Automating AI Research

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Louis Kirsch

under the supervision of
Prof. Jürgen Schmidhuber

June 2025

Dissertation Committee

Prof. Jeff Clune	University of British Columbia, Canada
Prof. David Silver	University College London, United Kingdom
Prof. Frank Hutter	Albert-Ludwigs-Universität Freiburg, Germany
Prof. Natasha Sharygina	Università della Svizzera italiana, Switzerland
Prof. Cesare Alippi	Università della Svizzera italiana, Switzerland

Dissertation submitted on 19 March 2025
Accepted on 3 June 2025
Final revision on 17 December 2025

Research Advisor	PhD Program Director
Prof. Jürgen Schmidhuber	**Prof. Walter Binder/ Prof. Stefan Wolf**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Louis Kirsch
Lugano, 3 June 2025

Abstract

A core element of Artificial General Intelligence (AGI) and ultimately Artificial Superintelligence (ASI) is the ability of AI to improve itself, including its own learning algorithm. This single capability, once developed by human researchers, could initiate an autonomously self-improving system capable of independently conducting all subsequent research. Toward this vision, this thesis advances the automation of AI research by developing methods for AI systems to discover general-purpose learning algorithms.

Today, machine learning (ML) research still relies heavily on human-designed learning algorithms, architectures, losses, optimizers, data, and other components. This process requires significant manual effort and the selection of appropriate inductive biases, limited by human creativity and knowledge. Meta-learning, or learning-to-learn, instead aims to automate the process of artificial intelligence (AI) research and promises to unlock greater capabilities with less manual effort.

In recent years, meta-learning has made significant progress in producing models that can learn very quickly from a few examples using in-context learning, fast-weights, and optimization-based methods. This tremendously helps in adapting to new, similar problems, but it does not automate AI research itself.

To address this limitation, we introduce meta-learners that discover general-purpose learning algorithms. The goal is the discovery of novel learning algorithms that can be reused across a wide range of tasks, similar to human-engineered learning algorithms. We present several novel methods addressing meta-generalization, including learned loss functions, weight-shared LSTMs that implement gradient descent in their recurrent dynamics, and black-box Transformers that learn how to in-context learn generally.

The main contributions include MetaGenRL, a novel off-policy gradient-based meta-RL algorithm that, for the first time, discovers general-purpose learning al-

gorithms with high performance on diverse robotic control tasks. Our Variable Shared Meta-Learners (VSML) make use of parameter sharing and sparsity in meta-learning to automatically discover powerful in-context learning algorithms that neither require explicit gradients at meta-test time nor weight updates. Next, we introduce SymLA, a method that builds on the neural network symmetries of VSML to improve the generalization of learning algorithms in meta-RL. In our general-purpose in-context learners (GPICL), we discover phase transitions where models transition from memorization to task identification, to general learning-to-learn. We identify crucial ingredients such as memory capacity and practical interventions in meta-training. We explore how these insights can be applied to meta-RL in generally learning agents (GLAs). Our results show that meta-learning is a powerful approach for generating general-purpose learning algorithms that can be effectively transferred to new and unseen environments.

Finally, we seek AI to self-improve while minimizing its dependence on human engineering. To this end, we explore self-referential systems that can recursively improve themselves without hard-wired meta-optimization. We extend this idea to AI scientists, large language models that can automate AI research by generating hypotheses, conducting experiments, and interpreting results.

Preface & Acknowledgements

Automating AI Research is humanity's final task.

The automation of AI research will be the last invention humanity ever needs to make. All other inventions will be made by AI itself. Throughout my research career, I have been fascinated by this idea. The rapid progress in this field is truly astounding. In just a few years, we've witnessed breakthroughs ranging from automating the discovery of new RL algorithms and general-purpose in-context learning to the early steps of automating science with large language models. When I started my PhD, the idea of fully automating AI research felt like a distant dream. Today, it seems within reach.

I feel honored to have made ambitious contributions to this final task and to have worked with one of the pioneers in the field, Jürgen Schmidhuber. There is one particular quote from Jürgen that has always resonated with me:

> *Since age 15 or so, the main goal of professor Jürgen Schmidhuber has been to build a self-improving Artificial Intelligence (AI) smarter than himself, then retire.*

The day will come when we can all retire, and AI can take over the task of improving itself.

I would like to thank all my co-authors, collaborators, and friends who have made this journey so joyful. Special thanks to Jürgen Schmidhuber, Sjoerd van Steenkiste, Francesco Faccio, Alexander Stanic, Vincent Herrmann, Aditya Ramesh, Anand Gopalakrishnan, Imanol Schlag, Paul Rauber, Chris Lu, Rylan Schäffer, James Harrison, Luke Metz, Jascha Sohl Dickstein, David Ha, Jeff Clune, Luisa Zintgraf, Akarsh Kumar, Yutian Chen, and Klaus Greff. Also, I would like to extend my gratitude to my family for their unwavering support and encouragement.

This thesis is based on several publications and reports [Kirsch et al., 2020b; Kirsch and Schmidhuber, 2021; Kirsch et al., 2022a; Kirsch and Schmidhuber, 2022a; Kirsch et al., 2022b, 2023]. Louis' work was supported by the ERC Advanced Grant (no: 742870) and computational resources by the Swiss National Supercomputing Centre (CSCS, projects s978, s1041, and s1127) [Kirsch et al., 2019; Kirsch and Schmiduber, 2020, 2021]. We also thank NVIDIA Corporation for donating several DGX machines as part of the Pioneers of AI Research Award, IBM for lending a Minsky machine, and weights & biases [Biewald, 2020] for their great experiment tracking software and support.

Contents

Chapter 1

Introduction

1.1 Automating AI research

It has long been a dream of computer scientists to build machines capable of performing any cognitive task that a human can do [Leibniz, 1666; Turing, 1950], often referred to as Artificial General Intelligence (AGI) [Legg, 2008; Goertzel, 2014]. An even more ambitious goal is to create systems that surpass human capabilities, termed Artificial Superintelligence (ASI) [Schmidhuber, 1987; Morris et al., 2023]. While the current literature often focuses on achieving or surpassing the wide range of human capabilities, this thesis takes a different perspective. We argue that the core ingredient of AGI and eventually ASI is the ability of AI to improve itself, including its own learning algorithm. If AI cannot improve itself, it will always be limited by the abilities of human researchers and can thus hardly be considered general. Conversely, if AI can improve itself, it might be the only capability that human researchers need to develop. All other capabilities can then be discovered by the AI itself through self-improvement, setting off an open-ended process of innovation. Such a self-improving system could then be considered an ASI.

To date, advances in machine learning (ML) are generally driven by human research and engineering [Ivakhnenko and Lapa, 1965; Schmidhuber, 2015; Silver et al., 2016]. This can be viewed as a human-driven open-ended process [Schmidhuber, 1997; Lehman and Stanley, 2011; Schmidhuber, 2013; Stanley, 2019] that continually creates new improvements, such as novel learning algorithms. At the same time, this process requires significant manual effort and the ability of researchers to choose appropriate inductive biases. Therefore, the

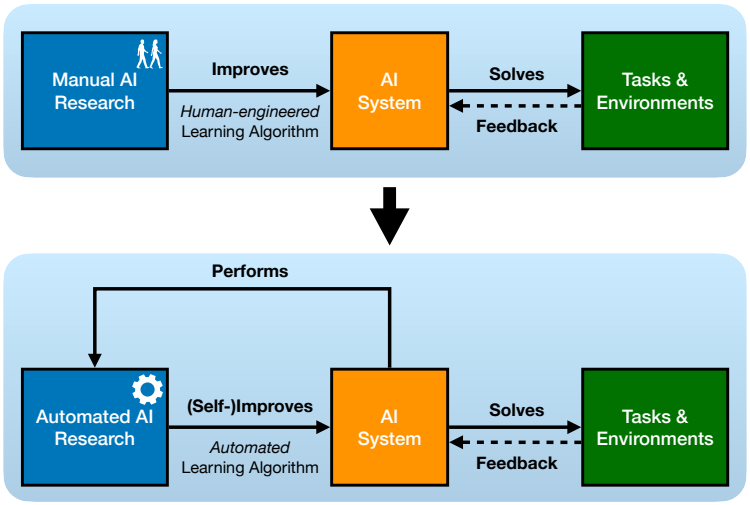


Figure 1.1: **How can we automate the process of AI research?** Conventionally the field of machine learning manually devises new methods to learn better predictors and agent behaviors from data. In this thesis, we investigate methods to apply this process to research itself, in effect automating the research process. Two important topics of interest are (1) how reusable the automatically discovered learning algorithms are, i.e. how well they generalize, and (2) how we can minimize the fixed inductive biases that are hardcoded into the system.

resulting ML systems are inherently limited by the ability of humans to understand and analyze the properties and behavior of the learning system.

Meta-learning aims to automate this research process (Figure 1.1). Its goal is to learn the learning algorithm (LA) itself, reducing the burden on the human designer to craft useful learning algorithms [Schmidhuber, 1987; Hochreiter et al., 2001; Clune, 2019; Hutter et al., 2019]. As learning, we describe a process that improves a model or agent behavior based on demonstrations, new observations, or feedback signals such as labels in supervised learning or rewards in reinforcement learning (RL). Such data must not be human designed, but can be generated by another automated system, or the agent itself in an unsupervised fashion. A key requirement for learning is an unknown or change in the environment or data distribution that warrants adaptation. In RL, learning is evident as an increase of the policy's expected return over time (return velocity). Meta-learning then corresponds to optimizing this process to iteratively increase the return velocity (resulting in return acceleration) [Schmidhuber et al., 1997]. Meta-learning is not limited to the automated discovery of the update rule of a neural network but may include architectural changes, training data changes, or

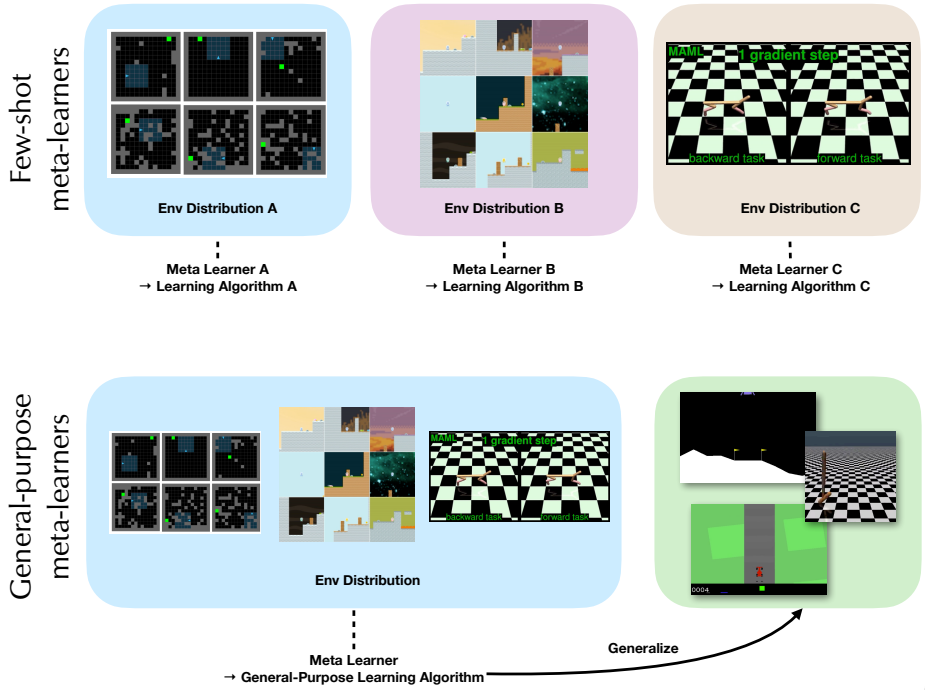


Figure 1.2: **From few-shot to general-purpose learning-to-learn.** Human-engineered learning algorithms are general-purpose in the sense of being applicable across a wide range of environments or tasks. This is not the case for most meta-learning approaches, often referred to as few-shot learners, that are good at learning very quickly on a specific distribution but do not generalize well. In contrast, general-purpose meta-learning is about automatically discovering novel learning algorithms that generalize broadly.

entirely other substrates of AI. In this thesis, we mainly focus on the former.

A common setup is to meta-train across several tasks or environments, the meta-training distribution, to successfully learn on unseen environments. These unseen tasks or environments correspond to the meta-test distribution.

Towards more general-purpose meta-learning Recent meta-learning has focused primarily on generalization from training tasks to similar test tasks, e.g., few-shot learning [Finn et al., 2017], or from training environments to similar test environments [Houthooft et al., 2018]. While this approach can lead to very efficient learning algorithms, generalization to novel, significantly different problems is quite limited. This contrasts with human-engineered LAs that generalize

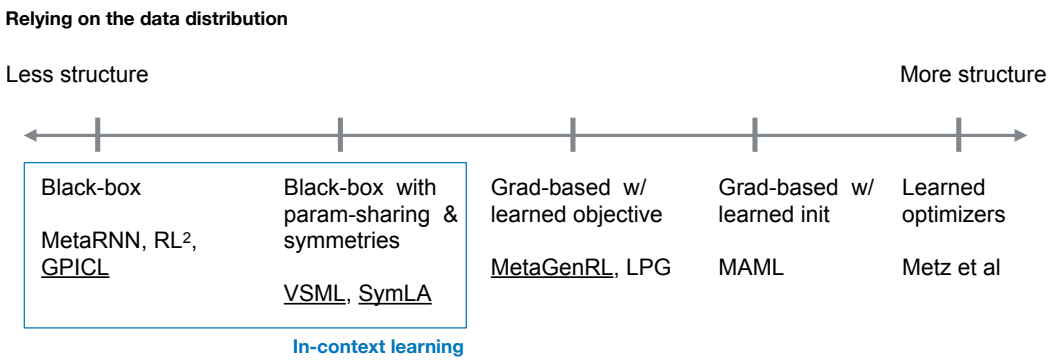


Figure 1.3: **Methods for (general-purpose) meta-learning can be classified along a continuum, from those with strong meta-test inductive biases to those that are more data-driven.** We propose various methods on this continuum, as underlined in the figure. Methods with more structure enable leveraging known insights from hand-crafted learning algorithms, whereas approaches with fewer inductive biases may automatically discover more expressive learning algorithms while still generalizing to significantly different problems.

across a wide range of datasets or environments. Without such generalization, meta-learned LAs cannot entirely replace human-engineered variants.

We propose an area of research that we refer to as *general-purpose meta-learning*, in which the objective is the meta-learning of expressive and novel general-purpose learning algorithms (Figure 1.2). This includes our recent work *MetaGenRL* Chapter 2 and the work of others [e.g. Alet et al., 2020; Oh et al., 2020; Xu et al., 2020], which demonstrate that meta-learning can successfully generate more general reinforcement learning (RL) algorithms that generalize across a wide spectrum of environments, e.g., from toy environments to Mujoco and Atari. These approaches, however, still rely on a large number of human-designed and unmodifiable inner-loop components, such as backpropagation. Our later work, *VSML* Chapter 3, does away with this dependency and shows that the backpropagation algorithm can be encoded in the activation spreading of parameter-shared (recurrent) neural networks, now referred to as in-context learners [Brown et al., 2020]. This enables the discovery of novel general-purpose learning algorithms for supervised learning without the use of hard-coded backpropagation. We also discuss the close relationship between updating NN activations and weights, relating it to fast weight programmers [Schmidhuber, 1992b, 1993a]. Based on VSML, in *SymLA* Chapter 4, we discuss how symmetries in the parameterization of reinforcement learning poli-

cies can help aid in the discovery of more general-purpose learning algorithms. *GPICL* Chapter 5 and *GLAs* Chapter 6 demonstrate that the inductive bias of parameter sharing can also be removed under certain conditions. As one increases the number of tasks that are meta-trained on, the model undergoes multiple algorithmic transitions in terms of its behavior, from multi-task learning, to task identification, to general learning to learn. Our contributions can be categorized along a continuum from inductive biases to more data-driven approaches, as visualized in Figure 1.3.

Towards reducing our reliance on human-engineered meta-optimization Despite these advances in automation, meta-learning creates a dependency on human engineering at the meta-level, where meta-learning algorithms must be designed. In principle, one could also meta-meta-learn this meta-algorithm [Schmidhuber, 1987, 1993b] but would still be left with some human engineering. In the optimal case, the human-engineered biases are minimized to the largest extent possible and rely on self-improvements from data or environment interaction to automatically develop better learners, meta-meta learners, and so on in a recursive and self-referential fashion [Schmidhuber et al., 1997; Schmidhuber, 2007]. To this end, we propose neural self-referential meta-learning systems that modify themselves without the need for explicit meta-optimization [Kirsch and Schmidhuber, 2022b] in Chapter 7.

In-context learning gives rise to automating scientific research in language space Language-based in-context learning creates another level of meta-learning – performing scientific research by writing code for experimentation and reasoning about hypotheses very much like a human scientist. This opens up the possibility of automating the scientific research process as a whole, including the generation of hypotheses, the design of experiments, and the interpretation of results. We refer to this as an AI Scientist, discussed in Chapter 8.

1.2 Background & Related work

To automate machine learning research, researchers in the fields of meta-learning [Schmidhuber, 1987; Bengio et al., 1991] and AutoML [Hutter et al., 2019] have developed a wide range of methods. These methods differ primarily in two aspects: the search space, which describes the kinds of ML systems that can be discovered, and the meta-optimizer, which determines how this search

space is navigated. For the search space, popular choices include the automated discovery of hyperparameters, architectures, gradient-based loss functions, initializations, symbolic computation graphs, code, natural language, or entirely neural approaches through weight or memory updates. For the meta-optimizer, options broadly include evolutionary methods (zeroth-order gradients), gradient-based approaches, or large language models. In this chapter, we provide an overview and describe these different types of meta-learning in both supervised and reinforcement learning. In each subsequent chapter, we will delve deeper into several of these approaches in the context of our contributions.

1.2.1 A description of meta-learning on multiple timescales

One popular perspective on meta-learning is to consider how it creates multiple timescales and levels of abstraction. Algorithm 1 illustrates the generic structure of meta-learning. In the case of two levels, we refer to the first level as the inner loop and the second level as the outer loop. The inner loop, corresponding to the learning algorithm A_ϕ^{inner}, iteratively updates the parameters θ of the model or policy π_θ. To learn, the algorithm receives feedback in the form of rewards $\{R_i\}$ or demonstrations $\{D_i\}$. These parameters can represent the weights of a neural network [Schmidhuber, 1992c, 1993a] or the memory (such as hidden or context state) of a neural network [Hochreiter et al., 2001]. The outer loop, operating on a slower timescale, updates the meta-parameters ϕ of the learning algorithm (using the meta-learner A^{outer}). Meta-training describes the optimization process of running the outer loop, which in turn executes the inner loop. Meta-testing involves holding the meta-parameters fixed while only running the inner loop. A common setup is to meta-train across several tasks or environments $\{T_i\}$, the meta-training distribution, to successfully learn on unseen environments. These unseen tasks or environments correspond to the meta-test distribution. What is the benefit of having multiple levels? We only need to design the meta-learner manually, while the inner learning algorithm can be discovered automatically and may go beyond the capabilities of human algorithm designs. The meta-learner is typically a simple and universal learning algorithm, whereas the inner learning algorithm may exhibit desirable properties such as better sample efficiency, exploration, or performance at convergence. In principle, additional levels of meta-learning can be added, but as we will discuss in Chapter 7, this does not solve the core goal of automating AI research: minimal human intervention. Instead, self-reference enables the learning algorithm to improve itself without the need for human-designed meta-learners.

In the following sections, we focus on describing different structures and parameterizations of the inner learning algorithm A_ϕ^{inner} as this choice directly affects properties such as expressivity, generalization, and efficiency of the discovered learning algorithms. Different types of outer meta-learners A^{outer} can usually be combined with any of these inner choices, such as evolutionary methods (zeroth-order gradients), gradient-based approaches, or large language models.

Algorithm 1 A generic meta-learning algorithm

Require: Initial meta-parameters ϕ parameterizing the learning algorithm A_ϕ^{inner} that updates a model or policy π_θ, a set of tasks or environments $\{T_i\}$ or demonstrations $\{D_i\}$, and a meta-learning algorithm A^{outer}

for each meta-iteration **do** ▷ Outer loop

  Initialize or carry-over model or policy parameters θ_i^0

  for each inner iteration **t do** ▷ Inner loop

    Evaluate the model π_{θ_i} on (a subset of) tasks $\{T_i\}$, yielding $\{R_i^t\}_{t=\mathbf{t}}$ and/or obtain demonstrations $\{D_i^t\}_{t=\mathbf{t}}$

    Update the model parameters $\theta_i^{\mathbf{t}} \leftarrow A_\phi^{\text{inner}}(\theta_i^{\mathbf{t}-1}, \{T_i\}, \{R_i^t\}_{t=\mathbf{t}}, \{D_i^t\}_{t=\mathbf{t}})$ ▷ Learning

  Update the meta-parameters $\phi \leftarrow A^{\text{outer}}(\phi, \{\theta_i^t\}, \{T_i\}, \{R_i^t\}, \{D_i^t\})$ ▷ Meta-Learning

1.2.2 Parameterizing gradient-based learning algorithms

Much of recent deep learning is based on optimization using gradient descent [Schmidhuber, 2015], where gradients are usually obtained using backpropagation [Linnainmaa, 1970]. In the context of meta-learning learning algorithms, it is thus natural to parameterize learning algorithms that make use of gradient descent (and backpropagation). Additional meta-parameters define how these gradients are derived or modified before being applied to the model parameters. We refer to this class of algorithms as gradient-based meta-learning.

These approaches include learning optimizers [Ravi and Larochelle, 2017; Andrychowicz et al., 2016], weight initialization and adaptation with a human-engineered RL algorithm [Finn et al., 2017], warping computed gradients [Flennerhag et al., 2020], meta-learning hyperparameters [Sutton, 1992; Schraudolph, 1999; Xu et al., 2018], or meta-learning loss functions corresponding to the learning algorithm [Houthooft et al., 2018; Kirsch et al., 2020b].

