental circumstances change over time and in unpredictable ways. Policies learned with reinforcement learning are notoriously brittle, meaning that even small changes in the environment can lead to failures (Rajeswaran et al., 2017; Zhang et al., 2018a; Song et al., 2019b) and the need for more training (Julian et al., 2020). Unfortunately, as also highlighted by Smith et al. (2022), the reward signal that was used during the simulation might not be available to use after training, e.g., in the real world. For this reason, it would be useful for the agent to be able to estimate a reward signal that it can learn from, using only information that is always available to it.

With this goal in mind, we introduce Evolved Internal Reward Reinforcement Learning (EIR-RL). As the name suggests, we evolve a function that creates a signal that can be used to optimize an RL agent. As long as the original goal stays the same, the agent can optimize its parameters indefinitely, and recover performance even in the face of a distributional shift. The evolved function can do this without relying on information beyond the inputs and outputs of the agent. In the proposed approach, a simple re-

current neural network (RNN) (Elman, 1990; Cheng et al., 2002) is evolved to provide a reward signal that can be used to optimize an RL agent. We test the EIR-RL approach on various reinforcement learning tasks of differing difficulty and reward structures. After optimization, EIR-RL is found to produce enhanced training speed and training stability of the RL agent, as well as the ability to recover performance in a changing environment without the need to refer to a reward signal granted by the environment.

Small changes in an individual's morphology or environmental conditions might render a specific solution of how to achieve a goal obsolete. However, with the ability to assess whether the results of one's own actions are desirable or not comes the possibility of improving maladaptive behavior, resulting in an overall more generally capable agent.

6.1 Related Work

Inverse reinforcement learning. This is a class of algorithms that, given some operational trajectories (i.e., expert trajectories), attempts to infer the reward function of the Markov Decision Process, so that the agents can learn how to solve problems where no reward function is initially available (Arora and Doshi, 2021). Since the reward function is naturally more robust and transferable than the policy (Abbeel and Ng, 2004), some studies use inverse reinforcement learning to obtain the reward function and achieve greater generalizability (Munzer et al., 2015; Melo and Lopes, 2010; Finn et al., 2016; Metelli et al., 2021).

As will be demonstrated in the experiments below, EIR-RL also enables agents to obtain better generalizability through the reward function. However, the reward function is obtained through evolution, not a priori by a policy or some operational demonstration. On the other hand, in order to evolve EIR-RL, an external reward from the environment is needed initially.

Continual reinforcement learning. This class of methods studies models that learn in a series of tasks. Generally, requirements for the continual reinforcement learning model include: (1) the model can learn incrementally without a fixed training set; (2) the tasks learned in the past can help the model learn better in the future (i.e., forward transfer); (3) the model will not forget the tasks learned in the past and learning new tasks should ideally improve the performance of the past tasks (i.e., backward transfer); and (4) the model can effectively adapt to changes in the environment and

recover quickly (Khetarpal et al., 2020; Hadsell et al., 2020). The EIR-RL has some characteristics of continual reinforcement learning because the internal reward network continually trains the policy when the agent interacts with the environment.

In the field of continual reinforcement learning, studies have developed different training and testing methods for demonstrating continual learning (Xie et al., 2020; Mendez et al., 2020; Huang et al., 2021). In this chapter, we did not design corresponding tests for different aspects of continuous learning but focused on testing the generalizability of EIR-RL for out-of-distribution (OOD) tasks without access to environmental rewards.

Intrinsic reward-based meta-reinforcement learning (Zheng et al., 2018, 2020; Stadie et al., 2020). These methods use a similar concept called *intrinsic rewards* and usually require second-order gradient descent in meta-training. Similar to some other meta-reinforcement learning methods for optimizing parameters (Duan et al., 2016; Xu et al., 2018; Zhou et al., 2020b), the goals of these methods are to achieve enhanced final performance (Stadie et al., 2020) or to obtain faster training speed for new tasks based on the meta-training results (Zheng et al., 2018, 2020). EIR-RL, on the other hand, meta-learns an internal reward that serves as a substitute for external rewards, with the potential added benefits of increased training speed and stability.

Evolved Loss Functions. Closely related to our approach are methods that meta-learn losses or gradients. Houthooft et al. (Houthooft et al., 2018) introduced an approach named Evolved Policy Gradients (EPG). This research carries out the evolution strategy in the outer loop of meta-learning as well. Therefore, when testing in new scenarios, EPG can theoretically get rid of extrinsic rewards and train based on the objective obtained in meta-training. EPG directly evolves strategy losses rather than internal rewards as inputs to an RL agent. To calculate the policy loss, the EPG employs an intricate setup, using convolution over a fixed number of steps to create context vectors. Context vectors, along with memory units updated by stochastic gradient descent, as well as experience vectors of the agent are used as inputs for a dense feedforward neural network to generate policy losses (Houthooft et al., 2018). To generate internal rewards, EIR-RL uses a simple RNN that takes state-action-pairs as input. In a similar vein of research, Gonzales and Mikkulainen (Gonzalez and Miikkulainen, 2022) were able to demonstrate theoretically that their TaylorGLO method (Gonzalez and Miikkulainen, 2021) for meta-learning loss functions with evolution acted

as a regularizer that prevented overfitting.

6.2 Approach: Evolving Internal Reward for Reinforcement Learning

The goal of EIR-RL is to obtain a function that can produce a reward signal that can be used for RL using only information that is directly available for the RL agent. We achieve this by optimizing the parameters of a neural network. We call this the Internal Reward (IR) network.

As input, the IR network takes the same observation as the policy network of the RL agent, as well as the resulting output of the policy network. The IR network outputs a single scalar that is used for optimizing the policy using Proximate Policy Optimization (PPO) (Schulman et al., 2017) instead of the reward scalar provided by the environment. Our approach thus requires two optimization processes: an inner- and an outer loop, as described in Algorithm 2, where 'ES' denotes evolution strategy, 'IRN' denotes internal reward network, 'ENV' denotes environment, 'ir' and 'er' denotes internal and extrinsic reward, respectively.

We use an evolution strategy (see Section 6.2.1) for the outer loop optimization of the IR network. In each generation, a population of IR networks is generated. The fitness of each IR individual is determined by evaluating the score of a policy that was optimized using the IR output in place of the reward provided by the environment. Importantly, the fitness is based on the true score in the environment, not the evolved reward signal. The parameters of the IR network are updated through evolution and stay fixed during the PPO training phase. From the point of view of evolution, each individual is born with a reward system in the form of the IR network and a learning system in the form of the PPO. Together, these innate systems provide a way of shaping a random policy into a functioning one. The fittest individuals will be the ones that have the reward system most suited for learning under the constraints of an objective function determined by the external environment.

6.2.1 Optimization

For the outer loop optimization of the IR network parameters, we use CMA-ES (Hansen et al., 2003; Hansen, 2006). For these experiments, a particular

Algorithm 2 Meta-training of EIR-RL

```
Initialize ES
for epoch = 1,··· do                                        ▷ Outer Loop
    for agent = 1, ··· , population_size do                 ▷ Inner Loop
        Sample vector ε from ES
        Initialize IRN parameters with ε
        Initialize POLICY network parameter φ randomly
        Sample ENV from task distribution
        Initialize state from ENV
        for t = 1,···, max_t do
            action = POLICY(state)                          ▷ Agent part
            ir = IRN(action, state)
            POLICY.update(ir)                               ▷ Based on Eq.1
            er, state = ENV(action)                         ▷ Environment part
        end for
        fitness = SUM(er) for each t
    end for
    Update ES by the fitness of each agent
end for
```

advantage of CMA-ES is the reduced reliance on a large population size in order to work well (Hansen, 2016). In our experiments, the IR networks are all small-sized neural networks with a relatively low number of parameters, making CMA-ES a favorable gradient-free candidate for the outer loop optimization of EIR-RL.

The fitness of an IR network is the performance of a policy that has been optimized using the IR signals as a reward. As mentioned above, the RL algorithm used for maximizing the IR signal was Proximal Policy Optimization (PPO) (Schulman et al., 2017). Different from the general PPO, the output of the internal reward network replaces the original extrinsic reward at each time step to train PPO (as shown in Fig. 6.1). In this case, the loss gradient of the policy is expressed as:

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T} \nabla_\theta \log\pi_\theta(a_t|s_t)(R_\varepsilon(a_{t+1}, s_{t+1}) + \gamma V_\upsilon(s_{t+1}) - V_\upsilon(s_t)) \tag{6.1}$$

In this loss, there are three functions parameterized by neural networks: internal reward network R_ε, actor network π_θ, and value network V_υ. The three networks assume orthogonal functions. The actor network outputs the actions of policy; the value network outputs the expected return under the current strategy; the internal reward network outputs estimated policy-independent rewards from the Markov Decision Process. Actor and value networks are updated based on the loss by using the PPO method (Schulman et al., 2017).

6.3 Experiments

6.3.1 Environments

For testing our approach, we used three different environments the OpenAI Gym. The environments described below represent various levels of difficulty to solve. Of special interest to our approach, these also provide different reward structures to learn from, each of which EIR-RL must be evolved to substitute. All experiments share the basic approach outlined in Algorithm 2

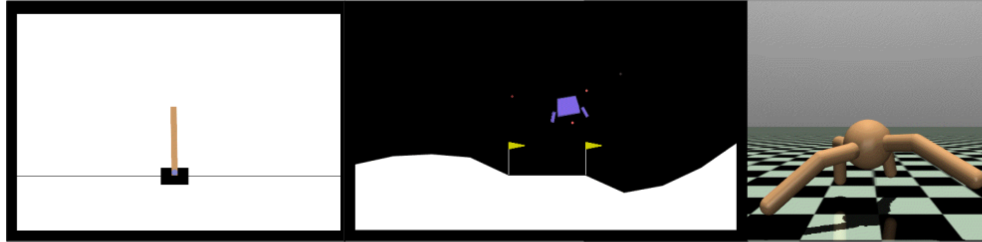


Figure 6.3: **Environments.** From left to right: `CartPole-v0`, `LunarLander-v2`, `Ant-v3`.

6.3.1.1 CartPole-v0

In this control task, a pole needs to be balanced on a cart. The task of the agent is to control the cart such that the pole does not fall (Barto et al., 1983). The environment has four inputs and two discrete actions. The reward returned by this environment is 1.0 for each time step that the pole is being balanced and 0.0 if it falls below a threshold and the balance has been lost. If the pole falls, the episode ends and no more rewards can be collected. If the agent is successful in balancing the pole, the episode ends after 200 time steps, and the agent thus achieves the maximum score of 200.

For the evolution strategy, the population is set to 80 and the outer loop is run for 150 generations. In the inner loop, each PPO has its value- and policy networks randomly initialized and is optimized for 100 episodes. The evolved internal reward is used for PPO training, and the score in terms of the original reward from the environment is used to determine the fitness given to the CMA-ES. The parameters of the internal reward network are initialized by CMA-ES and fixed during the inner loop.

At each time step, the policy network in EIR-RL receives the environment state vector of four elements and outputs the direction in which to move the cart (left/right). Meanwhile, the internal reward network also receives the state vector and outputs an internal reward signal. The PPO is updated every 256th time step, using a batch size of 32. For this simpler environment, the internal reward network is a small feedforward network of four layers (two hidden) with {4, 16, 8, 1} nodes per layer. This network has 225 parameters, which is thus the number of parameters optimized by the evolution strategy. The PPO consists of the actor network with {4, 32, 16, 1} nodes per layer and the critic network with {4, 16, 8, 1} nodes per layer. The hyperbolic tangent function is used to activate all hidden neurons.

6.3.1.2 LunarLander-v2

In the `LunarLander-v2` environment, the goal is to safely guide a small rocket ship to land within the correct landing grounds. Each new episode is procedurally generated with changing terrain. The agent receives an eight-dimensional vector as input from the environment and has to output one of four discrete actions to steer the rocket ship. This environment was chosen due to the particular reward structure provided by the environment. While there is a continuous reward signal that rewards the agent for steering closer to the landing pad, some events result in non-continuous reward signals. For example, crashing the rocket will result in a one-time penalty of -100 points. Crashing the rocket also results in ending the episode, so that no more points can be collected. Landing safely with the legs on the ground results in a one-time bonus of 10 points for each leg landed. Landing within the landing pad with both legs results in a bonus of 100 points. Due to this mix of continuous and sparse reward signals, this environment presents an interesting challenge for the EIR-RL method.

For the outer loop optimization, a population size of 64 is used, and optimization runs for 500 generations. The internal reward network is an RNN with {9,32,32,32,1} nodes per layer, where the last layer of size 32 is a non-recurrent dense layer to reduce the output to a single scalar, resulting in $3,521$ parameters to be optimized by CMA-ES in the outer loop. As input, the IR network took the observation from the environment as well as the discrete action of the policy network from the previous time step. The fitness of an IR network was determined after 100,000 time steps in the environment. The PPO updated policy- and critic parameters every 1024 steps, with a batch size of 32. Both the policy network and the critic network of the PPO had two hidden layers of 64 neurons activated by ReLU.

6.3.1.3 Ant-v3

This locomotion environment revolves around training the agent to control a simulated three-dimensional robot with four legs. The goal is to get the robot to walk as far as possible in a straight line. At each time step, the agent is rewarded for progress toward the goal direction, and for not falling. Excessive energy expenditure is penalized at each time step. If the robot falls, the episode ends. Otherwise, the episode ends after 1000 time steps. The input to the agent is a vector of 27 elements, containing proprioceptive

information about the robot. At each time step, the agent needs to output a vector of eight elements, describing the torque that is to be applied to each of the eight rotors of the robot, respectively.

In the `Ant-v3` environment, we run the training in two different scenarios. The first is simply the default setting of this environment. In the second scenario, one of the legs of the robot is sometimes shortened. Specifically, in each new episode, there is a probability 20% that no change is applied to the leg. The rest of the time, the length of the leg is sampled to be between 0.01 and 0.4 with uniform probability, where 0.4 is the original leg length. See Figure 6.2 for a robot with one leg shortened to 0.01. Only the third leg of the robot is selected for reduction, whereas all other legs are always maintained at the standard length. Earlier studies have shown that shortening a single leg of a quadruped by approximately 10% can significantly decrease the performance of a policy optimized with standard leg lengths only (Pedersen and Risi, 2021). In general, increasing the distribution of tasks during meta-training can improve the generalizability of a policy (Kirsch et al., 2022; Feng et al., 2022), so here we test whether the same is true for an evolved reward function. Apart from the leg-shortening in one scenario, all settings of the training are identical between the two scenarios.

The population for CMA-ES is set to 80, and optimization is run for 1,500 generations. In the inner loop, the PPO agent is allowed 500,000 time steps (at least 200 episodes) in this environment. The PPO is updated every 2048 time steps based on internal rewards The internal reward network is an RNN with {35,32,32,32,1} nodes per layer, where the last layer of size 32 is a non-recurrent dense layer to reduce the output to a single scalar. Thus a total of $4,353$ parameters are optimized by CMA-ES. The PPO consists of the policy feedforward network with {27,32,32,8} nodes per layer and the critic feedforward network with {27, 32, 16, 1} nodes per layer. All hidden neurons are activated by the hyperbolic tangent function.

In this more challenging environment, we pre-train the IR network before optimizing it in the outer loop. We do this by training a single PPO to perform well in the `Ant-v3` environment. Throughout this process, we collect the observations, actions, and external rewards for each time step. With this data set, the pre-training process is simply to optimize the IR network via gradient descent on the supervised regression problem with observations and actions as the input and the reward as the target. Preliminary experiments showed that the pre-trained internal reward network has a similar performance to environmental reward when training a new PPO network.

Finally, we used the parameters of the pre-trained internal reward network as the initial parameters of the evolutionary strategy. Another option to ease the optimization of the IR network was used by Houthooft et al. (Houthooft et al., 2018). To bootstrap the learning process, they initially used a combination of an external and an internal reward signal. Throughout the evolution, they gradually phased out the external part of the signal and in the end relied exclusively on the evolved reward signal. We tested a similar approach in preliminary experiments but found the pre-training with supervised learning to yield better results.

6.4 Results

Here we show the results of evaluating the optimized IR networks in different environments. We evaluate models on default environmental settings that were used for training, as well as altered settings novel to trained IR networks. In the plots below, 'Oracle' refers to a PPO trained with the reward provided by the environment. This is to emphasize that the primary motivation of EIR-RL is to enable optimization in the absence of externally provided reward signals. PPOs trained with either reward signal have the same specifications in terms of network sizes and other hyperparameters.

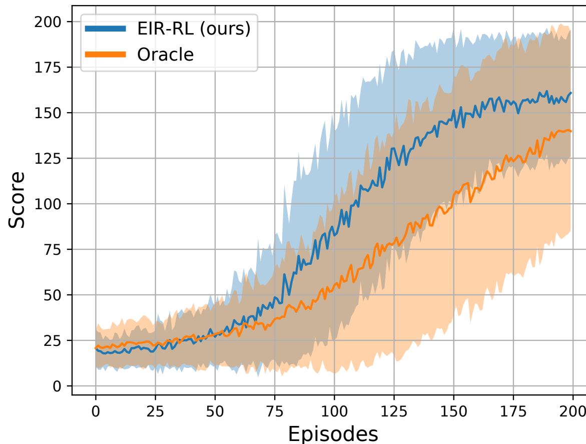


Figure 6.4: **Cart Pole: PPO Training Curves.** Comparing EIR-RL without access to environmental reward to training with access to environmental reward (Oracle) on the default `CartPole-v0` task. Means and standard deviations are calculated over 100 training runs. Training with EIR-RL results in faster improvements.

6.4.1 Cart Pole

For testing, we use the trained parameters for the internal reward network and random initializations for PPO networks. For comparison, we also train PPOs with the only difference being they are trained with extrinsic environmental rewards. Fig. 6.4 shows the training curves of using each reward signal on the default `CartPole-v0` task. The means and standard deviations in the plots are calculated from 100 PPO optimization runs, each lasting for 200 episodes. As can be seen, not only is it possible to optimize the PPO without any external reward, but compared to 'Oracle', optimizing with EIR-RL results in faster improvements of the policy.

We also test the evolved IR network's ability to provide an adequate reward signal under out-of-distribution environmental settings. In Fig. 6.5, training curves are shown for when the length of the pole is changed from 0.5 to 3.0. Once again, the means and standard deviations in the plots are calculated from 100 PPO optimization runs. While training with either reward signal eventually converges to similar scores, initial improvements happen quicker when using the internal reward network compared to the external environmental reward.

In Fig. 6.6, we plot the internal reward signal as well as each of the input values from the environment at each time step of a successful episode. The observation space includes cart position, cart velocity, pole angle, and pole angular velocity. Qualitatively, the evolved reward signal and the absolute value of the pole angular velocity seem to be highly inversely correlated. In contrast, the environmental reward is always 1.0, except for the last time step in the case of an unsuccessful episode. The IR network's reward signal thus provides more informative feedback to the agent based on the current observational state, which is likely the cause of the increased learning speed, seen in Fig. 6.4 and 6.5.

6.4.2 Lunar Lander

After the optimization of the IR network on the `LunarLander-v2` environment, we first compared PPO training with the reward given by the environment, and the reward from the IR network. The use of either reward signal did result in policies that could score above 200 in the environment. However, as can be seen in Figure 6.7, only the PPOs trained with the evolved reward signal were able to achieve and sustain a high moving average over

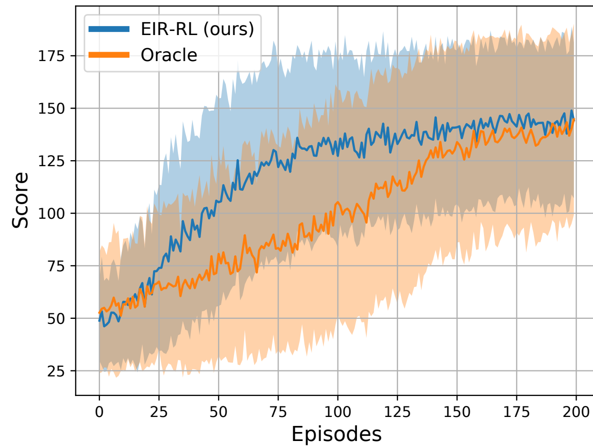


Figure 6.5: **Cart Pole: PPO Training Curves.** Training from scratch after the pole length is changed from 0.5 to 3.0 as an out-of-distribution task to test the performance of EIR-RL and 'Oracle'. Means and standard deviations are calculated over 100 training runs. Once again, initial improvements are faster with EIR-RL.

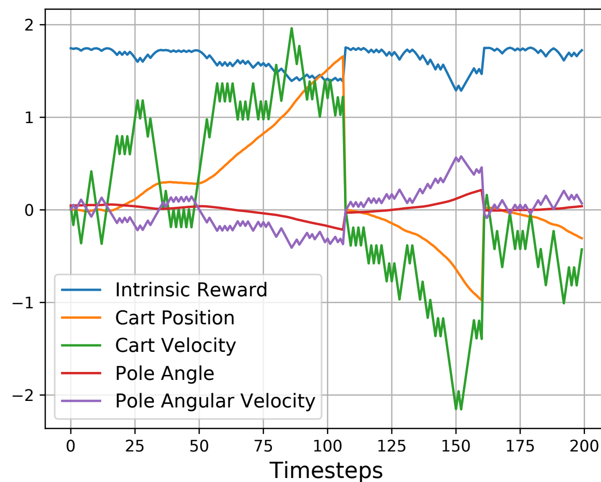


Figure 6.6: **Inputs and evolved reward signal over a successful Cart Pole episode.** The evolved reward seems to have a strong inverse correlation with the angular velocity of the pole.

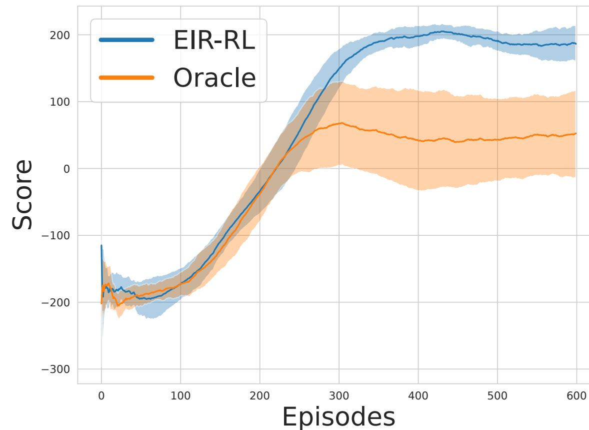


Figure 6.7: **Lunar Lander: PPO Training Curves.** Means and standard deviations over 10 different PPO runs. Plotted is the moving average of the objective (non-evolved) scores of the latest 100 episodes. While in both cases, the training leads to models that can achieve objective scores above 200, training with the evolved reward leads to better training stability that maintains high scores with continued training.

the latest 100 episodes, indicating improved training stability.

In Figure 6.8, the PPO is first optimized for 400 episodes in the default environmental settings as in Figure 6.7. Then, in the following 800 episodes, a strong wind (wind level 20 is the strongest possible in `LunarLander-v2`) is applied in the environment, making it harder to steer the rocket ship. The performance of EIR-RL initially suffers, but then gradually improves and is consistently well above a score of 100, which indicates that the rocket ship is not crashing. Once again, the PPO trained with the original rewards from the environment suffers from training instability and fails to sustain high performance. The curve labeled ”No Extra Training” shows how an agent trained only in the first 400 episodes performs when the wind is applied. As can be seen, the continued training with the original rewards from the environment results in a training curve similar to simply not optimizing anymore.

Figure 6.8: **Lunar Lander: PPO Training Curves.** Means and standard deviations over 5 different PPO runs. Plotted is the moving average of the objective (non-evolved) scores of the latest 20 episodes. The EIR-RL is able to recover most of the performance when strong wind is applied.

6.4.3 Quadruped

Figure 6.9 shows the PPO training curves using EIR-RL under both training scenarios described in section 6.3.1.3, as well as using the original reward from the environment. The means and standard deviations are calculated from 10 PPO runs for each curve. Curves of either of the EIR-RL training scenarios show a rapid increase in performance compared to training with the original external reward.

Figure 6.10 shows training curves under different out-of-distribution scenarios. In all cases, the PPO was first trained to perform well on the default task, and the plots show attempts to recover performance when the environmental setting is changed afterward. Three OOD leg length scenarios were examined for each of the four legs of the robot: fixed at 0.35, fixed at 0.01, and gradually shortened from 0.35 to 0.01. The shown curves are the averaged curves over 10 different PPO runs.