Algorithm 2 illustrates the general structure of gradient-based meta-learning. Model parameters θ are updated by a gradient-based update rule that is parameterized by ϕ. L_ϕ^{inner} could, for example, represent a meta-learned loss function (see Chapter 2) or a standard loss where ϕ corresponds to the weight initialization $\phi = \theta_0$ [Finn et al., 2017].

Algorithm 2 Gradient-based meta-learning algorithms

Require: Initial meta-parameters ϕ parameterizing the gradient-based learning algorithm $\nabla_\theta L_\phi^{\text{inner}}$ that updates a model or policy π_θ, a set of tasks or environments $\{T_i\}$ or demonstrations $\{D_i\}$, and a meta-learning algorithm A^{outer}

for each meta-iteration **do** ▷ Outer loop

  Initialize or carry over model or policy parameters θ_i^0

  for each inner iteration **t do** ▷ Inner loop

    Evaluate the model π_{θ_i} on (a subset of) tasks $\{T_i\}$, yielding $\{R_i^t\}_{t=\mathbf{t}}$ and/or obtain demonstrations $\{D_i^t\}_{t=\mathbf{t}}$

    Update the model parameters $\theta_i^{\mathbf{t}} \leftarrow \theta_i^{\mathbf{t}-1} - \nabla_{\theta_i} L_\phi^{\text{inner}}(\theta_i^{\mathbf{t}-1}, \{T_i\}, \{R_i^t\}_{t=\mathbf{t}}, \{D_i^t\}_{t=\mathbf{t}})$ ▷ Learning

  Update the meta-parameters $\phi \leftarrow A^{\text{outer}}(\phi, \{\theta_i^t\}, \{T_i\}, \{R_i^t\}, \{D_i^t\})$ ▷ Meta-Learning

Learning optimizers A straightforward meta-parameterization of gradient-based learners is to meta-learn the transformation from the gradient to the applied parameter update [Ravi and Larochelle, 2017; Li and Malik, 2017; Andrychowicz et al., 2016; Metz et al., 2020a]. This corresponds to automatically discovering novel optimizers, similar to those manually engineered by humans, e.g., Adam [Kingma and Ba, 2014]. This parameterization is applicable to both supervised and reinforcement learning, provided that a known algorithm is used to obtain the gradients. At the same time, this parameterization may limit the types of learning algorithms that are discovered. The main update signal, in the form of a gradient, is already provided as an input and does not need to be meta-learned.

Learning model initializations Instead of meta-parameterizing the optimizer directly, many recent meta-learning algorithms influence the learning trajectory indirectly by finding a model or policy initialization. This initialization is then fine-tuned using a fixed supervised or reinforcement learning algorithm [Finn et al., 2017; Grant et al., 2018; Yoon et al., 2018]. Despite still using a fixed known learning algorithm, this parameterization is, in principle, quite expressive [Finn and Levine, 2018]. In practice, these approaches rely mostly on feature reuse (transfer learning) instead of the discovery of novel learning algorithms [Raghu et al., 2020].

Learning objective functions A more expressive parameterization of gradient-based learning algorithms involves meta-learning the objective (or loss) function that is being differentiated itself. In supervised learning, these objective functions are a function of the output of the neural network and the labels. Human-designed objective functions include, for instance, the mean squared

error or cross-entropy loss. This parameterization can be particularly useful in reinforcement learning, where surrogate losses such as REINFORCE [Williams, 1992] or PPO [Schulman et al., 2017] are designed by researchers to circumvent the non-differentiability of the RL problem. Such meta-learning has previously been shown to be successful for learning quickly on new RL tasks similar to the meta-training distribution [Houthooft et al., 2018; Bechtle et al., 2021] and, in the context of our recent work, to generalize to significantly different environments [Kirsch et al., 2020b; Oh et al., 2020]. In supervised learning, learned objective functions have also been used for learning unsupervised representations [Metz et al., 2019a], aiding learning in downstream tasks. This can be viewed as replacing hand-crafted unsupervised objective functions, like the predictability of a sequence [Schmidhuber, 1991d, 1992a], with meta-learned ones. The previously mentioned works used neural parameterizations of objective functions. Techniques from architecture search have also been used to search for viable artificial curiosity objectives composed of primitive symbolic functions [Alet et al., 2020; Co-Reyes et al., 2021].

Learning intrinsic rewards A related notion to meta-learning objective functions is the meta-learning of intrinsic reward functions [Niekum et al., 2011; Zheng et al., 2018] connected to earlier work on subgoal generation [Schmidhuber, 1991b; Wiering and Schmidhuber, 1996a; Singh et al., 2004]. The main difference is that the learning algorithm remains fixed and is guided by an intrinsically generated reward signal instead of being fully parameterized.

1.2.3 Updating weights through fast weight programmers

Instead of relying on gradients to update neural network (NN) parameters, these updates can also be directly meta-parameterized. This is achieved by defining neural update rules that modify these weights, as described in Algorithm 3. NNs that generate or modify the weights of another NN—or even the same NN—were first introduced as fast weight programmers [Schmidhuber, 1992b, 1993a; Ba et al., 2016a; Schlag et al., 2021b]. Several variants of these have been developed, such as Hypernetworks [Ha et al., 2017], synaptic plasticity [Miconi et al., 2018; Najarro and Risi, 2020], or learned learning rules [Bengio et al., 1992; Gregor, 2020; Randazzo et al., 2020]. Fast weight programmers are also of interest outside meta-learning, for instance, to generate policies for a specific task or given a problem description [Harb et al., 2020; Faccio et al., 2021b, 2023].

Algorithm 3 A fast weight programmer (FWP) meta-learning algorithm

Require: Initial meta-parameters ϕ parameterizing the FWP neural network A_ϕ^{inner} that updates a model or policy π_θ, a set of tasks or environments $\{T_i\}$, and a meta-learning algorithm A^{outer}

for each meta-iteration **do** ▷ Outer loop

  Initialize or carry over model or policy parameters θ_i^0

  for each inner iteration $\mathbf{t}$ **do** ▷ Inner loop

    Evaluate the model π_{θ_i} on (a subset of) tasks $\{T_i\}$, yielding $\{R_i^t\}_{t=\mathbf{t}}$ and/or obtain demonstrations $\{D_i^t\}_{t=\mathbf{t}}$

    Update the model parameters $\theta_i^{\mathbf{t}} \leftarrow A_\phi^{\text{inner}}(\theta_i^{\mathbf{t}-1}, \{T_i\}, \{R_i^t\}_{t=\mathbf{t}}, \{D_i^t\}_{t=\mathbf{t}})$ where A_ϕ^{inner} is a neural network that takes input weights $\theta_i^{\mathbf{t}-1}$ to produce updated weights $\theta_i^{\mathbf{t}}$ ▷ Learning

  Update the meta-parameters $\phi \leftarrow A^{\text{outer}}(\phi, \{\theta_i^t\}, \{T_i\}, \{R_i^t\}, \{D_i^t\})$ ▷ Meta-Learning

1.2.4 In-context learning with black-box neural networks

Instead of relying on gradients or parameterized weight updates, learning can occur in any black-box neural network that receives a feedback signal or demonstration as input [Schmidhuber, 1993b]. This is known as memory-based meta-learning when a memory h_t is used to iteratively store and retrieve information that determines model or policy improvements. It has been shown that an RNN, such as an LSTM, can learn to implement a learning algorithm [Hochreiter et al., 2001] when the reward or demonstrations are provided as input [Schmidhuber, 1993b]. After meta-training, the learning algorithm is encoded in the weights of this RNN and determines learning during meta-testing. The activations serve as the memory to encode the learned program. We refer to these RNNs as in-context RNNs [Hochreiter et al., 2001; Duan et al., 2016; Wang et al., 2016] (Figure 1.4). This mechanism has also received substantial recent attention in Transformer models [Brown et al., 2020; Chan et al., 2022] under the name of in-context learning. In large language models (LLMs), demonstrations of a task in the input help solve language-based tasks at inference (meta-test) time [Brown et al., 2020]. In the case of Transformers, the memory h_t increases in dimensionality with the sequence length. Memory-based (in-context) learning and fast weight programmers are closely related when interpreting activations as weights that encode a program. This connection is established in our work on VSML [Kirsch and Schmidhuber, 2021], Chapter 3. Furthermore, Linear Transformers [Katharopoulos et al., 2020] are equivalent to certain fast weight programmers [Schmidhuber, 1993a; Schlag et al., 2021a]. In-context learning has also been interpreted from a Bayesian inference perspective [Ortega et al., 2019;

Mikulik et al., 2020; Nguyen and Grover, 2022; Müller et al., 2022].

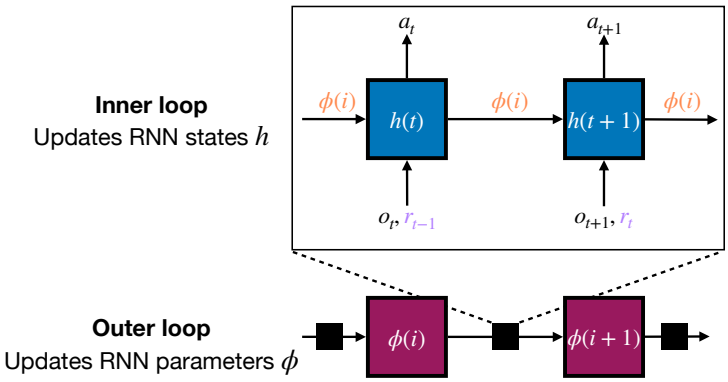


Figure 1.4: **In-context learning in neural networks.** A neural network, here recurrent, can implement a learning algorithm in its parameters ϕ when a feedback signal such as the reward r_t in RL is fed as an input in addition to the observation o_t to produce an action a_t. This allows learning to occur based on this signal in the memory/activations h. Meta-learning then corresponds to the optimization of ϕ.

Algorithm 4 In-context learning algorithms

Require: Initial meta-parameters ϕ parameterizing the in-context learning algorithm (neural network) A_ϕ^{inner} that updates the context / memory h_i, a set of tasks or environments $\{T_i\}$ or demonstrations $\{D_i\}$, and a meta-learning algorithm A^{outer}

for each meta-iteration **do** ▷ Outer loop

  Initialize or carry over the context / memory h^0

  for each inner iteration **t do** ▷ Inner loop

    Evaluate the model π_{h_i} on (a subset of) tasks $\{T_i\}$, yielding $\{R_i^t\}_{t=\mathbf{t}}$ and/or obtain demonstrations $\{D_i^t\}_{t=\mathbf{t}}$

    Update the context $h_i^{\mathbf{t}} \leftarrow A_\phi^{\text{inner}}(h_i^{\mathbf{t}-1}, \{T_i\}, \{R_i^t\}_{t=\mathbf{t}}, \{D_i^t\}_{t=\mathbf{t}})$ ▷ Learning

  Update the meta-parameters $\phi \leftarrow A^{\text{outer}}(\phi, \{h_i^t\}, \{T_i\}, \{R_i^t\}, \{D_i^t\})$ ▷ Meta-Learning

Algorithm 4 illustrates the general structure of in-context learning, where the model is now parameterized by the context $\theta := h$.

What is a supervised in-context learning algorithm? More concretely, a supervised in-context learning algorithm can be formalized as follows. Consider a mapping

$$\left(\{x_i, y_i\}_{i=1}^{N_D}, x'\right) \mapsto y' \tag{1.1}$$

from the training (support) set $D = \{x_i, y_i\}_{i=1}^{N_D}$ and a query input x' to the query's prediction y', where $x_i, x' \in \mathbb{R}^{N_x}$, $y_i, y' \in \mathbb{R}^{N_y}$, and $N_D, N_x, N_y \in \mathbb{N}^+$. The subset of these functions that qualify as learning algorithms A_ϕ^{inner} are those that improve their predictions y' given an increasingly larger training set D. Meta-learning then corresponds to finding these functions via meta-optimization. Typically, neural networks are used to approximate such functions. In-context learning differs from gradient-based meta-learning (such as MAML [Finn et al., 2017]) in that no explicit gradients are computed at meta-test time. All the learning mechanisms required are implicitly encoded in the black-box neural network.

1.2.5 Architectures and hyperparameters

In addition to the (automated) design of the learning algorithm, the neural architecture must also be selected. While many architectures, such as RNNs, LSTMs [Hochreiter and Schmidhuber, 1997b; Gers et al., 2000a], or Transformers [Schmidhuber, 1992b; Vaswani et al., 2017] (with recurrence in depth [Schmidhuber, 1993a] or width [Schuurmans et al., 2024]), are universal, improvements in neural architectures remain an active research field and thus also benefit from automation. This field of research is known as neural architecture search (NAS) [Elsken et al., 2019; Kirsch, 2017]. In this thesis, we do not focus on this design choice and assume fixed universal architectures. Furthermore, instead of learning the entire learning algorithm from scratch, many methods parameterize hand-crafted components and optimize these hyperparameters [Feurer et al., 2015; Golovin et al., 2017; Xu et al., 2018; Mukunthu et al., 2019]. The works in this thesis are concerned with higher-dimensional search spaces that can express novel learning algorithms beyond hyperparameter tuning.

1.2.6 Symbolic search spaces and programmers

Similar to how we can search over neural architectures and hyperparameters, we can also search over symbolic abstractions of learning algorithms. Such abstractions can include the equations of objective functions [Alet et al., 2020] or the entire computational graph that describes the machine learning algorithm expressed in code [Schmidhuber, 1987, 1994b; Real et al., 2020]. This symbolic representation can then be searched using zeroth-order optimization methods such as evolutionary algorithms. Because these representations are discrete in nature, direct gradient-based optimization is not possible, and solutions must

be mutated with mutation operators. This makes meta-optimization considerably harder and has so far limited the success of this research direction. Recent advances in large language models [Vaswani et al., 2017; Brown et al., 2020] offer a new way to approach this problem by defining a strong prior over code generation and mutation operators. We will discuss the prospects of this direction in Chapter 8 and how in-context learning (Chapter 3, Chapter 5) plays a major role in this approach. Meta-learning algorithms that search over code or other symbolic representations are illustrated in Algorithm 5. Here, we assume the programmer searches over a neural network-based learning algorithm. However, the same principle may apply to other substrates (sub-programs) of AI systems.

Algorithm 5 Programmer (code-based) meta-learning algorithms

Require: Initial program ϕ parameterizing the AI system / learning algorithm A_ϕ^{inner} that updates the neural network π_θ (or sub-program), a set of tasks or environments $\{T_i\}$ or demonstrations $\{D_i\}$, and a programmer A^{outer} (e.g. evolutionary or LLM-based)

for each meta-iteration **do** ▷ Outer loop (Programmer)

  Initialize or carry over the NN / code π_θ

  for each inner iteration $\mathbf{t}$ **do** ▷ Inner loop (Run the Program)

    Evaluate the NN / sub-program π_{θ_i} on (a subset of) tasks $\{T_i\}$, yielding $\{R_i^t\}_{t=\mathbf{t}}$ and/or obtain demonstrations $\{D_i\}_{t=\mathbf{t}}$

    Update NN / sub-program $\theta_i^{\mathbf{t}} \leftarrow A_\phi^{\text{inner}}(\theta_i^{\mathbf{t}-1}, \{T_i\}, \{R_i^t\}_{t=\mathbf{t}}, \{D_i\}_{t=\mathbf{t}})$ ▷ Learning

  Update the program describing the AI system

  $\phi \leftarrow A^{\text{outer}}(\phi, \{\theta_i^t\}, \{T_i\}, \{R_i^t\}, \{D_i^t\})$ ▷ Meta-Learning

1.2.7 Recursive self-improvement

The previously mentioned approaches to meta-learning all rely on a hierarchical structure where a *meta-optimizer* optimizes an inner *learning algorithm*. This creates a dependency on human engineering at the meta-level, as meta-learning algorithms must be designed. Have we then truly gained anything? Arguably, this meta-optimizer can be much simpler in its structure and does not need to be highly sample-efficient if the meta-learned learning algorithm later generalizes effectively. In principle, one could also meta-meta-learn this meta-algorithm [Schmidhuber, 1987, 1993b], but this would still involve some degree of human engineering. Ideally, human-engineered biases are minimized to the greatest extent possible, relying instead on self-improvements derived from data or environment interactions to automatically develop better learners, meta-meta-

learners, and so on in a recursive and self-referential manner [Schmidhuber et al., 1997; Schmidhuber, 2007]. In Chapter 7, we propose neural self-referential meta-learning systems that modify themselves without requiring explicit meta-optimization.

1.3 Contributions and key ideas

This is a brief summary of our contributions towards automating artificial intelligence research through general-purpose meta-learning and recursive self-improvement. We begin with our work *MetaGenRL* (Chapter 2), which meta-learns surrogate objective functions to automatically discover reinforcement learning algorithms. We demonstrate that meta-learning can successfully generate reinforcement learning algorithms that generalize across significantly different environments. Next, in *VSML* (Chapter 3), we remove the reliance on backpropagation at meta-test time. In this approach, the backpropagation algorithm is encoded in the activation spreading of parameter-shared recurrent neural networks. This enables the discovery of novel general-purpose learning algorithms for supervised learning without relying on hard-coded backpropagation. In *SymLA* (Chapter 4), we explore how symmetries in the parameterization of RL policies can aid in discovering more general-purpose RL algorithms. *GPICL* (Chapter 5) demonstrates that even without parameter sharing, and with a sufficiently rich task distribution, neural networks such as Transformers can learn-to-learn generally. We discuss in *GLAs* (Chapter 6) how this can be extended to Transformer-based RL agents trained using supervised learning techniques. Finally, we explore how to further reduce reliance on human-engineered meta-optimization to obtain *recursively self-improving systems* in Chapter 7. In *AI Scientist* (Chapter 8), we discuss how LLMs can be leveraged to automate AI research in the search space of code and natural language, based on the vast literature of human AI research.

The key ideas can be summarized as follows:

Chapter 2 to Chapter 6

Automating AI research requires general-purpose learning algorithms To automate AI research, meta-learned learning algorithms (LAs) must be as general-purpose and reusable as their hand-crafted counterparts. In contemporary meta-learning research, this is rarely the case [e.g. Duan et al., 2016; Wang et al., 2016; Finn et al., 2017; Houthooft et al., 2018]. We contribute a series of works, namely MetaGenRL, VSML, SymLA, GPICL, and GLAs, that demonstrate

the feasibility of the automated discovery of general-purpose LAs.

General-purpose meta-learners are on a spectrum of inductive bias and data distributional pressure Such general-purpose LAs can be obtained either by adding structure to the learning algorithm at meta-test time or by training over a broad enough data distribution to elicit generalization. The structure of the learning algorithm can take many forms. This can include the introduction of inductive bias based on known learning principles, such as hard-coding the use of gradients [Chapter 2 Kirsch et al., 2020b; Metz et al., 2019b; Oh et al., 2020]. Alternatively, the meta-learned LA can be regularized in various ways to encourage learning over memorization. It can be limited in capacity by bottlenecking the neural network (Chapter 3), the symbolic language that describes it [Real et al., 2020; Co-Reyes et al., 2021], or the inputs accessible to the LA. Furthermore, symmetries can be introduced as regularization (Chapter 4). Chapter 2 to Chapter 6

The difference between weights and activations is largely a semantic question Human-engineered learning algorithms usually update the weights of a neural network via gradient descent. Thus, it is natural to formulate meta-learning algorithms that also update the weights of a neural network with fast weight programmers [Schmidhuber, 1992b, 1993a; Ba et al., 2016a; Schlag et al., 2021b]. Does meta-learning require weight updates? Weight updates are neither a sufficient nor a necessary condition for meta-learning. Neural networks such as LSTMs and Transformers can perform learning-to-learn purely in-memory [Hochreiter et al., 2001] or in-context [Brown et al., 2020]. In fact, we demonstrate that activations in an RNN or LSTM can be interpreted as weight updates (Chapter 3). Later work has shown that linear Transformers are equivalent to certain fast weight programmers [Schlag et al., 2021a]. At the same time, fast weight programmers can also be used as a general method to increase the memory capacity of a neural network without any meta-learning involved [Schmidhuber, 1993a]. Chapter 3 Chapter 5

Memory capacity is a fundamental pillar of meta-learning If weight updates are not central to meta-learning, what makes a neural network a good meta-learner? In VSML (Chapter 3), we argue that a large memory capacity is a necessary component for a meta-learner that is general-purpose, due to the requirements of extracting a lot of information from the inference-time data provided in-context and storing it. We decouple meta-parameters from available memory through parameter sharing. In GPICL (Chapter 5), we show how memory capac- Chapter 3 Chapter 5

ity predicts in-context learning performance across various architectures, largely independent of the parameter count.

Chapter 3 Chapter 6 **Hand-crafted learning algorithms can be distilled into neural networks** In meta-learning, we usually meta-optimize for optimal learning behavior from scratch. In VSML, we demonstrate that we can also distill existing learning algorithms, such as gradient descent and backpropagation, into the activations of a neural network (Chapter 3). This may be used to bootstrap a learning algorithm from well-performing known learning algorithms and improve its performance from this initialization. This principle has later been used to distill RL algorithms into Transformers [Laskin et al., 2022]. Building on this, we use supervised learning to distill existing RL PPO learning trajectories into a general-purpose in-context learning agent in Chapter 6. We use this mechanism to accelerate existing RL algorithms via meta-learning while ensuring their generalization properties with sufficient data diversity.

Chapter 7 **Parameter sharing is a requirement for fully self-referential architectures** Parameter sharing can not only enable an increase in memory capacity but is also a crucial component for fully self-referential architectures. In order for a neural network to update all its parameters and activations, the parameters must be reused/shared (Chapter 7).

Chapter 7 **Recursive self-improvement is an architecture-agnostic phenomenon** Does recursive self-improvement [Schmidhuber, 1993b, 2007] in neural networks require updating all parameters in a neural network, i.e., a fully self-referential neural architecture? In Chapter 7, we show that, in principle, any memory-based or in-context learner can self-improve in an arbitrary manner, and this is not tied to a fully self-referential architecture.

Chapter 8 **LLMs and in-context learning form the basis for automated AI research with reasoning in natural language** In-context learning of today's Large Language Models (LLMs) can be used as the inner loop of a language-based automated scientist. This 'AI Scientist' is a new outer loop, replacing the original hand-crafted meta-learning algorithm. By training new AI models, we conjecture that methods like the AI Scientist will recursively self-improve its own capabilities in the future, leading to Artificial Super Intelligence.