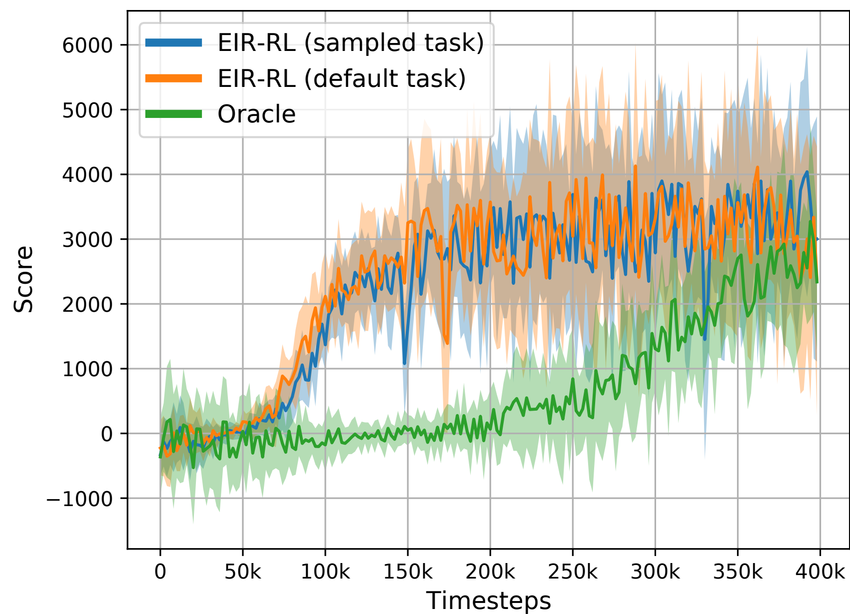


Figure 6.9: **Ant: PPO Training Curves.** The performance of EIR-RL evolved under default settings (orange) and sampled leg lengths (blue) in the `Ant-v3` environment, and policy network with access to environmental reward (Oracle) on the default `Ant-v3` task. Both versions of EIR-RL show significantly faster improvements compared to training with original rewards. Means and standard deviations are calculated from 10 training runs.

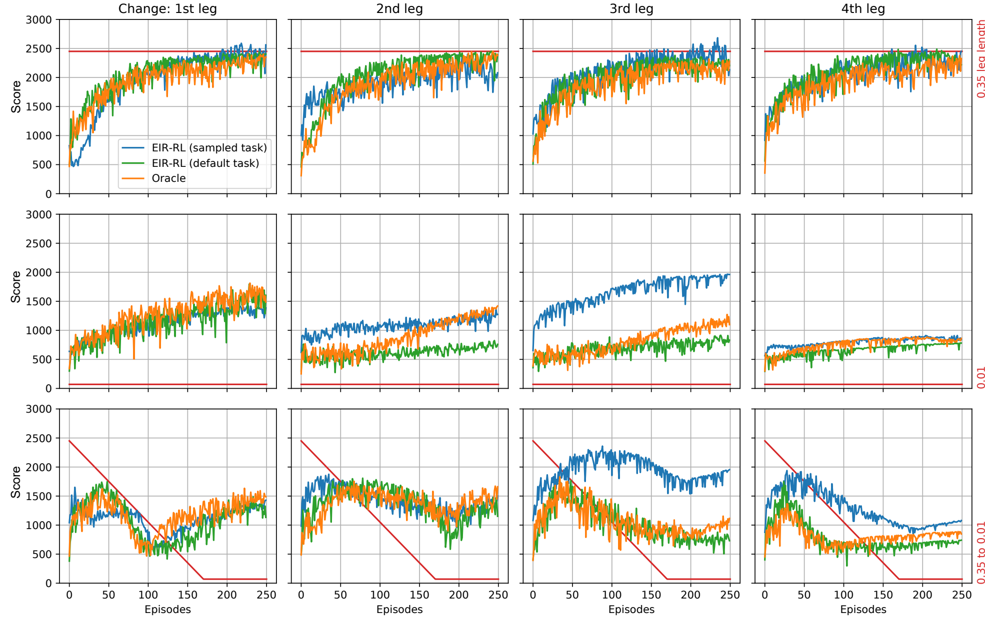


Figure 6.10: **Ant: PPO Training Curves.** Testing EIR-RL with different leg changes (indicated by the red lines) after an initial training phase under default settings. The IR network trained with changes to the third leg showed an advantage in the same setting. Otherwise, training curves are similar across the three different reward signals.

As can be seen, the EIR-RL trained with different sampled leg lengths (blue curves) of the third leg performs better when the third leg is changed during testing. The EIR-RL trained with sampled leg lengths has better performance only in one other scenario and that is when the fourth leg of the quadruped robot is gradually shortened from 0.35 to 0.01. However, the three methods have similar performance overall. Notably, no disadvantage in terms of training speed or performance can be detected in any of the scenarios - including the default settings in Figure 6.9 - when using the EIR-RL trained with sampled leg lengths.

In short, for new scenarios within the meta-training distribution, the training speed of EIR-RL is faster than that of 'Oracle'. In the out-of-distribution scenarios, the training speeds of EIR-RL and 'Oracle' are similar.

6.5 Discussion

In this chapter, we demonstrated several advantages of EIR-RL. The most obvious use case for the evolved reward function is in the case where additional training is needed, but no other reward signal is available. In this case, EIR-RL makes the additional training possible, even when circumstances differ from those under which the internal reward function was evolved. This robustness of the reward function to OOD situations compared to the lack of robustness often associated with policies learned in the same environments is likely due to the fact that every optimization step of the reward function in the outer loop comprises hundreds to thousands of policy optimization steps. This long time horizon makes the reward function unlikely to overfit to overly specific relations between observations, actions, and external rewards. For example, there are many different observation-action-pairs that would lead to similar rewards in the `Ant-v3` environment, and during the course of training an agent from scratch a plethora of such pairs are likely to have manifested at some point.

We also observe that the internal reward function can have advantages over the original reward signal provided by the simulation environments. These advantages seem to depend on the structure of the original rewards. In the `CartPole-v0` environment, using the evolved internal reward resulted in faster training of the PPO compared to the original reward signal. This is most likely due to the sparse reward structure of `CartPole-v0`, where the reward signal is always 1 except in the case of failures. The evolved reward function was able to produce a more instructive reward signal that in turn resulted in faster learning.

In the `LunarLander-v2`, training with the evolved internal reward signal resulted in enhanced training stability. Training with the original reward would yield functioning policies within the same number of episodes as training with the internal reward signal. However, sustained training with the original reward signal leads to large fluctuations in the training curve. The stable training curves when training internal rewards are likely due to how the fitness of individuals was evaluated for the evolution in the outer loop optimization. Each individual was evaluated on its performance after a fixed number of time steps in the environment (100,000 steps in `LunarLander-v2`). It is, in other words, not enough to have had a well-performing model at some point during this time frame. A successful individual would need to achieve a high performance before the 100,000th time step and sustain this perfor-

mance, thus implicitly selecting for training stability. Stability over sustained training is an important feature for models that potentially need indefinite training. If we cannot predict how the environment will change, it is unlikely that we can predict the timing of the changes. Instead, in order to deal with unforeseen distributional shifts it might be preferable to keep updating the model parameters to always be up-to-date without having to worry about performance decrease due to training instability.

In the `Ant-v3` environment, the evolved reward signals resulted in similar performance as the original reward under the default environmental settings, but with much faster improvements. Here, the faster improvements can again be attributed to the fact that the IR network was evaluated after a fixed number of time steps (500,000) in the environment. This created pressure to as quickly as possible make improvements that remained stable with sustained training. Further, the evolved internal reward has the advantage that it can be exposed to different settings during its evolution. Including a shortened leg in the training during evolution resulted in enhanced training compared to training with the original reward when the same leg was changed during meta-testing. This benefit was specific to that particular leg and did not lead to enhanced training compared to the original reward when other legs were changed during meta-testing. Importantly, the benefit in the additional scenario did not come at the expense of performance or training speed, nor did it require additional outer loop optimization of the IR network.

The advantages of the EIR-RL approach come at some cost. The evolution of the IR networks is a time-consuming process. This is because evaluating the fitness of a single individual involves optimizing a PPO agent from scratch for many time steps in the environment. Further, a limitation of the EIR-RL approach is that while EIR-RL can be used when a new behavior is required to reach the same goal, the approach only works as long as the original goal stays fixed. For example, in the `Ant-v3` environment, the IR network optimized in the experiments above can only reward behavior that moves the robot along the default goal-direction.

With these results and limitations in mind, we here propose directions for future studies. One promising approach might be to co-evolve the PPO parameters and the IR network in the outer loop optimizer using a diverse set of environmental settings (such as multiple leg changes, gravity settings, terrain variations, etc., in `Ant-v3`). In the current chapter, we confirmed that EIR-RL can be used for the optimization of randomly initialized PPOs at least as fast as training with the original reward, and most often faster. We

also demonstrated that when a leg-change was included in the training during the evolution of the IR network, the internal reward signal resulted in faster reinforcement learning for that setting than the original reward signal. By instead optimizing the PPO parameters in both the outer- and inner loop and focusing on adaptation with few gradient steps to many different settings, the approach would resemble the model-agnostic meta-learning (MAML) (Finn et al., 2017) framework, and other approaches leveraging the Baldwin effect (Fernando et al., 2018). Based on our findings in the experiments above, we expect that extending the MAML approach to include an evolved internal reward function would enhance both the training stability and the speed of adaptation compared to vanilla MAML.

In order to mitigate the EIR-RL's limitation when it comes to rewarding goals that are entirely different from each other (such as different goal-directions in `Ant-v3`), an interesting avenue of future research is to evolve IR networks in a goal-conditioned manner. By including a goal-embedding (Sukhbaatar et al., 2018; Liu et al., 2022; Islam et al., 2022) in the inputs of the IR network, a single function capable of rewarding behavior to separate objectives might be possible.

It is worth noting that even though the method introduced in this chapter is conceptually very different from evolving plasticity rules as in Chapter 4, these two approaches are similar in that the inner-loop parameters of both approaches are a function of the outer-loop parameters. This means that both EIR-RL and the evolution of Hebbian learning rules fall into the same category of meta-learning where an optimizer for the inner-loop is optimized. On the other hand, the plasticity rules and EIR-RL clearly deal with different aspects of the optimization of the inner-loop parameters. EIR-RL only generates a reward signal and relies on a different algorithm to make use of it to change the inner-loop parameters. In Chapter 4, no reward information was given to the plasticity rules. Given that *Evolve & Merge* and EIR-RL focus on different aspects, they could plausibly be combined. In this case, EIR-RL could be seen as a neuromodulatory function for the plasticity rules, which in turn would have to be extended with a term that weighs the generated reward signal as part of the synaptic update.

6.6 Conclusion

In this chapter, we presented an approach to evolve a reward function to use for reinforcement learning to replace the reward signal provided by the environment. The reward function only relied on information that was readily available to the agent during deployment after training. The main purpose of such an evolved function is to enable continued training after deployment, such that the agent can recover performance in the case the environment changes after the training phase.

In addition to enabling sustained policy updates after an initial training phase, we observed several advantages of training with the evolved internal reward signal. These observed benefits included faster training speed in default environments and improved training stability. When adding more environmental settings in the `Ant-v3` environment during the optimization of the internal reward network, an increased recovery of performance was observed if a similar change occurred during testing. This happened without showing any negative consequences in terms of recovery in settings unseen during training. This result is particularly encouraging, as potential environmental changes can be the reason for the need for extra training in the first place. The wider the range of changes that it is possible to recover from at a faster speed, the more useful will the evolved reward function be.

Overall, the experiments presented in this chapter demonstrate the importance of giving adequate rewards to reinforcement learners and that rewards tailored by evolution can make meaningful differences in the training of such learners. This is shown with a meta-learning design that is simple both conceptually and in terms amount of parameters evolved. The EIR-RL approach as presented here has the potential to be useful as is, but the simplicity of the method also makes it well-positioned to be combined with other meta-learning and reinforcement learning approaches in future studies of autonomous, adapting agents.

Chapter 7

Minimal Neural Network Models for Permutation Invariant Agents

This chapter focuses on a different aspect in which ANNs lack flexibility and that is when it comes to the structure of the network after training as well as the ANNs' reliance on the specific ordering of input elements to function.

This chapter aims to contribute to the trend of making the neural architectures of artificial agents more flexible (Kirsch et al., 2021; Tang and Ha, 2021). We do so by proposing a conceptually simple model that after optimization can output coherent actions for a performing agent, even when the inputs to the model are continually shuffled in short intervals. Throughout, we emphasize the minimal requirements that an ANN must follow in order to be invariant to both changes in size and permutations. For the latter, no parameter in the network can be optimized in relation to any specific index in the input vector. In order to be able to take in inputs with varying lengths after optimization, the input must somehow be aggregated to a representation, the size of which does not increase with the number of inputs. An example of such an aggregation is simply to take the average of a range of numbers; regardless of how many numbers there are in the range, their average will always be represented by a single number.

With these requirements in mind, we can choose the simplest solutions to each of them, in order to keep our model as minimal as possible. Thus, the model and its variations presented below do not include a Transformer layer like in the model of Tang and Ha (Tang and Ha, 2021). Indeed, in one

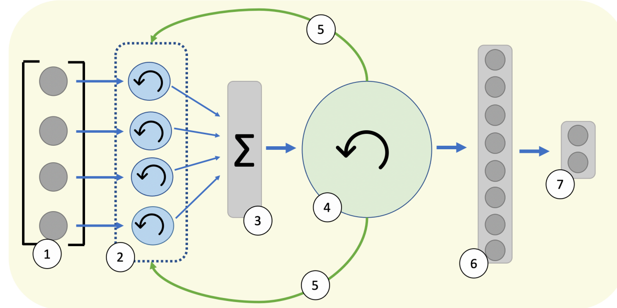


Figure 7.1: **Model Overview:** At each time step, the external input is presented to the network (1). Each element of the input vector is sent into a separate recurrent neural network (RNN) cell that we call *input units* (2). The RNNs in (2) all share the same parameters in their gates. This means that none of the parameters are evolved to be specific to a single element in the external input. The output vectors of the RNNs in (2) are then summed onto a single vector (3). This vector is then passed on to a single RNN (4). The output of this RNN is used to update the hidden states of the RNNs in (2). The same output is also propagated through a dense layer (6) that is connected to the final output of the model (7).

ablation study, we show that it is possible to evolve a network that is invariant to permutations of its input vector and to changes in the input size, even when all elements of the network are simple feedforward networks. Kirsch et al. (Kirsch et al., 2021) aims at meta-learning a black-box reinforcement learning algorithm with a shallow network structure that has invariance properties for both inputs and outputs. In terms of the model structure, our model is similar to that of Kirsch et al., but adding an integrator unit (explained further in Section 7.2) gives us the flexibility to choose to focus only on invariance properties for the input. This makes our model comparably easy to optimize. However, we also show how our model can easily be extended to also have invariance properties for the output.

7.1 Related Work

Transformer-based models have properties that allow them to take sequences of different lengths as input (Vaswani et al., 2017). Further, in language tasks where Transformers in recent years have been used to great effect (Devlin

et al., 2018; Brown et al., 2020), explicit steps must be taken in order to avoid invariance to permutation of the inputs. This is because it is most often useful to be able to interpret words differently depending on where in a sentence they occur. The use of Transformers in RL tasks is less frequent (but see (Chen et al., 2021; Gupta et al., 2022)). Recently, however, Tang and Ha (Tang and Ha, 2021) presented a model to control artificial agents that utilized the properties of Transformers to gain invariance properties for the input. The fact that their Transformer-based model complies with both conditions for permutation and size invariance, can be seen by considering that the key, query, and value transformations use shared parameters for all instances in the input. Then, by fixing the size of one of the transformation matrices (Q in their model), as opposed to letting it depend on the input, the attention matrix is always reduced to a representation of the same size, regardless of the number of elements in the input. Together, this means that parameters in the rest of the network can be optimized in relation to indices in the aggregated representation without being related to any specific indices in the input vector. This highlights the relatedness of the problems of size and permutation invariance. By meeting the condition for size invariance by aggregation of the inputs, the problem of input permutation invariance is contained in the part of the network prior to the aggregation. The rest of the network can thus be structured as any normal network. However, attention-based aggregation is not the only type that meets the conditions; as we show below, a simpler aggregation by averaging can also be used.

Plastic neural networks (Soltoggio et al., 2018; Coleman and Blair, 2012) (see Chapter 2, Section 2.5.3) is another class of models that under the right circumstances can be permutation invariant as well as size invariant. Although this field can be related to Transformers and Fast Weight Programmers (Schlag et al., 2021), plastic networks are usually framed quite differently, with a stronger emphasis on biological inspiration. Plastic networks are also more often used for RL tasks (Soltoggio et al., 2018). Not all plastic networks have the properties that we are interested in here. Interestingly, plastic neural networks, where a single plasticity mechanism governs all connections in randomly initialized networks, automatically meet conditions to be invariant to permutations in the input, as well as to changes in size. This observation might give a clue as to how biological neural networks achieve their high level of architectural flexibility. Biological brains learn complicated tasks as a whole (Caligiore et al., 2019) but vary in the number of neurons over a lifetime (Breedlove and Watson, 2013). Of course, the

brain is governed by a plethora of different plasticity mechanisms (Abbott and Nelson, 2000; Dan and Poo, 2004; Dayan, 2012), not just a single one, but not all parts of a brain might necessarily need to be invariant in relation to all other parts. Further, the conditions could also be met if instead of a single plasticity mechanism, there is a meta-function that organizes local plasticity mechanisms, just as the plasticity mechanisms in turn organize the individual connection strengths.

A recent approach that falls into this category is presented by. Yaman et al. (2021). In this work, the authors evolve a single discrete Hebbian rule to change the synapses of a randomly initialized network to solve a simple foraging task. They also test their rule's ability to control networks with more hidden neurons. More closely related to our method, is the work of Bertens and Lee (2020). They evolve a set of recurrent neural network cells and use them as basic units to form a network between them. In this approach, the synapses and neurons of the overall network are thus recurrent units, and the plastic parameters are the hidden states of the recurrent units. Since all synaptic recurrent units share parameters, and all neural recurrent units share parameters, the network is effectively governed by one overall, homogeneous non-linear plasticity mechanism. The network is shown to be permutation invariant, and this type of network was shown to be able to solve a simple t-maze task with non-stationary rewards.

Closely related to the approach of Bertens and Lee (2020) is the work of Kirsch et al. (2021). They evolve a shallow network structure where all connections consist of recurrent neural networks with shared parameters. Like Bertens and Lee, the goal of the approach of Kirsch et al. is to evolve a network with learning capabilities, and they showcase the abilities of their network to exhibit some level of learning in tasks unseen during training. As we extend our model to have invariance properties in the output vector as well as the input in Section 7.3.3.7, we show that we can have an intermediate integrator unit, between input and output units, and is therefore not bound to a shallow network structure like Kirsch et al. (2021). Common to all methods mentioned in this section is that no parameters are optimized in relation to any specific index in the network. Further, new elements can be added to the network architecture after optimization. This is possible as such added elements will be adapted by the same plasticity mechanism as all other elements in the network.

7.2 Approach: A Minimal Neural Model for Permutation Invariance

The requirement a network needs to meet in order to be invariant to permutations in the input is that no parameter can be optimized in relation to any specific element in the input. This has to hold throughout the entire network. As noted by Tang and Ha (Tang and Ha, 2021), a trivial strategy to achieve this is to constantly permute the input vector throughout the optimization phase. The logic behind this strategy is similar to that of data augmentation strategies used in some image classification studies (Taylor and Nitschke, 2018). If the original dataset does not include examples of rotated objects, a way to make the classifier more general is to augment the dataset with rotated copies of original images. Rotation is a continuous operation and meaningful interpolations can be made between rotations with different angles. However, permutations of indices are discrete and have no meaningful interpolations between them. This means that we would need to augment the dataset with every single possible permutation and thus potentially increase optimization time exponentially.

A visual presentation of the model introduced here is shown in Figure 7.1. When the input vector from the environment is presented to the model at each time step, each element of the input vector is passed into a separate *input unit*. These *input units* are a type of recurrent neural network called Gated Recurrent Unit (GRU) (Cho et al., 2014) with an added output gate. Other types of RNNs, such as the Long Short-Term Memory unit (Hochreiter and Schmidhuber, 1997), or any of the many other RNN variations (Yu et al., 2019) could also have been used. Importantly, the *input units* all share the same weight matrices. This means that regardless of how many elements there are in the input vector, we only optimize parameters for a single *input unit* and copy these to all the *input units*.

This separation of the input elements to *input units* with shared parameters is crucial for achieving input permutation invariance and is a major difference from how inputs are processed by traditional deep RNNs. Note, that even though the *input units* share their evolved weight matrices, their hidden states and their outputs are not necessarily the same at any given time, since these are influenced by the different inputs presented to each input unit. With this approach of routing the input elements into separate units, our method can be described as falling under the category of *instance*

slot models (reviewed by Greff, van Steenkiste & Schmidhuber (Greff et al., 2020)).

Each *input unit* outputs a vector of dimension $(m, 1)$. All these vectors are summed into a single vector of dimension $(m, 1)$ each element of this vector is divided by the number of *input units*. This averaging of the *input units*' outputs will have the same dimensionality of $(m, 1)$ regardless of the number of *input units*. This vector is analogous to the *global latent code* in the approach by Tang and Ha (Tang and Ha, 2021). It is then passed to another RNN (also a GRU with an output gate). We call this the *integrator unit*. It processes the summed outputs of the *input units* just as any normal GRU cell with an output gate would do. The output vector of the *integrator* is passed through two dense layers, the last of which projects to a vector with the number of elements that are needed to make an action in the given environment. The output vector of the *integrator* is also fed back to the *input units*. The *input units* get their hidden states updated through a separate set of GRU gates, the parameters of which are also shared between all the *input units*.

This model complies with both requirements for permutation and size invariance. First, no parameters are specifically optimized in relation to any specific input index. This is ensured by making the *input units* share their optimized parameters and averaging all their outputs to a single vector. The optimized parameters of the rest of the network are optimized in relation to indices of this aggregate of input units' outputs, but these cannot be traced back to any specific indices of the input vector. Second, averaging also means that we can add any number of the input units without disrupting the structure of the rest of the network, and without the need for additional optimized parameters. All RNN cells in all experiments below are GRUs (Cho et al., 2014) with an additional output gate. See Chapter 2, Section 2.2.1 for more details on RNNs and GRUs.

7.3 Experiments

We test the model described in Section 7.2, as well as several variations of it with different ablations and three different environments. The particulars of each of these are specified in the sections below.

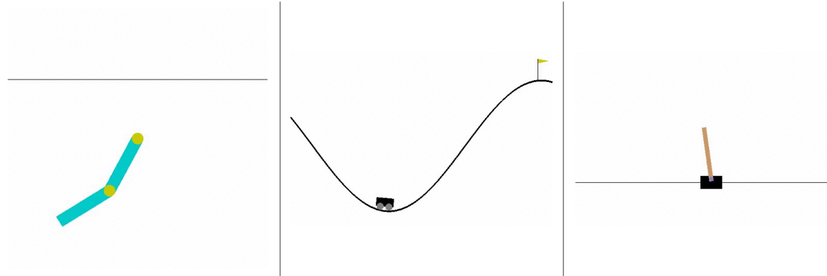


Figure 7.2: **Environments Used in Experiments.** From left to right: Acrobot-v1, MountainCar-v0, CartPole-v1

7.3.1 Environments

We test our model in multiple simple control tasks (Fig. 7.2) from the OpenAI gym suite (Brockman et al., 2016). In all experiments, the ordering of the inputs to the models stays fixed throughout the entire optimization time.

7.3.1.1 CartPole-v1

In this classic control task, a pole is balancing on a cart and the agent needs to control the cart such that the pole does not fall for as long as possible (Barto et al., 1983). The environment has four inputs and two discrete actions. A stopping condition score for the evolution strategy (see below) was set to 495 for this environment.

7.3.1.2 Acrobot-v1

The task in *Acrobot-v1* is to move a fixed 2-dimensional robotic arm with two joints such that the non-fixed end of the arm reaches a certain height (Sutton, 1995). The faster this is achieved the better the score, and a fitness point of -1 is given for each time step spent. The environment has six inputs and three discrete actions. We set the stopping condition to be an average score of -96.

7.3.1.3 MonutainCar-v0

The goal is to move a car from a valley on a one-dimensional track up a large hill (Moore, 1990). The car needs to build up momentum by first moving

Table 7.1: Hyperparameters for ES

Parameter	Value
Population Size	128
Learning Rate	0.1
Learning Rate Decay	0.9999
Learning Rate Limit	0.001
Sigma	0.1
Sigma Decay	0.999
Sigma Limit	0.01
Weight Decay	0

up a smaller hill on the opposite side. The environment has two inputs and three discrete actions. We set the stopping condition to be an average score of -105.