Chapter 2

MetaGenRL: Meta-learning gradient-based RL algorithms that generalize

Keywords *gradient-based, objective functions, reinforcement learning*
Article *Kirsch et al. [2020b] (preprint 2019)*

2.1 Introduction

The process of evolution has equipped humans with incredibly general learning algorithms. These algorithms enable us to solve a wide range of problems, even in the absence of extensive prior experiences. The algorithms that give rise to these capabilities are the result of *distilling* the collective experiences of many learners throughout the course of natural evolution. Essentially, by *learning from learning experiences* in this way, the resulting knowledge can be compactly encoded in the genetic code of an individual, giving rise to the general learning capabilities we observe today.

In contrast, Reinforcement Learning (RL) in artificial agents rarely follows this paradigm. The learning rules used to train agents are typically the result of years of human engineering and design (e.g., Williams [1992]; Wierstra et al. [2008]; Mnih et al. [2013]; Lillicrap et al. [2016]; Schulman et al. [2015a]). Consequently, artificial agents are inherently limited by the designer's ability to incorporate the right inductive biases to learn effectively from prior experiences.

Several works have proposed an alternative framework based on *meta-reinforcement learning* [Schmidhuber, 1993b; Wang et al., 2016; Duan et al., 2016; Finn et al., 2017; Houthooft et al., 2018; Clune, 2019]. Meta-RL distinguishes between learning-to-act in the environment (the reinforcement learning problem) and learning-to-learn (the meta-learning problem). This distinction allows learning itself to become a learning problem, enabling one to leverage prior learning experiences to meta-learn *general* learning rules that surpass human-engineered alternatives. However, while prior work has shown that learning rules can be meta-learned to generalize to slightly different environments or goals [Finn et al., 2017; Plappert et al., 2018; Houthooft et al., 2018], generalization to *entirely different* environments remains an open challenge.

In this work, we present *MetaGenRL*[1], a novel meta-reinforcement learning algorithm that meta-learns learning rules capable of generalizing to entirely different environments. MetaGenRL is inspired by the process of natural evolution and the ubiquitous description in research of RL algorithms as pseudo-objective functions optimized by gradient descent. Our approach distills the experiences of many agents into the parameters of a low-complexity objective function that determines how future individuals will learn. Similar to Evolved Policy Gradients (EPG; Houthooft et al. [2018]), it meta-learns neural objective functions that can be used to train complex agents with many parameters. However, unlike EPG, it is able to meta-learn using second-order gradients, which offers several advantages, as we will demonstrate. Furthermore, unlike recent meta-RL algorithms, *MetaGenRL* can generalize to new environments that are entirely different from those used for meta-training.

We evaluate MetaGenRL on a variety of continuous control tasks and compare it to RL^2 [Wang et al., 2016; Duan et al., 2016] and EPG, in addition to several human-engineered learning algorithms. Compared to RL^2, we find that MetaGenRL does not overfit and is able to train randomly initialized agents using meta-learned learning rules on *entirely different* environments. Compared to EPG, we find that MetaGenRL is more sample efficient and significantly outperforms it under a fixed budget of environment interactions. The results of an ablation study and additional analysis provide further insights into the benefits of our approach.

[1] Code is available at `http://louiskirsch.com/code/metagenrl`

2.2 Preliminaries

Notation We consider the standard MDP Reinforcement Learning setting defined by a tuple $e = (S, A, P, \rho_0, r, \gamma, T)$ consisting of states S, actions A, the transition probability distribution $P : S \times A \times S \to \mathbb{R}_+$, an initial state distribution $\rho_0 : S \to \mathbb{R}_+$, the reward function $r : S \times A \to [-R_{max}, R_{max}]$, a discount factor γ, and the episode length T. The objective for the probabilistic policy $\pi_\phi : S \times A \to \mathbb{R}_+$ parameterized by ϕ is to maximize the expected discounted return:

$$\mathbb{E}_\tau[\sum_{t=0}^{T-1} \gamma^t r_t], \text{ where } s_0 \sim \rho_0(s_0),\ a_t \sim \pi_\phi(a_t|s_t),\ s_{t+1} \sim P(s_{t+1}|s_t, a_t), r_t = r(s_t, a_t), \tag{2.1}$$

with $\tau = (s_0, a_0, r_0, s_1, ..., s_{T-1}, a_{T-1}, r_{T-1})$.

Human Engineered Gradient Estimators A popular gradient-based approach to maximizing Equation 2.1 is REINFORCE [Williams, 1992]. It directly differentiates Equation 2.1 with respect to ϕ using the likelihood ratio trick to derive gradient estimates of the form:

$$\nabla_\phi \mathbb{E}_\tau[L_{REINF}(\tau, \pi_\phi)] := \mathbb{E}_\tau[\nabla_\phi \sum_{t=0}^{T-1} \log \pi_\phi(a_t|s_t) \cdot \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'})]. \tag{2.2}$$

This basic estimator has become a building block for an entire class of policy-gradient algorithms of this form. For example, a popular extension from Schulman et al. [2015b] combines REINFORCE with a Generalized Advantage Estimate (GAE) to yield the following policy gradient estimator:

$$\nabla_\phi \mathbb{E}_\tau[L_{GAE}(\tau, \pi_\phi, V)] := \mathbb{E}_\tau[\nabla_\phi \sum_{t=0}^{T-1} \log \pi_\phi(a_t|s_t) \cdot A(\tau, V, t)]. \tag{2.3}$$

where $A(\tau, V, t)$ is the GAE and $V : S \to \mathbb{R}$ is a value function estimate. Several recent extensions include TRPO [Schulman et al., 2015a], which discourages bad policy updates using trust regions and iterative off-policy updates, or PPO [Schulman et al., 2017], which offers similar benefits using only first-order approximations.

Parametrized Objective Functions Many of the human-engineered policy gradient estimators [e.g. Williams, 1992; Schulman et al., 2015b,a, 2017] can be

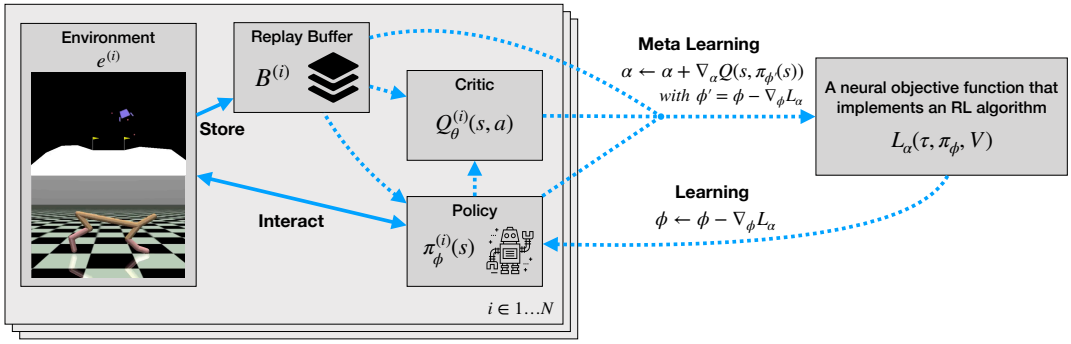


Figure 2.1: **A schematic of *MetaGenRL*.** On the left a population of agents ($i \in 1, \ldots, N$), where each member consist of a critic $Q_\theta^{(i)}$ and a policy $\pi_\phi^{(i)}$ that interact with a particular environment $e^{(i)}$ and store collected data in a corresponding replay buffer $B^{(i)}$. On the right a meta-learned neural objective function L_α that is shared across the population. Learning (dotted arrows) proceeds as follows: Each policy is updated by differentiating L_α, while the critic is updated using the usual TD-error (not shown). L_α is meta-learned by computing second-order gradients that can be obtained by differentiating through the critic.

viewed as specific implementations of a general objective function L that is differentiated with respect to the policy parameters

$$\nabla_\phi \mathbb{E}_\tau[L(\tau, \pi_\phi, V)], \tag{2.4}$$

where π_ϕ is the policy, τ is a sequence of environment interactions, and V is a value function.

Consequently, it is natural to consider a generic parameterization of L that, for specific choices of parameters α, can recover some of these estimators. In this work, we focus on *neural objective functions*, where L_α is implemented as a neural network. Our objective is to optimize the parameters α of this neural network to create a new learning algorithm that effectively maximizes Equation 2.1 across a wide range of diverse environments.

2.3 Meta-Learning neural objectives

In this work we propose *MetaGenRL*, a novel meta reinforcement learning algorithm that meta-learns neural objective functions of the form $L_\alpha(\tau, \pi_\phi, V)$. MetaGenRL makes use of value functions and second-order gradients, which makes it more sample efficient compared to prior work [Duan et al., 2016; Wang

et al., 2016; Houthooft et al., 2018]. More so, as we will demonstrate, MetaGenRL meta-learns objective functions that generalize to vastly different environments.

Our key insight is that a differentiable critic $Q_\theta : S \times A \to \mathbb{R}$ can be used to measure the effect of locally changing the objective function parameters α based on the quality of the corresponding policy gradients. This enables a population of agents to use and improve a single parameterized objective function L_α through interacting with a set of (potentially different) environments. During evaluation (meta-test time), the meta-learned objective function can then be used to train a randomly initialized RL agent in a new environment.

2.3.1 From DDPG to gradient-based meta-learning of neural objectives

We will formally introduce MetaGenRL as an extension of the DDPG actor-critic framework [Silver et al., 2014; Lillicrap et al., 2016]. In DDPG, a parameterized critic of the form $Q_\theta : S \times A \to \mathbb{R}$ transforms the non-differentiable RL reward maximization problem into a myopic value maximization problem for any $s_t \in S$. This is done by alternating between optimization of the critic Q_θ and the (here deterministic) policy π_ϕ. The critic is trained to minimize the TD-error by following:

$$\nabla_\theta \sum_{(s_t, a_t, r_t, s_{t+1})} (Q_\theta(s_t, a_t) - y_t)^2, \text{ where } y_t = r_t + \gamma \cdot Q_\theta(s_{t+1}, \pi_\phi(s_{t+1})), \quad (2.5)$$

and the dependence of y_t on the parameter vector θ is ignored. The policy π_ϕ is improved to increase the expected return from arbitrary states by following the gradient $\nabla_\phi \sum_{s_t} Q_\theta(s_t, \pi_\phi(s_t))$. Both gradients can be computed entirely off-policy by sampling trajectories from a replay buffer.

MetaGenRL builds on this idea of differentiating the critic Q_θ with respect to the policy parameters. It incorporates a parameterized objective function L_α that is used to improve the policy (i.e. by following the gradient $\nabla_\phi L_\alpha$), which adds one extra level of indirection: The critic Q_θ improves L_α, while L_α improves the policy π_ϕ. By first differentiating with respect to the objective function parameters α, and then with respect to the policy parameters ϕ, the critic can be used

Algorithm 6 MetaGenRL: Meta-Training

Require: $p(e)$ a distribution of environments
$P \Leftarrow \{(e_1 \sim p(e), \phi_1, \theta_1, B_1 \leftarrow \varnothing), \ldots\}$ ▷ Randomly initialize population of agents
Randomly initialize objective function L_α
while L_α has not converged **do**
for $e, \phi, \theta, B \in P$ **do** ▷ For each agent i in parallel
if extend replay buffer B **then**
Extend B using π_ϕ in e
Sample trajectories from B
Update critic Q_θ using TD-error
Update policy by following $\nabla_\phi L_\alpha$
Compute objective function gradient Δ_i for agent i according to Equation 2.6
Sum gradients $\sum_i \Delta_i$ to update L_α

to measure the effect of updating π_ϕ using L_α on the estimated return[2]:

$$\nabla_\alpha Q_\theta(s_t, \pi_{\phi'}(s_t)), \text{ where } \phi' = \phi - \nabla_\phi L_\alpha(\tau, x(\phi), V). \tag{2.6}$$

This constitutes a type of second order gradient $\nabla_\alpha \nabla_\phi$ that can be used to meta-train L_α to provide better updates to the policy parameters in the future. In practice we will use batching to optimize Equation 2.6 over multiple trajectories τ.

Similarly to human-engineered policy-gradient estimators from Section 2.2, the objective function $L_\alpha(\tau, x(\phi), V)$ receives as inputs an episode trajectory $\tau = (s_{0:T-1}, a_{0:T-1}, r_{0:T-1})$, the value function estimates V, and an auxiliary input $x(\phi)$ (previously π_ϕ) that can be differentiated with respect to the policy parameters. The latter is critical to be able to differentiate with respect to ϕ and in the simplest case it consists of the action as predicted by the policy. Extensions of this may include the hidden state of the policy or other policy-predicted quantities. While Equation 2.6 is used for meta-learning L_α, the objective function L_α itself is used for policy learning by following $\nabla_\phi L_\alpha(\tau, x(\phi), V)$. See Figure 2.1 for an overview. MetaGenRL consists of two phases: During meta-training, we alternate between critic updates, objective function updates, and policy updates

[2] In case of a probabilistic policy $\pi_\phi(a_t|s_t)$ the following becomes an expectation under π_ϕ and a reparameterizable form is required [Williams, 1988; Kingma and Welling, 2014; Rezende et al., 2014]. Here we focus on learning deterministic target policies.

to meta-learn an objective function L_α as described in Algorithm 6. During meta-testing in Algorithm 7, we take the learned objective function L_α and keep it fixed while training a randomly initialized policy in a new environment to assess its performance.

We note that the inputs to L_α are sampled from a replay buffer rather than solely using on-policy data. If L_α were to represent a REINFORCE-type objective then it would mean that differentiating L_α yields *biased* policy gradient estimates. In our experiments we will find that the gradients from L_α work much better in comparison to a biased off-policy REINFORCE algorithm, and to an importance-sampled unbiased REINFORCE algorithm, while also improving over the popular on-policy REINFORCE and PPO algorithms.

2.3.2 Parametrizing the objective function

We implement L_α using an LSTM [Gers et al., 2000b; Hochreiter and Schmidhuber, 1997a] that iterates over τ in reverse order and depends on the current policy action $\pi_\phi(s_t)$ (see Figure 2.2). At every time-step L_α receives the reward r_t, taken action a_t, predicted action by the current policy $\pi_\phi(s_t)$, the time t, and value function estimates V_t, V_{t+1}[3]. At each step the LSTM outputs the objective value l_t, all of which are summed to yield a single scalar output value that can be differentiated with respect to ϕ. In order to accommodate varying action dimensionalities across different environments, both $\pi_\phi(s_t)$ and a_t are first convolved and then averaged to obtain an action embedding that does not depend on the action dimensionality. Additional details, including suggestions for more expressive alternatives are available in Section A.2.

By presenting the trajectory in reverse order to the LSTM (and L_α correspondingly), it is able to assign credit to an action a_t based on its *future* impact on the reward, similar to policy gradient estimators. More so, as a general function approximator using these inputs, the LSTM is in principle able to learn different variance and bias reduction techniques, akin to advantage estimates, generalized advantage estimates, or importance weights[4]. Due to these properties, we expect the class of objective functions that is supported to somewhat relate to a

[3]The value estimates are derived from the Q-function, i.e. $V_t = Q_\theta(s_t, \pi_\phi(s_t))$, and are treated as a constant input. Hence, the gradient $\nabla_\phi L_\alpha$ can not flow backwards through Q_θ, which ensures that L_α can not naively learn to implement a DDPG-like objective function.

[4]We note that in practice it is is difficult to assess whether the meta-learned object function incorporates bias / variance reduction techniques, especially because MetaGenRL is unlikely to recover *known* techniques.

REINFORCE [Williams, 1992] estimator that uses generalized advantage estimation [Schulman et al., 2015b].

Algorithm 7 MetaGenRL: Meta-Testing

Require: A test environment e, and an objective function L_α
Randomly initialize π_ϕ, V_θ, $B \leftarrow \varnothing$
while π_ϕ has not converged **do**
  if extend replay buffer B **then**
    Extend B using π_ϕ in e
  Sample trajectories from B
  Update V_θ using TD-error
  Update policy by following $\nabla_\phi L_\alpha$

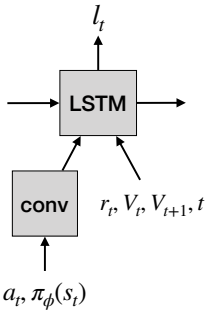


Figure 2.2: **The architecture of the objective function.** The learning algorithm is described by an objective function $L_\alpha(\tau, x(\phi), V)$ parameterized by α that processes taken and predicted actions with an element-wise convolution and processes an entire trajectory τ with an LSTM.

2.3.3 Generality and efficiency of MetaGenRL

MetaGenRL offers a general framework for meta-learning objective functions that can represent a wide range of learning algorithms. In particular, it is only required that both π_ϕ and L_α can be differentiated w.r.t. to the policy parameters ϕ. In the present work, we use this flexibility to leverage population-based meta-optimization, increase sample efficiency through off-policy second-order gradients, and to improve the generalization capabilities of meta-learned objective functions.

Population-Based A general objective function should be applicable to a wide range of environments and agent parameters. To this extent MetaGenRL is able to leverage the collective experience of *multiple* agents to perform meta-learning by using a *single* objective function L_α shared among a population of agents that each act in their own (potentially different) environment. Each agent locally computes Equation 2.6 over a batch of trajectories, and the resulting gradients are combined to update L_α. Thus, the relevant learning experience of each individual agent is compressed into the objective function that is available to the entire population at any given time.

Sample Efficiency An alternative to learning neural objective functions using a population of agents is through evolution as in EPG [Houthooft et al., 2018]. However, we expect meta-learning using second-order gradients as in MetaGenRL to be much more sample efficient. This is due to off-policy training of the objective function L_α and its subsequent off-policy use to improve the policy. Indeed, unlike in evolution there is no need to train multiple randomly initialized agents in their entirety in order to evaluate the objective function, thus speeding up credit assignment. Rather, at any point in time, any information that is deemed useful for future environment interactions can directly be incorporated into the objective function. Finally, using the formulation in Equation 2.6 one can measure the effects of improving the policy using L_α for *multiple* steps by increasing the corresponding number of gradient steps before applying Q_θ, which we will explore in Figure 2.5.2.

Meta-Generalization The focus of this work is to learn *general* learning rules that during test-time can be applied to vastly different environments. A strict separation between the policy and the learning rule, the functional form of the latter, and training across many environments all contribute to this. Regarding the former, a clear separation between the policy and the learning rule as in MetaGenRL is expected to be advantageous for two reasons. Firstly, it allows us to specify the number of parameters of the learning rule independent of the policy and critic parameters. For example, our implementation of L_α uses only $15K$ parameters for the objective function compared to $384K$ parameters for the policy and critic. Hence, we are able to only use a short description length for the learning rule. A second advantage that is gained is that the meta-learner is unable to directly change the policy and must, therefore, learn to make use of the objective function. This makes it difficult for the meta-learner to *overfit* to the training environments.

Computational cost Sample efficiency is an important factor to consider when designing meta-learning systems. This is due to either limited data availability or the high cost of environment interactions in many RL settings, such as with the physical world or in complex simulations. After considering sample efficiency, the compute and space cost of meta-training and meta-testing should be considered. Here, we can directly analyze the complexity of each iteration step, whereas the overall computational cost can be estimated based on our empirical number of iterations until convergence. An iteration of meta-training in MetaGenRL consists of several stages with various runtime complexities. Data

collection has a complexity of $O(N_P N_s N_\phi)$ where N_P is the size of the agent population, N_s is the number of newly collected transitions per agent, and N_ϕ is the number of floating point operations in a forward pass of an agent π_ϕ. The critic update has a complexity of $O(N_P N_B(N_\phi + N_\theta))$ where N_B is the number of transitions sampled from the buffer B and N_θ is the number of floating point operations in a forward pass of the critic. Policy learning has a complexity of $O(N_P N_B(N_\phi + N_\alpha))$ where N_α is the number of floating points operations in a forward pass of the objective function L_α. Finally, the objective function update has a complexity of $O(N_P N_B(N_\phi N_\alpha + N_\theta))$ due to the second order gradient $\nabla_\alpha \nabla_\phi$ in Equation 2.6. In total, meta-training has an iteration runtime complexity of $O(N_P(N_s N_\phi + N_B(N_\phi N_\alpha + N_\theta)))$. Space complexity is dominated by the size of the buffer B. In contrast, meta-testing is significantly cheaper. Each iteration involves only a single agent, and no objective function update, resulting in a computational complexity of $O(N_s N_\phi + N_B(N_\phi + N_\theta + N_\alpha))$. Arguably, this is the more relevant computational complexity when the meta-learned learning algorithms generalizes well, as intended in this work.

2.4 Related work

Among the earliest pursuits in meta-learning are meta-hierarchies of genetic algorithms [Schmidhuber, 1987] and learning update rules in supervised learning [Bengio et al., 1991]. While the former introduced a general framework of entire meta-hierarchies, it relied on discrete non-differentiable programs. The latter proposed local update rules that included free parameters, which could be learned using gradients in a supervised setting. Schmidhuber [1993b] conceptualized a differentiable self-referential RNN that could address and modify its own weights.

Hochreiter et al. [2001] introduced differentiable meta-learning using RNNs to scale to larger problem instances, today known as in-context learning. By giving an RNN access to its prediction error, it could implement its own meta-learning algorithm, where the weights are the meta-learned parameters, and the hidden states the subject of learning. This was later extended to the RL setting [Wang et al., 2016; Duan et al., 2016; Santoro et al., 2016; Mishra et al., 2018] (here refered to as RL^2). As we show empirically in our work, meta-learning with RL^2 does not generalize well. It lacks a clear separation between policy and objective function, which makes it easy to overfit on training environments. This is exacerbated by the imbalance of $O(n^2)$ meta-learned parameters to learn $O(n)$ activations, unlike in MetaGenRL.

Many other recent meta-learning algorithms learn a policy parameter initialization that is later fine-tuned using a fixed reinforcement learning algorithm [Finn et al., 2017; Schulman et al., 2017; Grant et al., 2018; Yoon et al., 2018]. Different from MetaGenRL, these approaches use second order gradients on the same policy parameter vector instead of using a separate objective function. Albeit in principle general [Finn and Levine, 2018], it was later shown that these approaches perform more feature reuse than meta-learning, relying largely on the human-engineered learning algorithm [Raghu et al., 2020].

Objective functions have been learned prior to MetaGenRL. Houthooft et al. [2018] evolve an objective function that is later used to train an agent. Unlike MetaGenRL, this approach is extremely costly in terms of the number of environment interactions required to evaluate and update the objective function. Most recently, Bechtle et al. [2021] introduced learned loss functions for reinforcement learning that also make use of second-order gradients, but use a policy gradient estimator instead of a Q-function. Similar to other work, their focus is only on narrow task distributions. Learned objective functions have also been used for learning unsupervised representations [Metz et al., 2019a], DDPG-like meta-gradients for hyperparameter search [Xu et al., 2018], and learning from human demonstrations [Yu et al., 2018]. Concurrent to our work, Alet et al. [2020] uses techniques from architecture search to search for viable artificial curiosity objectives that are composed of primitive objective functions.

Li and Malik [2017] and Andrychowicz et al. [2016] conduct meta-learning by learning optimizers that update parameters ϕ by modulating the gradient of some fixed objective function L: $\Delta\phi = f_\alpha(\nabla_\phi L)$ where α is learned. They differ from MetaGenRL in that they only modulate the gradient of a fixed objective function L instead of learning L itself.

Another connection exists to meta-learned intrinsic reward functions [Niekum et al., 2011; Zheng et al., 2018; Jaderberg et al., 2019] which are related to earlier work on subgoal generation [Schmidhuber, 1991b; Wiering and Schmidhuber, 1996a; Singh et al., 2004]. Choosing $\nabla_\phi L_\alpha = \tilde{\nabla}_\phi \sum_{t=1}^{T} \bar{r}_t(\tau)$, where $\bar{r}_t$ is a meta-learned reward and $\tilde{\nabla}_\theta$ is a gradient estimator (such as a value based or policy gradient based estimator) reveals that meta-learning objective functions includes meta-learning the gradient estimatior $\tilde{\nabla}$ itself as long as it is expressible by a gradient ∇_θ on an objective L_α. In contrast, for intrinsic reward functions, the gradient estimator $\tilde{\nabla}$ is normally fixed.

Finally, we note that positive transfer between different tasks (reward functions) as well as environments (e.g. different Atari games) has been shown previously

Table 2.1: **An evaluation of MetaGenRL and baselines across various meta-training and meta-testing regimes.** Mean return across multiple seeds (MetaGenRL: 6 meta-train $\times$ 2 meta-test seeds, RL^2: 6 meta-train $\times$ 2 meta-test seeds, EPG: 3 meta-train $\times$ 2 meta-test seeds) obtained by training randomly initialized agents during meta-test time on previously seen environments (cyan) and on unseen environments (brown). Boldface highlights the best meta-learned algorithm. The mean returns (6 seeds) of several human-engineered algorithms are also listed.

Training \Testing		Cheetah	Hopper	Lunar
Cheetah & Hopper	**MetaGenRL**	2185	**2439**	**18**
	EPG	-571	20	-540
	RL^2	**5180**	289	-479
Lunar & Cheetah	**MetaGenRL**	**2552**	**2363**	258
	EPG	-701	8	-707
	RL^2	2218	5	**283**
Lunar & Hopper & Walker & Ant	**MetaGenRL (40 agents)**	3106	2869	201
Cheetah & Lunar & Walker & Ant		3331	2452	-71
Cheetah & Hopper & Walker & Ant		2541	2345	-148
	PPO	1455	1894	187
	DDPG / TD3	8315	2718	288
	off-policy REINFORCE (GAE)	-88	1804	168
	on-policy REINFORCE (GAE)	38	565	120

in the context of transfer learning [Kistler et al., 1997; Parisotto et al., 2015; Rusu et al., 2016, 2019; Nichol et al., 2018] and meta-critic learning across tasks [Sung et al., 2017]. In contrast to this work, the approaches that have shown to be successful in this domain rely entirely on human-engineered learning algorithms.

2.5 Experiments

We investigate the learning and generalization capabilities of MetaGenRL on several continuous control benchmarks including *HalfCheetah (Cheetah)* and *Hopper* from MuJoCo [Todorov et al., 2012], and *LunarLanderContinuous (Lunar)* from OpenAI gym [Brockman et al., 2016]. These environments differ significantly in terms of the properties of the underlying system that is to be controlled, and in terms of the dynamics that have to be learned to complete the environment. Hence, by training meta-RL algorithms on one environment and testing on other environments they provide a reasonable measure of out-of-distribution

generalization.

In our experiments, we will mainly compare to EPG [Houthooft et al., 2018] and to RL^2 [Duan et al., 2016; Wang et al., 2016] to evaluate the efficacy of our approach. We will also compare to several fixed model-free RL algorithms to measure how well the algorithms meta-learned by MetaGenRL compare to these handcrafted alternatives. Unless otherwise mentioned, we will meta-train MetaGenRL using 20 agents that are distributed equally over the indicated training environments[5]. Meta-learning uses clipped double-Q learning, delayed policy & objective updates, and target policy smoothing from TD3 [Fujimoto et al., 2018]. We will allow for $600K$ environment interactions per agent during meta-training and then meta-test the objective function for $1M$ interactions. Further details are available in Section A.2.

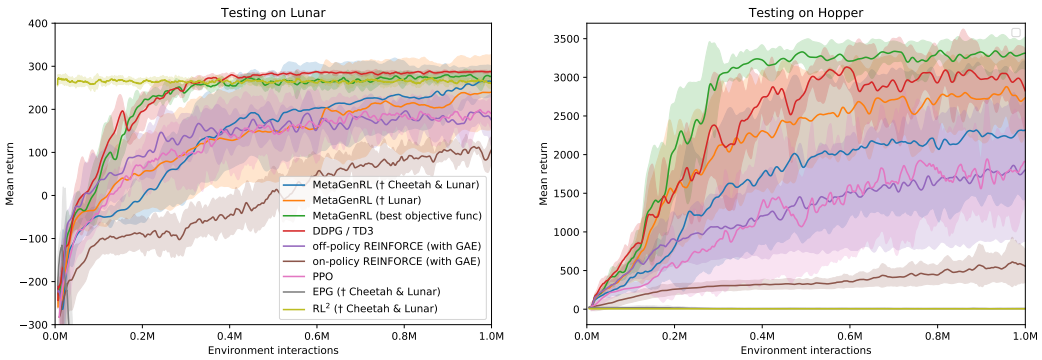
2.5.1 Comparison to prior work

Evaluating on previously seen environments We meta-train MetaGenRL on *Lunar* and compare its ability to train a randomly initialized agent at test-time (i.e. using the learned objective function and keeping it fixed) to DDPG, PPO, and on- and off-policy REINFORCE (both using GAE [Schulman et al., 2015b]) across multiple seeds. Figure 2.3a shows that MetaGenRL markedly outperforms both the REINFORCE baselines and PPO. Compared to DDPG, which finds the optimal policy, MetaGenRL performs only slightly worse on average although the presence of outliers increases its variance. In particular, we find that some meta-test agents get 'stuck' for some time before reaching the optimal policy (see Section A.1.2 for additional analysis). Indeed, when evaluating only the best meta-learned objective function that was obtained during meta-training (MetaGenRL (best objective func) in Figure 2.3a) we are able to observe a strong reduction in variance and even better performance.

We also report results (Figure 2.3a) when meta-training MetaGenRL on both *Lunar* and *Cheetah*, and compare to EPG and RL^2 that were meta-trained on these same environments[6]. For MetaGenRL we were able to obtain similar performance to meta-training on only *Lunar* in this case. In contrast, for EPG it can be observed that even one billion environment interactions is insufficient to find a

[5] An ablation study in Section A.1.3 revealed that a large number of agents is indeed required.

[6] In order to ensure a good baseline we allowed for a maximum of $100M$ environment interactions for RL^2 and $1B$ for EPG, which is more than eight / eighty times the amount used by MetaGenRL. Regarding EPG, this did require us to reduce the total number of seeds to 3 meta-train $\times$ 2 meta-test seeds.



(a) Previously seen *Lunar* environment. *(b)* Unseen *Hopper* environment.

Figure 2.3: **MetaGenRL uniquely generalize to completely unseen environments with different observations, actions, and dynamics.** Comparing the test-time training behavior of the meta-learned objective functions by MetaGenRL to other (meta) reinforcement learning algorithms. We train randomly initialized agents on (a) environments that were encountered during training, and (b) on significantly different environments that were unseen. Training environments are denoted by † in the legend. All runs are shown with mean and standard deviation computed over multiple random seeds (MetaGenRL: 6 meta-train $\times$ 2 meta-test seeds, RL^2: 6 meta-train $\times$ 2 meta-test seeds, EPG: 3 meta-train $\times$ 2 meta-test seeds, and 6 seeds for all others).

good objective function (in Figure 2.3a quickly dropping below -300). Finally, we find that RL^2 reaches the optimal policy after 100 million meta-training iterations, and that its performance is unaffected by additional steps during testing on *Lunar*. We note that RL^2 does not separate the policy and the learning rule and indeed in a similar 'within distribution' evaluation, RL^2 was found successful [Wang et al., 2016; Duan et al., 2016].

Table 2.1 provides a similar comparison for two other environments. Here we find that in general MetaGenRL is able to outperform the REINFORCE baselines and PPO, and in most cases (except for *Cheetah*) performs similar to DDPG[7]. We also find that MetaGenRL consistently outperforms EPG, and often RL^2. For an analysis of meta-training on more than two environments we refer to Section A.1.

[7]We emphasize that the neural objective function under consideration is unable to implement DDPG and only uses a constant value estimate (i.e. $\nabla_\phi V = 0$ by using gradient stopping) during meta testing.

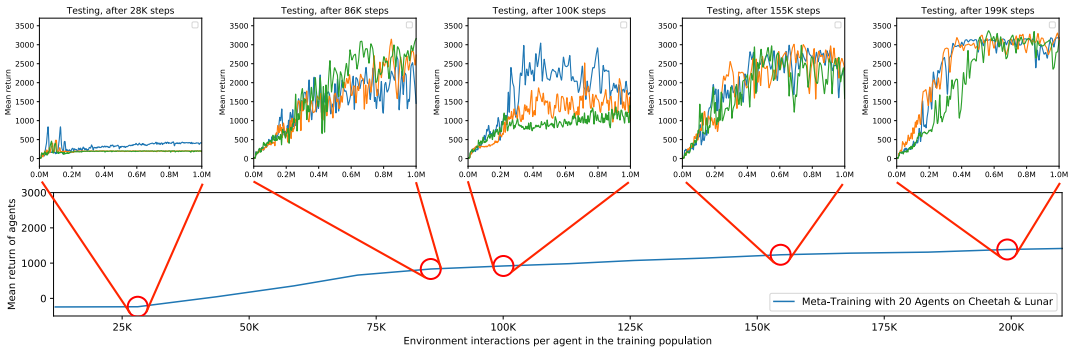


Figure 2.4: **We visualize how meta-test time behavior improves over the course of meta-training.** Meta-training with 20 agents on *Cheetah* and *Lunar*. We test the objective function at five stages of meta-training by using it to train three randomly initialized agents on *Hopper*.

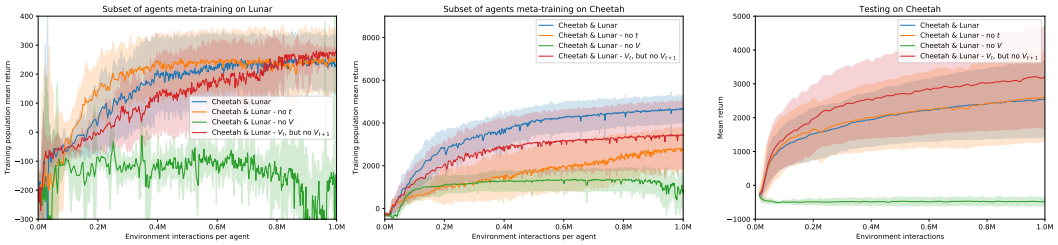
Generalization to vastly different environments We evaluate the same objective functions learned by MetaGenRL, EPG and the recurrent dynamics by RL^2 on *Hopper*, which is significantly different compared to the meta-training environments. Figure 2.3b shows that the learned objective function by MetaGenRL continues to outperform both PPO and our implementations of REINFORCE, while the best performing configuration is even able to outperform DDPG.

When comparing to related meta-RL approaches, we find that MetaGenRL is significantly better in this case. The performance of EPG remains poor, which was expected given what was observed on previously seen environments. On the other hand, we now find that the RL^2 baseline fails completely (resulting in a flat low-reward evaluation), suggesting that the learned learning rule that was previously found to be successful is in fact entirely overfitted to the environments that were seen during meta-training. We were able to observe similar results when using different train and test environment splits as reported in Table 2.1, and in Section A.1.

2.5.2 Analysis

Meta-Training Progression of Objective Functions

Previously we focused on test-time training randomly initialized agents using an objective function that was meta-trained for a total of $600K$ steps (corresponding to a total of $12M$ environment interactions across the entire population). We will now investigate the quality of the objective functions during meta-training.



(a) Meta-training on *Lunar* & *Cheetah* *(b)* Meta-testing on *Cheetah*

Figure 2.5: **Ablating the objective function inputs.** We meta-train MetaGenRL using several alternative parametrizations of L_α on a) *Lunar* and *Cheetah*, and b) present results of testing on *Cheetah*. During meta-training a representative example of a single agent population is shown with shaded regions denoting standard deviation across the population. Meta-test results are reported as per usual across 6 meta-train $\times$ 2 meta-test seeds.

Figure 2.4 displays the result of evaluating an objective function on *Hopper* at different intervals during meta-training on *Cheetah* and *Lunar*. Initially ($28K$ steps) it can be seen that due to lack of meta-training there is only a marginal improvement in the return obtained during test time. However, after only meta-training for $86K$ steps we find (perhaps surprisingly) that the meta-trained objective function is already able to make consistent progress in optimizing a randomly initialized agent during test-time. On the other hand, we observe large variances at test-time during this phase of meta-training. Throughout the remaining stages of meta-training we then observe an increase in convergence speed, more stable updates, and a lower variance across seeds.

Ablation study

We conduct an ablation study of the neural objective function that was described in Section 2.3.2. In particular, we assess the dependence of L_α on the value estimates V_t,V_{t+1} and on the time component that could to some extent be learned. Other ablations, including limiting access to the action chosen or to the received reward, are expected to be disastrous for generalization to any other environment (or reward function) and therefore not explored.

Dependence on t We use a parameterized objective function of the form $L_\alpha(a_t, r_t, V_t, \pi_\phi(s_t)|t \in 0, ..., T-1)$ as in Figure 2.2 except that it does not receive information about the time-step t at each step. Although information about the

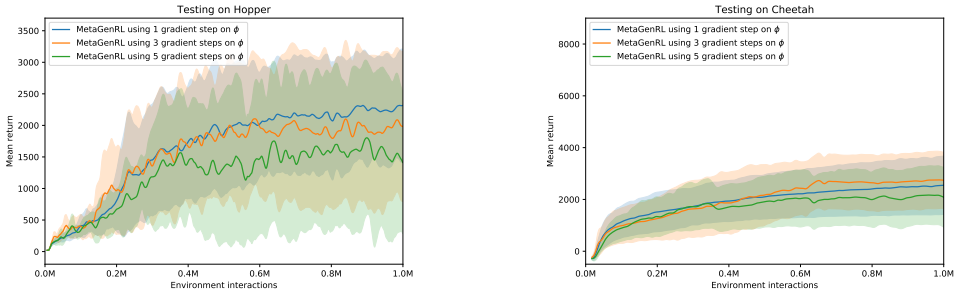


Figure 2.6: **Ablating the inner gradient steps.** We meta-train MetaGenRL on the *LunarLander* and *HalfCheetah* environments using one, three, and five inner gradient steps on ϕ. Meta-test results are reported across 3 meta-train $\times$ 2 meta-test seeds.

current time-step is required in order to learn (for example) a generalized advantage estimate [Schulman et al., 2015b], the LSTM could in principle learn such time tracking on it own, and we expect only minor effects on meta-training and during meta-testing. Indeed in Figure 2.5b it can be seen that the neural objective function performs well without access to t, although it converges slower on *Cheetah* during meta-training (Figure 2.5a).

Dependence on V We use a parameterized objective function of the form $L_\alpha(a_t, r_t, t, \pi_\phi(s_t)|t \in 0, ..., T-1)$ as in Figure 2.2 except that it does not receive any information about the value estimates at time-step t. There exist reinforcement learning algorithms that work without value function estimates (eg. Amari [1967]; Williams [1992]; Schmidhuber and Zhao [1999]), although in the absence of an alternative baseline these often have a large variance. Similar results are observed for this ablation in Figure 2.5a during meta-training where a possibly large variance appears to affect meta-training. Correspondingly during test-time (Figure 2.5b) we do not find any meaningful training progress to take place. In contrast, we find that we can remove the dependence on *one* of the value function estimates, i.e. remove V_{t+1} but keep V_t, which during some runs even increases performance.

Multiple gradient steps

We analyze the effect of making *multiple* gradient updates to the policy using L_α before applying the critic to compute second-order gradients with respect to the objective function parameters as in Equation 2.6. While in previous experiments

we have only considered applying a single update, multiple gradient updates might better capture long term effects of the objective function. At the same time, moving further away from the current policy parameters could reduce the overall quality of the second-order gradients. Indeed, in Figure 2.6 it can be observed that using 3 gradient steps already slightly increases the variance during test-time training on *Hopper* and *Cheetah* after meta-training on *LunarLander* and *Cheetah*. Similarly, we find that further increasing the number of gradient steps to 5 harms performance.

2.6 Conclusion

We have presented MetaGenRL, a novel off-policy gradient-based meta reinforcement learning algorithm that leverages a population of DDPG-like agents to meta-learn general objective functions. Unlike related methods the meta-learned objective functions do not only generalize in narrow task distributions but show similar performance on entirely different tasks while markedly outperforming REINFORCE and PPO. We have argued that this generality is due to MetaGenRL's explicit separation of the policy and learning rule, the functional form of the latter, and training across multiple agents and environments. Furthermore, the use of second order gradients increases MetaGenRL's sample efficiency by several orders of magnitude compared to EPG [Houthooft et al., 2018].

An exciting direction for future work is to further expand the expressivity of the meta-learned objective functions. Indeed, in our current implementation, the objective function is unable to observe the environment or the hidden state of the (recurrent) policy. These extensions are especially interesting as they may allow more complicated curiosity-based [Schmidhuber, 1991a; Houthooft et al., 2018; Pathak et al., 2017] or model-based [Schmidhuber, 1990; Racanière et al., 2017; Ha and Schmidhuber, 2018] algorithms to be learned. To this extent, it will be important to develop introspection methods that analyze the learned objective function and to scale MetaGenRL to make use of many more environments and agents.

2.7 Follow-up work

Since the publication of MetaGenRL, several works have been exploring the automatic discovery of reinforcement learning algorithms with a strong focus on

generalizability of the learned learning algorithms. For example, researchers have meta-learned artificial curiosity algorithms [Alet et al., 2020] and reinforcement learning algorithms [Co-Reyes et al., 2021] by describing them using symbolic objective functions. In the space of meta-gradients, Oh et al. [2020] have searched over an expressive space of objective functions that can also represent value functions, showing strong performance and generalization to unseen environments. Lu et al. [2022] have used evolutionary algorithms to search over PPO-like policy gradient algorithms with a more constrained search space, leading to the discovery of more interpretable and generalizable algorithms.

Chapter 3

VSML: Meta-learning backpropagation and improving it

Keywords *in-context learning, backpropagation, supervised learning, fast weight programmers*
Article *Kirsch and Schmidhuber [2021] (preprint 2020)*

Our previously proposed MetaGenRL (Chapter 2) still relies on the known backpropagation algorithm to optimize an objective function at meta-test time. This reliance on backpropagation may limit the learning algorithms that can be automatically discovered. How can we further reduce the inductive biases in the learning algorithms that we discover? In VSML, we investigate the implementation of the entire learning algorithm and inference in a neural network without any hardcoded gradient descent, which we refer to as in-context learning. In contrast to previous research, our objective is to discover new *general-purpose* in-context learning algorithms that can be applied to a wide range of tasks.

3.1 Introduction

The shift from standard machine learning to meta-learning involves learning the learning algorithm (LA) itself, reducing the burden on the human designer to craft useful learning algorithms [Schmidhuber, 1987]. Recent meta-learning has primarily focused on generalization from training tasks to similar test tasks, e.g., few-shot learning [Finn et al., 2017], or from training environments to similar test environments [Houthooft et al., 2018]. This contrasts human-engineered LAs that generalize across a wide range of datasets or environments. With-

out such generalization, meta-learned LAs can not entirely replace human-engineered variants. More recently, meta-learning was demonstrated to also successfully generate more general LAs that generalize across wide spectra of environments [Kirsch et al., 2020b; Alet et al., 2020; Oh et al., 2020], e.g., from toy environments to Mujoco and Atari.

Unfortunately, however, many recent approaches, such as our previously proposed *MetaGenRL* (Chapter 2), still rely on a large number of human-designed and unmodifiable inner-loop components such as backpropagation. How could we further reduce the inductive biases in our discovered learning algorithms? One approach is to implement the entire learning algorithm and inference in a neural network, such as an RNN or a Transformer. Today, this is commonly known as in-context learning [Brown et al., 2020], but has also been referred to as memory-based learning [Duan et al., 2016; Wang et al., 2016; Ortega et al., 2019] or black-box learning-to-learn [Kirsch and Schmidhuber, 2021]. Although this is an expressive method of meta-learning, it also quickly leads to overfitting and poses difficulties in meta-learning learning algorithms that can be applied to completely unseen tasks. Such a generalization has not been demonstrated prior to this work.