7.3.2 Optimization Details

We use an evolutionary strategy (Salimans et al., 2017) (ES) to optimize the parameters of the system. We use an off-the-shelf implementation of ES (Ha, 2017a) with its default hyperparameters, except that we set weight decay to zero (unless stated otherwise, hyperparameter configurations of all experiments match those in Table 7.1). However, for experiments in the Mountain Car experiments, weight decay is set to 0.01. This is because successes in this environment are initially very sparse, and if all individuals score the same, there is an increased risk of the evolution getting stuck. Weight decay helps evolution differentiate between individuals. The implementation uses mirrored sampling, fitness ranking, and the Adam optimizer for optimization. This optimization implementation is similar to that used by Palm et al. (Palm et al., 2021; Palm, 2020), and Pedersen and Risi (Pedersen and Risi, 2021). Every 20th generation, the mean solution of the population is evaluated over 128 episodes, and if it achieves an acceptable average score, the solution is saved and the evolution run is ended. For all experiments, if a solution was not found within $5,000$ generations, the run was terminated.

7.3.3 Model Ablations

7.3.3.1 Full Model

We refer to the model described in Section 7.2 as the full model. It is fully specified by the following weights matrices: Four weight matrices and bias vectors in the *input units* that adjust the units' hidden states according to the external input following equations (1) through (5) in Section 7.2. Three more weight matrices and bias vectors in the *input units* adjust the units' hidden states according to the feedback from the *integrator* following equations (1) through (4). The feedback only serves to adjust the hidden states of the units, and there are thus no output gates for the feedback. Then, there are the four weight matrices and bias vectors in the *integrator* and two weight matrices and bias vectors for the dense layers that determine the final output of the model. The sizes of the matrices of the GRUs are fully determined by their input sizes, the size of their hidden states, and the output sizes. These hyperparameters are summarized in 7.2. Each *input unit* receives a value from a particular environmental input element, copied eight times to produce a vector. The input size of eight for the *input units* was chosen partly to dictate the sizes of the following matrices, and to ensure that the input would be able to impact the hidden state better than if it had just been represented by a single value. The sizes of the networks for different tasks only differ in the output vector. CartPole-v1 has two discrete actions, whereas MountainCar-v0 and Acrobot-v1 both have three. For each time step in any environment, the index of the largest element of the output vector becomes the action taken by the agent at that time step. At the beginning of each episode, all hidden states are initialized with noise from a Normal Distribution, $\mathcal{N}(0, 0.05)$. The total number of optimized parameters in the network is $24,064$ for the Cart-Pole environment and $24,096$ for the two other environments.

7.3.3.2 No Feedback Model

We run the same experiments with a variation of the full model with the only difference being that the *input units* do not receive a feedback signal from the *integrator*. The total number of optimized parameters in the network is $5,584$ for the Cart-Pole environment and $5,616$ for the two other environments.

Table 7.2: Network Size Specifications

Name	Size
Input Unit In	8
Input Unit Hidden	16
Input Unit Out	24
Inp. Un. Feedback In.	24
Integrator In	24
Integrator Hidden	16
Integrator Out	24
Dense 1	32
Dense 2	environment dependent

7.3.3.3 Integrator as Feedforward Network

In another variation of the full model, we skip the *integrator* and send the averaged outputs of the *input units* directly to the first dense layer. The *input units* receive the output of the first dense layer as feedback to adjust their hidden states. The total number of optimized parameters for Cart-Pole: $20,904$, others: $20,928$.

7.3.3.4 Input Units as Feedforward Networks

In this variation, the *input units* are not GRUs but simple feedforward networks with multiple hidden layers. As in the other variations, the optimized parameters in the weight matrices are shared between the input units. The number of hidden units in the layers of the input units are from beginning to end: 8, 32, 24, 24, 24. The activation function for all the layers is *tanh*. The rest of the network is identical to the full model with no feedback. The total number of optimized parameters for Cart-Pole: $6,064$, others: $6,096$.

7.3.3.5 No RNNs

Finally, we run experiments with a model with no recurrence in the network at all; *input units* are feedforward networks with multiple hidden layers and their averaged outputs are sent through multiple dense layers. The size of the *input units* is the same as in Section 7.3.3.4, and the average output is then sent through layers of sizes: 24, 32, 24, 24, 16, 32, before being projected

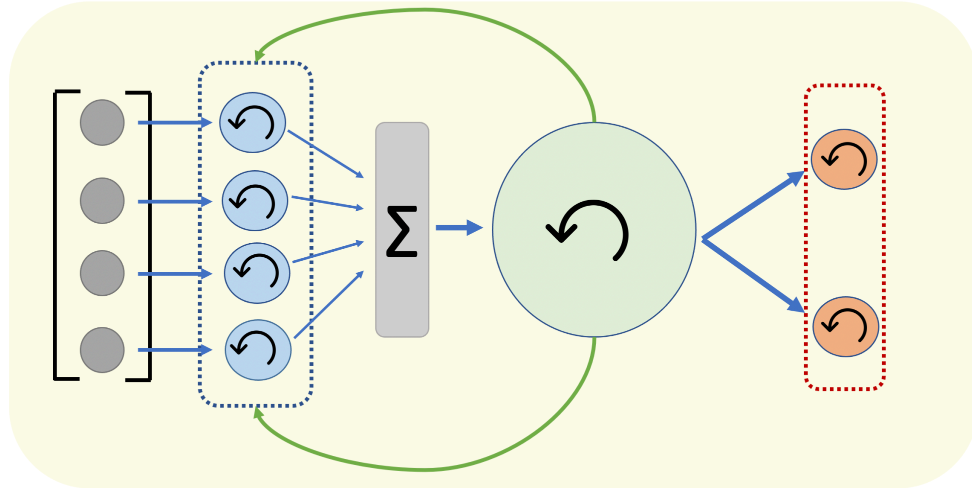


Figure 7.3: **Model Extended to Output Permutation Invariance:** The first part of this model is identical to that in Fig. 7.1. However, instead of dense layers projecting to the output, this model has two recurrent units with shared parameters as the final output nodes.

to the number of actions in the given environment. The total number of optimized parameters for Cart-Pole: $5,760$, others: $5,792$.

7.3.3.6 Standard RNN

In addition to variations of our proposed model, we run experiments with a traditional RNN structure. This consists of a GRU unit with additional output gates following equations (1) through (5) in Section 7.2. The input size of this RNN is equal to the input vector given by the environment in which it is optimized. Its hidden state has 16 elements, and so does its output. It is connected to two dense layers, one with 32 hidden notes, and one that has a number of nodes equal to the number of possible actions in the environment. Total number of optimized parameters for Cart-Pole: $1,954$, Mountain Car: $1,859$, Acrobot: $2,115$.

7.3.3.7 Output Permutations

In our last experiment, we adapt the full model to also be invariant to permutations and changes in size to the output of the network. The first half

of this model is identical the that of the full model. However, instead of projecting to a dense layer, the *integrator* projects to two units of the same type as the *input units,* but with their own set of shared optimized parameters. In this experiment, these *output units* have weight matrices of sizes equal to those of the *input units*, but this is not required. The *output units* have no weight matrices for feedback. Only experiments on the Cart-Pole environment are done with this model and the total number of optimized parameters is $24,176$. For these evolution runs, weight decay was set to 0.01. Further, the fitness of each individual of a generation was here the average performance over four episodes, instead of just a single episode. We extend the optimization in this way, as we are now attempting to solve another problem on top of what was solved by the previous models.

7.4 Results

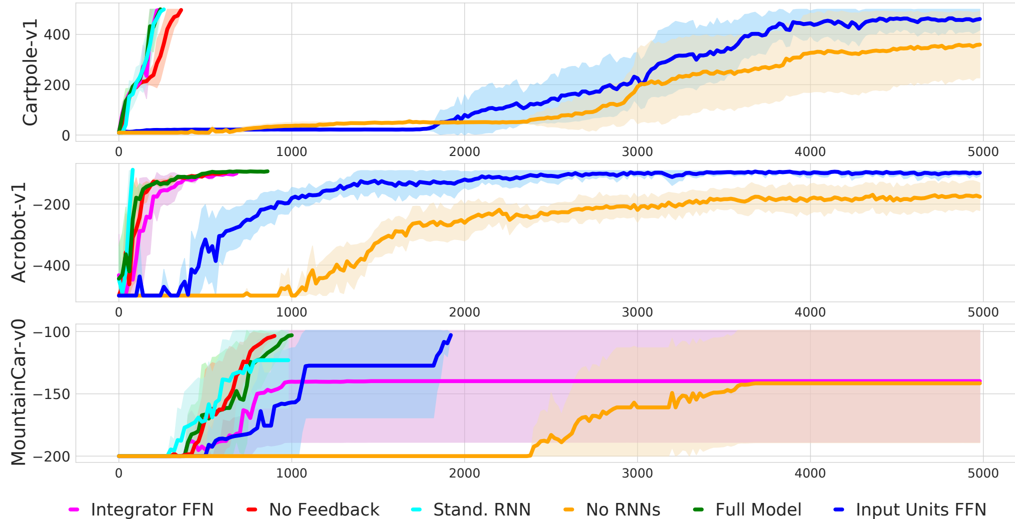


Figure 7.4: **Training Curves:** Means and standard deviations. Each curve represents the mean of five independent evolution runs with a specific method. The full model as described in Figure 7.1 and its variation without feedback from the integrator to the *input units* tend to find solutions to the tasks quickly. The same is true for the standard RNN. When the *input units* are feedforward networks rather than RNNs, evolution in most cases did not end up finding a solution within the set time limit of 5000 generations.

Training curves of each experiment are shown in Figure 7.4, except for the experiment described in Section 7.3.3.7 that is presented in Figure 7.5. Each curve represents the mean of five independent evolutionary runs. From Figure 7.4, it is clear that evolution tends to find solutions much faster with some models than with others. Specifically, the full model, the full model without feedback, and the standard RNN find solutions in the matter of hundreds of generations for all problems. On the contrary, for the models where the *input units* are not RNNs, a solution was often not found within the time limit of 5,000 generations.

Full Model		
Env.	Input Doubling	No Perm.
Cart-Pole	488 ± 68	485 ± 77
Acrobot	-106 ± 48	-109 ± 58
Mountain Car	-100 ± 6	-99 ± 6
Standard RNN		
Cart-Pole	N/A	172 ± 210
Acrobot	N/A	-396 ± 168
Mountain Car	N/A	-149 ± 48
No RNNs		
Cart-Pole	496 ± 35	496 ± 36
Acrobot	-119± 63	-118 ± 63
Mountain Car	-106 ± 7	-105 ± 8

Table 7.3: Table of Results. Means and standard deviations over 1000 episodes. Numbers are rounded to the nearest integer. For each method, we choose the run with the highest population mean score at the end of evolution. Input Doubling means that each element of the input vector is copied. E.g., in the Cart-Pole environment, this means that there are eight input elements and therefore also eight *input units* instead of four. No. Perm. means that the input vector is not permuted online. However, at the beginning of each new episode, the ordering is randomized.

Table 7.4: **Online Permutations Evaluations** Same as in Table 7.3 Results show that variations of our model both with and without recurrent dynamics are able to do well in the tasks, even when the input is permuted online several times. Note, however, as seen in Figure 7.4, evolution runs of the model without recurrent dynamics, did not reliably result in a solution within the set time limit. The standard RNN model does not do well in any of these scenarios.

Full Model				
Env.	Every 100	Every 50	Every 10	Every 5
Cart-Pole	489 ± 68	489 ± 68	490 ± 66	482 ± 87
Acrobot	-107 ±55	-107 ± 57	-108 ± 61	-105 ± 45
Mountain Car	-100 ± 6	-100 ± 6	-100 ± 6	-108 ± 22
Standard RNN				
Cart-Pole	73 ± 87	48 ± 51	20 ± 15	23 ± 13
Acrobot	-293 ± 152	-296 ± 138	-279 ± 108	-280 ± 106
Mountain Car	-146 ± 46	-171 ± 41	-180 ± 35	-193± 21
No RNNs				
Cart-Pole	495 ± 43	496 ± 37	498 ± 29	496±37
Acrobot	-122±69	-120±70	-124±76	-117± 62
Mountain Car	-105 ± 7	-105±7	-105±7	-105±7

After optimization, we are interested in how well the models do with online permutations of the input vectors. In Tables 7.3 and 7.4, such evaluations are shown for an optimized full model, the model consisting of feedforward units only, and the standard RNN model. Across the board, the models designed for invariance tend to achieve similar scores under all conditions, even when the inputs are shuffled at intervals as frequent as every 5th time step. The standard RNN fails under all permutation conditions. Following Tang and Ha (Tang and Ha, 2021), we also evaluate the models when given a larger input vector than seen during optimization with redundant values. Here too, the models designed for invariance show no signs of deterioration in performance. It was not possible to evaluate the standard RNN using a doubled input size, due to its rigid structure.

Figure 7.5 shows that it takes longer for evolution to find solutions to the Cart-Pole environment when the full model is extended to also have *output units* with shared parameters, but that solutions are consistently found.

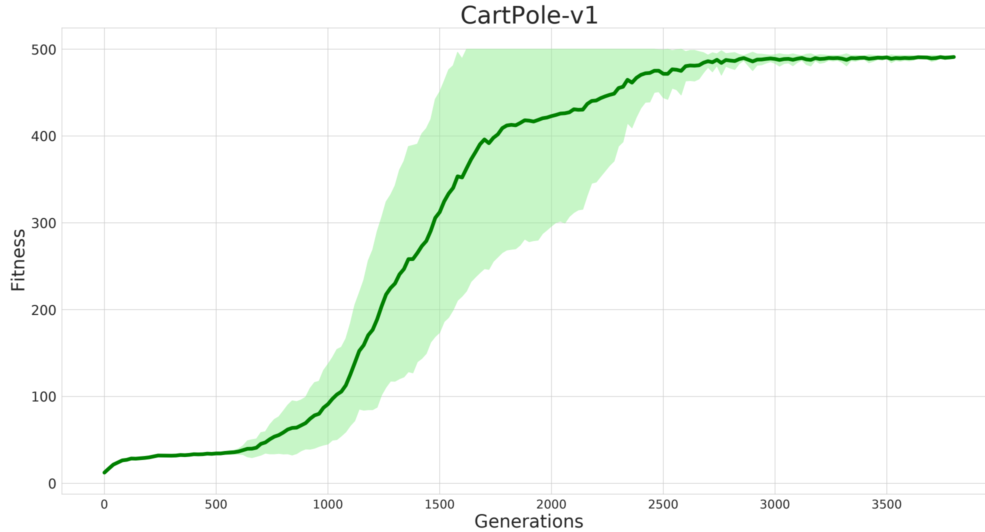


Figure 7.5: **Training Curve for Model with Output Units.** Means and standard variations of five independent evolution runs.

Further, in Figure 7.6, we see that the optimized model does not tend to perform well with frequent permutations of both the input and output vectors. However, as shown by the left-most box plot in the figure, the model performs well under random orderings, as long as they stay fixed during the episode.

We can look closer at how the *input units* behave under conditions without and with online input permutations. Such cases are presented for a full model performing in the Cart-Pole environment in Figure 7.7 and Figure 7.8 respectively, where we see the 16 hidden state elements of each of the four *input units* over a full episode. Figure 7.7 shows that when no permutations occur, the mode of each *input unit* tends to look similar throughout the episode. However, as can be seen in Figure 7.8, the *input units* are able to quickly switch roles in response to a permutation.

7.5 Discussion

In this chapter, we demonstrate that the requirements needed for making a network invariant to permutation and size changes of the external inputs can be met by relatively simple models. Importantly, no parameters can be

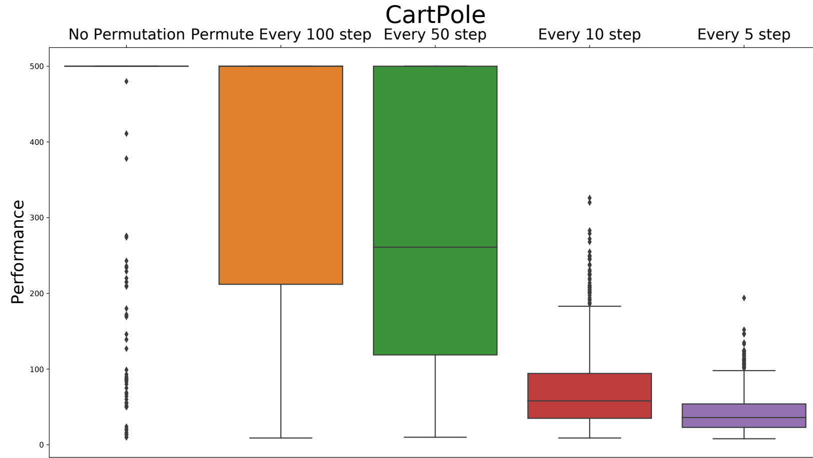


Figure 7.6: **Performance Under Online Permutation of Input and Output.** The model performs well under random permutations of both the input and output when the random ordering is fixed during the episode. However, online permutations make the model fail at increasing levels.

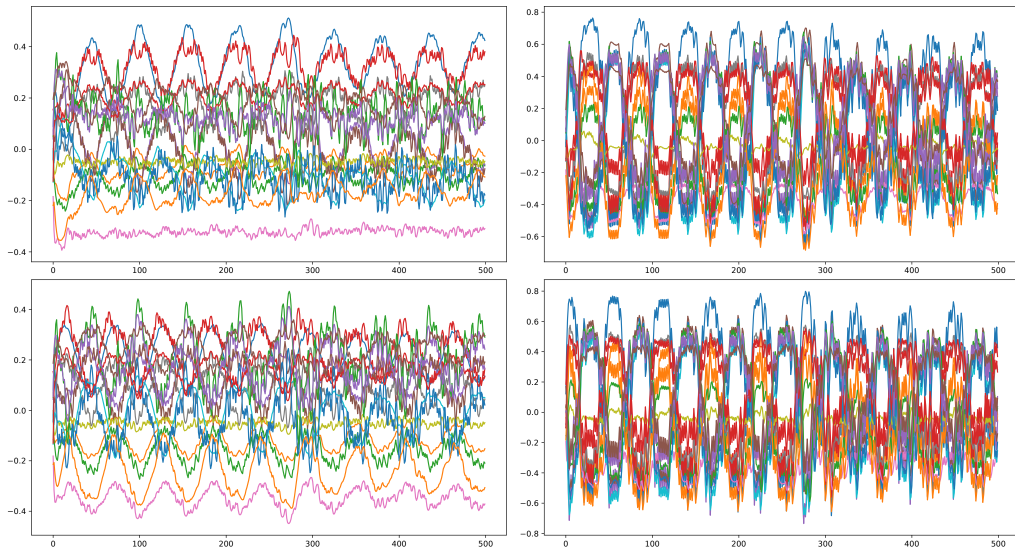


Figure 7.7: **CartPole Hidden States: No Shuffling** The 16 hidden state elements of each of the four input units over a full episode in the Cart-Pole environment. The units seem to have separate, fixed roles.

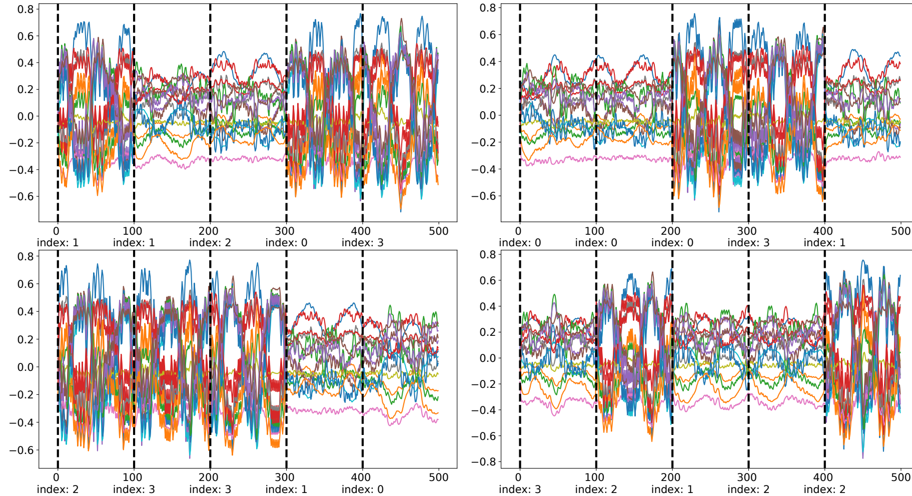


Figure 7.8: **CartPole Hidden States: Shuffle Every 100th Step** Black dotted lines indicate the times at which the input vector was randomly permuted. The hidden states rapidly adapt their activity levels in response to permutations.

optimized in relation to any specific index of the input vector, and projections of the input must at some point in the network be aggregated to a representation that does not grow with the number of inputs.

The simple solution we use in the model presented here is to average the outputs of input units with shared parameters. Even when optimized with fixed inputs, the models are remarkably robust to frequent permutations of the input vector. The solution does not specify that the *input units* need to be recurrent neural networks. Indeed, we find that in the environments we use for experimentation here, it is possible to find solutions solely using feedforward networks. However, such solutions tend to be more difficult to find compared to when *input units* are RNNs. This effect might only be exacerbated in more difficult environments. The use of feedback from the *integrator* did not tend to make a difference in our experiments. This could be due to the simplicity of the environments, as Tang and Ha (Tang and Ha, 2021) report that their analogous *input units* need to get the model's previous outputs as additional inputs in order to work. One might expect that the more overlapping the values of the input vector can be, the more need there is for some form of global signal as well as a memory of previous

inputs.
We further show that it is simple to extend the model to also be able to work with different permutations of the output vector. This is done by following the same principle as for the input vector: no parameters in the network can optimized in relation to a specific index of the output vector. We solve this by having *output units* with shared parameters that receive a common input from the *integrator*. However, the problem of invariance to permutations of the output vector is different in important ways. First of all, the *output units* almost certainly need to be recurrent, as having different hidden states is the only way that the units are able to give different outputs in response to the identical inputs they are presented with from the *integrator*. Second, dealing with online permutations of the output vector is much harder than with the input vector as indicated by the results in Figure 7.6. This is not surprising considering that every time the output vector is permuted, the next action of the network will be random. For the Cart-Pole environment with only two actions that are oppositely directed, this might be somewhat feasible, as only a single random action is needed in order to have a perfect overview of all actions. For environments with larger numbers of available actions, the agent would have to behave randomly for potentially many time steps before being able to settle into a learned behavior. Still, even though rapid online permutations of the output might be insurmountable at large scales, the properties of a model like the one presented in Section 7.3.3.7 can still be interesting. In this model, the number of parameters to be optimized is completely decoupled from the size of the external input and the number of actions in the environment. As such, the model can potentially be optimized on multiple different environments that do not need to share the input and output spaces. This idea is not unlike the one presented by Kirsch et al. (Kirsch et al., 2021). However, Kirsch et al. are in their work aiming for evolving a black-box reinforcement learning algorithm. While the optimization procedure of the model presented in this chapter could be altered to mimic that of Kirsch et al., our model currently does not take any rewards into account but focuses solely on solving the problems of invariance. With the model presented here, we get these invariance properties with only a small fraction of the optimization time reported by Kirsch et al. on the same problems.

7.5.1 Future Directions

Having shown that our model can be reliably evolved to be invariant to permutations on simple problems, it is worth considering how the model might be scaled up to bigger problems. In future experiments, we also aim to use our model to solve continuous control problems with more inputs and outputs. Trivially, it is always possible to make the model more expressive by increasing the sizes of the weight matrices throughout the network. However, there are other ways the model can be expanded, while still conforming to the laid out restriction. It should for example be possible to add a layer of *input units* parallel to the ones in the model in Figure 7.1. Each element in the input vector would thus be sent through two different *input units*. The added *input units* would still have to share optimized parameters with each other, but, importantly, not with the "original" layer of *input units*. The averaged output vectors of the *input unit* layers can then be concatenated and sent through the network as in Figure 7.1. An added, separate set of optimized parameters for processing the input could allow for more specialization, without hurting the ability to deal with permutations of the input.

7.6 Conclusion

The world is a messy place and it might not always be possible for us to fully anticipate how inputs will be presented to agents meant to perform in it. With this chapter, we contribute to the efforts of making artificial agents more adaptable to changes to their inputs. We do so by being explicit about what the models at the very least will need to be able to adapt to permutations and size changes and use these restrictions to develop simple models that adhere to them. We hope that this will inspire even more research in making more adaptable artificial agents.

We have now seen several approaches that extend the capabilities of ANNs in different bio-inspired manners. Chapter 4 presented an algorithm for increasing the robustness of ANNs with relatively small rule sets. In Chapter 5 the need for optimizing synaptic weights was alleviated as parameterized neurons in random networks were shown to be expressive enough to perform in RL-tasks. In Chapter 6 RL agents were able to maintain their performance in novel circumstances through the guidance of an evolved reward

function. Finally, minimal models with invariance properties were evolved in this chapter. In the next chapter, a framework is presented that combines the ideas of plasticity with a few learning rules, parameterized neurons, learning from rewards, and structural flexibility.