Our objective then is to discover novel general-purpose learning algorithms purely by using in-context learning. To this end, is it possible to implement modifiable versions of backpropagation or related algorithms as part of the end-to-end differentiable activation dynamics of a neural net (NN), instead of inserting them as separate fixed routines? Here, we propose the Variable Shared Meta Learning (VSML) principle for this purpose. It introduces a novel way of using sparsity and weight-sharing in NNs for meta-learning. We build on the arguably simplest neural meta-learner, an in-context learner in the form of a recurrent neural network (in-context RNN / Meta RNN) [Hochreiter et al., 2001; Duan et al., 2016; Wang et al., 2016], by replicating the RNN many times. The resulting system can be viewed as many RNNs passing messages to each other, or as one big RNN with a sparse shared weight matrix, or as a system meta-learning the functionality and learning algorithm of each neuron in another neural network. VSML generalizes the principle behind end-to-end differentiable fast weight programmers [Schmidhuber, 1992b, 1993a; Ba et al., 2016a; Schlag et al., 2021b], hyper networks [Ha et al., 2017], learned learning rules [Bengio et al., 1992; Gregor, 2020; Randazzo et al., 2020], and hebbian-like synaptic plasticity [Miconi et al., 2018; Najarro and Risi, 2020].

Our mechanism, VSML, can implement backpropagation solely in the forward-

dynamics of an RNN. Consequently, it enables meta-optimization of backprop-like algorithms. We envision a future where novel methods of credit assignment can be meta-learned while still generalizing across vastly different tasks. This may lead to improvements in sample efficiency, memory efficiency, continual learning, and others. As a first step, our system meta-learns online in-context LAs from scratch that frequently learn faster than gradient descent and generalize to datasets outside of the meta-training distribution (e.g., from MNIST to Fashion MNIST). VSML is the first neural in-context learner without hardcoded backpropagation that shows such strong generalization. Introspection reveals that our meta-learned LAs learn through fast association in a way that is qualitatively different from gradient descent.

3.2 Background

Deep learning-based meta-learning that does not rely on fixed gradient descent in the inner loop has historically fallen into two categories, 1) Learnable weight update mechanisms that allow for changing the parameters of an NN to implement a learning rule (Fast Weight Programmers, FWPs), and 2) Learning algorithms implemented in black-box models such as recurrent neural networks (in-context RNNs), now commonly known as in-context learning.

Fast weight programmers (FWPs) In a standard NN, the weights are updated by a fixed LA. This framework can be extended to meta-learning by defining an explicit architecture that allows for modifying these weights. This weight-update architecture augments a standard NN architecture. NNs that generate or change the weights of another or the same NN are known as fast weight programmers (FWPs) [Schmidhuber, 1992b, 1993a; Ba et al., 2016a; Schlag and Schmidhuber, 2017], hypernetworks [Ha et al., 2017], NNs with synaptic plasticity [Miconi et al., 2018, 2019; Najarro and Risi, 2020] or learned learning rules [Bengio et al., 1991; Gregor, 2020; Randazzo et al., 2020]. Often these architectures make use of local Hebbian-like update rules or outer-products, and we summarize this category as FWPs. In FWPs the variables V_L that are subject to learning are the weights of the network, whereas the meta-variables V_M that implement the LA are defined by the weight-update architecture. Note that the dimensionality of V_L and V_M can be defined independently of each other and often V_M are reused in a coordinate-wise fashion for V_L resulting in $|V_L| \gg |V_M|$, where $|\cdot|$ is the number of elements.

In-context / Black-box learning It was shown that a black-box neural network such as an LSTM or Transformer can learn to implement an LA [Hochreiter et al., 2001] when the reward or error is given as input [Schmidhuber, 1993b]. After meta-training, the LA is encoded in the weights θ of the model and determines *in-context* learning during meta-testing. The activations serve as the memory used for the LA solution. In-context learners are conceptually simpler than FWPs as no additional weight update rules with many degrees of freedom need to be defined. Here, we focus on recurrent in-context learners, referred to as in-context RNNs [Hochreiter et al., 2001; Duan et al., 2016; Wang et al., 2016].

Overfitting of in-context RNNs An in-context RNN with N neurons has $O(N)$ activations (used for inner learning, referred to as learned variables V_L). In contrast, there are $O(N^2)$ parameters θ (used for meta-learning, referred to as meta variables V_M). This means that the learning algorithm is largely overparameterized, whereas the available memory for learning is very small, making this approach prone to overfitting [Section 2.3.3 Kirsch et al., 2020b]. As a result, RNN parameters often encode task-specific solutions instead of generic LAs. Meta-learning a simple and generalizing LA would benefit from $|V_L| \gg |V_M|$. A large $|V_L|$ is also required to allow the discovered learning algorithm to learn and store complex processes. Previous approaches have tried to solve this problem by adding architectural complexity through external memory mechanisms [Sun, 1991; Mozer and Das, 1993; Santoro et al., 2016; Mishra et al., 2018; Schlag et al., 2021b].

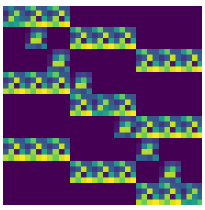
In-context learning

Terminology: The concept of in-context learning has appeared in the literature under many different names. We use the following terms interchangeably

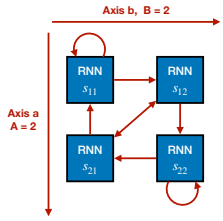
- in-context learning
- memory-based meta-learning
- black-box meta-learning

3.3 Variable Shared Meta Learning (VSML)

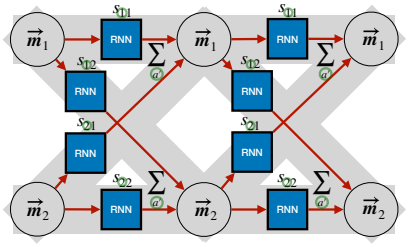
In VSML we build on the simplicity of in-context learning RNNs while ensuring that $|V_L| \gg |V_M|$. We do this by reusing the same few parameters $V_M := \theta$ many times in an RNN and introducing sparsity in the connectivity. This yields



(a) (RNN) Viewed as a single RNN (structured weight matrix)



(b) (MP) Viewed as many sub-RNNs with message passing



(c) (CN) Viewed as an NN with meta-learned complex neurons

Figure 3.1: **Different perspectives on VSML:** (a) (RNN) A single in-context RNN [Hochreiter et al., 2001] where entries in the weight matrix are shared or zero. (b) (MP) VSML consists of many sub-RNNs with shared parameters θ passing messages between each other. (c) (CN) VSML implements an NN with complex neurons (here 2 neurons). θ determines the nature of weights, how these are used in the neural computation, and the LA by which those are updated. Each weight $w_{ab} \in \mathbb{R}$ is represented by the multi-dimensional RNN state $s_{ab} \in \mathbb{R}^N$. Neuron activations correspond to messages $\overrightarrow{m}$ passed between sub-RNNs.

several interpretations for VSML:

(RNN) **VSML as a *single in-context RNN* with a sparse shared weight matrix (Figure 3.1a).** The most general description.

(MP) **VSML as *message passing* between RNNs (Figure 3.1b).** We choose a simple sharing and sparsity scheme for the weight matrix such that it corresponds to multiple RNNs with shared parameters that exchange information.

(CN) **VSML as *complex neurons* with learned updates (Figure 3.1c).** When choosing a specific connectivity between RNNs, states / activations V_L of these RNNs can be interpreted as the weights of a conventional NN, consequently blurring the distinction between a weight and an activation. The learned state updates (in-context learning) may then implement learned weight updates (FWPs).

Introducing variable sharing to in-context RNNs We begin by formalizing in-context RNNs which often use multiplicative gates such as the LSTM [Gers et al., 2000b; Hochreiter and Schmidhuber, 1997a] or its variant GRU [Cho et al., 2014]. For notational simplicity, we consider a vanilla RNN. Let $s \in \mathbb{R}^N$ be

the hidden state of an RNN. The update for an element $j \in \{1, \dots, N\}$ is given by

$$s_j \leftarrow f_{\text{RNN}}(s)_j = \sigma(\sum_i s_i W_{ij}), \tag{3.1}$$

where σ is a non-linear activation function, $W \in \mathbb{R}^{N \times N}$, and the bias is omitted for simplicity. We also omit inputs by assuming a subset of s to be externally provided. Each application of Equation 3.1 reflects a single time tick in the RNN.

We now introduce variable sharing (reusing W) into the RNN by duplicating the computation along two axes of size A, B (here $A = B$, which will later be relaxed) giving $s \in \mathbb{R}^{A \times B \times N}$. For $a \in \{1, \dots, A\}, b \in \{1, \dots, B\}$ we have

$$s_{abj} \leftarrow f_{\text{RNN}}(s_{ab})_j = \sigma(\sum_i s_{abi} W_{ij}). \tag{3.2}$$

This can be viewed as multiple RNNs arranged on a 2-dimensional grid, with shared parameters that update independent states. Here, we chose a particular arrangement (two axes) that will facilitate the interpretation (CN) of RNNs as weights.

VSML as message passing between RNNs The computation so far describes $A \cdot B$ independent RNNs. We connect those by passing messages (interpretation (MP))

$$s_{ab} \leftarrow f_{\text{RNN}}(s_{ab}, \overrightarrow{m}_a), \tag{3.3}$$

where the message $\overrightarrow{m}_a = \sum_{a'} f_{\overrightarrow{m}}(s_{a'a})$ with $a \in \{1, \dots, A = B\}$, $f_{\overrightarrow{m}} : \mathbb{R}^N \to \mathbb{R}^{N'}$ is fed as an additional input to each RNN. This is related to Graph Neural Networks [Sperduti, 1994; Wu et al., 2020]. Summing over the axis A (elements a') corresponds to an RNN connectivity mimicking those of weights in an NN (to facilitate interpretation (CN)). We emphasise that other schemes based on different kinds of message passing and graph connectivity are possible. For a simple $f_{\overrightarrow{m}}$ defined by the matrix $C \in \mathbb{R}^{N \times N}$, we may equivalently write

$$s_{abj} \leftarrow \sigma(\sum_i s_{abi} W_{ij} + \sum_{a'} f_{\overrightarrow{m}}(s_{a'a})_j) = \sigma(\sum_i s_{abi} W_{ij} + \sum_{a',i} s_{a'ai} C_{ij}). \tag{3.4}$$

This constitutes the minimal version of VSML with $\theta := (W, C)$ and is visualized in Figure 3.1b.

VSML as an in-context RNN with a sparse shared weight matrix It is trivial to see that with $A = 1$ and $B = 1$ we obtain a single RNN and Equation 3.4 recovers the original in-context RNN Equation 3.1. In the general case, we can derive an equivalent formulation that corresponds to a single standard RNN with a single matrix $\tilde{W}$ that has entries of zero and shared entries

$$s_{abj} \leftarrow \sigma(\sum_{c,d,i} s_{cdi} \tilde{W}_{cdiabj}), \tag{3.5}$$

where the six axes can be flattened to obtain the two axes. For Equation 3.4 and Equation 3.5 to be equivalent, $\tilde{W}$ must satisfy (derivation in Section B.1)

$$\tilde{W}_{cdiabj} = \begin{cases} C_{ij}, & \text{if } d = a \wedge (d \neq b \vee c \neq a). \\ W_{ij}, & \text{if } d \neq a \wedge d = b \wedge c = a. \\ C_{ij} + W_{ij}, & \text{if } d = a \wedge d = b \wedge c = a. \\ 0, & \text{otherwise.} \end{cases} \tag{3.6}$$

This corresponds to interpretation (RNN) with the weight matrix visualized in Figure 3.1a. To distinguish between the single sparse shared RNN and the connected RNNs, we now call the latter *sub-RNNs*.

VSML as complex neurons with learned updates The arrangement and connectivity of the sub-RNNs as described in the previous paragraphs corresponds to that of weights in a standard NN. Thus, in interpretation (CN), VSML can be viewed as defining complex neurons where each sub-RNN corresponds to a weight in a standard NN as visualized in Figure 3.1c. All these sub-RNNs share the same parameters but have distinct states. The previous Equation 3.3 corresponds to a single NN layer that is run recurrently. We will generalize this to other architectures in the next section. A corresponds to the dimensionality of the inputs and B to that of the outputs in that layer.

The role of weights in a standard neural network is now assigned to the states of RNNs. This allows these RNNs to define both the neural forward computation as well as the learning algorithm that determines how the network is updated (where the mechanism is shared across the network). In the case of backpropagation, this would correspond to the forward and backward passes being implemented purely in the recurrent dynamics. We will demonstrate the practical feasibility of this in Section 3.3.2. The emergence of RNN states as weights quickly leads to confusing terminology when RNNs have 'meta weights'. Instead, we simply refer to the RNN 'weights' θ as meta variables V_M and the RNN activations as learned variables V_L.

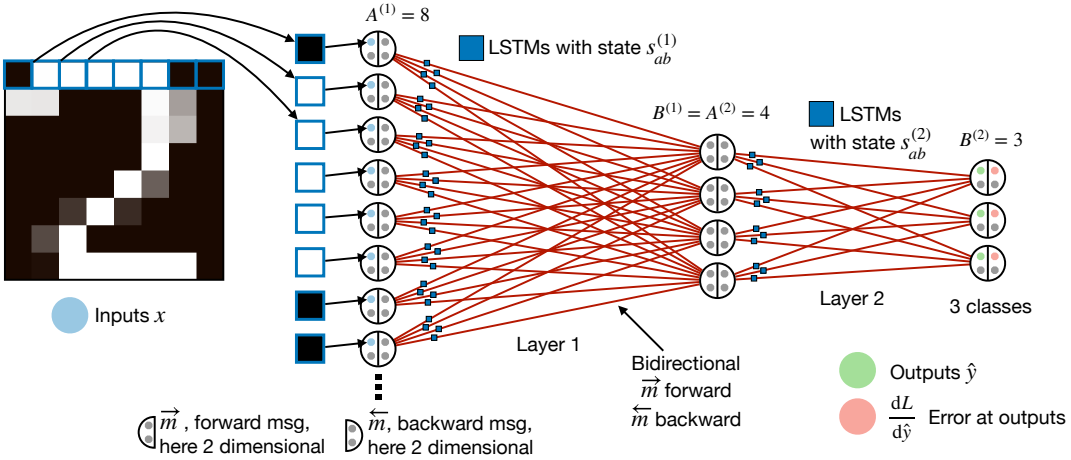


Figure 3.2: **The neural interpretation of VSML** replaces all weights of a standard NN with tiny LSTMs using shared parameters (resembling complex neurons). This allows these LSTMs to define both the neural forward computation as well as the learning algorithm that determines how the network is updated. Information flows forward and backward in the network through multi-dimensional messages $\overrightarrow{m}$ and $\overleftarrow{m}$, generalizing the dynamics of an NN trained using backpropagation.

With this interpretation, VSML can be seen as a generalization of learned learning rules [Bengio et al., 1991; Gregor, 2020; Randazzo et al., 2020] and Hebbian-like differentiable mechanisms or fast weight programmers (FWPs) more generally [Schmidhuber, 1992b, 1993a; Miconi et al., 2018, 2019] where RNNs replace explicit weight updates. The learned state updates in VSML (in-context learning) may then implement learned weight updates (FWPs). While more expressive, compared to a simple symbolic learning rule, these learned weight updates in the form of RNNs are computationally more expensive, which is something we discuss in more detail in Section 3.7.

In standard NNs, weights and activations have multiplicative interactions. For VSML RNNs to mimic such computation we require multiplicative interactions between parts of the state s. Fortunately, LSTMs already incorporate this through gating and can be directly used in place of RNNs.

Stacking VSML RNNs and feeding inputs To get a structure similar to one of the non-recurrent deep feed-forward architectures (FNNs), we stack multiple VSML RNNs where their states are untied and their parameters

are tied.[1] This is visualized with two layers in Figure 3.2 where the states $s^{(2)}$ of the second column of sub-RNNs are distinct from the first column $s^{(1)}$. The parameters $A^{(k)}$ and $B^{(k)}$ describing layer sizes can then be varied for each layer $k \in \{1, \ldots, K\}$ constrained by $A^{(k)} = B^{(k-1)}$. The updated Equation 3.3 with distinct layers k is given by $s_{ab}^{(k)} \leftarrow f_{\text{RNN}}(s_{ab}^{(k)}, \overrightarrow{m}_a^{(k)})$ where $\overrightarrow{m}_b^{(k+1)} := \sum_{a'} f_{\overrightarrow{m}}(s_{a'b}^{(k)})$ with $b \in \{1, \ldots, B^{(k)} = A^{(k+1)}\}$. To prevent information from flowing only forward in the network, we use an additional backward message

$$s_{ab}^{(k)} \leftarrow f_{RNN}(s_{ab}^{(k)}, \overrightarrow{m}_a^{(k)}, \overleftarrow{m}_b^{(k)}), \tag{3.7}$$

where $\overleftarrow{m}_a^{(k-1)} := \sum_{b'} f_{\overleftarrow{m}}(s_{ab'}^{(k)})$ with $a \in \{1, \ldots, A^{(k)} = B^{(k-1)}\}$ (visualized in Figure 3.3). The backward transformation is given by $f_{\overleftarrow{m}} : \mathbb{R}^N \to \mathbb{R}^{N''}$.

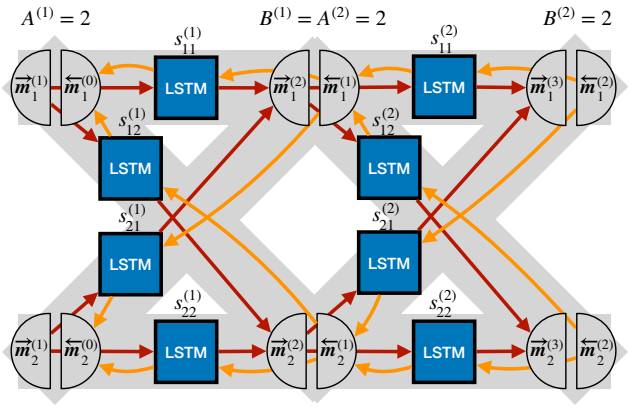


Figure 3.3: **A more detailed visualization of VSML.** It visualizes the forward messages $\overrightarrow{m}$ and backward messages $\overleftarrow{m}$ to form a two-layer NN with shared LSTM parameters but distinct states.

Similarly, other architectures can be explicitly constructed (e.g. convolutional NNs, Figure B.2.2). Some architectures may be learned implicitly if positional information is fed into each sub-RNN (Section B.3). We then update all states $s^{(k)}$ in sequence $1, \ldots, K$ to mimic sequential layer execution. We may also apply multiple RNN ticks for each layer k.

To provide the VSML RNN with data, each time we execute the operations of the first layer, a single new datum $x \in \mathbb{R}^{A(1)}$ (e.g. one flattened image) is distributed across all sub-RNNs. In our present experiments, we match the axis $A(1)$ to the input datum dimensionality such that each dimension (e.g., pixel) is fed to different RNNs. This corresponds to initializing the forward message $\overrightarrow{m}_{a1}^{(1)} := x_a$ (padding $\overrightarrow{m}$ with zeros). Similarly, we read the output $\hat{y} \in \mathbb{R}^{B(K)}$ from $\hat{y}_a := \overrightarrow{m}_{a1}^{(K+1)}$. Finally, we feed the error $e \in \mathbb{R}^{B(K)}$ at the output such that $\overleftarrow{m}_{b1}^{(K)} := e_b$. See Figure 3.2 for a visualization. Alternatively, multiple input or output dimensions could be patched together and fed into fewer sub-RNNs.

[1] The resultant architecture as a whole is still recurrent. Note that even standard FNNs are recurrent if the LA (backpropagation) is taken into account.

3.3.1 Meta-learning general-purpose learning algorithms from scratch

Having formalized VSML, we can now use end-to-end meta-learning to create LAs from scratch in Algorithm 8. We simply optimize the LSTM parameters θ to minimize the sum of prediction losses over many datapoints $(x, y) \in \{(x_1, y_1), \ldots, (x_T, y_T)\} \subset D$ starting with random states $V_L := \{s_{ab}^{(k)}\}$. We focus on meta-learning online LAs where one example is fed at a time as done in in-context RNNs [Hochreiter et al., 2001; Wang et al., 2016; Duan et al., 2016]. Meta training may be performed using end-to-end gradient descent or gradient-free optimization such as evolutionary strategies [Wierstra et al., 2008; Salimans et al., 2017]. The latter is significantly more efficient on VSML compared to standard NNs due to the small parameter space θ. Crucially, during meta-testing, no explicit gradient descent is used. In this phase, LSTM states are updated and messages are passed corresponding to the instructions in the blue box in Algorithm 8.

Algorithm 8 VSML: Meta Training

Require: Dataset(s) $D = \{(x_i, y_i)\}$
$\theta \leftarrow$ initialize LSTM parameters
while meta loss has not converged **do** ▷ Outer loop in parallel over datasets D
$\{s_{ab}^{(k)}\} \leftarrow$ initialize LSTM states $\quad \forall a, b, k$
for $(x, y) \in \{(x_1, y_1), \ldots, (x_T, y_T)\} \subset D$ **do** ▷ Inner loop over T examples
$\overrightarrow{m}_{a1}^{(1)} := x_a \quad \forall a$ ▷ Initialize from input image x
for $k \in \{1, \ldots, K\}$ **do** ▷ Iterating over K layers
$s_{ab}^{(k)} \leftarrow f_{RNN}(s_{ab}^{(k)}, \overrightarrow{m}_a^{(k)}, \overleftarrow{m}_b^{(k)}) \quad \forall a, b$ ▷ Equation 3.7
$\overrightarrow{m}_b^{(k+1)} := \sum_{a'} f_{\overrightarrow{m}}(s_{a'b}^{(k)}) \quad \forall b$ ▷ Create forward message
$\overleftarrow{m}_a^{(k-1)} := \sum_{b'} f_{\overleftarrow{m}}(s_{ab'}^{(k)}) \quad \forall a$ ▷ Create backward message
$y'_a := \overrightarrow{m}_{a1}^{(K+1)} \quad \forall a$ ▷ Read output
$e := \nabla_{y'} L(y', y)$ ▷ Compute error at outputs using loss L
$\overleftarrow{m}_{b1}^{(K)} := e_b \quad \forall b$ ▷ Input errors
$\theta \leftarrow \theta - \alpha \nabla_\theta \sum_{t=1}^{T} L(y'(t), y(t))$, obtaining ∇_θ either by
- back-propagation through the inner loop
- evolution strategies, using a search distribution $p(\theta)$

3.3.2 Learning to implement backpropagation in RNNs

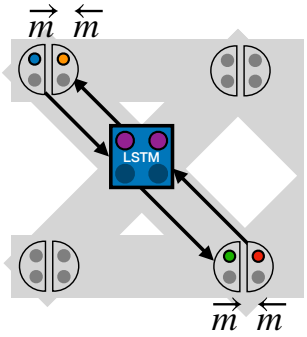


Figure 3.4: **VSML can implement backpropagation as a special case.** To do so, we optimize the VSML RNN to use and update weights w and biases b as part of the state s_{ab} in each sub-RNN. Inputs are pre-synaptic x and error e. Outputs are post-synaptic $\hat{y}$ and error $\hat{e}'$.

As an alternative to end-to-end meta-learning, we also demonstrated that VSML can implement backpropagation in its recurrent dynamics. Due to the algorithm's ubiquitous use, it seems desirable to be able to meta-learn backpropagation-like algorithms. Here we investigate how VSML can learn to implement backpropagation purely in its recurrent dynamics. We do this by optimizing θ to (1) store a weight w and bias b as a subset of each state s_{ab}, (2) compute $y = \tanh(x)w + b$ to implement neural forward computation, and (3) update w and b according to the backpropagation algorithm [Linnainmaa, 1970]. We call this process *learning algorithm cloning* and it is visualized in Figure 3.4.

We designate an element of each message $\overrightarrow{m}_a^{(k)}$, $\overleftarrow{m}_b^{(k)}$, $f_{\overrightarrow{m}}(s_{ab}^{(k)})$, $f_{\overleftarrow{m}}(s_{ab}^{(k)})$ as the input x, error e, and output $\hat{y}$ and error $\hat{e}'$. Similarly, we set $w := s_{ab1}$ and $b := s_{ab2}$. We then optimize θ via gradient descent to regress $\hat{y}$, Δw, Δb, and $\hat{e}'$ toward their respective targets. We can either generate the training dataset $D := \{(x, w, b, y, e, e')_i\}$ randomly or run a 'shadow' NN on some supervised problem and fit the VSML RNN to its activations and parameter updates. Multiple iterations in the VSML RNN would then correspond to evaluating the network and updating it via backpropagation. The activations from the forward pass necessary for credit assignment could be memorized as part of the state s or be explicitly stored and fed back. For simplicity, we chose the latter to clone backpropagation. We continuously run the VSML RNN forward, alternately running the layers in order $1, \ldots, K$ and in reverse order $K, \ldots, 1$.[2]

3.4 Experiments

[2] Executing layers in reverse order is not strictly necessary as information always also flows backwards through $\overleftarrow{m}$ but makes LA cloning easier.

First, we demonstrate the capabilities of the VSML RNN by showing that it can implement neural forward computation and backpropagation in its recurrent dynamics on the *MNIST* [LeCun et al., 2010] and *Fashion MNIST* [Xiao et al., 2017] dataset. Then, we show how we can meta-learn an LA from scratch on one set of datasets and then successfully apply it to another (out of distribution). Such generalization is enabled by extensive variable sharing where we have very few meta variables $|V_M| \approx 2,400$ (RNN parameters θ) and many learned variables $|V_L| \approx 257,200$. We also investigate the robustness of the discovered LA. Finally, we introspect the meta-learned LA and compare it to gradient descent.

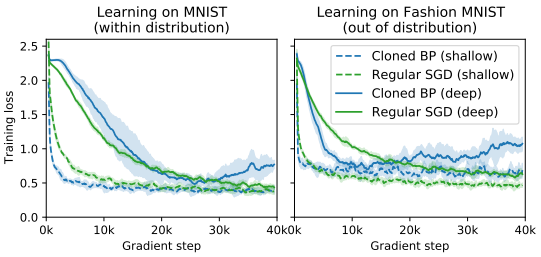


Figure 3.5: **Meta-testing learned backpropagation in VSML by running the LSTMs in forward-mode.** The VSML RNN is optimized to implement backpropagation in its recurrent dynamics on MNIST, then tested both on MNIST and Fashion MNIST where it performs learning purely by unrolling the LSTM. We test on shallow and deep architectures (single hidden layer of 32 units). Standard deviations are over 6 seeds.

Our implementation uses LSTMs and the message interpretation from Equation 3.7. Hyperparameters, training details, and additional experiments can be found in the appendix.

3.4.1 VSML RNNs can implement backpropagation

As described in Section 3.3.2, we optimize the VSML RNN to implement backpropagation. We structure the sub-RNNs to mimic a feed-forward NN with either one hidden layer or no hidden layers. To obtain training targets, we instantiate a standard NN, the shadow network, and feed it MNIST data. After cloning, we then run the LA encoded in the VSML RNN on the MNIST and Fashion MNIST dataset and observe that it performs learning purely in its recurrent dynamics, making explicit gradient calculations unnecessary. Figure 3.5 shows the learning curve on these two datasets. Notably, learning works both on MNIST (within distribution) and on Fashion MNIST (out of distribution). We observe that the loss is decently minimized, albeit regular gradient descent still outperforms our cloned backpropagation. This may be due to non-zero errors during learning algorithm cloning, in particular when these errors accumulate in

the deeper architecture. It is also possible that the VSML states ('weights') deviate too far from ranges seen during cloning, in particular in the deep case when the loss starts increasing. We obtain 87% (deep) and 90% (shallow) test accuracy on MNIST and 76% (deep) and 80% (shallow) on Fashion MNIST (focusing on successful cloning over performance).

3.4.2 Meta learning supervised learning from scratch

In the previous experiments, we have established that VSML is expressive enough to meta-optimize backpropagation-like algorithms. Instead of cloning an LA, we now meta-learn from scratch as described in Section 3.3.1. We use a single layer ($K = 1$) from input to output dimension and run it for two RNN ticks per image with $N = 16$ and $\overrightarrow{M} = \overleftarrow{M} = 8$. First, the VSML RNN is meta trained end-to-end using evolutionary strategies (ES) [Salimans et al., 2017] on MNIST to minimize the sum of cross-entropies over 500 data points starting from random state initializations. As each image is unique and θ can not memorize the data, we are implicitly optimizing the VSML RNN to generalize to future inputs given all inputs it has seen so far. We do not pre-train θ with a human-engineered LA.

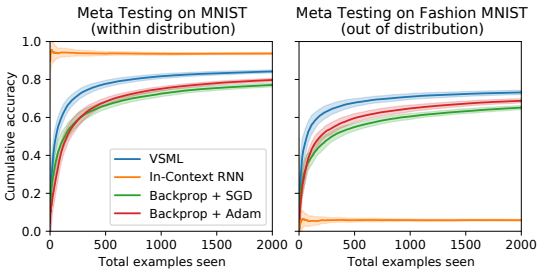


Figure 3.6: **Meta-learning novel generalizing learning algorithms with VSML.** The VSML RNN can be directly meta trained on MNIST to minimize the sum of errors when classifying online starting from a random state initialization. This allows for faster learning during meta-testing compared to online gradient descent with Adam on the same dataset and even generalizes to a different dataset (Fashion MNIST). In comparison, a standard in-context RNN [Hochreiter et al., 2001] strongly overfits in the same setting. Standard deviations are over 128 seeds.

During meta-testing on MNIST (Figure 3.6) we plot the cumulative accuracy on all previous inputs on the y axis ($\frac{1}{T}\sum_{t=1}^{T} c_t$ after example T with binary c_t indicating prediction correctness). For each example, the prediction when this example was fed to the RNN is used, thus measuring sample efficient learning. This evaluation protocol is similar to the one used in in-context RNNs [Wang et al., 2016; Duan et al., 2016]. We observe that learning is considerably faster compared to the baseline of online

gradient descent (no mini batching, the learning rate appropriately tuned). One possibility is that VSML simply overfits to the training distribution. We reject this possibility by meta-testing the same unmodified RNN on a different dataset, here Fashion MNIST. Learning still works well, meaning we have meta-learned a fairly general LA (although performance at convergence still slightly lacks behind). This generalization is achieved without using any hardcoded gradients during meta-testing purely by running the RNN forward. In comparison to VSML, an in-context RNN heavily overfits.

3.4.3 Robustness to varying inputs and outputs

A defining property of VSML is that the same parameters θ can be used to learn on varying input and output sizes. Further, the architecture and thus the meta-learned LA is invariant to the order of inputs and outputs. In this experiment, we investigate how robust we are to such changes. We meta train across MNIST with 3, 4, 6, and 7 classes. Likewise, we train across rescaled versions with 14x14, 28x28, and 32x32 pixels. We also randomly project all inputs using a linear transformation, with the transformation fixed for all inner learning steps. In Figure 3.7 we meta test on 6 configurations that were not seen during meta-training. Performance on all of these configurations is comparable to the unchanged reference from the previous section. In particular, the invariance to random projections suggests that we have meta-learned a learning algorithm beyond transferring learned representations [compare Finn et al., 2017; Triantafillou et al., 2020; Tseng et al., 2020].

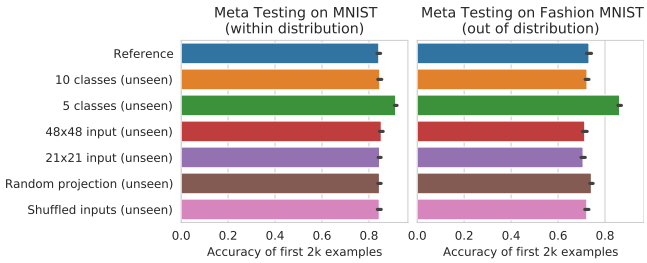


Figure 3.7: **VSML exhibits strong robustness to unseen tasks.** The meta-learned learning algorithm is robust to permutations and size changes in the inputs and outputs. All six configurations have not been seen during training and perform comparable to the unchanged reference. Standard deviations are over 32 seeds.

3.4.4 Varying datasets

To better understand how different meta-training distributions and meta test datasets affect VSML RNNs and our baselines, we present several different com-

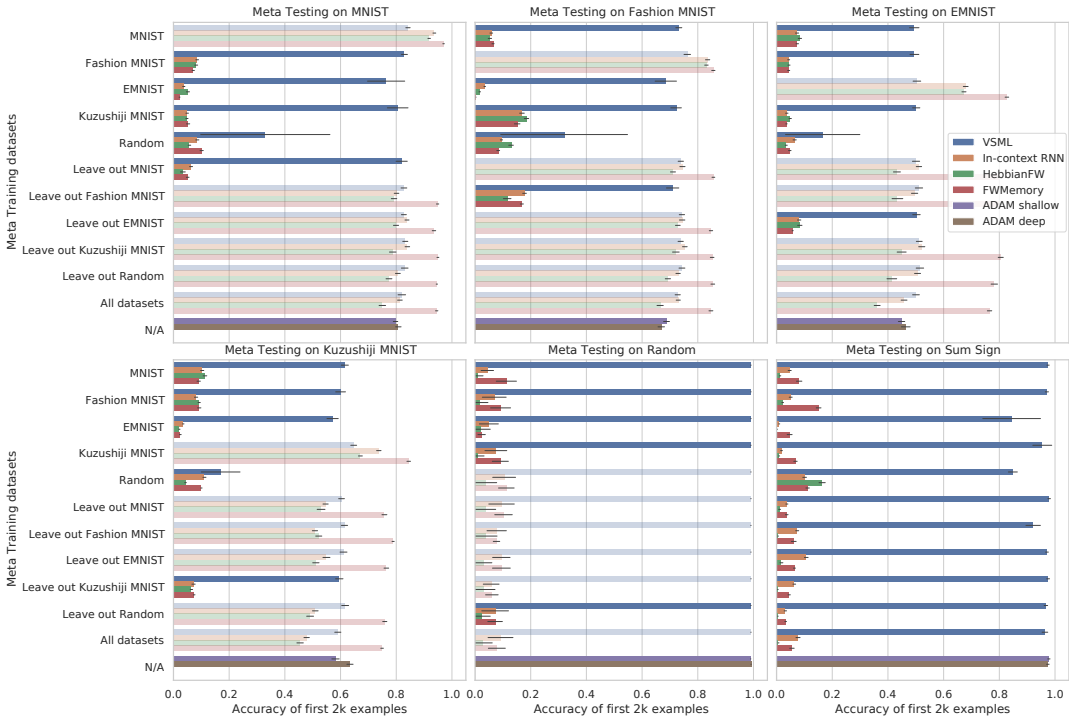


Figure 3.8: **An evaluation of VSML's online learning capabilities across various datasets.** Cumulative accuracy in % after having seen 2k training examples evaluated after each prediction starting with random states (VSML, in-context RNN, HebbianFW, FWMemory) or random parameters (SGD). Standard deviations are over 32 meta test training runs. Meta testing is done on the official test set of each dataset. Meta training is on subsets of datasets excluding the Sum Sign dataset. Unseen tasks, most relevant from a general-purpose LA perspective, are opaque.

binations in Figure 3.8. The opaque bars represent tasks that were not seen during meta-training and are thus most relevant for this analysis. This includes four additional datasets: (1) *Kuzushiji MNIST* [Clanuwat et al., 2018] with 10 classes, (2) *EMNIST* [Cohen et al., 2017] with 62 classes, (3) A randomly generated classification dataset (*Random*) with 20 data points that changes with each step in the outer loop, and (4) *Sum Sign* which generates random inputs and requires classifying the sign of the sum of all inputs. Meta training is done over 500 randomly drawn samples per outer iteration. Each algorithm is meta trained for 10k outer iterations. Inputs are randomly projected as in Section 3.4.3 (for VSML; the baselines did not benefit from these augmentations). We again report the cumulative accuracy on all data seen since the beginning of meta test training. We compare to SGD with a single layer, matching the architecture of VSML, and a hidden layer, matching the number of weights to the size of V_L in VSML. We also have included a Hebbian fast weight baseline [Miconi et al., 2018] and an external (fast weight) memory approach [Schlag et al., 2021b].

We observe that VSML generalizes much better than in-context RNNs, Hebbian fast weights, and the external memory. These baselines overfit to the training environments. Notably, VSML even generalizes to the unseen tasks *Random* and *Sum Sign* which have no shared structure with the other datasets. In many cases VSML's performance is similar to SGD but a little more sample efficient in the beginning of training (learning curves in Section B.2). This suggests that our meta-learned LAs are good at quickly associating new inputs with their labels. We further investigate this in the next Section 3.5.

3.5 Analysis

Given that VSML seems to learn faster than online gradient descent in many cases we would like to qualitatively investigate how learning differs. We first meta train on the full MNIST dataset as before. During meta-testing, we plot the output probabilities for each digit against the number of samples seen in Figure 3.9. We highlight the ground truth input class □ as well as the predicted class ◯. In this case, our meta test dataset consists of MNIST digits with two examples of each type. The same digit is always repeated twice. This allows us to observe and visualize the effect with only a few examples. We have done the same introspection with the full dataset in Section B.2.

We observe that in VSML almost all failed predictions are followed by the correct prediction with high certainty. In contrast, SGD makes many incorrect predic-

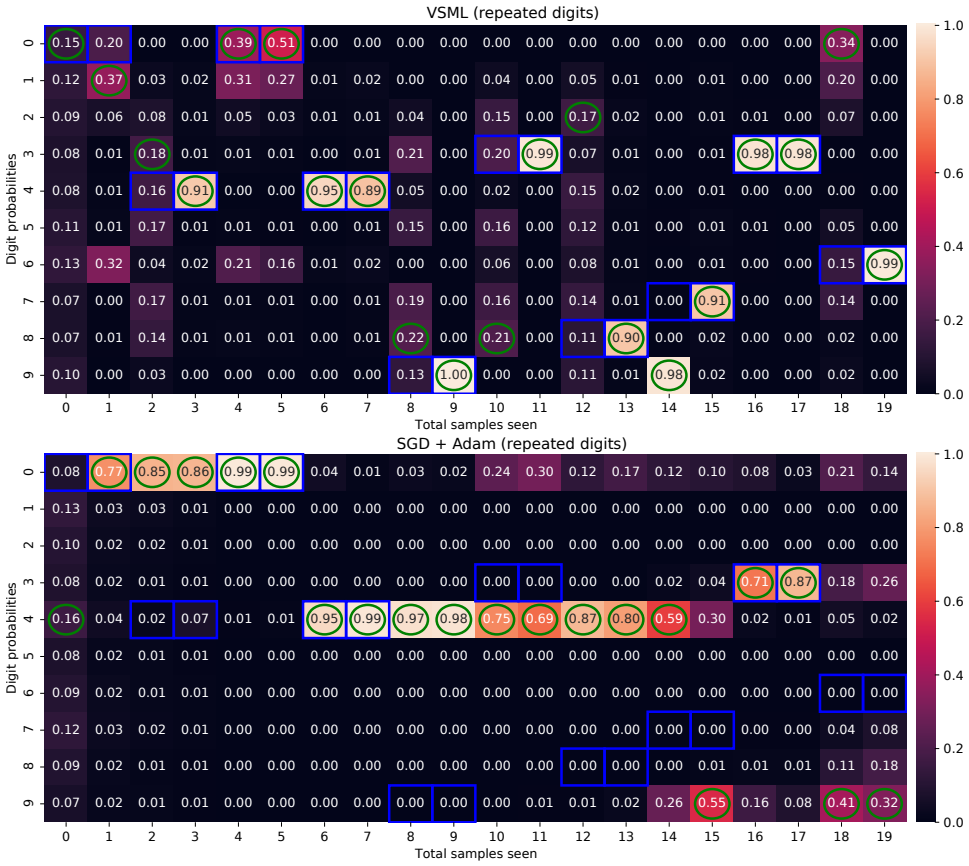


Figure 3.9: **An introspection of VSML's discovered learning algorithm.** We introspect how output probabilities change after observing an input and the prediction error when meta-testing on MNIST with two examples for each type. We highlight the ground truth class □ as well as the predicted class ◯. The top plot shows VSML quickly associating the input images with the right label, almost always making the right prediction the second time with high confidence. The bottom plot shows the same dataset processed by SGD with Adam which fails to learn quickly.

tions and fails to adapt correctly in only 20 steps. It seems that SGD learns qualitatively different from VSML. The VSML RNN meta-learns to quickly associate new inputs with their class whereas SGD fails to do so. We tried several different SGD learning rates and considered multiple steps on the same input. In both cases, SGD does not behave similar to VSML, either learning much slower or forgetting previous examples. As evident from the high accuracies in Figure 3.8, VSML does not only memorize inputs using this strategy of fast association, but the associations generalize to future unseen inputs.

3.6 Related work

Memory based meta-learning (in-context RNNs) Standard RNNs [Hochreiter et al., 2001; Duan et al., 2016; Wang et al., 2016] can implement a simple in-context learning algorithm (see Section 3.2). Unfortunately, the LA encoded in the RNN parameters is largely over-parameterized, which leads to overfitting. VSML demonstrates that weight sharing can address this issue resulting in more general-purpose LAs.

Learned Learning Rules / Fast Weights NNs that generate or change the weights of another or the same NN are known as fast weight programmers [Schmidhuber, 1992b], hypernetworks [Ha et al., 2017], NNs with synaptic plasticity [Miconi et al., 2018] or learned learning rules [Bengio et al., 1991] (see Section 3.2). In VSML we do not require explicit architectures for weight updates, as weights are emergent from RNN state updates. In addition to the learning rule, we implicitly learn how the neural forward computation is defined. In succession to this work, fast weights have also been used to meta-learn more general LAs [Sandler et al., 2021].

Learned gradient-based optimizers Meta-learning has been used to find optimizers that update the parameters of a model by taking the loss and gradient with respect to these parameters as input [Ravi and Larochelle, 2017; Andrychowicz et al., 2016; Li and Malik, 2017; Metz et al., 2020a]. In this work, we are interested in meta-learning that does not rely on fixed gradient calculation in the inner loop.

Discrete program search An interesting alternative to distributed variable updates in VSML is meta-learning via discrete program search [Schmidhuber, 1994b; Real et al., 2020]. In this paradigm, a separate programming language needs to be defined that gives rise to neural computation when its instructions are combined. This led to the automated rediscovery of backpropagation [Real

et al., 2020]. In VSML, we demonstrate that a symbolic programming language is not required, and general-purpose LAs can be discovered and encoded in variable-shared RNNs. Search over neural network parameters is usually easier compared to symbolic program search due to smoothness in the loss landscape.

Multi-agent systems In the reinforcement learning setting, multiple agents can be modeled with shared parameters [Sims, Karl, 1994; Pathak et al., 2019; Huang et al., 2020], also in the context of meta-learning [Rosa et al., 2019a]. This is related to the sharing of variables in VSML, depending on how the boundary between the agent and the environment is drawn. Unlike these works, we demonstrate the advantage of variable sharing in meta-learning more general-purpose LAs and present a weight update interpretation.

3.7 Discussion and limitations

The research community has perfected the art of leveraging backpropagation for learning for a long time. At the same time, there are open questions such as how to minimize memory requirements, learn effectively online and continuously, learn sample efficiently, learn without separate backward phases, and others. VSML suggests that instead of building on top of backpropagation as a fixed routine, meta-learning offers an alternative to discover general-purpose LAs. Nevertheless, this work is only a proof of concept—until now we have only investigated small-scale problems and performance does not yet quite match the mini-batched setting with large quantities of data. In particular, we observed premature convergence of the solution at meta-test time which calls for further investigations. Scaling our system to harder problems and larger meta-task distributions will be important future work.

The computational cost of the current VSML variant is also higher than that of standard backpropagation. If we run a sub-RNN for each weight in a standard NN with W weights, the cost is in $O(WN^2)$, where N is the state size of a sub-RNN. If N is small enough and our experiments suggest that small N may be feasible, this may be an acceptable cost. As N is not expected to grow with the problem size (as opposed to W), this may be negligible for large problems. Furthermore, VSML is not bound to the interpretation of a sub-RNN as a single weight. Future work may relax this particular choice. The total complexity of an iteration of meta-training with evolution strategies is $O(N_P TWN^2)$ where N_P is the size of the population and T is the number of examples VSML is learning

from. It further requires space of $O(WN+N^2)$ to store all the forward activations at the current time step WN and meta-parameters N^2. For a gradient-based meta-optimizer N_P corresponds to the batch size, which is usually smaller. The space requirements increase due to storing all forward activations across time $O(TWN+N^2)$. Meta-testing has a runtime complexity of $O(TWN^2)$ and space complexity of $O(WN+N^2)$.