Chapter 8

Evolution of Structurally Flexible Adaptive Neural Networks: Toward a Model for General Learners

In this chapter, we present our approach called Structurally Flexible Adaptive Neural Networks (SFANN). With this approach, we aim to leverage the concepts of plasticity, optimized neural units, and structural flexibility, which were themes of earlier chapters of this thesis, to develop a parameterized reinforcement learning function capable of learning and adapting to multiple environments. Plasticity plays a crucial role in transforming an initially random connectivity structure into a functional one, allowing the network to adapt its connections based on experience. By incorporating trainable parameters within neural units and employing learning rules, we enable complex neural interactions using a limited set of rules. The key to achieving a versatile learning algorithm lies in structural flexibility, which liberates the model from being tied to specific input and output spaces.

The primary objective is to devise a parameterized learning algorithm that exhibits fast adaptation across various environments, regardless of their dimensionality, and the permutations of the input and output elements. If successful, such a model could serve as a general foundation model (Bommasani et al., 2021; Yang et al., 2023) for black-box meta-reinforcement learning.

Our proposed approach builds on the Variable Shared Meta-Learning

framework (Kirsch and Schmidhuber, 2020) and generalizes it by introducing dynamic neurons and by relaxing the constraint that all units in the network have to share the same parameters. These extensions contribute to resolving the symmetry dilemma. The symmetry dilemma, described in more detail later in the chapter, summarizes the insights that inform most of the design decisions of the SFANN approach. In short, the symmetry dilemma is that on the one hand, we need the learning mechanism of the network to be symmetric enough so that it can be used flexibly on any network structure and for any input/output space. On the other hand, the learning mechanism must be able to produce structures that are not symmetric, such that signals propagated through them can be processed appropriately.

This chapter presents promising preliminary results showcasing a single parameter set capable of solving tasks with different input and output sizes. The optimized model demonstrates its proficiency in simple point navigation tasks with both discrete and continuous output spaces. Remarkably, the trained model exhibits the ability to perform actions that were unseen during training, illustrating its generalization capabilities. Starting with random connectivity, random initial synaptic and neural states, and random numbers of hidden neurons, the evolved classes of plastic synapses and dynamic neurons organize during the lifetime to achieve better performance in the given environment.

The SFANNs presented in this chapter adhere to the requirements for permutation and size invariance as discussed in the previous chapter. However, it is important to note that the objective of these networks differs from the models presented in Chapter 7.

In the current chapter, the focus is on enabling synapses and neurons to form functional networks of any shape and size guided by a reward signal. Unlike the models in Chapter 7, where robustness to online permutations was a primary goal, these networks aim to acquire functional network configurations through learning.

It is worth noting that as these networks become proficient in performing tasks with a particular network configuration, they lose their structural flexibility and may at the end of the agent's lifetime not readily adapt to permutation, unlike the models in Chapter 7. In other words, while the models satisfy the requirement of not optimizing parameters relative to specific positions in the network on an evolutionary scale, they do not do so at a lifetime scale. This is by design, as we prioritize reward-based learning to perform well across environments, rather than being tolerant to rapid online

permutations, as the former might be of more general interest to real-world scenarios. However, the observations made in Chapter 7 are valuable in that they inform how the outer-loop parameters must be optimized.

8.1 Related Work

The approach described in this chapter is related to other studies on black-box meta-reinforcement learning (see Chapter 2, Section 2.5), plastic neural networks (see Chapter 2, Section 2.5.3), and permutation invariant models (see Chapter 7). In addition to the areas of work that have already been described in detail elsewhere in this thesis, there are some topics that are more specifically related to the approach presented in this chapter.

8.1.0.1 Graph Neural Networks:

The network type presented in this chapter consists of neurons and synapses that are themselves recurrent neural networks, more specifically LSTMS. This means that they all have hidden state vectors associated with them, and these are updated using local information. This is similar to how Graph Neural Networks (GNNs) work. GNNs are designed to analyze data that is represented as graphs (Wu et al., 2020). Nodes in a GNN have internal states and pass information to their neighbors through parameterized message functions that are optimized during the network's training phase (Zhou et al., 2020a). A neural network with neurons and synapses being LSTMs can be seen as a graph where some nodes (the synapses) can only have two neighbors (neurons), whereas the ”neuron” nodes can have many ”synaptic” nodes as neighbors. In the neural networks of this chapter, these different types of nodes also differ in how they propagate information and update their states. However, the idea of optimizing how nodes with internal states interact with each other is analogous to that of GNNs. GNNs have been used to control simulated robots showing that they can take advantage of the fact that robot morphology can be expressed as a graph (Wang et al., 2018). In contrast, the networks presented in this chapter do not make assumptions about the spatial structure of inputs or outputs of the model but are only concerned with how the units of the network can best be updated to transform inputs into outputs in a manner that maximizes rewards.

8.1.0.2 Foundation Models:

Foundation models refer to models with large numbers of parameters trained on huge, varied datasets (Bommasani et al., 2021). These models require a great amount of resources to optimize. However, it has recently been shown that such models can be fine-tuned to specific tasks at lower costs than would have been required to train a new model for the specific task from scratch. The initial large investment in the general foundation model thus pays off in the form of reduced cost of new specialized models. Foundation models have mainly been used within the fields of language- and image processing, as these are generally the areas where access to large amounts of data is the most abundant (Zhou et al., 2023). Large language models in particular have in recent years achieved great success, being trained on large amounts of unlabelled text. Recently, the notion of foundation models has also surfaced in the context of reinforcement learning (Yang et al., 2023). Some examples have been leveraging large pre-trained language models to speed up offline reinforcement learning (Reid et al., 2022). Other examples of foundation models are optimized both on language, vision, and RL tasks, as with GATO (Reed et al., 2022) and RoboCat (Bousmalis et al., 2023). In the case of GATO and RoboCat, these models rely on supervised learning on demonstrative trajectories in the environments they attempt to solve.

A foundation model optimized via reinforcement learning was recently introduced as AdA (Team et al., 2023). This framework also utilizes sequence-modelling with versions using either RNNs or Transformers. This type of foundation model was able to be trained on variable context lengths, resulting in the varying and flexible use of memory. However, the model depends on fixed-sized embeddings of the observations and actions that make up the sequences. This makes the approach less suited to be used across environments that require differing observation and action sizes.

Due to the structural flexibility of SFANNs, the parameters of the SFANNs approach presented in this chapter could be exposed to any reinforcement learning environment in existence during its evolution. As such, this approach provides an ideal candidate for a foundation model for black-box reinforcement learning that only relies on reward signals and is trained by directly interacting in the environments.

8.2 Structural Flexibility through Parameter Sharing Neuron and Synapse Classes

This section will explain the details of the proposed network, how information propagates through it, how synaptic strengths are updated, how the property of structural flexibility is gained, as well as the architecture of the network.

8.2.1 Information Propagation and Update of Synapses and Neurons

The networks presented here consist of synapses and neurons that are all represented by LSTMs (Hochreiter and Schmidhuber, 1997) introduced in Chapter 2, Section 2.2.1. The LSTMs that make up the synapses of the network can be conceptualized as non-linear history-dependent plasticity rules, responsible for updating synaptic strengths. The neurons are responsible for integrating information sent to them, also in a history-dependent manner. Inputs and outputs of both synapses and neurons are vectors rather than scalars as in standard neural networks. Being LSTMs, all synapses and neurons maintain a hidden state that is updated as information propagates through the network.

Neurons and synapses differ in the way they are updated. Neurons in the network behave the most like normal LSTMs. As input, a neuron receives the summed signal propagated to it from other neurons through connecting synapses. To this input, the reward signal from the previous time step is concatenated. This input is then used to update the cell state and hidden state of the neuron using the usual LSTM update procedure. The input and the updated hidden state are then concatenated and sent through an output gate. The output vector is then passed along the outgoing synapses of the neuron.

An output vector is propagated through a synapse using an element-wise multiplication of the output vector and the hidden state of the synapse. The resulting vector is then added to all other vectors arriving at the same post-synaptic neuron and the result of this is divided by the number of synapses. When the post-synaptic neuron has been updated and created an output vector, the output vectors of the pre-synaptic neuron and the post-synaptic neuron, as well as the reward signal from the previous time step, are concatenated and used as input for updating the hidden state of the synapse.

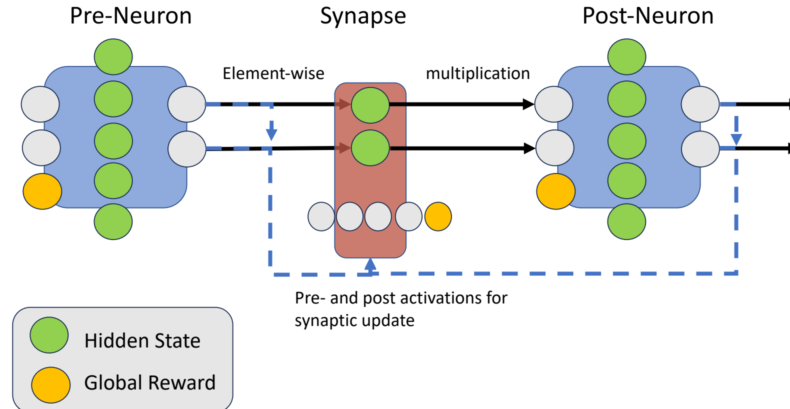


Figure 8.1: **Pre- and post-neurons connected by synapse** Representation of the flow of activity from one neuron to another through a synapse. The output of the pre-neuron is first weighted by the synapse using element-wise multiplication with the hidden state of the synapse (solid black arrows). The output of both pre- and post-neurons are then used for updating the hidden state of the synapse (dotted blue arrows). The global reward signal provided by the environment is concatenated to the input every time a hidden state is updated.

In sum, a neuron uses information from all incoming synapses to update its state and create an output vector. Output vectors of neurons are propagated through synapses via simple element-wise multiplication with hidden states of synapses. The hidden state of a synapse is updated using the output vectors of the two neurons that it connects. The reward signal from the environment is available for the update of the hidden states for all neurons and synapses. See Figure 8.1

8.2.2 The Symmetry Dilemma

Structural flexibility is achieved in much the same way as the permutation invariance property of Chapter 7. It is needed for the reasons explained by Kirsch et al. (2021): the symmetry of backpropagation and its accompanying ability to optimize any network permutation and size are absent in most

black-box meta-reinforcement learning approaches. This ties the learned reinforcement learning algorithms to specific input and output spaces, making them of limited general use.

The challenge that comes with constructing a parameterized approach that is symmetric and invariant to permutations and sizes, is that the approach must be capable of resulting in information processing that is not symmetric. A completely symmetric network where all output neurons always end up having the same values is not very useful. Even a case where output values can differ, but are too tightly connected to each other, such that, e.g., two outputs always have the same interval between them, is problematic for environments that require continuous outputs.

This problem is what we might call *the symmetry dilemma* of black-box meta-reinforcement learning. We have to optimize the outer-loop parameters in a symmetric manner, such that they do not break the necessary conditions for permutation and size invariance, but also in a manner that can result in inner-loop parameters that are arbitrarily asymmetric. Kirsch and Schmidhuber (2020) introduced a fully connected network where all synapses were LSTMs sharing the same parameters. Such a network is clearly symmetric in its meta-optimized parameters, and the idea is that LSTMs that share parameters can still develop different hidden states over time, potentially resulting in an asymmetric network inner-loop parameter structure.

However, there are some downsides to such a network architecture. One downside has to do with the option of adding more trainable outer-loop parameters. From the standpoint of traditional machine learning, a surprising finding in research in deep neural networks is that overparameterization can be beneficial both in terms of trainability and generalization (Rocks and Mehta, 2022; Hasson et al., 2020). A network such as the one presented by Kirsch and Schmidhuber (2020) (Kirsch and Schmidhuber, 2020), where synapses all share LSTM parameters can only increase parameterization by increasing the size of the LSTM unit. By allowing multiple classes of neurons and synapses, outer-loop parameters can be added by adding more classes. One reason that this is useful is that the computational cost of propagating through the network increases with the size and number of the network's units, not with the number of different parameter sets of these units.

The second downside has to do with the symmetry dilemma: a fully connected structure of identical LSTM units is vulnerable to hidden states of the synapses converging to similar values. When the network is first initialized, the initial hidden states (the inner-loop parameters) of the synapses are

randomly sampled, and this is what makes the network asymmetric. The approach is fully reliant on the hidden states to be able to diverge from each other and not fall into the same attractors, even though the parameters that update them are identical and inputs are the same for all synapses that are connected to the same pre-neuron. This makes it challenging for the output neurons to be independent of each other, which is especially problematic in environments that require continuous outputs. The risk of having units converge to the same hidden states is even bigger in deep structures of homogeneous LSTM parameters. This can be seen by considering the case where two synapses connecting the same input element to different hidden neurons become similar to each other. The more similar they get, the greater the risk that their respective post-synaptic neurons become similar in their activations. This can result in a vicious circle where these neurons propagate more similar signals forward through all their synapses, the hidden states of which now have an even greater risk of falling into the same attractor and becoming identical. In shallow networks used for supervised learning, as in some experiments in Kirsch and Schmidhuber (2020), this risk is smaller, as the error vector can enforce more differentiated signals used to update the synapses. However, in reinforcement learning any error or reward signal usually provides less fine-grained information.

The approach presented in this chapter is greatly influenced by that of Kirsch and Schmidhuber (2020) and Kirsch et al. (2021), but attempts to mitigate these downsides through the introduction of parameterized neurons, and LSTMs with different parameter sets.

It is possible to introduce these additions to a permutation and size-invariant network without optimizing any parameters relative to a fixed position in the network. This is done through additional randomization of the network initialization beyond just randomly sampled hidden states. Any aspect of the network configuration that, if fixed throughout the outer-loop optimization phase, would result in the outer-loop parameters overfitting to it must be presented with variation during the optimization phase for the parameters to be invariant to it. The reason that hidden states can be initialized randomly to form an asymmetric structure in the network of Kirsch et al. (2021) and still maintain permutation and size invariant properties, is that the the outer-loop optimization process has no information about these random initializations, and the outer-loop optimization does not take any specific initialization of the hidden states into account.

The same logic can be applied to different aspects of randomness in the

initialization of the networks in the inner-loop. Rather than initializing a synapse with random noise, some synapses can be deleted at random, making for more sparse connectivity. As long as synapses are dropped out randomly at initialization, the outer-loop parameters cannot overfit to a specific connectivity. This point was also noted by Bertens and Lee (2020).

Having parameterized neurons rather than just synapses provides additional opportunities for asymmetries. Trivially, the hidden states of neurons also can have random initializations. Further, with more neurons in a network come more synapses that can potentially be dropped out to make the structure more asymmetric.

8.2.3 Neural and Synaptic Diversity

Another advantage of adding hidden layers of neurons is that it also creates multiple layers of synapses. As noted in Chapter 7, all input units must share parameters to be invariant to permutations of the input elements. However, synapses from the input to a hidden layer, do not need to share parameters with synapses between other layers to adhere to the requirements for invariance. In addition, skip connections from earlier to later layers can also have unique synapses without breaking any invariance requirements. In fact, only synapses that connect the same inputs to the same outputs must share parameters. This means that if the inputs can be divided into different groups in a manner that can be generalized across any environment, then synapses connecting to different groups can have different parameters as well, even if they project onto the same hidden neurons. An example of this is if the network gets its outputs from the previous time step concatenated to the observation vector from the environment. In this case, the previous can be projected to the first hidden layer using a different set of synapse parameters than that of the observation from the environment.

Having multiple sets of parameters for synapses relieves the responsibility for each of them to work across potentially significantly differing contexts throughout the network, and this might make solutions easier to find for the outer-loop optimization process. In Chapter 3, we saw that adding parameters to the outer-loop increases the ability of the outer-loop to alter the loss landscape of the inner-loop.

Of course, the addition of parameterized neurons in and of itself adds a new parameter set that can complement the synaptic parameters. Further, just like synapses of different layers do not need to share parameters, the

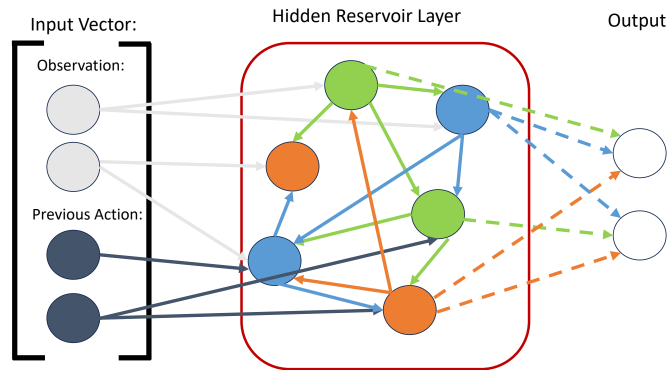


Figure 8.2: **Structurally Flexible Adaptive Neural Network** The input vector consists of the observation of the environment and the previous output of the network. This is sent through a reservoir layer with random connectivity. Neurons that share color are from the same class and share LSTM parameters. Neurons of a class are always pre-neurons to synapses of the same class. Different classes of synapses project within the reservoir (solid) and to the output layer (dotted).

same is true for neurons.

Compared to a shallow network structure with homogeneous synapses there is thus ample opportunity to increase the parameter space beyond increasing the size of the LSTMs that make up the network.

8.2.4 Network Architecture

Rather than dividing the hidden nodes into multiple distinct layers, a single reservoir (Lukoševičius and Jaeger, 2009) of different neuron classes makes up the network. An illustration can be found in Figure 8.2 There is thus an adjacency matrix encompassing all hidden neurons in the entire network. Each class of neurons has a class of synapses associated with it. The associated synapse class always serves as the vehicle for the outputs of the class of neurons. A neuron can project to any other neuron within the reservoir regardless of their classes, but always through the same class of synapse.

In the networks used for the experiments below, three classes of neurons

are used. Networks are initialized in two steps. These initialization steps include different elements of randomness that are supposed to mitigate the symmetry dilemma described above. First, the size of the reservoir is randomly chosen from a range of integers. Neurons and their associated synapses are placed randomly in the reservoir. When a new network is initialized, the input vector consisting of the observation and the previous action produced by the network projects to the hidden neurons in a fully connected manner. Hidden neurons also project to the output of the network, again with each class of neuron having a specific class of synapse to send each activation through.

Step two of the initialization is to randomly remove synapses from all layers. Unless stated otherwise, each synapse in the experiments below has an independent probability of 50 % of being removed from the network.

8.2.5 From Input to Action

At each time step, the observations from the environment are concatenated with the network's output of the previous time step. At the very first time step, each vector of zeros is put in the place of the previous output vector. Each element of this combined input vector to the network is sent through a synapse to the hidden neurons in the reservoir. One class of synapses projects elements from the observation, and another class of synapses projects from the previous output. Since a synapse takes a vector as input, each element of the input vector is copied to match the number of input channels of the synapses.

When the activity of the input synapses reaches the hidden neurons, these are updated according to their LSTM parameters. The hidden neurons then send their outputs to all the other hidden neurons in the reservoir that they are connected with. The outputs of the neurons are used to update the hidden state of the input synapses. The neurons are then updated, and so are the synapses connecting the hidden neurons. This is repeated so that hidden neurons get to project to each other, update their hidden states, and update the synapses twice. After this, the neurons send their outputs through the output synapses of the network. These projects to the action units. Action units are not LSTMs. They are simply an average of the values of output synapses connected to them, respectively. Since the outputs of synapses are vectors, and the environments only require a single scalar per action, the average values of the first output channel of the synapses are used as action

Table 8.1: SFANN Hyperparameters

Neuron Hidden Size	15
Synapse Hidden Size	2
Total Number of Outer-loop Parameters	4,290
Reservoir Size Range	15-30
Neural Ouput Activation Function	ReLU
Number of Neuron Classes	3
Number of Synapse Classes	8
Number of Reservoir Ticks per Time step	2

elements, and the values of the rest of the output channel are discarded.

The activation of the output units is then determined by whether the environment expects a discrete or continuous action. If a discrete action is needed, the action vector is turned into a probability distribution using a softmax activation. These probabilities are used to sample an action from a categorical distribution. If a continuous action is needed, the action vector is first activated by the hyperbolic tangent function. The resulting vector is then used as the mean in a multivariate normal distribution with a standard deviation that has the value of 1 at the first time step in the lifetime of the agent and decays at every time step by a factor of 0.975. The sampling of actions serves to dual purpose of aiding early exploration and de-correlating the activations of continuous actions by forcing different activations and thus different local feedback that updates the output synapses.

The elements of the final action vector are in both cases used to update the hidden states of the output synapses.

Table 8.1 provides an overview of the numbers relevant to the networks in the experiments below, such as the number of different neurons and synapses, their sizes, and the number of hidden units in the networks.

8.3 Experiments

8.3.1 Environments

As a proof-of-concept, we use the approach in two variations of a simple point navigation task, one with a discrete action space and one with a continuous action space. In this environment, the agent has to find a specific point on a square surface. The agents start at the origin of a square two-dimensional coordinate system with the points (-10,-10), (-10,10), (10,-10), (10,10) marking the border of the area wherein the agent is allowed to move. At each time step, the observation from the environment is the point in the coordinate system that the agent currently occupies.

In the version that requires a continuous action, the agent outputs at each time step a vector with two elements that are added to its current location. In the version that requires a discrete action, the agent outputs an integer in the range of [0,3]. These, respectively, result in adding or subtracting 1 from one of the coordinates of the agent's current location.

Whenever a new episode is created, a goal point is randomly sampled to be anywhere within the boundaries of the coordinate system. At each time step, the reward information given by the environment is the difference between the current distance to the goal and the distance to the goal at the previous time step. If the agent reaches a distance that is less than 1.5 from the goal point, the agent receives a reward of +10, and the episode ends. If the agent has not reached the goal point within 50 time steps, the episode ends as well with a reward signal of -10.

8.3.2 Training Setup

The training of the type of network described in Section 8.2 is set up as described in the following. The parameters of the neural and synaptic units are optimized using CMA-ES (see Section 2.4.3 in Chapter 2).

The fitness assigned to each individual is determined as the average score over four lifetimes in the environments, two lifetimes with discrete actions and two lifetimes with continuous actions. A lifetime of an agent consists of a total of 300 time steps in the environment, repeating the same episode the goal point is reached or the episode has reached its time limit of 50 time steps. For each episode during the lifetime of the agent, the accumulated reward for each time step is recorded. The lifetime score of the agent is then the

weighted sum of recorded episode scores. Scores obtained in episodes later in the agent's lifetime are weighted higher than scores obtained early in life. The exact weight of each episode is calculated using the softmax activation on a range of enumerating each episode of the lifetime. This weighting method constitutes a convenient way of emphasizing the importance of performance later in the lifetime, while keeping the magnitude of the lifetime score similar to the simple average, as all the weights sum to one. A lifetime score above 9.5 can be considered successful with these settings.

The reason for assigning a larger weight to later episodes is to encourage learning over time and to maintain performance. Agents that perform well in the first half of their lifetimes, but then start to fail, should be differentiated from agents that with experience are able to improve their scores. Letting scores from the end of the agent's lifetime be more important for the lifetime score also serves to not punish early exploration.

For each of the four lifetimes in a generation, a common initialization of the network structure is chosen for all individuals in the population. This means that all individuals have the same reservoir layer size, the same random placement of neuron classes in the reservoir layer, as well as the same deleted synapses. The individuals differ only in the parameters of the neural and synaptic classes, as well as the hidden state initializations. The choice of making sure that all individuals were initialized with the same network structure was made to get a more fair evaluation of the parameters. In the graphs below, this is the approach named **SFANN**.

When an individual is initialized for a new episode, the number of input and output units are set to fit the specifics of the particular environment. This means that networks initialized for the environment with a discrete action space, have four output units and therefore also four elements being concatenated to the observation vector with two elements. Networks initialized for the environment with a continuous action space are initialized with just two output units.

8.3.2.1 Necessity of rewards:

To show that reward feedback is vital for this setup, a training setup identical to the one described above is run with the only exception being that rewards are always set to zero. The approach is referred to as **No Reward**.

8.3.2.2 Shallow Networks:

As an indication of the usefulness of the hidden reservoir of neurons, experiments are also run with shallow network structures for comparison. In these networks, there are no hidden neurons and only two different synapse classes, the synapses connected to input elements, and the synapses connected to elements of the action vector of the previous time step. All the synapses project directly to the output units.