Meta-optimization is also prone to local minima. In particular, when the number of ticks between input and feedback increases (e.g. deeper architectures), credit assignment becomes harder. Early experiments suggest that diverse meta-task distributions can help mitigate these issues. Additionally, learning horizons are limited when using backprop-based meta-optimization. Using ES allowed for training across longer horizons and more stable optimization.

VSML can also be viewed as regularizing the NN weights that encode the LA through a representational bottleneck. It is conceivable that LA generalization as obtained by VSML can also be achieved through other regularization techniques. Unlike most regularizers, VSML also introduces substantial reuse of the same learning principle and permutation invariance through variable sharing.

3.8 Conclusion

We introduced Variable Shared Meta Learning (VSML), a simple principle of weight sharing and sparsity for meta-learning powerful learning algorithms (LAs). Our implementation replaces the weights of a neural network with tiny LSTMs that share parameters. We discuss connections to in-context learning, fast weight generators (hyper networks), and learned learning rules.

Using *learning algorithm cloning*, VSML RNNs can learn to implement the backpropagation algorithm and its parameter updates encoded implicitly in the recurrent dynamics. On MNIST it learns to predict well without any human-designed explicit computational graph for gradient calculation.

VSML can meta-learn from scratch supervised LAs that do not explicitly rely on gradient computation and that *generalize to unseen datasets*. Introspection reveals that VSML LAs learn by fast association in a way that is qualitatively different from stochastic gradient descent. This leads to gains in sample efficiency. Future work will focus on reinforcement learning settings, improvements of meta-learning, larger task distributions, and learning over longer horizons.

3.9 Follow-up work

Since the publication of VSML, in-context learning in Transformers has become hugely popular. In particular, research has demonstrated that Transformers can implement increasingly general learning algorithms [Garg et al., 2022; Kirsch et al., 2022b; Müller et al., 2022; Hollmann et al., 2022] such as gradient descent [Von Oswald et al., 2023; Akyürek et al., 2022]. Works in RL have shown that learning algorithm cloning can also be used to distill and speed up existing RL algorithms into Transformers [Laskin et al., 2022]. Symmetries in neural networks modifying other neural networks have been further explored as 'neural functionals' [Navon et al., 2023; Zhou et al., 2024; Herrmann et al., 2024].

Chapter 4

SymLA: Introducing symmetries to in-context reinforcement learning

Keywords *in-context learning, reinforcement learning*
Article *Kirsch et al. [2022a] (preprint 2021)*

In the next work, we extend VSML (Chapter 3) to the reinforcement learning setting. We refer to such in-context reinforcement learning agents as *symmetric learning agents* (SymLA).

4.1 Introduction

Meta reinforcement learning (RL) attempts to discover new RL algorithms automatically from environment interaction. In so-called in-context learning (or black-box) approaches, the policy and the learning algorithm are jointly represented by a single neural network. These methods are very flexible, but they tend to underperform compared to human-engineered RL algorithms in terms of generalisation to new, unseen environments. In this work, we explore the role of symmetries in meta-generalisation. We show that our previously introduced meta-RL approach (Chapter 2) that meta-learns an objective for backpropagation-based learning exhibits certain symmetries (specifically the reuse of the learning rule, and invariance to input and output permutations) that are not present in typical in-context meta-RL systems. We hypothesise that these symmetries can play an important role in meta-generalisation. Building off VSML (Chapter 3), we develop an in-context meta-RL system that exhibits these same symmetries. We show through careful experimentation that incorporating these symmetries

can lead to algorithms with a greater ability to generalise to unseen action & observation spaces, tasks, and environments.

Recent work in meta reinforcement learning (RL) has begun to tackle the challenging problem of automatically discovering general-purpose RL algorithms [Kirsch et al., 2020b; Alet et al., 2020; Oh et al., 2020]. These methods learn to reinforcement learn by optimizing for earned reward over the lifetimes of many agents in multiple environments. If the discovered learning principles are sufficiently general-purpose, then the learned algorithms should generalise to significantly different unseen environments. Depending on the structure of the learned algorithm, these methods can be partitioned into backpropagation-based methods, which learn to use the backpropagation algorithm to reinforcement learn, and in-context learning (black-box-based) methods, in which a single (typically recurrent) neural network jointly specifies the agent and RL algorithm [Wang et al., 2016; Duan et al., 2016]. While backpropagation-based methods are more prevalent due to their relative ease of implementation and theoretical guarantees, in-context learning methods are expressive and have the potential to avoid some of the issues with backpropagation-based optimization, such as higher memory requirements, catastrophic forgetting, and differentiability.

Unfortunately, in-context learning methods have not yet been successful at discovering general-purpose RL algorithms that compete with the generality of human-engineered algorithms. In this work, we show that in-context methods exploit fewer symmetries than backpropagation-based methods. We hypothesise that introducing more symmetries to in-context meta-learners can improve their generalisation capabilities. We test this hypothesis by introducing a number of symmetries into an existing in-context meta-learning algorithm, including (1) the use of the same learned learning rule across all nodes of the neural network (NN), (2) the flexibility to work with any input, output, and architecture sizes, and (3) invariance to permutations of the inputs and outputs (for dense layers). Permutation invariance implies that for any permutation of inputs and outputs the learning algorithm produces the same policy. As we show, this is similar to dense NNs trained with backpropagation that also exhibit permutation invariance. We refer to such agents as *symmetric learning agents* (SymLA).

To introduce these symmetries, we build on variable shared meta-learning (VSML Chapter 3), which we adapt to the RL setting. VSML arranges multiple RNNs like weights in a NN and performs message passing between these RNNs. We then perform meta training and meta testing similar to in-context

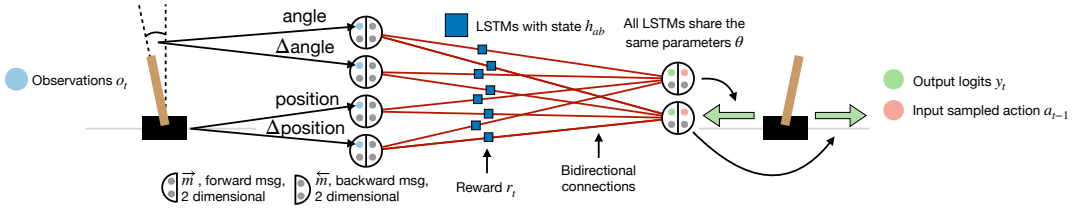


Figure 4.1: **The architecture for the proposed *symmetric learning agents* (SymLA)** that we use to investigate black-box learning algorithm with symmetries. Weights in a neural network are replaced with small parameter-shared RNNs. Activations in the original network correspond to messages passed between RNNs, both in the forward $\overrightarrow{m}$ and backward $\overleftarrow{m}$ direction in the network. These messages may contain external information such as the environment observation, previously taken actions, and rewards from the environment.

learning RNNs (ICL RNNs), also known as RL^2 [Wang et al., 2016; Duan et al., 2016]. We experimentally validate SymLA on bandits, classic control, and grid worlds, comparing generalisation capabilities to in-context RNNs. SymLA improves generalisation when varying action dimensions, permuting observations and actions, and significantly changing tasks and environments.

4.2 Preliminaries

4.2.1 Reinforcement Learning

The RL setting in this work follows the standard (PO)MDP formulation. At each time step, $t = 1, 2, \ldots$ the agent receives a new observation $o_t \in \mathcal{O}$ generated from the environment state $s_t \in \mathcal{S}$ and then takes an action $a_t \in \mathcal{A}$ sampled from its (recurrent) policy $a_t \sim \pi_\theta(\cdot|o_{1:t}, a_{1:t-1})$. The agent receives a reward $r_t \in \mathcal{R} \subset \mathbb{R}$ and the next state via $(s_{t+1}, r_t) \sim e(\cdot|s_t, a_t)$. The initial environment state s_1 is drawn from the initial state distribution $s_1 \sim p_{\text{init}}(\cdot)$. The goal is to find the optimal policy parameters θ^* that maximise the expected return $R = \mathbb{E}[\sum_{t=1}^{T} \gamma^t r_t]$ where T is the episode length, and $0 < \gamma \leq 1$ is a discount factor ($T = \infty$, $\gamma < 1$ for non-episodic MDPs).

4.2.2 Meta Reinforcement Learning

The meta reinforcement learning setting is concerned with discovering novel agents that learn throughout their multi-episode lifetime ($L \geq T$) by making

use of rewards r_t to update their behavior. This can be formulated as maximizing $\arg\max_\theta \mathbb{E}_{e\sim p(e)}[\mathbb{E}_{a_t\sim\pi_\theta, r_t\sim e}[\sum_{t=1}^{L}\gamma^t r_t]]$ where $p(e)$ is a distribution of meta-training environments. The objective itself is similar to a multi-task setting. In this work, we discuss how the structure of the agent influences the degree to which it *learns* and *generalises* in novel tasks and environments. We seek to discover *general-purpose* learning algorithms that generalise outside the meta-training distribution.

We can think of an agent that learns throughout its lifetime as a history-dependent map $a_t, h_t = f(h_{t-1}, o_t, r_{t-1}, a_{t-1})$ that produces an action a_t and new agent state h_t given its previous state h_{t-1}, an observation o_t, environment reward r_{t-1}, and previous action a_{t-1}. In the case of backpropagation-based learning, f is decomposed into: (1) a *stationary* policy $\pi_\theta^{(s)}$ that maps the current state into an action, $a_t = \pi_\theta^{(s)}(o_t)$; and (2) a backpropagation-based update rule that optimizes a given objective J by propagating the error signal backwards and updating the policy in fixed intervals (e.g. after each episode). In its simplest form, for any dense layer $k \in \{1, \dots, K\}$ of a NN policy with size $A^{(k)} \times B^{(k)}$, inputs $x^{(k)}$, outputs $x^{(k+1)}$, and weights $w^{(k)} \subset \theta$, the backpropagation update rule is given by

$$x_b^{(k+1)} = \sum_a x_a^{(k)} w_{ab}^{(k)} \qquad \text{(forward pass)} \tag{4.1}$$

$$\delta_a^{(k-1)} = \sum_b \delta_b^{(k)} w_{ab}^{(k)} \qquad \text{(backward pass)} \tag{4.2}$$

$$\Delta w_{ab}^{(k)} = -\alpha \frac{\partial J}{\partial w_{ab}^{(k)}} = -\alpha x_a^{(k)} \delta_b^{(k)} \qquad \text{(update)} \tag{4.3}$$

where $a \in \{1, \dots, A^{(k)}\}$, $b \in \{1, \dots, B^{(k)}\}$, α is the learning rate, δ are error terms, and the agent state h corresponds to parameters θ. The initial error is given by the gradient at the NN outputs, $\delta^{(k)} = \frac{\partial J}{\partial x^{(K+1)}}$. Transformations such as non-linearities are omitted here. Works in meta-reinforcement learning that take this approach parameterise the objective J_ϕ and meta-learn its parameters [Kirsch et al., 2020b; Oh et al., 2020].

In contrast, in-context meta RL [Duan et al., 2016; Wang et al., 2016] meta-learns f directly in the form of a single *non-stationary* policy π_θ with memory. Parameters of f represent the learning algorithm (no explicit J_ϕ) while the state h represents the policy. In the simplest form of an RNN representation of f, given a current hidden state h and inputs o, r, a (concatenated $[\cdot]$), updates to

the policy take the form

$$a_b, h_b \leftarrow f_\theta(h, o, r, a)_b = \sigma(\sum_a [h, o, r, a]_a v_{ab}), \tag{4.4}$$

with parameters $\theta = v$ and activation function σ, omitting the bias term. We refer to this as the in-context RNN or MetaRNN. The inputs must include, beyond the observation o, the previous reward r and action a, so that the meta-learner can learn to associate past actions with rewards [Schmidhuber, 1993b; Wang et al., 2016]. Further, black-box systems do not reset the state h between episode boundaries, so that the learning algorithm can accumulate knowledge throughout the agent's lifetime.

4.3 Symmetries in meta-RL

In this section, we demonstrate how the learning dynamics in backpropagation-based systems (Equation 4.3) differ from the learning dynamics in black-box systems (Equation 4.4), and how this affects the generalisation of black-box methods to novel environments.

4.3.1 Symmetries in backpropagation-based meta-RL

We first identify symmetries that backpropagation-based systems exhibit and discuss how they affect the generalisability of the learned learning algorithms.

1. **Symmetric learning rule.** In Equation 4.3, each parameter w_{ab} is updated by the same update rule based on information from the forward and backward pass. Meta-learning an objective J_ϕ affects the updates of each parameter symmetrically through backpropagation.

2. **Flexible input, output, and architecture sizes.** Because the same rule is applied everywhere, the learning algorithm can be applied to arbitrarily sized neural networks, including variations in input and output sizes. This involves varying A and B and the number of layers, affecting how often the learning rule is applied and how many parameters are being learned.

3. **Invariance to input and output permutations.** Given a permutation of inputs and outputs in a layer, defined by the bijections $\rho : \mathbb{N} \to \mathbb{N}$ and $\rho' : \mathbb{N} \to \mathbb{N}$, the learning rule is applied as $x^{(k+1)}_{\rho'(b)} = \sum_a x^{(k)}_{\rho(a)} w^{(k)}_{ab}$, $\delta^{(k-1)}_{\rho(a)} = \sum_b \delta^{(k)}_{\rho'(b)} w^{(k)}_{ab}$, and $\Delta w^{(k)}_{ab} = -\alpha x^{(k)}_{\rho(a)} \delta^{(k)}_{\rho'(b)}$. Let w' be a weight matrix with

$w'^{(k)}_{\rho(a)\rho'(b)} = w^{(k)}_{a,b}$, then we can equivalently write $x^{(k+1)}_{\rho'(b)} = \sum_a x^{(k)}_{\rho(a)} w'^{(k)}_{\rho(a)\rho'(b)}$, $\delta^{(k-1)}_{\rho(a)} = \sum_b \delta^{(k)}_{\rho'(b)} w'^{(k)}_{\rho(a)\rho'(b)}$, and $\Delta w'^{(k)}_{\rho(a)\rho'(b)} = -\alpha x^{(k)}_{\rho(a)} \delta^{(k)}_{\rho'(b)}$. If all elements of $w'^{(k)}$ are initialized i.i.d., we can interchangeably use w in place of w' in the above updates. By doing so, we recover the original learning rule equations for any a, b. Thus, the learning algorithm is invariant to input and output permutations.

While backpropagation has inherent symmetries, these symmetries would be violated if the objective function J_ϕ would be asymmetric. Formally, when permuting the NN outputs $y = x^{(K+1)}$ such that $y'_b = y_{\rho'(b)}$, J_ϕ should satisfy that the gradient under the permutation is also a permutation

$$\frac{\partial J_\phi(y')}{\partial y'_b} = \left[\frac{\partial J_\phi(y)}{\partial y}\right]_{\rho'(b)} \tag{4.5}$$

where the environment accepts the action permuted by ρ' in the case of $J_\phi(y')$. This is the case for policy gradients, for instance, if the action selection $\pi(a|s)$ is permuted according to ρ'. When meta-learning objective functions, prior work carefully designed the objective function J_ϕ to be symmetric. In MetaGenRL [Kirsch et al., 2020b], taken actions were processed element-wise with the policy outputs and sum-reduced by the loss function. In LPG [Oh et al., 2020], taken actions and policy outputs were not directly fed to J_ϕ, but instead only the log probability of the action distribution was used.

4.3.2 Insufficient symmetries in black-box meta-RL

Black-box meta-learning methods are appealing as they require few hard-coded biases and are flexible enough to represent a wide range of possible learning algorithms. We hypothesize that this comes at the cost of the tendency to overfit to the given meta training environment(s) resulting in overly specialized learning algorithms.

Learning dynamics in backpropagation-based systems (Equation 4.3) differ significantly from learning dynamics in black-box systems (Equation 4.4). In particular, meta-learning J_ϕ is significantly more constrained, since J_ϕ can only indirectly affect each policy parameter $w^{(k)}_{ab}$ through the *same* learning rule from Equation 4.3. In contrast, in black-box systems (Equation 4.4), each policy state h_b is directly controlled by *unique* meta-parameters (vector $v_{\cdot b}$), thereby encouraging the black-box meta-learner to construct specific update rules for each element of the policy state. This results in sensitivity to permutations in inputs and out-

puts. Furthermore, input and output spaces must retain the same size as those are directly dependent on the number of RNN parameters.

As an example, consider a meta-training distribution of two-armed bandits where the expected payout of the first arm is much larger than the second. If we meta-train an in-context RNN on these environments then when meta-testing the in-context RNN will have learned to immediately increase the probability of pulling the first arm, independent of any observed rewards. If instead the action probability is adapted using REINFORCE or a meta-learned symmetric objective function then, due to the implicit symmetries, the learning algorithm could not differentiate between the two arms to favor one over the other. While the in-context RNN behavior is optimal when meta-testing on the same meta-training distribution, it completely fails to generalise to other distributions. Thus, the in-context RNN results in a non-learning, biased solution, whereas the backpropagation-based approach results in a learning solution. In the former case, the learning algorithm is overfitted to only produce a fixed policy that always samples the first arm. In the latter case, the learning algorithm is unbiased and will learn a policy from observed rewards to sample the first arm. Beyond bandits, for reasonably sized meta-training distributions, we may have any number of biases in the data that an in-context RNN will inherit, impeding generalisation to unseen tasks and environments.

4.4 Adding symmetries to black-box meta-RL

A solution to the illustrated over-fitting problem with black-box methods is the introduction of symmetries into the parameterisation of the policy. This can be achieved by generalising the forward pass (Equation 4.1), backward pass (Equation 4.2), and element-wise update (Equation 4.3) to parameterized versions. We further subsume the loss computation into these parameterized update rules. Together, they form a single recurrent policy with additional symmetries. Prior work on variable shared meta-learning (VSML Chapter 3) [Kirsch and Schmidhuber, 2021] used similar principles to meta-learn supervised learning algorithms. In the following, we extend their approach to deal with the RL setting.

4.4.1 Variable Shared Meta Learning

We now recap VSML from Chapter 3. VSML describes neural architectures for meta-learning with parameter sharing. This can be motivated by meta-learning how to update weights [Bengio et al., 1992; Schmidhuber, 1993a] where the up-

date rule is shared across the network. Instead of designing a meta network that defines the weight updates explicitly, we arrange small parameter-shared RNNs (LSTMs) like weights in a NN and perform message passing between those.

In VSML, each weight w_{ab} with $w \in \mathbb{R}^{A \times B}$ in a NN is replaced by a small RNN with parameters θ and hidden state $h_{ab} \in \mathbb{R}^N$. We restrict ourselves to dense NN layers here, where w corresponds to the weights of that layer with input size A and output size B. This can be adapted to other architectures such as CNNs if necessary. All these RNNs share the same parameters θ, defining both what information propagates in the neural network, as well as how states are updated to implement learning. Each RNN with state h_{ab} receives the analogue to the previous activation, here called the vectorized forward message $\overrightarrow{m}_a \in \mathbb{R}^{\overrightarrow{M}}$, and the backward message $\overleftarrow{m}_b \in \mathbb{R}^{\overleftarrow{M}}$ for information flowing backwards in the network (asynchronously). The backward message may contain information relevant to credit assignment, but is not constrained to this. The RNN update equation (compare Equation 4.3 and 4.4) is then given by

$$h_{ab}^{(k)} \leftarrow f_{\mathrm{RNN}}(h_{ab}^{(k)}, \overrightarrow{m}_a^{(k)}, \overleftarrow{m}_b^{(k)}) \tag{4.6}$$

for layer k where $k \in \{1, \dots, K\}$ and $a \in \{1, \dots, A^{(k)}\}, b \in \{1, \dots, B^{(k)}\}$. Similarly, new forward messages are created by transforming the RNN states using a function $f_{\overrightarrow{m}} : \mathbb{R}^N \to \mathbb{R}^{\overrightarrow{M}}$ (compare Equation 4.1) such that

$$\overrightarrow{m}_b^{(k+1)} = \sum_a f_{\overrightarrow{m}}(h_{ab}^{(k)}) \tag{4.7}$$

defines the new forward message for layer $k+1$ with $b \in \{1, \dots, B^{(k)} = A^{(k+1)}\}$. The backward message is given by $f_{\overleftarrow{m}} : \mathbb{R}^N \to \mathbb{R}^{\overleftarrow{M}}$ (compare Equation 4.2) such that

$$\overleftarrow{m}_a^{(k-1)} = \sum_b f_{\overleftarrow{m}}(h_{ab}^{(k)}) \tag{4.8}$$

and $a \in \{1, \dots, A^{(k)} = B^{(k-1)}\}$. For simplicity, we use θ below to denote all of the VSML parameters, including those of the RNN and forward and backward message functions.

In the following, we derive a black-box meta reinforcement learner based on VSML (visualized in Figure 4.1).

4.4.2 RL agent inputs and outputs

At each time step in the environment, the agent's inputs consist of the previously taken action a_{t-1}, current observation o_t and previous reward r_{t-1}. We feed r_{t-1}

as an additional input to each RNN, the observation $o_t \in \mathbb{R}^{A^{(1)}}$ to the first layer ($\overrightarrow{m}_{.1}^{(1)} := o_t$), and the action $a_{t-1} \in \{0,1\}^{B^{(K)}}$ (one-hot encoded) to the last layer ($\overleftarrow{m}_{.1}^{(K)} := a_{t-1}$). The index 1 refers to the first dimension of the $\overrightarrow{M}$ or $\overleftarrow{M}$-dimensional message. We interpret the agent's output message $y = \overrightarrow{m}_{.1}^{(K+1)}$ as the unnormalized logits of a categorical distribution over actions. While we focus on discrete actions only in our present experiments, this can be adapted for probabilistic or deterministic continuous control.