Three variations of experiments with shallow connections are run: 1) The synapse classes have the same number of parameters as the synapse classes in the main experiment (**Small Shallow**). Half of all the connections are still deleted at step two of the network initialization. In this setting, the total number of outer-loop parameters is just 144. 2) The synapse classes have a combined number of parameters that are similar to the combined number of parameters in the main experiments (**Big Shallow**). Half of all the connections are still deleted at step two of the network initialization. 3) The synapse classes have a combined number of parameters that are similar to the combined number of parameters in the main experiments, and no connections are deleted as part of initializing the networks (**Full Shallow**). The total number of outer-loop parameters in this setting is 5,040.

8.3.2.3 Standard LSTM:

We also optimize a standard **LSTM** with a number of parameters similar to that of the network of the main experiment as a policy in the two environments. Since the standard LSTM is not flexible in its input and output space, the LSTM is trained with fixed input and output sizes, and the parts that are not needed in a particular environment are simply masked by zeros and ignored. The fixed number of outputs of this LSTM is 10; two elements are to be used as continuous actions, four for the discrete environment, and four extra outputs are to be used during the evaluation of the models with unseen actions. Actions are sampled using the output vectors in the same manner as for the main experiments. The LSTM has a total number of optimized parameters of 5106.

8.3.2.4 Random LSTM:

The last set of models that were run was named **Random LSTM**. The difference between this model compared to the standard LSTM was that in every

new lifetime, the order of the observations and the actions were randomly shuffled to encourage permutation invariant properties. Further, an extra input element that was always set to zero doing training was concatenated to the input vector so that it is possible to also use the models in a variation of the point-finding environment that has three observation elements, described in more detail in Section 8.4. The total number of outer-loop parameters in this setting is 5218.

8.4 Results

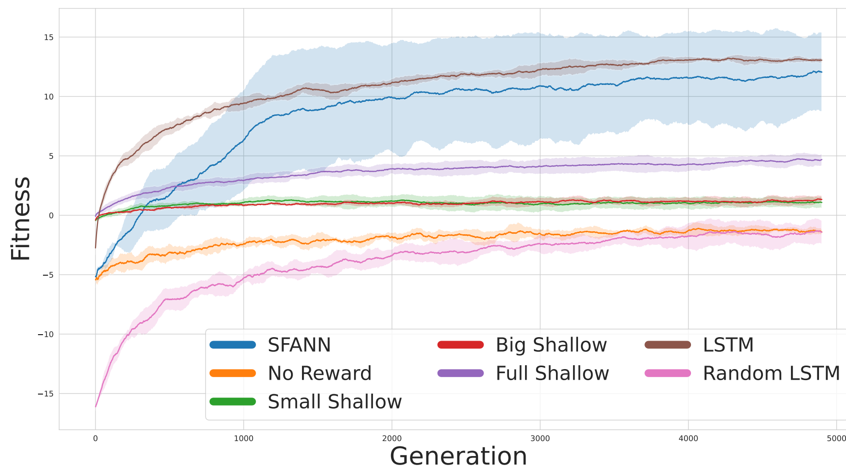


Figure 8.3: **Training plots.** Shown are the average and standard deviations of the population means of five runs for each model. The graphs depict the moving averages over 100 generations. Only the SFANN approach with neurons and the standard LSTM progress to achieve population means above a lifetime score of 10.

Training Curves of the different runs are shown in Figure 8.3. Only our proposed SFANN approach and the standard LSTM resulted in well-performing solutions within the 5,000 generations of evolution. The fully connected shallow version of SFANN with no hidden neurons did achieve better-than-random results but did not reach a solution that consistently solved the

tasks in the environments. The other shallow versions did not improve significantly during their evolution phases, and the population means remained close to zero. The LSTM with randomly selected input and output ordering never achieved a positive population mean. Unsurprisingly, with no reward information, the SFANN approach did not learn to solve the tasks in the environments either.

Figure 8.4 summarizes evaluations of the performances in 100 different tasks in the training environments. These are shown for the **SFANN**, the standard **LSTM**, and the fully connected shallow SFANN, **Full Shallow**. The lifetimes in these evaluations consisted of 10 episodes, regardless of the episodes' lengths.

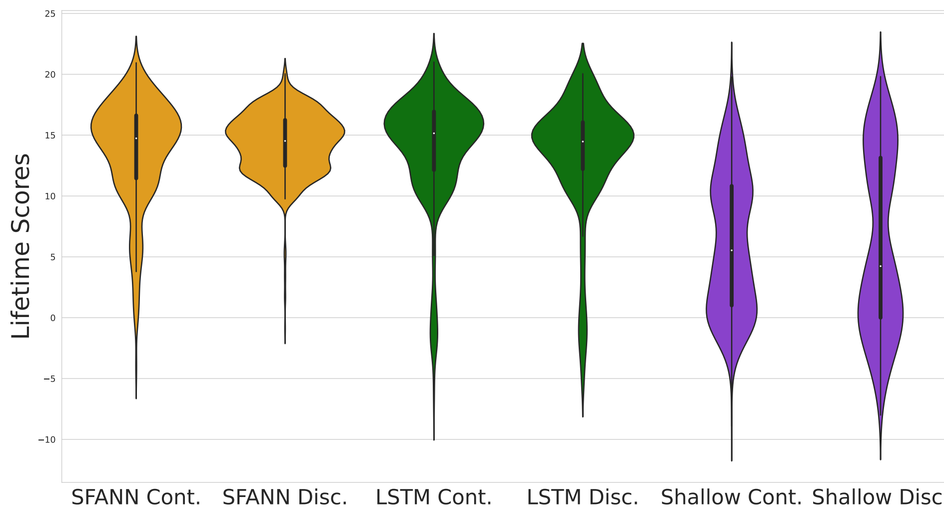


Figure 8.4: **Lifetime Performance in Training Environments.** Yellow violin represents scores of **SFANNs**. Green plots represent scores of the standard **LSTM**. Blue plots represent scores of the **Full Shallow**. In the environments that were seen during training, the LSTM scores are comparable to the SFANN scores. The shallow SFANN did not achieve good scores consistently. 100 lifetimes were run for each model in each of the respective environments.

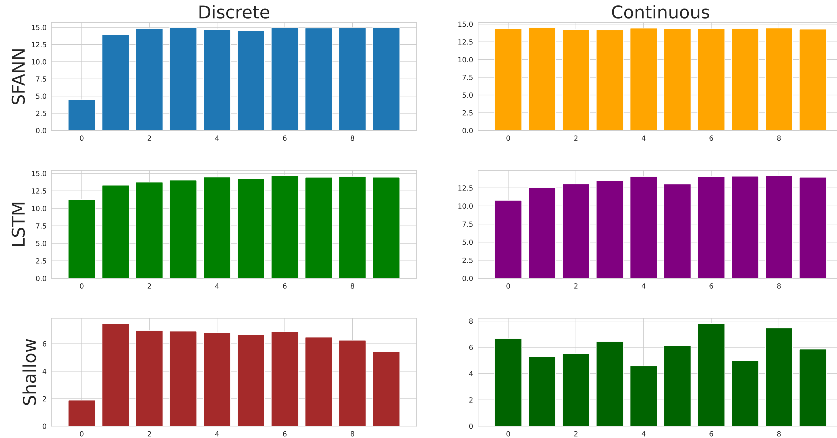


Figure 8.5: **Lifetime Progress in Training Environments.** Bar plots showing the average score of each episode that when combined make up the lifetime scores shown in Figure 8.4. Each bar represents the average score in the episode of its respective enumeration. Most models have lower scores in earlier episodes and achieve better scores upon repetition. Notable, SFANN has the same average score across all episodes in the continuous version of the environment, indicating rapid learning of the correct direction.

Figure 8.5 shows the models' average scores in each of the 10 tasks across the 100 tasks. The **SFANN** and **LSTM** are both able to sustain high scores throughout the lifetime, whereas the shallow SFANN fails to retain a reached score. On average, both SFANN and LSTM get high scores even in the very first episode, and in the continuous version of the environment, the SFANN is able to consistently find a fast route to the goal point within the first episode. This is not the case in the discrete version, but the agent is then, on average, able to solve the task in the second episode instead. Agents controlled by the LSTM tend to find the goal point in the first episode, and then find faster routes to the goal point in the subsequent episodes.

8.4.1 Evaluation in Novel Environments

To investigate whether our model has achieved the desired versatile learning ability, we test the trained models in variations of the point-finding environ-

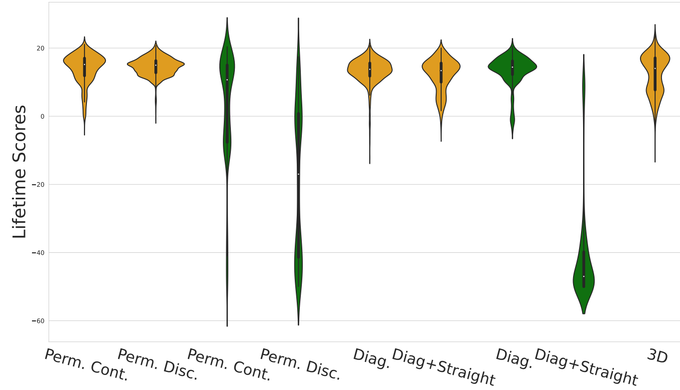


Figure 8.6: **Performance in Unseen Environments.** Yellow: SFANNs. Green: standard LSTM. Performances are in environments not seen during training. The SFANN approach generally does well in new environments. The LSTM only succeeds in the case where the straight movements (up, down, left, right) are switched for diagonal actions. Permuting the inputs and outputs severely decreases the consistency of the LSTM score compared to the scores in Figure 8.4. 100 lifetimes were run for each model in each of the respective environments.

ment that were not seen during the outer-loop optimization phase.

One variation is simply to randomly change the ordering of the observation and output elements of the model at the beginning of each new lifetime. Two other variations are introduced to the environment with a discrete action space. In one of these, the four straight movements (up, down, left, right) are replaced with four diagonal movements. In the other variation, the diagonal movements are, instead, added to the action space, such that there are eight available actions instead of four.

The last variation of the point-finding environment has a continuous action space, but a dimension is added to the environment, such that instead of moving around in a square, the agent is now searching for the point in a cube. This increases both the observation and action space to contain one extra element.

We only tested the **SFANN** and **LSTM**, as these were the only approaches that were successful in the training environments.

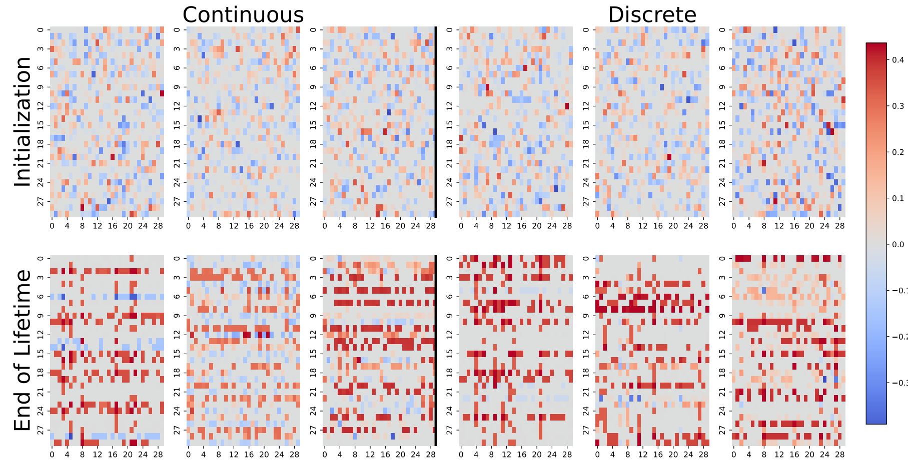


Figure 8.7: **Reservoir Connectivity Matrices Resulting after Lifetime.** The top row displays random initializations of the reservoir connectivity matrices of six different networks, three deployed in the continuous version of the environment and three deployed in the discrete version of the environment. On the bottom is shown how these connectivity matrices look after having spent 10 episodes in their respective environments. Since the hidden states of the synapses that make up the connectivity have two dimensions, each square in the plots depicted here is the sum of the two dimensions in its position.

8.4.2 Investigating Evolved Neurons

Figure 8.7 shows examples of different connectivity initializations of the hidden reservoir layer and the connectivity resulting from a successful lifetime in the training environments. The connectivity is depicted in the form of the summed hidden state elements of the synapses that connect neurons in the reservoir. These visualizations demonstrate that the SFANN can end up with different-looking weight matrices that all have similar performance.

Figure 8.8, shows the **SFANNs** performance across 100 lifetimes of each of the described training and novel environments depending on the number of neurons in the hidden reservoir. From this figure, the trend is that larger reservoirs have a more stable performance across many lifetimes. This is

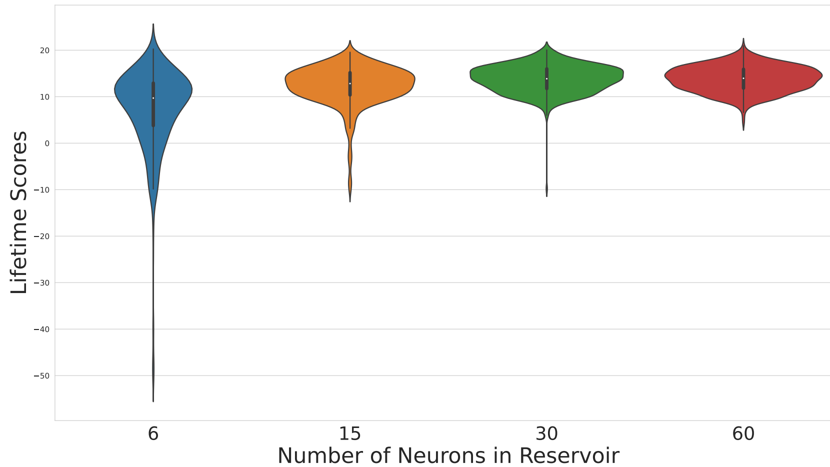


Figure 8.8: **Performance of SFANNs of Different Sizes** The violin plots summarize lifetime scores gathered from five different environments: the two training environments, the two environments that include diagonal movements, and the 3D environment. Each violin plot thus consists of 500 values, 100 values for each environment. The violin plots are separated by the number of neurons that were in the reservoir of the SFANN solving the environments. The trained parameters were the same across different network sizes, as well as all hyperparameters other than reservoir size. Less variation in performance is seen in the networks with larger reservoirs.

despite the fact, that the networks never had more than 30 hidden neurons during the outer-loop optimization phase. However, the connection matrix of the reservoir increases quadratically with the number of neurons, and a network of 60 neurons takes significantly more time to run than a network of, e.g., 15 or 30 neurons.

8.5 Discussion and Future Work

This chapter introduced a network type that can have outer-loop parameters optimized flexibly across environments with different input and output shapes, and learn online to optimize rewards during the lifetimes of the agents

that it controls.

Networks with shallow structures failed to improve on the same tasks that the networks containing hidden neurons performed well in. This shows that at least in this specific setup, the neurons played an important role.

The evaluation of the shallow structures revealed that these networks failed to properly de-correlate the output elements from each other, continuing to output very similar values for all output units throughout their lifetimes. This points toward a failure to resolve the symmetry dilemma for structurally flexible black-box meta-reinforcement learning, explained in Section 8.2.2. The reservoir of hidden neurons provided one way of mitigating this dilemma.

The SFANN was tested in a very simple environment. However, it should be noted, that due to the permutation invariance of the network, a network initialized in for any of the environments has no built-in knowledge about which directions in the coordinate system the observation vector corresponds to. Further, there is no built-in knowledge about the direction of its actions. These must be inferred during the agent's lifetime through the obtained rewards.

The standard LSTM network was able to solve both environments when given the appropriate number of actions to train with, and, as can be seen in Figure 8.3, had a faster training progression compared to the SFANN. However, as seen in Figure 8.6, the LSTM performed poorly on nearly all unseen tasks, with the exception of when the straight movements were replaced by random ones. Testing the standard LSTM on the three-dimensional version of the environment was not even an option, since the models were only trained with two inputs. To improve the chances of the LSTM models performing well in the test environments, we also trained LSTM models where input- and output elements were randomly shuffled at the beginning of the lifetime of each agent. Further, a third input element that was set to zero during the training phase was concatenated to the observations so that we would have the option of testing the models in the environment that required an extra input. However, these models never progressed enough to get a population with a positive mean score. Note, that the random LSTMs should be invariant to permutations to inputs and outputs since no parameters were trained relative to a specific ordering. However, as noted in Chapter 7, setting up the training in this way, requires that all possible permutations are shown during training. The issue is exacerbated when we are unsure how large we might need to have the input and output spaces be after training. The safest

option would be to choose the maximum sizes that we could need, but this of course extends the number of permutations combinatorially.

The SFANN method has a high number of hyperparameters and design choices. Some of these include how many parameters to use in the LSTMs that make up the units in the network, how many different neuron- and synapse classes should be included in the network, and the percentage of connections to delete as part of the initialization. Ideally, a hyperparameter sweep could be conducted with a grid search to empirically support the optimal settings. Alternatively - and more feasible in terms of computational resources - as discussed in more detail in Section 9.2.3 of Chapter 9, some of these hyperparameters could be included as parameters in the outer-loop and thus be optimized instead of being manually chosen by researchers.

On the other hand, the method also eliminates some choices that have to be made for other types of networks before training can begin. Most notably, we do not need to decide on the exact number of actions we would like to available for the agent after training. Even with a number of actions never seen during training, our networks could still learn to solve the task during its lifetime. Further, it is not required to select the number of hidden nodes beforehand, but only a range. This means that we after training can evaluate networks of different sizes and, for example, identify the smallest number of neurons needed and gain a functioning model with minimal computational expenditure.

These freedoms create the potential for the networks to be candidate models for foundation models for black-box meta-reinforcement learning. This would require training in as wide a range of environments as possible. Having trained a set of parameters to be able to improve on a wide range of tasks and environments would be a step toward a general learner. The more environments the SFANN is able to generalize across, the more useful it would potentially be in future tasks. The ultimate goal is to be able to put the trained neurons and synapses together in a network for any reinforcement learning-type task and let the network improve through interactions in the network. However, scaling up the approach to many more environments could face some obstacles.

The variations on the point-finding environments used in this chapter had the same range of values in the observations and the rewards. Training across a diverse set of tasks would require normalization of inputs and rewards. A more challenging issue might be the determination of the length of lifetimes that an agent should have within an environment during the training phase.

Environments with challenging tasks might require much more exploration and many more attempts than others, but it might not be obvious beforehand exactly how much time should be allocated to each specific environment. One strategy might be to simply have very long lifetimes in all environments, eliminating the need for choosing lifetimes on an individual basis for each environment. However, this would exacerbate the already costly training.

The experiments in this chapter used environments with small input and output sizes. All challenges related to the symmetry dilemma are magnified when input and output sizes are increased. More specifically, more input elements potentially make it harder for the input synapses with shared parameters to distinguish them and make properly individualized synaptic strengths for each of them. In the same vein, more outputs mean a greater risk that some of them do not become sufficiently de-correlated in their activations to provide adequate continuous actions. More experiments are needed to determine whether more classes of neurons and synapses in the hidden reservoir are sufficient to mitigate the symmetry dilemma in high-dimensional environments.

A point of discussion can also be made related to the design choice of utilizing a reservoir where hidden neurons of different classes are placed. The benefit of a reservoir layer is that it with its adjacency matrix in principle can implement any feedforward or residual network only constrained by its size and number of times per forward propagation the reservoir neurons are allowed to be updated. Of course, with the randomized initialization of the adjacency matrix, it is very unlikely that the connectivity is ordered such that activations propagate in a pure feedforward manner. Mixing neurons and synapses in the same layer is also an effective way of making the network less symmetric. At the same time, updating the hidden neurons multiple times by using a common adjacency matrix encompassing all neurons, gives the neurons the opportunity to incorporate information about the current state of all connected neurons into their own update.

However, breaking the symmetry of the network by placing neurons in the same layer with a random adjacency matrix requires the training phase to incorporate different permutations of the adjacency matrix and ordering of neurons so that the outer-loop parameters do not overfit to any specific configuration. This state of affairs resembles that which was criticized in Chapter 7, Section 7.2: achieving a network with permutation invariance by simply shuffling the observation vector throughout the training process so that the weights of the network do not overfit to a particular permutation.

This idea was discarded because of its inherent inscalability that stems from the fact that the number of needed permutations increases combinatorially with the number of elements in the observation. So what is different here? There are two points that make the need for training with multiple permutations more acceptable with the networks presented in this chapter. First, as opposed to the example in Chapter 7, Section 7.2 we do not expect that it is required that all possible permutations of the adjacency network are present during training. This is because of the black-box reinforcement learning capabilities of the LSTMs that control all units of our networks. All units get information about the global reward at each time step, and each unit can be seen as a distinct reinforcement learner that is evolved to adapt across different tasks. From the perspective of a neuron or synapse in the network, even if the environment and the task stay the same throughout optimization, it will be placed in different parts of the network in each new initialization of the network. It will thus be under much the same conditions as an LSTM trained as a policy network across different tasks. For this reason, it should be feasible for the units to adapt to structures not seen during the training phase, limiting how much the training needs to be extended from all possible permutations to just a subset of these.

Another reason that it is preferred to rely on training with multiple permutations of the hidden layer rather than the input layer is that the amount of permutations of the adjacency matrix in the hidden reservoir is to a lesser degree affected by the specific environment and to a higher degree by the choices we make in terms of the number of neurons and how many synapses are deleted. This is preferable when we want to optimize across different environments. Besides, these choices could be subject to optimization rather than being manually decided, which is clearly not the case for the size of the observation vector.

One could imagine other types of models to train for structural flexibility and adaptation. One possibility could be a sequence-to-sequence model (Yousuf et al., 2021) that takes input elements from the environment one at a time and outputs action elements one at a time. This model would have the same size regardless of the number of elements in the input- and output spaces. However, such a model would have some significant downsides. First, in many environments, such as the point-finding environments of this chapter, there is no meaningful sequential information between the inputs, but the processing of each new element would be affected by the previous elements. Further, the model would still depend on seeing many permuta-

tions of input and output vectors during training to avoid overfitting to a specific ordering in a way that might be harmful to performance in novel environments. Lastly, while it is generally true for all models that an increased input- and/or output space will result in increased computational requirements, this is especially true for sequence-to-sequence models as the one imagined here, which would need an entirely new forward propagation for each new element in the input.

Another option could be to use Set Transformers that are specifically designed to be permutation invariant and would not rely on being exposed to all permutations during the training phase (Lee et al., 2019). To make this type of model invariant to the length of the observation vectors from different environments, the set would be the observation elements for a given environment, instead of sequences of multiple time steps like in the case of the AdA framework. However, the standard architecture of Set Transformers does include fully connected feedforward layers in the network. This limits the possibility of varying the size of the networks after the outer-loop optimization phase, as was done in Figure 8.8 for the SFANNs. This inflexibility would also make it more difficult to eventually extend the approach to include a developmental growth phase. The potential of using a developmental algorithm as part of the SFANN foundation model is discussed in more detail in Chapter 9, Section 9.2.4.

On the other hand, attention mechanisms have proved to be powerful tools across multiple domains (Lin et al., 2022), and there could be ways of mitigating the structural inflexibility of the Set Transformers. Future studies could explore this as well as the possibility of combining attention mechanisms with the SFANN framework.

In future studies an interesting extension to the training and testing of the models' ability to learn multiple tasks during a lifetime. How difficult is it for the networks to reconfigure themselves to solve a new task after having already organized from its random initialization to solve one task? In order to be successful in such task switching, it will likely be necessary to incorporate multiple tasks into the agents' lifetimes during training.

Chapter 9

General Discussion and Future Research

This chapter will build upon the results and the discussions of the previous chapters. The general discussion will then be followed by suggestions for future studies that could further advance the pursuit of adaptive artificial agents.

9.1 General Discussion

The topics of chapters of this thesis have centered around approaches aimed at furthering the adaptive capabilities of artificial agents. Most approaches involved some degree of meta-learning as well as inspiration from biological neural networks. We have studied different ingredients that each have the potential to contribute to the goal of making artificial agents more robust to changes in their environments, making them more adaptable. Part of such a contribution has come in the form of a framework for thinking about meta-learning and how the two optimization loops affect each other as described in Chapter 3. However, the main contributions have come from the experiments in Chapters 4 to 8 as summed up below.

In Chapter 4, we presented experiments involving plastic neural networks and showed that unique learning rules can be merged together to form smaller rule sets without decreasing the performance. The activity-dependent Hebbian learning rules used in these experiments were shown to yield agents more robust to morphology changes than static networks.

A less explored aspect of activity-dependent plasticity in artificial neural networks is the activity of the neurons themselves. In Chapter 5, neuro-centric parameters were optimized to enable different artificial neurons to have unique, history-dependent outputs to identical inputs. The expressiveness of neuro-centric parameters in networks with fixed random connectivity matrices was shown to be sufficient to perform well in several environments.

In Chapter 6, a simple RNN was evolved to provide a reward signal for a gradient-based RL algorithm, and the learned reward was shown to accelerate learning in both seen and unseen scenarios. Chapter 7 revolved around structural flexibility and the principles that must be adhered to in order to build a network that has this kind of flexibility. Structural flexibility is important for enabling a network to work across environments that have different input and output sizes.

The concept of structural flexibility was further explored in Chapter 8, where some of the ideas of the earlier chapters were combined. Here a set of preliminary experiments were presented where both synapses and neurons were represented by small RNNs with shared parameters. It was shown that such networks could exhibit flexibility in both input-, output-, and hidden layers. This meant that the same set of parameters could be concurrently optimized on tasks that had different input and output sizes. Driven by a reward signal, a network could be initialized randomly and then self-organize into a functional network over the lifetime of the agent.

Together, these experiments provided validation for different biology-inspired ideas, showing that they could work independently. In Chapter 8, the first steps toward putting these ideas together in a combined approach were taken. However, much work is still needed in order to enhance how these are optimized and improve the possible final structures. Such work requires outlining new experiments, as well as considerations of the overall framework. To this end, several topics that are general to the presented experiments are worth discussing, before suggesting potential future studies.

9.1.1 Bio-Inspiration and the Bitter Lesson

Our methods have been inspired by biological systems and proposed ways of implementing some similar capabilities. However, there are no claims that the proposed methods are biologically realistic at the level of implementation. So what is the value of biological inspiration? In recent years, several researchers have argued for using biology and in particular neuroscience as

inspiration in developing new AI approaches (Zador et al., 2023; Poo, 2018; Kudithipudi et al., 2022). One of the main arguments is that since today's agents are still far from having achieved the same level of adaptiveness to novel situations as animals have, it makes sense to take inspiration from the biological counterparts of ANNs.

However, Sutton (2019) pointed out that historically, it is the approaches that are best able to leverage increasing computational resources that tend to best stand the test of time: "We have to learn the bitter lesson that building in how we think we think does not work in the long run." In outlining different paths toward general AI, Clune (2019) argues that approaches that seek to mimic biological brains are not likely to be the fastest to get us to general AI. This is because such approaches by design do not take advantage of potential abstractions that are functionally but not implementationally equivalent to, e.g., neocortical columns.

More important than achieving biological realism in models is whether an approach, when pushed to its limits, holds the potential to yield stronger more general artificial intelligence. Approaches that closely resemble biological implementations of intelligence could be considered advantageous in this regard since we know that biological implementations have resulted in intelligent beings. However, despite the growing body of neuroscientific knowledge (Kandel et al., 2000), the exact ingredients necessary for the emergence of intelligence remain elusive. This uncertainty poses a significant challenge in implementing biologically inspired intelligence, as researchers must rely on educated guesses, and at times, arbitrary assumptions when optimizing our models. Each assumption that requires handcrafted programming carries the risk of leading down an ultimately unfruitful path. It could demand substantial resources to realize that a chosen ingredient, or even a set of ingredients, is leading to a dead-end.

That is not to say that any research that does not ultimately result in strong and general artificial intelligence is useless. AI research has produced numerous practical and useful tools that have found applications across multiple domains. Furthermore, it is often impossible to predict the future implications of a particular approach (Stanley and Lehman, 2015). Following the line of thinking presented by Sutton and Clune, it is possible that the most general and wide-ranging approach has the best chance of discovering potentially narrow and specific paths.

The stance taken in this thesis is that the objective of drawing inspiration from biological brains is not solely to achieve biologically accurate or realistic

solutions. Rather, the emphasis lies in incorporating mechanisms known to benefit biological systems into our approaches. However, this should not hinder the ability of the approach to leverage increasing computing power. In other words, the preferred methods are those capable of expressing solutions that are closely aligned with biological implementations of intelligence as a subset of the broader range of possible solutions. While biologically plausible mechanisms hold value, the optimization process should have the flexibility to explore other effective strategies without being constrained to exclusively biological solutions.

By adopting this approach, we acknowledge the potential benefits of incorporating biologically inspired elements while remaining open to alternative approaches that may leverage computational advantages. This approach allows for the exploration and discovery of efficient and effective solutions, regardless of their strict adherence to biological systems. The optimization process should be able to identify and utilize biologically plausible mechanisms if they prove advantageous, but it should not be limited to them in order to encourage the exploration of a wider solution space.

Comparisons between artificial and biological neural network representations have suggested that artificial neural networks can find solutions that represent inputs in ways that resemble representations in biological neural networks (Khaligh-Razavi and Kriegeskorte, 2014). Evolution of the type of network presented in Chapter 8 with neurons and synapses being represented by RNNs arguably has the potential to result in structures and representations that resemble biological networks even more. The reason for this is that like biological neural networks, synapses and neurons are dynamical systems (Izhikevich, 2007). The parameterized RNNs that make up different neuron and synapse types have the potential to result in behavior that imitates biological neurons and synapses. However, in contrast to their counterparts found in spiking neural networks, the RNN representations are general enough to also encompass functions that are biologically implausible.

In sum, the position of this thesis on the discussion on the value of biological inspiration is that methods that can incorporate some of the properties found in biological networks are preferred to ones that cannot, as long as this does not detract from the methods' ability to find other solutions that are not faithful to any biological realism. It is further argued that networks of recurrent units with partially shared parameters, for which the blueprint is described in Chapter 8, represent an approach that has the potential to live up to this preferred criteria. To make that argument even stronger, some

suggestions for extending and improving the evolution of such networks can be found in the Discussion section of Chapter 8, as well as Sections 9.2.4 and 9.2.3 below.

9.1.2 Learning versus Adaptation

The experiments presented in this thesis encompass both learning and adaptation, prompting a discussion on the preference between the two and the circumstances that warrant each approach. In this context, learning and adaptation can be seen as points on a spectrum. The distinction lies in the extent of change occurring to the agent's starting point and the effort required to enact that change. Effort can be loosely measured by the time it takes to implement the desired modifications. Adaptation involves smaller changes that can be achieved with relatively little effort. Conversely, an agent capable of learning can enact more significant changes but at a higher cost in terms of effort.

The abilities of both adaptation and learning can be cultivated through a meta-learning setup. Referring to the framework of meta-learning described in Chapter 3, the determination of whether an agent exhibits learning or adaptation depends on how the outer-loop optimization process influences the inner-loop's ability to optimize within changing inner-loop loss landscapes. As discussed in Chapter 3, the outer-loop can impact inner-loop optimization through one or a combination of three different ways. To achieve an inner-loop that demonstrates what we classify as adaptation, the outer-loop could, in any combination, (1) identify a starting point that is close to good solutions in multiple inner-loop loss landscapes, (2) adapt the loss landscapes of multiple inner-loops to position good solutions at similar points, or (3) optimize the inner-loop optimizer to search only in areas that have proven promising throughout the training phase.

One could argue that for learning since all the same ways of improving the inner-loop are available, the only difference between learning and adaptation in the meta-learning setting is how many resources are allowed in the inner-loop. While this is in principle true, some practical considerations make the two regimens differ. It should be noted that for adaptation, problems like catastrophic forgetting and the stability-plasticity dilemma are not as prominent problems as for learning. It is, for example, not as important to simultaneously maintain the ability to control many variations of a robot morphology if these abilities can be picked up quickly, should the morphology

change from one setting to another. The longer a skill takes to learn, the more costly it is to forget. The problem of catastrophic forgetting was not the focus of the experiments of this thesis, but this should ideally be taken into account when setting up the meta-learning procedure. The outer-loop can mitigate catastrophic forgetting in several ways. Riemer et al. (2018) added a term to their objective function to minimize forgetting with new inner-loop updates. Beaulieu et al. (2020) optimized two networks in the outer-loop one of which modulated the outputs of the other thus modifying the inner-loop updates of the other network to encourage updates that maintained earlier learned skills. In terms of adapting the inner-loop loss landscapes themselves, one strategy would be to make the optimal areas for the inner-loop parameters as wide as possible so that it takes larger movements in the parameter space to lose an ability once it has been learned.

It is important to note that optimizing an agent to possess the ability to learn necessitates a longer inner-loop training setup. This distinction presents a significant difference between the training setups described in Chapters 4 and 5 compared to the training setup required for the experiments in Chapter 8. In the latter, agents have lifetimes spanning multiple repeated episodes, making the training process more computationally expensive.

These observations underscore the increased complexity involved in optimizing for learning as opposed to adaptation. However, there is no inherent reason to believe that it is impossible to optimize for both learning and adaptation within the same agent, provided that the neural architecture can accommodate it. One possible approach could involve allowing shorter inner-loop training times for variations of essential skills to encourage adaptation while allocating more time for other tasks sampled within the inner-loop.

By designing training setups that strike a balance between learning and adaptation, we can harness the benefits of both capabilities, leading to more versatile and capable agents. This nuanced approach recognizes the potential trade-offs involved and seeks to optimize the agent's neural architecture and training procedures accordingly.

9.1.3 Limitations of Episodic Tasks

All experiments have used episodic environments for testing the proposed approaches. However, real-world use-cases might, generally speaking, not be structured in such clearly defined episodes. One natural example of episodes in the real world could be the day-night-cycle. However, a full day could

contain many different tasks that are hierarchically interwoven and that need to be solved at different timescales, and thus very different from the simple environments with a single defined task per episode.

On the other hand, there is nothing in particular in the presented approaches that excludes them from being optimized in non-episodic environments. The Hebbian learning approach used in chapter 4 only relies on the inputs to adapt its parameters. Likewise, the permutation invariant models presented in Chapter 7 re-configure their input units online with no regard to when in the episode the permutation occurs.

The episodic structure is arguably more important for the approaches that involve learning from a reward signal, namely the approaches from Chapter 6 and Chapter 8. Episodic structures are in these cases important insofar as they determine the reward structure of the environment. For example, if rewards are only provided at the end of an episode, the length of episodes determines how sparse the reward signal is, and thus how often the parameters of the network can be adapted towards maximizing the rewards. In the case of the point navigation task, it would be significantly harder for an individual starting with random connectivity to learn anything if it was only rewarded at the end of each episode instead of at each time step. This is because there would be significantly fewer opportunities to update the inner-loop parameters in a goal-directed manner. Learning usually becomes harder, when feedback is rare as demonstrated by, e.g., Dubey et al. (2018) in experiments where usual human priors were rendered useless in modified video games. As a result, humans had a difficult time self-evaluating progress and the reward signal effectively became more sparse. Showing that an individual is able to improve with a sparse reward signal should therefore be considered more convincing than learning from dense, informative rewards as was the case in Chapter 8.

In sum, the use of episodic environments is less important in and of itself than the consequences the episodic structure might have on the reward structure of the environment.

9.1.4 Scaling Up Experiments

The experiments presented in the thesis have mainly served the purpose of proving the viability of proposed approaches, rather than improving the state-of-the-art for the task environments in which they were tested. All series of experiments could be scaled up both in terms of the number of

environments tested as well as the number of replications in each environment. To enhance the comprehensiveness and robustness of the findings, it would be beneficial to scale up the experiments in terms of the number of environments tested and the number of replications conducted in each environment. Naturally, such an expansion would require additional resources. Nevertheless, investing in the scaling up of these experiments would solidify the knowledge acquired thus far, while also enabling a more nuanced investigation of the approaches through statistical tests of the performances of models with small variations from the main ones. Such up-scaling would also help map out which types of environments the different approaches are most suited to solve. With this additional knowledge, necessary adaptations to the approaches could become apparent, if, for example, there is a certain type of environment wherein the approach systematically fails to improve.

9.1.5 Rewards for Reinforcing Adaptation and Learning

Only the experiments of Chapter 6 and 8 involved a reward signal in the inner-loop. In Chapter 8, the reward signal given to the agent during its lifetime was the same signal used to calculate the fitness for outer-loop optimization. In Chapter 6, the whole point was to optimize a reward signal for inner-loop learning different from the original reward signal. It is not a requirement that the reward signals for the two loops are the same, although this is the case in much of the meta-learning literature. Most often, however, we might want the reward signals for the two loops to be aligned, so that optimizing one reward signal also improves the optimization of the other. As done in Chapter 8, it likely makes sense for most purposes to have the reward signal provided to the outer-loop be as close as possible to reflecting the performance of the task we are actually interested in solving. As for the reward signal provided to the agent during its lifetime in the inner-loop, the type of feedback signal provided to it is less important, as long as it results in behavior that solves the task in its given environment. Considerations of the reward signal given to the agent become increasingly important when the goal is to optimize the agent over tasks sampled from multiple domains. If these tasks do not share a reward structure, it creates an added challenge for generalization, in that the agent potentially has to be able to cope with different input- and output sizes, different magnitudes of input values, as

well as different guiding signals to improve from. This extra challenge could be mitigated by shaping the reward signal given to the agent to be consistent across different environments.

Part of the solution could be to include a generic reward signal that can always be used across environments. A framework like SFANN, where inputs to the network's units are already in the form of vectors, is not limited to reward signals that consist of a single scalar. The reward signal could be a vector as well. A simple example could be a signal after each episode indicating whether or not the agent achieved a new personal high-score. Such a signal could be present in any environment with scores and it could make up its own element of a reward vector. Other generic extensions of the reward signal could come from the literature on intrinsic motivation (Aubret et al., 2019). However, for SFANN, most of these techniques could be problematic in that there is an element of learning of the intrinsic motivation. One intrinsic motivation technique uses a measure of surprise that is determined by the ability of e.g., an autoencoder associated with the agent to reconstruct the input (Hafez et al., 2019; Bougie and Ichise, 2020). Such a method is not easily transferable to the SFANN framework, as the inputs across different environments do not have the same sizes, and the same autoencoder could not be used. This also exposes why the EIR-RL of Chapter 6 does not fit easily in the SFANN framework: The RNN of EIR-RL takes the input from the environment as well as the previous action of the agent as inputs, and none of these have fixed sizes across environments. Within the framework of SFANN, the closest solution to EIR-RL would be to have one or more neuron classes within the network have their outputs concatenated to the reward signal that is sent to all the other neurons. However, since the connectivity of such neurons would themselves also necessarily be random initially, they would first need to be guided by a different reward signal before potentially being useful later in the lifetime of the agent.

9.1.6 Evolving General Learners: Ending Evolution or Open-Ended Evolution?

As highlighted earlier, the training process plays a crucial role in developing artificial agents with learning capabilities. One key consideration is providing the agent with sufficient time to learn the given task within its lifetime. The extent to which an agent can become a general learner depends on the

diversity of learning tasks it encounters during the outer-loop optimization process. For instance, the study by Kirsch et al. (2022) on emergence in transformers demonstrates the importance of a varied set of training tasks in shaping the agent's learning abilities.

However, determining the appropriate set of training tasks or designing an effective curriculum is not a straightforward task. It raises the question of the feasibility for researchers to anticipate the specific tasks that are necessary to foster interesting, useful, and general learning behaviors. As Clune (2019) emphasizes, the selection of a suitable training set or curriculum is a non-trivial challenge. Given this complexity, it may be unrealistic to expect that training an agent with a finite outer-loop alone will result in an agent capable of learning on a sufficiently general level to excel in unstructured real-world scenarios.

Open-endedness in training approaches offers an alternative to handcrafting a fixed training set, providing a more dynamic and adaptable learning environment. Open-endedness can be described as a characteristic of systems that generate novel and diverse outputs continuously, without predetermined limits (Stanley et al., 2017). In this context, agents can be rewarded for tasks of interest, but the specific conditions of these tasks are not fixed throughout the optimization process.

The SFANN architecture aligns well with open-ended settings due to its structural flexibility. Allowing individuals to modify the number of neurons in any layer of the network facilitates convenient co-evolution of agents' bodies and brains. For instance, in an environment where agents start with small and simple body morphologies and networks, mutations that extend the brains and bodies simultaneously expand the possibilities of evolutionary processes. This flexibility contrasts with cases where the neural network, and consequently the inputs and outputs of the body's sensors, remain fixed in size indefinitely. Using a more open-ended training regimen and leveraging adaptable neural network architectures, researchers can explore the emergence of diverse and novel behaviors that go beyond handcrafted tasks and promote more flexible and robust learning in artificial agents.

9.2 Future Research

9.2.1 Evolve & Merge Neural Parameters

A natural progression from the experiments conducted in Chapters 4 and 5 would be to explore the combination of both approaches into a unified framework. There are several possible ways to achieve this combination. One approach could involve optimizing the parameters of both Hebbian learning rules and neural units simultaneously but keeping them separate during the merging process. This means that the learning rule set and neural unit set could be merged independently of each other. To maximize the effectiveness of this merging process, it may be beneficial to stagger the events of merging the learning rules and the neural units, allowing each newly reduced parameter set to adapt and recover from any temporary performance dip before reducing the other parameter set.

The motivation behind combining the optimization of neural parameters and local activity-dependent learning rules is the potential synergy that could arise from these parameter sets. A hypothesis for this study could be that by combining the two sets, it is possible to achieve a smaller combined parameter size while maintaining or even improving performance compared to using the evolve and merge approach on each parameter set separately. Control experiments should be conducted with networks of similar sizes, where only the learning rules or only the neural units are evolved and merged. In all cases, the merging of parameter sets should continue until a permanent drop in performance is observed, indicating that there are too few parameters to adequately solve the given task.

An interesting question to address in this study is whether it is more effective to first reduce one parameter set as much as possible before attempting to reduce the other set, or if it is preferable to interweave the merging events of the two parameter sets. Exploring the optimal sequencing of parameter set merging could shed light on the dynamics of the combined approach and provide insights into the best strategies for reducing parameter size without sacrificing performance.

9.2.2 Automatic Evolve & Merge

Another future study involving the evolve and merge approach involves extending the approach to have the timing and intervals of parameter merging

be controlled by the evolution algorithm, rather than being manually decided by the researcher. This could be realized as part of a genetic algorithm as a mutational operator. More specifically, in each new generation each individual would have some probability of undergoing parameter merging. Given that parameter reductions are associated with a loss of performance, some mechanism of innovation protection should be put in place, so that individuals have some time to recover performance after a merging event, without the risk of going extinct immediately.

Additionally, an accompanying mutational operator could be introduced to split learning rules or neural unit parameters, thereby expanding the number of optimizable parameters. By dynamically changing the parameter count as part of the optimization process, the approach could automatically converge at the optimal number of parameters, eliminating the need for manual decisions by researchers.

It would be intriguing to investigate whether replications with different seeds would yield consistent numbers of parameters for the same environment and network size, or if there would be significant variance across seeds. Comparing the performance and the final number of parameters obtained using this approach with NEAT (Stanley and Miikkulainen, 2002), another genetic algorithm that evolves the number of trainable parameters could provide valuable insights into the relative effectiveness of different optimization strategies.

Furthermore, once the final number of parameters is obtained, it might also be enlightening to try and optimize standard fully connected networks with a similar number of trainable parameters on the same task. This comparison could contribute to the discussion on different types of parameters in artificial neural networks. Many studies compare the trainable parameter counts across different models but is a parameter just a parameter? How can we take into account the context in which a parameter is embedded when determining its value in terms of how much it contributes to the expressiveness of the overall model?

9.2.3 Dynamic Genome Size for Networks of Neural Units

In the same vein as a more dynamic, extended version of evolve and merge, it would likely be useful for the optimization of structurally flexible neu-

ral networks consisting of RNN units to have more of what are currently hyperparameters optimized rather than hard-coded.

The experiments in Chapter 8 were preliminary and there are many ways the approach could be enhanced to potentially be much more expressive and to better fit the description of a path towards more general artificial intelligence as discussed in Section 9.1.1. Having an optimizable genome that not only optimizes the parameters of the LSTM gates. A way to better leverage computational power would be to have mutational operators decide:

- The range that the number of hidden neurons in the reservoir can drawn from.
- The number of different neural classes in the network.
- The proportion of the total number of neurons each neuron class should take up in the network.
- The probability that each neural class has to connect to each of the other neural classes.
- The number of times the reservoir of hidden neurons should project to itself before the activity is propagated to the output of the model.
- The initial values of the hidden states and cell states of each neuron class and its associated synapses.

Making these aspects of the model subject to evolution instead of manual tuning could greatly enhance the applicability of the method, especially when the goal is to use the same neural units in several different contexts. As the optimization process becomes more intricate with multiple moving parts, it becomes increasingly challenging for researchers to accurately determine adequate hyperparameter values.

These extensions of the optimization process could in principle result in one specific network with no room for variation in network initializations for different individuals. In an extreme case, a network might be evolved with a fixed number of neurons in the reservoir, where each hidden neuron belongs to its own distinct class, with the predetermined connection probabilities to other neurons being exclusively either one hundred or zero. While it remains to be seen whether such a network would be able to generalize across multiple environments with varying input and output sizes, it is within the realm of possibility for the approach to yield such a solution.

9.2.4 Developmental Phase to Supplement Symmetric Randomness

A central theme explored in Chapter 7 and 8 revolves around the concept of structural flexibility and the necessary conditions for achieving it within a model. The fundamental requirement is that no parameters in the network should be optimized with respect to a fixed position, as this would result in parameter overfitting to the position and thus the loss of structural flexibility. To mitigate this issue, parameter sharing across neural units can be employed, which prevents parameters from being overly specific to a particular network position. As demonstrated in Chapter 8, it is possible to meet this requirement while still allowing for diversity within the network, by subjecting the connectivity matrix of hidden neural units to randomness during network initialization.

In addition to relying solely on randomness for network initialization, an extension of the approach could involve incorporating a developmental phase to guide the network's structure throughout the agent's lifetime. The structural flexibility potentially allows for leveraging a developmental phase of the network structure throughout the lifetime of the agent.

Introducing a developmental phase that progressively grows the network, as opposed to directly initializing it based on probabilities, offers several potential advantages. First, it may result in reduced performance variability within a parameter set, as the developmental phase could yield networks with more consistent morphologies across different initializations. Additionally, a developmental process could help avoid network structures that, by random chance, are incapable of learning. Using randomness for making an asymmetric network structure does come with a challenge related to the amount of training data required for the outer-loop optimization problem. If, for example, random connections are removed at initialization, there is a risk that there simply are not sufficient connections present from one layer to the next to propagate information from the input needed to solve the task. This would make the outer-loop parameters achieve a bad fitness score, regardless of how great the parameter set would be in a functional network structure. This in turn makes the optimization process more noisy and could stifle progress, especially early in the optimization phase. There are some ways, not related to growth, that might mitigate this issue. One solution is to calculate the fitness of a parameter set as an average over more lifetimes, that are initialized independently from each other. This would reduce the

risk of any parameter set being unlucky. However, it would also increase the required training data, likely many-fold. Another solution could be to increase redundancy in the network by simply using more hidden neurons, making it more likely that some circuits or connectivity patterns are present across different initializations. However, added redundancy comes with an added cost of propagating information through the network. A developmental process that grows the network structures in a directed manner based on interactions with the environment might prove to be the better solution. Some form of algorithmic growth not only makes sure that dysfunctional structures are avoided, but it could also serve as a bottleneck for the number of different structures and network permutations that may appear during the training phase. If this could be achieved, this would mean that the demand on outer-loop parameters' ability to generalize over essentially all possible network structures would be relaxed.

Another advantage of the incorporation of a developmental phase could facilitate the creation of networks with reservoirs of minimal size necessary to solve a given task. By initially starting with a small network and allowing it to grow as needed, potential redundancies that arise from randomly selecting network size can be minimized.

Another potential benefit lies in the opportunity to synchronize the development of the neural network with the agent's body. An environment where the agent begins with a simple body and brain could serve as a curriculum, with initial stages involving constrained and simple actions that gradually evolve into more complex behaviors.

Furthermore, a developmental phase can better tailor the network's structure to different tasks. In the SFANN framework presented in Chapter 8, certain aspects of the network structure, such as zero entries in the connectivity matrices and the number of neurons in the hidden layer, are determined during network initialization without considering the task. However, for specific tasks, certain network morphologies might be more effective than others. Thus, incorporating the task environment into the network's developmental phase allows for the incorporation of task-specific inductive biases early on, enhancing the learning capabilities of dynamic neurons and synapses throughout the agent's lifetime in that particular task.

A developmental phase could also be useful for sequentially switching between environments of different dimensionality in inputs or outputs. We might have a network initialized in one environment and have it learn a functional network configuration here. If we then wish to solve a new envi-

ronment, leveraging some of the skills learned in the first, a developmental phase during the initial interactions in the new environment could determine how the network should grow or adapt its structure beyond synapse strengths in a way that preserves the learned skills from the original environment.

Designing a useful developmental phase that can grow the network during the lifetime is not a trivial task, and potentially adds a lot of moving parts to the approach. A simple idea might just employ pruning of connections (Sietsma and Dow, 1991; Blalock et al., 2020). For example, if part of the hidden state of a synapse reaches a certain threshold during a developmental phase, this could serve as a signal for it to be removed.

Alternatively, more sophisticated developmental algorithms could be used. One example could be the recent approach to a neural developmental program by Najarro et al. (2023). Here one neural network guides the development of another network based on the activations of the growing network. This, of course, comes with the added complexity of having to optimize the neural developmental program.

The growth signal could also come from the evolved neurons themselves. In the same way as some neurons in the work of Soltoggio et al. (2008) come to have a dedicated responsibility for exerting neuromodulatory signals, evolved neurons in SFANN could be responsible for signaling when more neurons need to be added to the network.

Chapter 10

Conclusion

The successes of ANNs have inspired research in a plethora of new directions, with new network architectures and mechanisms being introduced at a fast pace.

The purpose of this thesis has been to contribute to the field of research in adaptive agents controlled by ANNs. Despite all the advancements that have been made with ANNs in recent years, agents that can adapt to and learn from their surroundings to the same degree as animals – let alone humans – still elude us. For this reason, approaches introduced throughout this thesis have sought to build on the tradition of taking inspiration from the mechanisms in biological neural networks that are believed to contribute to their remarkable abilities to learn and adapt. The work in this thesis has revolved around synaptic plasticity, neural diversity, and structural flexibility.

The contributions of the thesis culminated with the introduction of the SFANN approach in Chapter 8, as well as the suggestions for its extensions in Chapter 9. SFANN combines the concepts used in earlier chapters and implements them in a way that generalizes the Variable Shared Meta-Learning framework. A key component of this was to identify and attempt to mitigate the symmetry dilemma for black-box meta-reinforcement learning with structural flexibility properties. The way to identify the symmetry dilemma was paved for by the reasoning of Chapter 7 which was used to construct minimal permutation invariant neural network models. One of the observations made in Chapter 7 was that even if the *Evolve & Merge* approach of Chapter 4 was able to significantly reduce the number of the plasticity rules of the network, the rules would still be tied to the particular network architecture that the rules were trained in, as the rules were optimized relative to specific

coordinates of the connectivity matrices in the network. The only way that the evolved plasticity rules could be used in other network structures was if it was possible to merge the rules into a single rule for the whole network. This seemed unlikely exactly because of the symmetry dilemma.

An important component of learning is the feedback provided to learn from. This was the focus of Chapter 6. The evolved internal reward network was shown to enable faster learning for the RL agents, and as argued in Chapter 9, even if the exact same method might not be compatible with the SFANN framework, incorporating similar mechanisms is likely a promising direction for enhancing the SFANN framework as well. This ties into the common ultimate goal of the contributions of the thesis to achieve artificial agents that adapt to their environments.

The SFANN was argued to have the potential to be trained in a wide range of environments for black-box meta-reinforcement learning due to its plasticity and flexible structure. It was further argued that the neural and synaptic diversity of SFANN makes it better capable of coping with the dangers posed by the symmetry dilemma compared to similar earlier approaches. Experiments in simple environments with discrete and continuous output spaces of different sizes showed that this framework indeed showed promising learning abilities across different network structures. Further developments to this promising framework could evolve networks of any complexity in terms of the number of different neural and synaptic classes as well as their respective probabilities of connecting with each other. Whether the approach will evolve structures that resemble structures found in biological neural networks remains to be seen. SFANNs contain at least some key ingredients to make this possible, in that the networks consist of sub-units that are all dynamical systems, with history-dependent neurons being connected by plastic synapses.

Bibliography

Momin Abbas, Quan Xiao, Lisha Chen, Pin-Yu Chen, and Tianyi Chen. Sharp-maml: Sharpness-aware model-agnostic meta learning. In *International conference on machine learning*, pages 10–32. PMLR, 2022.

Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

Larry F Abbott and Sacha B Nelson. Synaptic plasticity: taming the beast. *Nature neuroscience*, 3(11):1178–1183, 2000.

Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence*, 297:103500, 2021.

Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

Arthur Aubret, Laetitia Matignon, and Salima Hassas. A survey on intrinsic motivation in reinforcement learning. *arXiv preprint arXiv:1908.06976*, 2019.

Bruno B Averbeck and Vincent D Costa. Motivational neural circuits underlying reinforcement learning. *Nature Neuroscience*, 20(4):505–512, 2017.

Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23, 1993.

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

Mina Basirat and Peter M Roth. The quest for the golden activation function. *arXiv preprint arXiv:1808.00783*, 2018.

Gary K Beauchamp. Why do we like sweet taste: a bitter tale? *Physiology & behavior*, 164:432–437, 2016.

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

Lou Beaulieu-Laroche, Enrique HS Toloza, Marie-Sophie Van der Goes, Mathieu Lafourcade, Derrick Barnagian, Ziv M Williams, Emad N Eskandar, Matthew P Frosch, Sydney S Cash, and Mark T Harnett. Enhanced dendritic compartmentalization in human cortical neurons. *Cell*, 175(3): 643–651, 2018.

Jacob Beck, Risto Vuorio, Evan Zheran Liu, Zheng Xiong, Luisa Zintgraf, Chelsea Finn, and Shimon Whiteson. A survey of meta-reinforcement learning. *arXiv preprint arXiv:2301.08028*, 2023.

Harkirat Singh Behl, Atılım Güneş Baydin, and Philip HS Torr. Alpha maml: Adaptive model-agnostic meta-learning. *arXiv preprint arXiv:1905.07435*, 2019.

Eseoghene Ben-Iwhiwhu, Pawel Ladosz, Jeffery Dick, Wen-Hua Chen, Praveen Pilly, and Andrea Soltoggio. Evolving inborn knowledge for fast adaptation in dynamic pomdp problems. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 280–288, 2020.

Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

David Beniaguev, Idan Segev, and Michael London. Single cortical neurons as deep artificial neural networks. *Neuron*, 109(17):2727–2739, 2021.

José Manuel Benítez, Juan Luis Castro, and Ignacio Requena. Are artificial neural networks black boxes? *IEEE Transactions on neural networks*, 8 (5):1156–1164, 1997.

Peter J Bentley and Sanjeev Kumar. Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In *GECCO*, volume 99, pages 35–43, 1999.

Paul Bertens and Seong-Whan Lee. Network of evolvable neural units can learn synaptic learning rules and spiking dynamics. *Nature Machine Intelligence*, 2(12):791–799, 2020.

Garrett Bingham and Risto Miikkulainen. Discovering parametric activation functions. *Neural Networks*, 148:48–65, 2022.

Garrett Bingham, William Macke, and Risto Miikkulainen. Evolutionary optimization of deep learning activation functions. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 289–296, 2020.

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Nicolas Bougie and Ryutaro Ichise. Skill-based curiosity for intrinsically motivated reinforcement learning. *Machine Learning*, 109:493–512, 2020.

Konstantinos Bousmalis, Giulia Vezzani, Dushyant Rao, Coline Devin, Alex X. Lee, Maria Bauza, Todor Davchev, Yuxiang Zhou, Agrim Gupta, Akhil Raju, Antoine Laurens, Claudio Fantacci, Valentin Dalibard, Martina Zambelli, Murilo Martins, Rugile Pevceviciute, Michiel Blokzijl, Misha Denil, Nathan Batchelor, Thomas Lampe, Emilio Parisotto, Konrad Żołna, Scott Reed, Sergio Gómez Colmenarejo, Jon Scholz, Abbas Abdolmaleki,

Oliver Groth, Jean-Baptiste Regli, Oleg Sushkov, Tom Rothörl, José Enrique Chen, Yusuf Aytar, Dave Barker, Joy Ortiz, Martin Riedmiller, Jost Tobias Springenberg, Raia Hadsell, Francesco Nori, and Nicolas Heess. Robocat: A self-improving foundation agent for robotic manipulation, 2023.

S. Marc Breedlove and Neil V Watson. *Biological psychology: An introduction to behavioral, cognitive, and clinical neuroscience.* Sinauer Associates, 2013.

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

György Buzsáki. Neural syntax: cell assemblies, synapsembles, and readers. *Neuron*, 68(3):362–385, 2010.

Daniele Caligiore, Michael A Arbib, R Chris Miall, and Gianluca Baldassarre. The super-learning hypothesis: Integrating learning processes across cortex, cerebellum and basal ganglia. *Neuroscience & Biobehavioral Reviews*, 100:19–34, 2019.

Sinan Çalışır and Meltem Kurt Pehlivanoğlu. Model-free reinforcement learning algorithms: A survey. In *2019 27th signal processing and communications applications conference (SIU)*, pages 1–4. IEEE, 2019.

Christian Carvelli, Djordje Grbic, and Sebastian Risi. Evolving hypernetworks for game-playing agents. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 71–72, 2020.

David J Chalmers. The evolution of learning: An experiment in genetic connectionism. In *Connectionist Models*, pages 81–90. Elsevier, 1991.

Mathieu Chalvidal, Thomas Serre, and Rufin Van-Rullen. Meta-reinforcement learning with self-modifying networks. In *36th Conference*

on Neural Information Processing Systems (NeurIPS 2022), pages 1–19, 2022.

David T Chau, Robert M Roth, and Alan I Green. The neural circuitry of reward and its relevance to psychiatric disorders. *Current psychiatry reports*, 6(5):391–399, 2004.

Nitin Kumar Chauhan and Krishna Singh. A review on conventional machine learning vs deep learning. In *2018 International conference on computing, power and communication technologies (GUCON)*, pages 347–352. IEEE, 2018.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Misha Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34, 2021.

Yuan-chu Cheng, Wei-Min Qi, and Wei-You Cai. Dynamic properties of elman and modified elman neural network. In *Proceedings. International Conference on Machine Learning and Cybernetics*, volume 2, pages 637–640. IEEE, 2002.

Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve, et al. On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward. *Proceedings of the National Academy of Sciences*, 118(1):e1907856118, 2021.

Jeff Clune. Ai-gas: Ai-generating algorithms, an alternate paradigm for producing general artificial intelligence. *arXiv preprint arXiv:1905.10985*, 2019.

Cédric Colas, Vashisht Madhavan, Joost Huizinga, and Jeff Clune. Scaling map-elites to deep neuroevolution. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 67–75, 2020.

Oliver J Coleman and Alan D Blair. Evolving plastic neural networks for online learning: review and future directions. In *Australasian Joint Conference on Artificial Intelligence*, pages 326–337. Springer, 2012.

Jack Collins, Shelvin Chand, Anthony Vanderkop, and David Howard. A review of physics simulators for robotic applications. *IEEE Access*, 9: 51416–51431, 2021.

Xiaodong Cui, Vaibhava Goel, and Brian Kingsbury. Data augmentation for deep neural network acoustic modeling. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 23(9):1469–1477, 2015.

Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503–507, 2015.

Yang Dan and Mu-ming Poo. Spike timing-dependent plasticity of neural circuits. *Neuron*, 44(1):23–30, 2004.

Peter Dayan. Twenty-five lessons from computational neuromodulation. *Neuron*, 76(1):240–256, 2012.

Daniel C Dennett. Darwin's dangerous idea. *The Sciences*, 35(3):34–40, 1995.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

Rachit Dubey, Pulkit Agrawal, Deepak Pathak, Thomas L Griffiths, and Alexei A Efros. Investigating human priors for playing video games. *arXiv preprint arXiv:1802.10217*, 2018.

Kai Olav Ellefsen, Jean-Baptiste Mouret, and Jeff Clune. Neural modularity helps organisms evolve to learn new skills without forgetting old skills. *PLoS Comput Biol*, 11(4):e1004128, 2015.

Benjamin Ellenberger. Pybullet gymperium. `https://github.com/benelot/pybullet-gym`, 2018.

Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

B. Fasel. An introduction to bio-inspired artificial neural network architectures. *Acta neurologica belgica*, 103(1):6–12, 2003.

Daniel E Feldman. The spike-timing dependence of plasticity. *Neuron*, 75 (4):556–571, 2012.

Gilbert Feng, Hongbo Zhang, et al. Genloco: Generalized locomotion controllers for quadrupedal robots. *arXiv preprint arXiv:2209.05309*, 2022.

Chrisantha Fernando, Jakub Sygnowski, Simon Osindero, Jane Wang, Tom Schaul, Denis Teplyashin, Pablo Sprechmann, Alexander Pritzel, and Andrei Rusu. Meta-learning by the baldwin effect. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1313–1320, 2018.

Chelsea Finn, Sergey Levine, and Pieter Abbeel. Guided cost learning: Deep inverse optimal control via policy optimization. In *International conference on machine learning*, pages 49–58. PMLR, 2016.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135. PMLR, 2017.

Dario Floreano and Claudio Mattiussi. *Bio-inspired artificial intelligence: theories, methods, and technologies.* MIT press, 2008.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. Stabilizing the lottery ticket hypothesis. *arXiv preprint arXiv:1903.01611*, 2019.

Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

Adam Gaier and David Ha. Weight agnostic neural networks. *Advances in neural information processing systems*, 32, 2019.

Luíza C Garaffa, Abdullah Aljuffri, Cezar Reinbrecht, Said Hamdioui, Mottaqiallah Taouil, and Johanna Sepulveda. Revealing the secrets of spiking neural networks: The case of izhikevich neuron. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 514–518. IEEE, 2021.

Wulfram Gerstner. Associative memory in a network ofbiological'neurons. *Advances in neural information processing systems*, 3, 1990.

Faustino Gomez, Jan Koutník, and Jürgen Schmidhuber. Compressed network complexity search. In *International Conference on Parallel Problem Solving from Nature*, pages 316–326. Springer, 2012.

Santiago Gonzalez and Risto Miikkulainen. Optimizing loss functions through multi-variate taylor polynomial parameterization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 305–313, 2021.

Santiago Gonzalez and Risto Miikkulainen. Effective regularization through loss-function metalearning. *arXiv preprint arXiv:2010.00788*, 2022.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

William T Greenough and James E Black. Induction of brain structure by experience: Substrates. In *Developmental behavioral neuroscience: The Minnesota symposia on child psychology*, page 155, 2013.

Klaus Greff, Sjoerd van Steenkiste, and Jürgen Schmidhuber. On the binding problem in artificial neural networks. *arXiv preprint arXiv:2012.05208*, 2020.

Frederic Gruau, Darrell Whitley, and Larry Pyeatt. A comparison between cellular encoding and direct encoding for genetic neural networks. In *Proceedings of the 1st annual conference on genetic programming*, pages 81–89, 1996.

Agrim Gupta, Linxi Fan, Surya Ganguli, and Li Fei-Fei. Metamorph: Learning universal controllers with transformers. *arXiv preprint arXiv:2203.11931*, 2022.

David Ha. Evolving stable strategies. *blog.otoro.net*, 2017a. URL http://blog.otoro.net/2017/11/12/evolving-stable-strategies/.

David Ha. A visual guide to evolution strategies. *blog. otoro. net*, 2017b.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. *Advances in neural information processing systems*, 31, 2018.

David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

Raia Hadsell, Dushyant Rao, Andrei A Rusu, and Razvan Pascanu. Embracing change: Continual learning in deep neural networks. *Trends in cognitive sciences*, 24(12):1028–1040, 2020.

Muhammad Burhan Hafez, Cornelius Weber, Matthias Kerzel, and Stefan Wermter. Deep intrinsically motivated continuous actor-critic for efficient robotic visuomotor skill learning. *Paladyn, Journal of Behavioral Robotics*, 10(1):14–29, 2019.

Alexander Hagg, Maximilian Mensing, and Alexander Asteroth. Evolving parsimonious networks by mixing activation functions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 425–432, 2017.

Robert E Hampson and Sam A Deadwyler. Neural population recording in behaving animals: Constituents of a neural code for behavioral decisions. 2009.

Nikolaus Hansen. The cma evolution strategy: a comparing review. *Towards a new evolutionary computation*, pages 75–102, 2006.

Nikolaus Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18, 2003.

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

Uri Hasson, Samuel A Nastase, and Ariel Goldstein. Direct fit to nature: An evolutionary perspective on biological and artificial neural networks. *Neuron*, 105(3):416–434, 2020.

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.

Kun He, Yan Wang, and John Hopcroft. A powerful generative model using random weights for the deep image representation. *Advances in Neural Information Processing Systems*, 29, 2016.

Geoffrey E Hinton and Steven J Nowlan. How learning can guide evolution. *Adaptive individuals in evolving populations: models and algorithms*, 26: 447–454, 1987.

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

Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

Kari L Hoffmann. A face in the crowd: Which groups of neurons process face stimuli, and how do they interact? 2009.

Christian Holscher. How could populations of neurons encode information? In *Information Processing by Neuronal Populations*, pages 3–20. Cambridge University Press, 2008.

Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-learning in neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(9):5149–5169, 2021.

Rein Houthooft, Yuhua Chen, Phillip Isola, Bradly Stadie, Filip Wolski, OpenAI Jonathan Ho, and Pieter Abbeel. Evolved policy gradients. *Advances in Neural Information Processing Systems*, 31, 2018.

Yizhou Huang, Kevin Xie, Homanga Bharadhwaj, and Florian Shkurti. Continual model-based reinforcement learning with hypernetworks. In *2021*

IEEE International Conference on Robotics and Automation (ICRA), pages 799–805. IEEE, 2021.

Michael R Hunsaker and Raymond P Kesner. The operation of pattern separation and pattern completion processes associated with different attributes or domains of memory. *Neuroscience & Biobehavioral Reviews*, 37 (1):36–58, 2013.

Maximilian Igl, Kamil Ciosek, Yingzhen Li, Sebastian Tschiatschek, Cheng Zhang, Sam Devlin, and Katja Hofmann. Generalization in reinforcement learning with selective noise injection and information bottleneck. *arXiv preprint arXiv:1910.12911*, 2019.

Riashat Islam, Hongyu Zang, Anirudh Goyal, Alex Lamb, Kenji Kawaguchi, Xin Li, Romain Laroche, Yoshua Bengio, and Remi Tachet Des Combes. Discrete factorial representations as an abstraction for goal conditioned reinforcement learning. *arXiv preprint arXiv:2211.00247*, 2022.

Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

Eugene M Izhikevich. Polychronization: computation with spikes. *Neural computation*, 18(2):245–282, 2006.

Eugene M Izhikevich. *Dynamical systems in neuroscience*. MIT press, 2007.

Khurram Javed and Martha White. Meta-learning representations for continual learning. *Advances in neural information processing systems*, 32, 2019.

Taewon Jeong and Heeyoung Kim. Ood-maml: Meta-learning for few-shot out-of-distribution detection and classification. *Advances in Neural Information Processing Systems*, 33:3907–3916, 2020.

Michael I Jordan. Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier, 1997.

Ryan Julian, Benjamin Swanson, Gaurav S Sukhatme, Sergey Levine, Chelsea Finn, and Karol Hausman. Never stop learning: The effectiveness of fine-tuning in robotic reinforcement learning. *arXiv preprint arXiv:2004.10190*, 2020.

Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural level generation. In *NeurIPS Workshop on Deep Reinforcement Learning*, 2018.

Eric R Kandel, James H Schwartz, Thomas M Jessell, Steven Siegelbaum, A James Hudspeth, and Sarah Mack. *Principles of neural science*, volume 4. McGraw-hill New York, 2000.

Mert Kayaalp, Stefan Vlaski, and Ali H Sayed. Dif-maml: Decentralized multi-agent meta-learning. *IEEE Open Journal of Signal Processing*, 3: 71–93, 2022.

James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.

Seyed-Mahdi Khaligh-Razavi and Nikolaus Kriegeskorte. Deep supervised, but not unsupervised, models may explain it cortical representation. *PLoS computational biology*, 10(11):e1003915, 2014.

Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. Transformers in vision: A survey. *ACM computing surveys (CSUR)*, 54(10s):1–41, 2022.

Khimya Khetarpal, Matthew Riemer, Irina Rish, and Doina Precup. Towards continual reinforcement learning: A review and perspectives. *arXiv preprint arXiv:2012.13490*, 2020.

Louis Kirsch and Jürgen Schmidhuber. Meta learning backpropagation and improving it. *arXiv preprint arXiv:2012.14905*, 2020.

Louis Kirsch, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. *arXiv preprint arXiv:1910.04098*, 2019.

Louis Kirsch, Sebastian Flennerhag, Hado van Hasselt, Abram Friesen, Junhyuk Oh, and Yutian Chen. Introducing symmetries to black box meta reinforcement learning. *arXiv preprint arXiv:2109.10781*, 2021.

Louis Kirsch, James Harrison, et al. General-purpose in-context learning by meta-learning transformers. *arXiv preprint arXiv:2212.04458*, 2022.

Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32 (11):1238–1274, 2013.

Jan Koutnik, Faustino Gomez, and Jürgen Schmidhuber. Evolving neural networks in compressed weight space. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 619–626, 2010.

Dhireesha Kudithipudi, Mario Aguilar-Simon, Jonathan Babb, Maxim Bazhenov, Douglas Blackiston, Josh Bongard, Andrew P Brna, Suraj Chakravarthi Raja, Nick Cheney, Jeff Clune, et al. Biological underpinnings for lifelong learning machines. *Nature Machine Intelligence*, 4(3): 196–210, 2022.

Thomas S Kuhn. *The structure of scientific revolutions*. University of Chicago press, 2012.

Anders Lansner. Associative memory models: from the cell-assembly theory to biophysically detailed cortex simulations. *Trends in neurosciences*, 32 (3):178–186, 2009.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International conference on machine learning*, pages 3744–3753. PMLR, 2019.

Richard C Lewontin. The units of selection. *Annual review of ecology and systematics*, 1(1):1–18, 1970.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Jian-Ping Li, Marton E Balazs, Geoffrey T Parks, and P John Clarkson. A species conserving genetic algorithm for multimodal function optimization. *Evolutionary computation*, 10(3):207–234, 2002.

Laura Lillien. Neural development: instructions for neural diversity. *Current Biology*, 7(3):R168–R171, 1997.

Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *AI Open*, 2022.

Zachary C Lipton. Stuck in a what? adventures in weight space. *arXiv preprint arXiv:1602.07320*, 2016.

Hanxiao Liu, Andy Brock, Karen Simonyan, and Quoc Le. Evolving normalization-activation layers. *Advances in Neural Information Processing Systems*, 33:13539–13550, 2020.

Minghuan Liu, Menghui Zhu, and Weinan Zhang. Goal-conditioned reinforcement learning: Problems and solutions. *arXiv preprint arXiv:2201.08299*, 2022.

Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer science review*, 3(3):127–149, 2009.

Eran Malach, Gilad Yehudai, Shai Shalev-Schwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. In *International Conference on Machine Learning*, pages 6682–6691. PMLR, 2020.

Eve Marder, LF Abbott, Gina G Turrigiano, Zheng Liu, and Jorge Golowasch. Memory from the dynamics of intrinsic membrane currents. *Proceedings of the national academy of sciences*, 93(24):13481–13486, 1996.

James L McClelland, David E Rumelhart, PDP Research Group, et al. *Parallel distributed processing, volume 2: Explorations in the microstructure of cognition: Psychological and biological models*, volume 2. MIT press, 1987.

Francisco S Melo and Manuel Lopes. Learning from demonstration using mdp induced metrics. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 385–401. Springer, 2010.

Jorge Mendez, Boyu Wang, and Eric Eaton. Lifelong policy gradient learning of factored policies for faster training without forgetting. *Advances in Neural Information Processing Systems*, 33:14398–14409, 2020.

Alberto Maria Metelli, Giorgia Ramponi, Alessandro Concetti, and Marcello Restelli. Provably efficient learning of transferable rewards. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 7665–7676. PMLR, 18–24 Jul 2021. URL `https://proceedings.mlr.press/v139/metelli21a.html`.

Thomas Miconi. Learning to learn with backpropagation of hebbian plasticity. *arXiv preprint arXiv:1609.02228*, 2016.

Jae-eun Kang Miller, Inbal Ayzenshtat, Luis Carrillo-Reid, and Rafael Yuste. Visual stimuli recruit intrinsically generated cortical ensembles. *Proceedings of the National Academy of Sciences*, 111(38):E4053–E4061, 2014.

Thomas M Moerland, Joost Broekens, Aske Plaat, Catholijn M Jonker, et al. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, 2023.

Andrew William Moore. Efficient memory-based learning for robot control. 1990.

Jean-Baptiste Mouret and Paul Tonelli. Artificial evolution of plastic neural networks: a few key concepts. In *Growing adaptive machines*, pages 251–261. Springer, 2014.

Nils Müller and Tobias Glasmachers. Challenges in high-dimensional reinforcement learning with evolution strategies. In *International Conference on Parallel Problem Solving from Nature*, pages 411–423. Springer, 2018.

Tsendsuren Munkhdalai and Hong Yu. Meta networks. In *International Conference on Machine Learning*, pages 2554–2563. PMLR, 2017.

Thibaut Munzer, Bilal Piot, Matthieu Geist, Olivier Pietquin, and Manuel Lopes. Inverse reinforcement learning in relational domains. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

Elias Najarro and Sebastian Risi. Meta-learning through hebbian plasticity in random networks. *Advances in Neural Information Processing Systems*, 33, 2020.

Elias Najarro, Shyam Sudhakaran, and Sebastian Risi. Towards self-assembling artificial neural networks through neural developmental programs. In *ALIFE 2023: Ghost in the Machine: Proceedings of the 2023 Artificial Life Conference*. MIT Press, 2023.

Ali Bou Nassif, Ismail Shahin, Imtinan Attili, Mohammad Azzeh, and Khaled Shaalan. Speech recognition using deep neural networks: A systematic review. *IEEE access*, 7:19143–19165, 2019.

Emre O Neftci and Bruno B Averbeck. Reinforcement learning in artificial and biological systems. *Nature Machine Intelligence*, 1(3):133–143, 2019.

Thanh Nguyen, Tung Luu, Trung Pham, Sanzhar Rakhimkul, and Chang D Yoo. Robust maml: prioritization task buffer with adaptive learning process for model-agnostic meta-learning. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3460–3464. IEEE, 2021.

Yael Niv, Daphna Joel, Isaac Meilijson, and Eytan Ruppin. Evolution of reinforcement learning in uncertain environments: A simple explanation for complex foraging behaviors. 2002.

Ben Norman and Jeff Clune. First-explore, then exploit: Meta-learning intelligent exploration. *arXiv preprint arXiv:2307.02276*, 2023.

Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. arxiv 2018. *arXiv preprint arXiv:1811.03378*, 2018.

Jeff Orchard and Lin Wang. The evolution of a generalized neural learning rule. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 4688–4694. IEEE, 2016.

Norman R Pace. Mapping the tree of life: progress and prospects. *Microbiology and molecular biology reviews*, 73(4):565–576, 2009.

Rasmus Berg Palm. Evostrat. https://github.com/rasmusbergpalm/evostrat, 2020.

Rasmus Berg Palm, Elias Najarro, and Sebastian Risi. Testing the genomic bottleneck hypothesis in hebbian meta-learning. In *NeurIPS 2020 Workshop on Pre-registration in Machine Learning*, pages 100–110. PMLR, 2021.

Evgenia Papavasileiou, Jan Cornelis, and Bart Jansen. A systematic literature review of the successors of “neuroevolution of augmenting topologies”. *Evolutionary Computation*, 29(1):1–73, 2021.

Fabio Pardo. Tonic: A deep reinforcement learning library for fast prototyping and benchmarking. *arXiv preprint arXiv:2011.07537*, 2020.

Joachim Winther Pedersen and Sebastian Risi. Evolving and merging hebbian learning rules: increasing generalization by decreasing the number of rules. *arXiv preprint arXiv:2104.07959*, 2021.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Michael Pfeiffer and Thomas Pfeil. Deep learning with spiking neurons: opportunities and challenges. *Frontiers in neuroscience*, 12:774, 2018.

Harry A Pierson and Michael S Gashler. Deep learning in robotics: a review of recent research. *Advanced Robotics*, 31(16):821–835, 2017.

Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization: An overview. *Swarm intelligence*, 1:33–57, 2007.

Mu-ming Poo. Towards brain-inspired artificial intelligence, 2018.

Jonathan D Power and Bradley L Schlaggar. Neural plasticity across the lifespan. *Wiley Interdisciplinary Reviews: Developmental Biology*, 6(1): e216, 2017.

Joseph L Price and Wayne C Drevets. Neural circuits underlying the pathophysiology of mood disorders. *Trends in cognitive sciences*, 16(1):61–71, 2012.

Trevor D Price, Anna Qvarnström, and Darren E Irwin. The role of phenotypic plasticity in driving genetic evolution. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 270(1523):1433–1440, 2003.

J. Andrew Pruszynski and Joel Zylberberg. The language of the brain: real-world neural population codes. *Current opinion in neurobiology*, 58:30–36, 2019.

Friedemann Pulvermüller, Max Garagnani, and Thomas Wennekers. Thinking in circuits: toward neurobiological explanation in cognitive neuroscience. *Biological cybernetics*, 108(5):573–593, 2014.

Aravind Rajeswaran, Kendall Lowrey, Emanuel V Todorov, and Sham M Kakade. Towards generalization and simplicity in continuous control. *Advances in Neural Information Processing Systems*, 30, 2017.

Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. What's hidden in a randomly weighted neural network? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11893–11902, 2020.

Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, et al. A generalist agent. *arXiv preprint arXiv:2205.06175*, 2022.

Machel Reid, Yutaro Yamada, and Shixiang Shane Gu. Can wikipedia help offline reinforcement learning? *arXiv preprint arXiv:2201.12122*, 2022.

Matthew Riemer, Ignacio Cases, Robert Ajemian, Miao Liu, Irina Rish, Yuhai Tu, and Gerald Tesauro. Learning to learn without forgetting by maximizing transfer and minimizing interference. *arXiv preprint arXiv:1810.11910*, 2018.

Sebastian Risi and Kenneth O Stanley. Indirectly encoding neural plasticity as a pattern of local rules. In *International Conference on Simulation of Adaptive Behavior*, pages 533–543. Springer, 2010.

Sebastian Risi and Kenneth O Stanley. Enhancing es-hyperneat to evolve more complex regular neural networks. In *Proceedings of the Genetic and*

Evolutionary Computation Conference (GECCO 2011). New York, NY: ACM. http://eplex. cs. ucf. edu/papers/risi_gecco11. pdf, 2011.

Sebastian Risi and Kenneth O Stanley. An enhanced hypercube-based encoding for evolving the placement, density, and connectivity of neurons. *Artificial life*, 18(4):331–363, 2012a.

Sebastian Risi and Kenneth O Stanley. A unified approach to evolving plasticity and neural geometry. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2012b.

Sebastian Risi and Kenneth O Stanley. Confronting the challenge of learning a flexible neural controller for a diversity of morphologies. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 255–262, 2013.

Sebastian Risi, Charles E Hughes, and Kenneth O Stanley. Evolving plastic neural networks with novelty search. *Adaptive Behavior*, 18(6):470–491, 2010.

Jason W Rocks and Pankaj Mehta. Memorizing without overfitting: Bias, variance, and interpolation in overparameterized models. *Physical review research*, 4(1):013201, 2022.

Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks. *arXiv preprint arXiv:1801.01078*, 2017.

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

Shreya Saxena and John P Cunningham. Towards the neural population doctrine. *Current opinion in neurobiology*, 55:103–111, 2019.

Samuel M Scheiner. Genetics and evolution of phenotypic plasticity. *Annual review of ecology and systematics*, 24(1):35–68, 1993.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pages 9355–9366. PMLR, 2021.

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

John Schulman, Filip Wolski, et al. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Anthony Sclafani, Khalid Touzani, and Richard J Bodnar. Dopamine and learned food preferences. *Physiology & behavior*, 104(1):64–68, 2011.

Jermyn Z See, Craig A Atencio, Vikaas S Sohal, and Christoph E Schreiner. Coordinated neuronal ensembles in primary auditory cortical columns. *Elife*, 7:e35587, 2018.

Chris Sekirnjak and Sascha Du Lac. Intrinsic firing dynamics of vestibular nucleus neurons. *Journal of Neuroscience*, 22(6):2083–2095, 2002.

Jocelyn Sietsma and Robert JF Dow. Creating artificial neural networks that generalize. *Neural networks*, 4(1):67–79, 1991.

Laura Smith, Ilya Kostrikov, and Sergey Levine. A walk in the park: Learning to walk in 20 minutes with model-free reinforcement learning. *arXiv preprint arXiv:2208.07860*, 2022.

Emilie C Snell-Rood. An overview of the evolutionary causes and consequences of behavioural plasticity. *Animal Behaviour*, 85(5):1004–1011, 2013.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

Ivan Soltesz et al. *Diversity in the neuronal machine: order and variability in interneuronal microcircuits*. Oxford University Press, 2006.

Andrea Soltoggio, John A Bullinaria, Claudio Mattiussi, Peter Dürr, and Dario Floreano. Evolutionary advantages of neuromodulated plasticity in dynamic, reward-based scenarios. In *Proceedings of the 11th international conference on artificial life (Alife XI)*, number CONF, pages 569–576. MIT Press, 2008.

Andrea Soltoggio, Kenneth O Stanley, and Sebastian Risi. Born to learn: the inspiration, progress, and future of evolved plastic artificial neural networks. *Neural Networks*, 108:48–67, 2018.

Sen Song, Kenneth D Miller, and Larry F Abbott. Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature neuroscience*, 3(9):919–926, 2000.

Xingyou Song, Wenbo Gao, Yuxiang Yang, Krzysztof Choromanski, Aldo Pacchiano, and Yunhao Tang. Es-maml: Simple hessian-free meta learning. *arXiv preprint arXiv:1910.01215*, 2019a.

Xingyou Song, Yiding Jiang, et al. Observational overfitting in reinforcement learning. *arXiv preprint arXiv:1912.02975*, 2019b.

Bradly Stadie, Lunjun Zhang, and Jimmy Ba. Learning intrinsic rewards as a bi-level optimization problem. In *Conference on Uncertainty in Artificial Intelligence*, pages 111–120. PMLR, 2020.

Kenneth O Stanley and Joel Lehman. *Why greatness cannot be planned: The myth of the objective*. Springer, 2015.

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

Kenneth O Stanley and Risto Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130, 2003.

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

Kenneth O Stanley, Joel Lehman, and Lisa Soros. Open-endedness: The last grand challenge you've never heard of. *While open-endedness could be a force for discovering intelligence, it could also be a component of AI itself*, 2017.

Joan Stiles. Neural plasticity and cognitive development. *Developmental neuropsychology*, 18(2):237–272, 2000.

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.

Sainbayar Sukhbaatar, Emily Denton, Arthur Szlam, and Rob Fergus. Learning goal embeddings via self-play for hierarchical reinforcement learning. *arXiv preprint arXiv:1811.09083*, 2018.

Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems*, 8, 1995.

Richard S Sutton. The bitter lesson. *Incomplete Ideas (blog)*, 13(1), 2019.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Yujin Tang and David Ha. The sensory neuron as a transformer: Permutation-invariant neural networks for reinforcement learning. *Advances in Neural Information Processing Systems*, 34, 2021.

Yujin Tang, Duong Nguyen, and David Ha. Neuroevolution of self-interpretable agents. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pages 414–424, 2020.

Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, 2019.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6):1–28, 2022.

Luke Taylor and Geoff Nitschke. Improving deep learning with generic data augmentation. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1542–1547. IEEE, 2018.

Adaptive Agent Team, Jakob Bauer, Kate Baumli, Satinder Baveja, Feryal Behbahani, Avishkar Bhoopchand, Nathalie Bradley-Schmieg, Michael Chang, Natalie Clay, Adrian Collister, et al. Human-timescale adaptation in an open-ended task space. *arXiv preprint arXiv:2301.07608*, 2023.

Yingjie Tian, Xiaoxi Zhao, and Wei Huang. Meta-learning approaches for learning-to-learn in deep learning: A survey. *Neurocomputing*, 494:203–223, 2022.

Paul Tonelli and Jean-Baptiste Mouret. On the relationships between generative encodings, regularity, and learning abilities when evolving plastic artificial neural networks. *PLOS ONE*, 8 (11):1–12, 11 2013. doi: 10.1371/journal.pone.0079138. URL https://doi.org/10.1371/journal.pone.0079138.

Peter D Turney. Myths and legends of the baldwin effect. *arXiv preprint cs/0212036*, 2002.

Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Deep image prior. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9446–9454, 2018.

Joseba Urzelai and Dario Floreano. Evolution of adaptive synapses: Robots with fast adaptive behavior in new environments. *Evolutionary computation*, 9(4):495–524, 2001.

Arjen van Ooyen. Activity-dependent neural network development. *Network: Computation in Neural Systems*, 5(3):401–423, 1994.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, Eftychios Protopapadakis, et al. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

Scott Waddell. Reinforcement signalling in drosophila; dopamine does it all after all. *Current opinion in neurobiology*, 23(3):324–329, 2013.

Jane X Wang. Meta-learning in natural and artificial intelligence. *Current Opinion in Behavioral Sciences*, 38:90–95, 2021.

Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn, 2017.

Jindong Wang, Cuiling Lan, Chang Liu, Yidong Ouyang, Tao Qin, Wang Lu, Yiqiang Chen, Wenjun Zeng, and Philip Yu. Generalizing to unseen domains: A survey on domain generalization. *IEEE Transactions on Knowledge and Data Engineering*, 2022.

Lin Wang, Junteng Zheng, and Jeff Orchard. Evolving generalized modulatory learning: Unifying neuromodulation and synaptic plasticity. *IEEE Transactions on Cognitive and Developmental Systems*, 12(4):797–808, 2019.

Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. Nervenet: Learning structured policy with graph neural networks. In *International conference on learning representations*, 2018.

Dante Francisco Wasmuht, Eelke Spaak, Timothy J Buschman, Earl K Miller, and Mark G Stokes. Intrinsic neuronal dynamics predict distinct functional roles during working memory. *Nature communications*, 9(1): 3499, 2018.

Mary Jane West-Eberhard. Phenotypic plasticity and the origins of diversity. *Annual review of Ecology and Systematics*, 20(1):249–278, 1989.

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *The Journal of Machine Learning Research*, 15(1):949–980, 2014.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.

Mitchell Wortsman, Vivek Ramanujan, Rosanne Liu, Aniruddha Kembhavi, Mohammad Rastegari, Jason Yosinski, and Ali Farhadi. Supermasks in

superposition. *Advances in Neural Information Processing Systems*, 33: 15173–15184, 2020.

Philipp Wu, Alejandro Escontrela, Danijar Hafner, Ken Goldberg, and Pieter Abbeel. Daydreamer: World models for physical robot learning. *arXiv preprint arXiv:2206.14176*, 2022.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

Annie Xie, James Harrison, and Chelsea Finn. Deep reinforcement learning amidst lifelong non-stationarity. *arXiv preprint arXiv:2006.10701*, 2020.

Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-gradient reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

Anil Yaman, Giovanni Iacca, Decebal Constantin Mocanu, George Fletcher, and Mykola Pechenizkiy. Learning with delayed synaptic plasticity. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 152–160, 2019.

Anil Yaman, Giovanni Iacca, Decebal Constantin Mocanu, Matt Coler, George Fletcher, and Mykola Pechenizkiy. Evolving plasticity for autonomous learning under changing environmental conditions. *Evolutionary computation*, 29(3):391–414, 2021.

Sherry Yang, Ofir Nachum, Yilun Du, Jason Wei, Pieter Abbeel, and Dale Schuurmans. Foundation models for decision making: Problems, methods, and opportunities. *arXiv preprint arXiv:2303.04129*, 2023.

Hana Yousuf, Michael Lahzi, Said A Salloum, and Khaled Shaalan. A systematic review on sequence-to-sequence learning with neural network and its models. *International Journal of Electrical & Computer Engineering (2088-8708)*, 11(3), 2021.

Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural computation*, 31(7):1235–1270, 2019.

Anthony M Zador. A critique of pure learning and what artificial neural networks can learn from animal brains. *Nature communications*, 10(1): 1–7, 2019.

Anthony M Zador, Sean Escola, Blake Richards, Bence Ölveczky, Yoshua Bengio, Kwabena Boahen, Matthew Botvinick, Dmitri Chklovskii, Anne Churchland, Claudia Clopath, et al. Catalyzing next-generation artificial intelligence through neuroai. *Nature communications*, 14(1):1597, 2023.

Amy Zhang, Nicolas Ballas, and Joelle Pineau. A dissection of overfitting and generalization in continuous reinforcement learning. *arXiv preprint arXiv:1806.07937*, 2018a.

Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018b.

Chenyang Zhao, Olivier Sigaud, Freek Stulp, and Timothy M Hospedales. Investigating generalisation in continuous deep reinforcement learning. *arXiv preprint arXiv:1902.07015*, 2019.

Zeyu Zheng, Junhyuk Oh, and Satinder Singh. On learning intrinsic rewards for policy gradient methods. *Advances in Neural Information Processing Systems*, 31, 2018.

Zeyu Zheng, Junhyuk Oh, Matteo Hessel, Zhongwen Xu, Manuel Kroiss, Hado Van Hasselt, David Silver, and Satinder Singh. What can learned intrinsic rewards capture? In *International Conference on Machine Learning*, pages 11436–11446. PMLR, 2020.

Ce Zhou, Qian Li, Chen Li, Jun Yu, Yixin Liu, Guangjing Wang, Kai Zhang, Cheng Ji, Qiben Yan, Lifang He, et al. A comprehensive survey on pretrained foundation models: A history from bert to chatgpt. *arXiv preprint arXiv:2302.09419*, 2023.

Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. *arXiv preprint arXiv:1905.01067*, 2019.

Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020a.

Kaiyang Zhou, Ziwei Liu, Yu Qiao, Tao Xiang, and Chen Change Loy. Domain generalization: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

Wei Zhou, Yiying Li, Yongxin Yang, Huaimin Wang, and Timothy Hospedales. Online meta-critic learning for off-policy actor-critic methods. *Advances in Neural Information Processing Systems*, 33:17662–17673, 2020b.

A Appendix Section 1

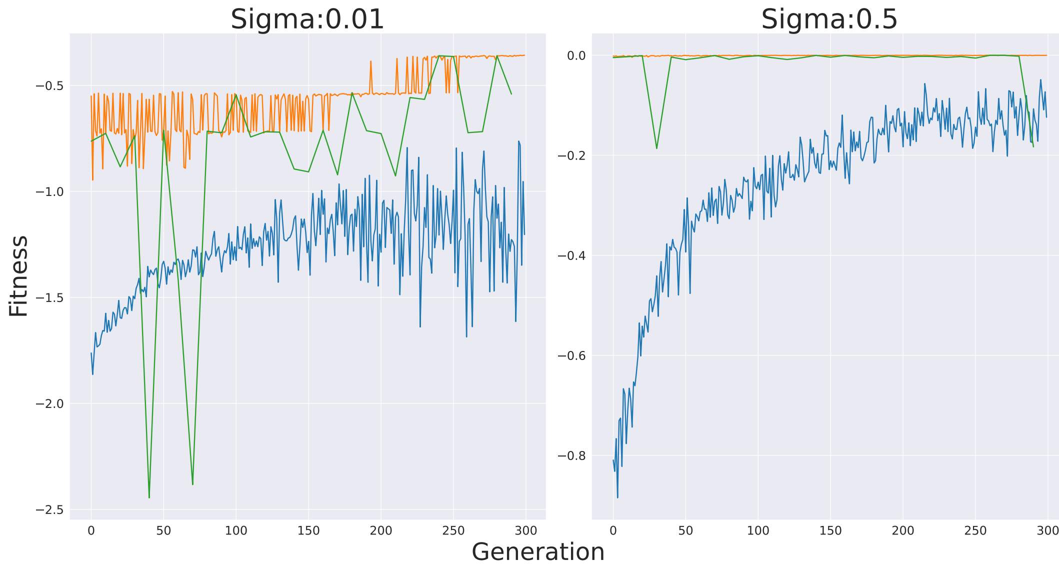


Figure A1: **Training curves for outer-loop with 100 parameters.** Training curves for inner-loop sigmas look similar to when 10 parameters were optimized in the outer-loop. As better top solutions are found in the run with an inner-loop sigma of 0.01, the population mean becomes more volatile.

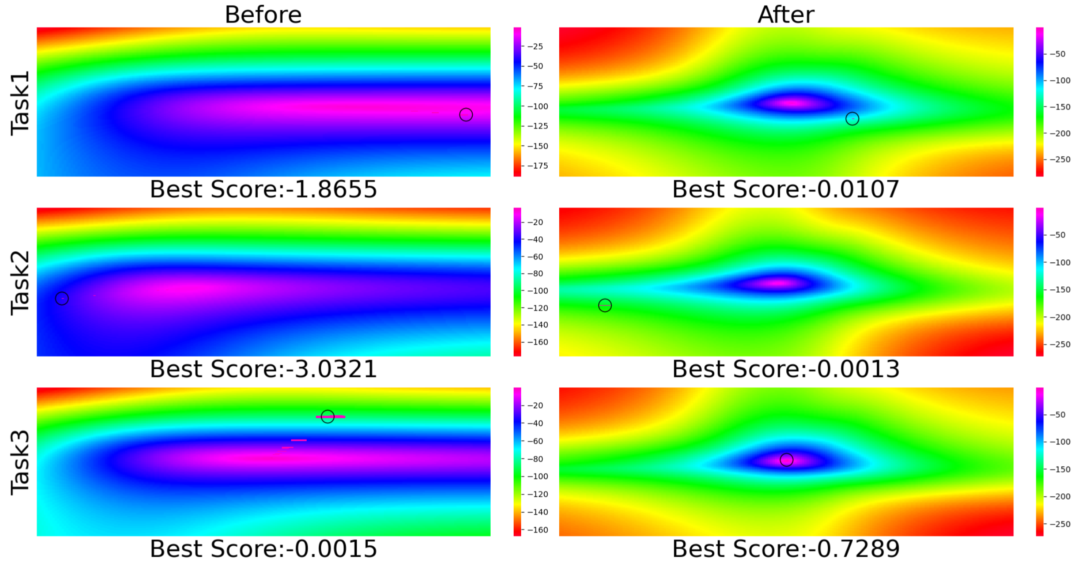


Figure A2: **Changing inner-loss landscapes with inner-loop sigma 0.01 and 100 outer-loop parameters.** Large changes in the inner-loop loss landscapes occur after outer-loop optimization. The optimal point in task 3 is worse than before optimization but is now placed almost directly at the starting point of the inner-loop optimization, and therefore very easy to find. The overall shape of the optimized loss landscapes and the isolated placements of the optimal points in task 1 and 2, makes it plausible that in these loss landscapes, the inner-loop optimizer always settles on points very close the starting points.

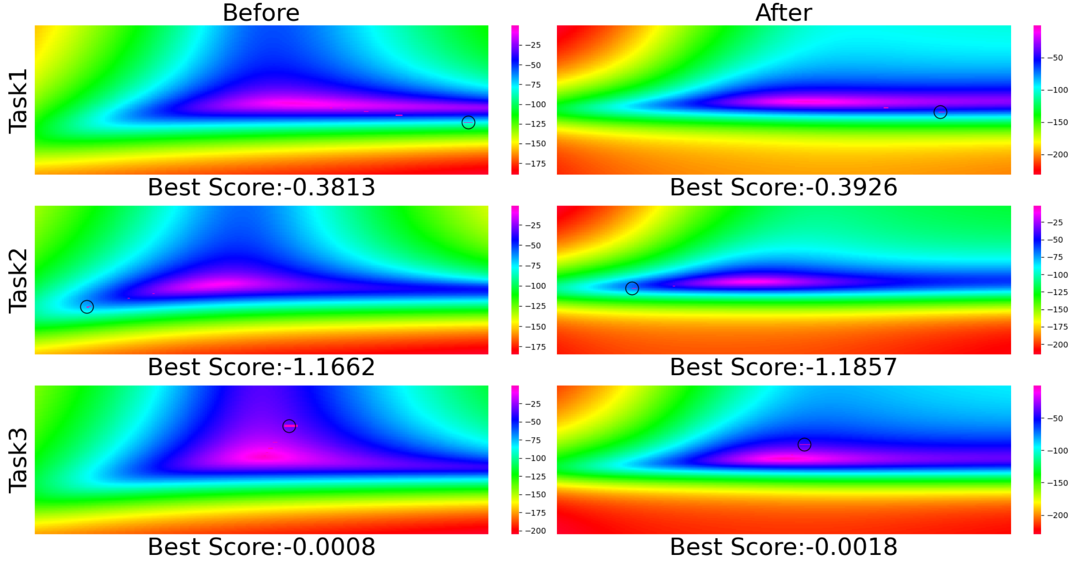


Figure A3: **Changing inner-loss landscapes with inner-loop sigma 0.5 and 100 outer-loop parameters.** Once again, the inner-loop loss landscapes change to a smaller degree throughout outer-loop optimization when the inner-loop sigma is larger. Loss landscapes and their optimal points are more similar before and after outer-loop optimization than in Figure A2.

Figure A2 shows how the inner-loop loss landscapes changed when the inner-loop sigma was 0.01, the landscapes has changed more than in any other loss landscapes. The loss landscape for task number three has after optimization an optimal point with worse performance than before. However, its new optimal point is very close to the starting parameters of the inner-loop optimization. Further, when examining the optimal points of the two other tasks after optimization, we can see that they are not located near the origin, and are isolated in areas of low performance. It seems plausible that the outer-loop parameters have shaped the inner-loop loss landscapes such that the solutions in area around the starting parameters are well-performing if not exactly optimal. This would be a way for the outer-loop to increase the probability of getting a good mean score across all three tasks.