4.4.3 Architecture recurrence and reward signal

Instead of using multiple layers ($K > 1$), in this work we use a single layer ($K = 1$). In Equation 4.6, RNNs in the same layer can not coordinate directly as their messages are only passed to the next and previous layer. To give that single layer sufficient expressivity, we make it 'recurrent' by processing the layer's own messages $\overrightarrow{m}_b^{(k+1)}$ and $\overleftarrow{m}_a^{(k-1)}$. We empirically found that this architectural choice enabled faster convergence during training compared to $K > 1$. The network thus has two levels of recurrence: (1) Each RNN that corresponds to a weight of a standard NN and (2) messages that are generated according to Equation 4.7 and 4.8 and fed back into the same layer. Furthermore, each RNN receives the current reward signal r_{t-1} as input. The update equation is given by

$$h_{ab}^{(k)} \leftarrow f_{\text{RNN}}(h_{ab}^{(k)}, \underbrace{\overrightarrow{m}_a^{(k)}, \overleftarrow{m}_b^{(k)}, r_{t-1}}_{\text{environment inputs}}, \underbrace{\overrightarrow{m}_b^{(k+1)}, \overleftarrow{m}_a^{(k-1)}}_{\text{from previous step}}) \tag{4.9}$$

where $a \in \{1, \ldots, A^{(k)}\}, b \in \{1, \ldots, B^{(k)}\}$. As we only use a single layer, $k = 1$, we apply the update multiple times (multiple micro ticks) for each step in the environment. This can also be viewed as multiple layers with shared parameters, where parameters correspond to states h. For pseudo code, see Algorithm 14 in the appendix.

4.4.4 Symmetries in SymLA

By incorporating the above changes to inputs, outputs, and architecture, we arrive at a black-box meta RL method with symmetries, here represented by our proposed *symmetric learning agents* (SymLA). By construction, SymLA exhibits the same symmetries as those described in Section 4.3.1, despite not using the backpropagation algorithm.

1. **Symmetric learning rule.** The learning rule as defined by Equation 4.9 is replicated across $a \in \{1, \ldots, A\}$ and $b \in \{1, \ldots, B\}$ with the same parameter θ.

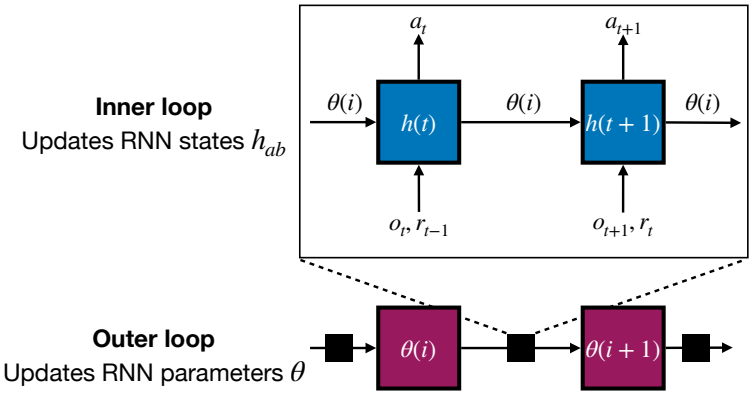


Figure 4.2: **A visualization of SymLA's inner and outer loop.** In SymLA, the inner loop recurrently updates all RNN states $h_{ab}(t)$ for agent steps $t \in \{1, \dots, L\}$ starting with randomly initialized states h_{ab}. Based on feedback r_t, RNN states can be used as memory for learning. The learning algorithm encoded in the RNN parameters θ is updated in the outer loop by meta-training using ES.

2. **Flexible input, output, and architecture sizes.** Changes in A, B, and K correspond to input, output, and architecture size. This does not affect the number of meta-parameters and therefore these quantities can also be varied at meta-test time.

3. **Invariance to input and output permutations.** When permuting messages using bijections ρ and ρ', the state update becomes $h_{ab}^{(k)} \leftarrow f_{\text{RNN}}(h_{ab}^{(k)}, \overrightarrow{m}_{\rho(a)}^{(k)}, \overleftarrow{m}_{\rho'(b)}^{(k)}, r_{t-1}, \overrightarrow{m}_{\rho'(b)}^{(k+1)}, \overleftarrow{m}_{\rho(a)}^{(k-1)})$, and the message transformations are $\overrightarrow{m}_{\rho'(b)}^{(k+1)} = \sum_a f_{\overrightarrow{m}}(h_{ab}^{(k)})$ and $\overleftarrow{m}_{\rho(a)}^{(k-1)} = \sum_b f_{\overleftarrow{m}}(h_{ab}^{(k)})$. Similar to backpropagation, when RNN states h_{ab} are initialized i.i.d., we can use $h_{\rho(a),\rho'(b)}$ in place of h_{ab} to recover the original Equations 4.7, 4.8, 4.9.

4.4.5 Learning / Inner loop

Learning corresponds to updating RNN states h_{ab} (see Figure 4.2). This is the same as the in-context RNN [Wang et al., 2016; Duan et al., 2016] but with a more structured neural model. For fixed RNN parameters θ which encode the learning algorithm, we randomly initialize all states h_{ab}. Next, the agent steps through the environment, updating h_{ab} in each step. If the environment is episodic with T steps, the agent is run for a lifetime of $L \geq T$ steps with environment resets in-between, carrying the agent state h_{ab} over.

4.4.6 Meta-Learning / Outer loop

Each outer loop step unrolls the inner loop for L environment steps to update θ. The SymLA objective is to maximize the agent's lifetime sum of rewards, i.e. $\sum_{t=1}^{L} r_t(\theta)$. We optimize this objective using evolutionary strategies [Wierstra et al., 2008; Salimans et al., 2017] by following the gradient

$$\nabla_\theta \mathbb{E}_{\phi\sim\mathcal{N}(\phi|\theta,\Sigma)}[\mathbb{E}_{e\sim p(e)}[\sum_{t=1}^{L} r_t^{(e)}(\phi)]]. \tag{4.10}$$

with some fixed diagonal covariance matrix Σ and environments $e \sim p(e)$. We chose evolution strategies due to its ability to optimize over long inner-loop horizons without memory constraints that occur due to backpropagation-based meta optimization. Furthermore, it was shown that meta-loss landscapes are difficult to navigate and the search distribution helps in smoothing those [Metz et al., 2019b]. The computational cost of meta-training and meta-testing in SymLA is equivalent to the one described in VSML (Chapter 3).

4.5 Experiments

Equipped with a symmetric black-box learner, we now investigate how its learning properties differ from a standard in-context RNN. Firstly, we learn to learn on bandits from Wang et al. [2016] where the meta-training environments are similar to the meta-test environments. Secondly, we demonstrate generalisation to unseen action spaces, applying the learned algorithm to bandits with varying numbers of arms at meta-test time—something that in-context RNNs are not capable of. Thirdly, we demonstrate how symmetries improve generalisation to unseen observation spaces by creating permutations of observations and actions in classic control benchmarks. Fourthly, we show how permutation invariance leads to generalisation to unseen tasks by learning about states and their associated rewards at meta-test time. Finally, we demonstrate how symmetries result in better learning algorithms for unseen environments, generalising from a grid world to CartPole. Hyper-parameters are in Appendix C.2.

4.5.1 Learning to learn on similar environments

We first compare SymLA and the in-context RNN on the two-armed (dependent) bandit experiments from Wang et al. [2016] where there is no large variation in the meta-test environments. These consist of five different settings of varying difficulty that we use for meta-training and meta-testing (see Appendix C.1). There

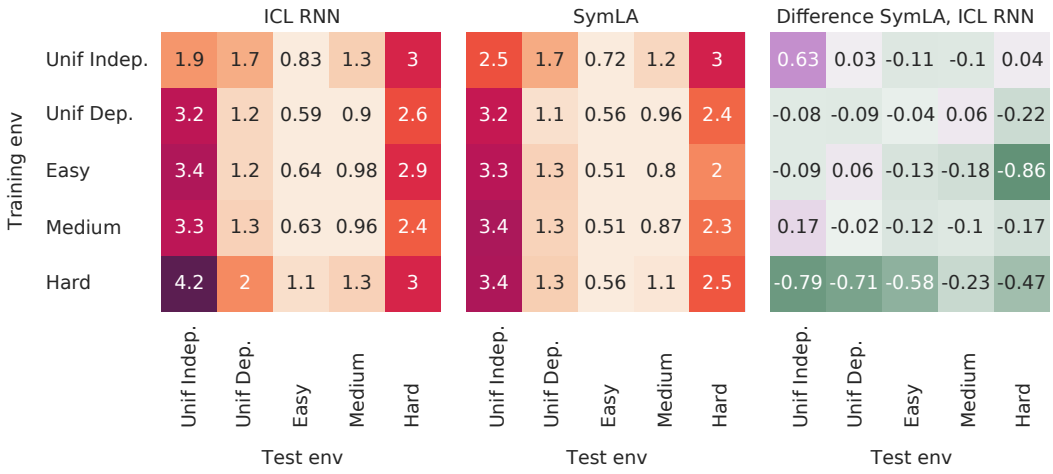


Figure 4.3: **SymLA is competitive to in-context RNNs on bandit tasks.** We compare SymLA to a standard in-context RNN on a set of bandit benchmarks from Wang et al. [2016]. We train (y-axis) and test (x-axis) on two-armed bandits of varying difficulties. We report expected cumulative regret across 3 meta-training and 100 meta-testing runs with 100 arm-pulls (smaller is better). We observe that SymLA tends to perform comparably to the in-context RNN.

are no observations (no context), only two arms, and a meta-training distribution where each arm has the same marginal distribution of payouts. Thus, we expect the symmetries from SymLA to have no significant effect on performance. We meta-train for an agent lifetime of $L = 100$ arm-pulls and report the expected cumulative regret at meta-test time in Figure 4.3. We meta-train on each of the five settings, and meta-test across all settings. The performance of the in-context RNN reproduces the average performance of Wang et al. [2016], here trained with ES instead of A2C. When using symmetries (as in SymLA), we recover a similar performance compared to the in-context RNN.

4.5.2 Generalisation to unseen action spaces

In contrast to the in-context RNN, in SymLA we can vary the number of arms at meta-test time. The architecture of SymLA allows to change the network size arbitrarily by replicating existing RNNs, thus adding or removing arms at meta-test time while retaining the same meta-parameters from meta-training. In Figure 4.4 we train on different numbers of arms and test on seen and unseen configurations. All arms are independently drawn from the uniform distribution $p_i \sim U[0, 1]$. We observe that SymLA works well within-distribution (diagonal)

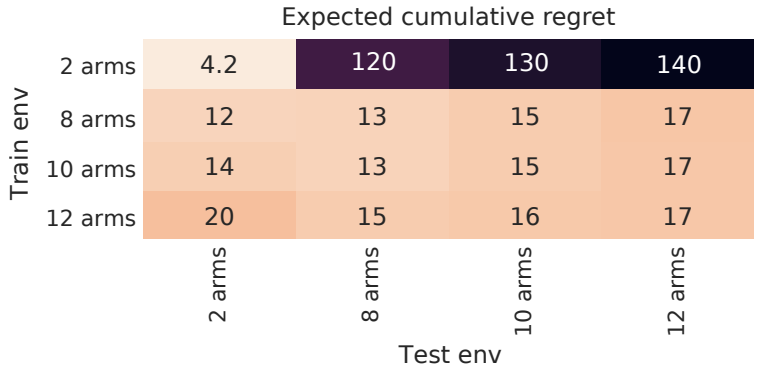


Figure 4.4: **SymLA generalizes to unseen numbers of arms by adding or removing LSTMs at meta-test time.** We meta-train and meta-test SymLA on varying numbers of independent arms to measure generalisation performance on unseen configurations. We do this by adding or removing RNNs to accommodate the additional output units. The number of meta-parameters remains constant. We report expected cumulative regret across 3 meta-training and 100 meta-testing runs with 100 arm-pulls (smaller is better). Particularly relevant are the out-of-distribution scenarios (off-diagonal).

and generalises to unseen numbers of arms (off-diagonal). We also observe that for two arms a more specialized solution can be discovered, impeding generalisation when only training on this configuration.

4.5.3 Generalisation to unseen observation spaces

In the next experiments we want to specifically analyze the permutation invariance created by our architecture. In the previous bandit environments, actions occurred in all permutations in the training distribution. In contrast, RL environments usually have some structure to their observations and actions. For example in CartPole the first observation is usually the pole angle and the first action describes moving to the left. Human-engineered learning algorithms are usually invariant to permutations and thus generalise to new problems with different structure. The same should apply for our black-box agent with symmetries.

We demonstrate this property in the classic control tasks *CartPole*, *Acrobot*, and *MountainCar*. We meta-train on each environment respectively with the original observation and action order. We then meta-test on either (1) the same configuration or (2) across a permuted version. The results are visualized in

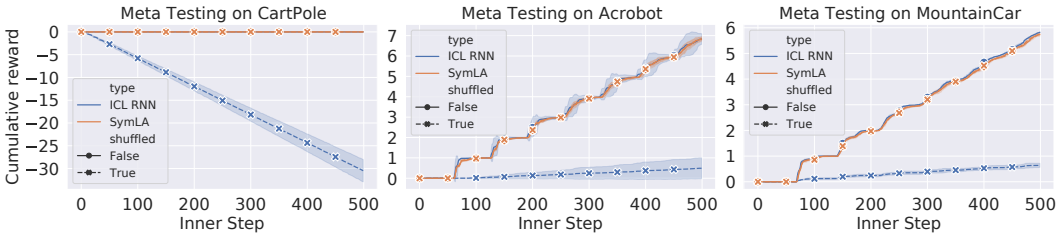


Figure 4.5: **SymLA's architecture is inherently permutation invariant.** When meta-training on standard CartPole, Acrobot, and MountainCar, the performance of the in-context RNN and SymLA are comparable. We then meta-test with shuffled observations and actions. In this setting, SymLA still performs well as it has meta-learned to identify observations and actions at meta-test time. In contrast, the in-context RNN fails to do so. Standard deviations are over 3 meta-training and 100 meta-testing runs.

Figure 4.5. Due to the built-in symmetries, the performance does not degrade in the shuffled setting. Instead, our method quickly learns about the ordering of the relevant observations and actions at meta-test time. In comparison, the in-context RNN baseline fails on the permuted setting where it was not trained on, indicating over-specialization. Thus, symmetries help to generalise to observation permutations that were not encountered during meta training.

4.5.4 Generalisation to unseen tasks

The permutation invariance has further reaching consequences. It extends to learning about tasks at meta-test time. This enables generalisation to unseen tasks. We construct a grid world environment (see Figure 4.6) with two object types: A trap and a heart. The agent and the two objects (one of each type) are randomly positioned every episode. Collecting the heart gives a reward of $+1$, whereas the trap gives -1. All other rewards are zero. The agent observes its own position and the position of both objects. The observation is constructed as an image with binary channels for the position and each object type.

When meta-training on this environment, at meta-test time we observe in Figure 4.6 that the in-context RNN learns to directly collect hearts in each episode throughout its lifetime. This is due to having overfitted to the association of hearts with positive rewards. In comparison, SymLA starts with near-zero rewards and learns through interactions which actions need to be taken when receiving particular observations to collect the heart instead of the trap. With suf-

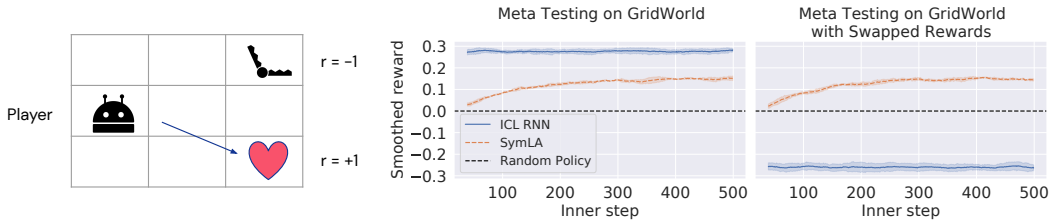


Figure 4.6: **We extend the permutation invariant property to concepts** - varying the rewards associated with different object types (+1 and -1) in a grid world environment (left). SymLA is forced to learn about the rewards of object types at meta-test time (starting at near zero reward and increasing the reward intake over time). When switching the rewards and running the same learner, the in-context RNN collects the wrong rewards, whereas SymLA still infers the correct relationships. Standard deviations are over 3 meta-training and 100 meta-testing runs.

ficient environment interactions L we would expect SymLA, if it implemented a general-purpose RL algorithm, to eventually (after sufficient learning) match the average reward per time of the in-context RNN in the non-shuffled grid world. Next, we swap the rewards of the trap and heart, i.e. the trap now gives a positive reward, whereas the heart gives a negative reward. This is equivalent to swapping the input channels corresponding to the heart and trap. We observe that SymLA still generalises, learning at meta-test time about observations and their associated rewards. In contrast, the in-context RNN now collects the wrong item, receiving negative rewards. These results show that black-box meta RL with symmetries discovers a more general update rule that is less specific to the training tasks than typical in-context RNNs.

4.5.5 Generalisation to unseen environments

We have demonstrated how permutation invariance can lead to increased generalisation. But can SymLA also generalise between entirely different environments? We show-case how meta-training on a grid world environment allows generalisation to CartPole. To simplify credit-assignment, we use a dense-reward grid world where the reward is proportional to the change in distance toward a target position. Both the target position, as well as the agent position are randomized. The agent observes its own position, all obstacles, and the target position as a binary image with multiple channels. In the CartPole environment the agent is rewarded for being as upright and centered as possible [Tun-

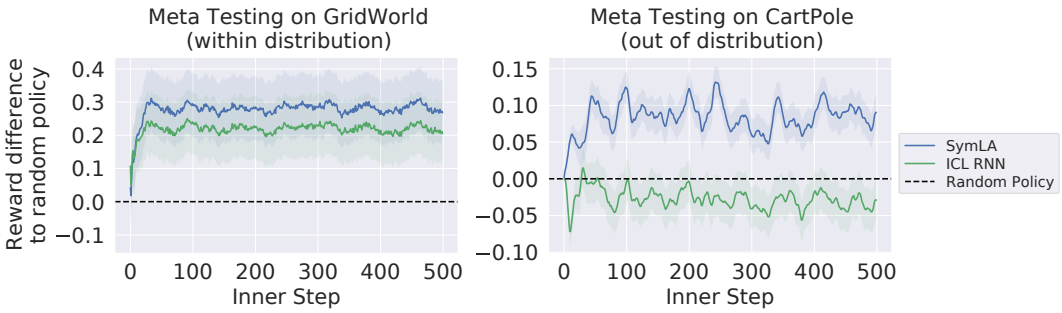


Figure 4.7: **Generalisation capabilities of SymLA from GridWorld to CartPole.** We meta-train the learning algorithm on GridWorld. We then meta-test on GridWorld and CartPole and report standard error of the mean and mean rewards (100 seeds) relative to a random policy - this highlights the learning process. While SymLA generalises from GridWorld to CartPole, the in-context RNN does not.

yasuvunakool et al., 2020]. Further, during meta-training, we randomly project observations linearly for each lifetime. This is necessary as in the grid world environment all observations are binary whereas the CartPole environment has continuously varying observations. This mismatch would inhibit generalisation. In Figure 4.7 we demonstrate that meta-training with SymLA only on the GridWorld environment allows reusing the same meta-learned learning algorithm to the CartPole environment. In contrast, the in-context RNN does not exhibit such generalisation. This suggests that meta-learning with symmetries has the potential to produce learning algorithms that generalize between significantly different environments.

4.6 Related work

Black-Box Meta RL Black-box meta RL can be implemented by policies that receive the reward signal as input [Schmidhuber, 1993b] and use memory to learn, such as recurrence in RNNs [Hochreiter et al., 2001; Wang et al., 2016; Duan et al., 2016]. These approaches do not feature the symmetries discussed in this work which leads to a tendency of overfitting.

Learned Learning Rules & Fast Weights In the supervised and reinforcement learning contexts, learned learning rules [Bengio et al., 1992] or fast weights [Schmidhuber, 1992b, 1993a; Miconi et al., 2018; Schlag et al., 2021b;

Najarro and Risi, 2020] describe (meta-)learned mechanisms (slow weights) that update fast weights to implement learning. This often involves outer-products and can be generalised to black-box meta-learning with parameter sharing [Kirsch and Schmidhuber, 2021]. None of these approaches feature all of the symmetries we discuss above to meta learn RL algorithms.

Backpropagation-based Meta RL Alternatives to black-box meta RL include learning a weight initialization and adapting it with a human-engineered RL algorithm [Finn et al., 2017], warping computed gradients [Flennerhag et al., 2020], meta-learning hyper-parameters [Sutton, 1992; Xu et al., 2018] or meta-learning objective functions corresponding to the learning algorithm [Houthooft et al., 2018; Kirsch et al., 2020b; Xu et al., 2020; Oh et al., 2020; Bechtle et al., 2021].

Neural Network Symmetries Symmetries in neural networks have mainly been investigated to reflect the structure of the input data. This includes applications of convolutions [Fukushima, 1979], deep sets [Zaheer et al., 2017], graph neural networks [Wu et al., 2020], geometric deep learning [Bronstein et al., 2017], or meta-learning symmetries [Zhou et al., 2021]. In contrast, our work focuses on the structure and symmetries of learning algorithms. While many meta-learning algorithms exhibit symmetries [Bengio et al., 1992], in particular backpropagation-based meta-learning [Andrychowicz et al., 2016; Finn et al., 2017; Flennerhag et al., 2020; Kirsch et al., 2020b], the effects of these symmetries have not been discussed in detail. In this work, we provide such a discussion and experimental investigation in the context of meta-RL.

4.7 Conclusion

In this work, we identified symmetries that exist in backpropagation-based methods for meta RL but are missing from black-box methods. We hypothesized that these symmetries lead to better generalisation of the resulting learning algorithms. To test this, we extended a black-box meta-learning method [Kirsch and Schmidhuber, 2021] that exhibits these same symmetries to the meta RL setting. This resulted in SymLA, a flexible black-box meta RL algorithm that is less prone to over-fitting compared to in-context RNNs. We demonstrated generalisation to varying numbers of arms in bandit experiments (unseen action spaces), permuted observations and actions with no degradation in performance (unseen observation spaces), and observed the tendency of the meta-learned RL algo-

rithm to learn about states and their associated rewards at meta-test time (unseen tasks). Finally, we showed that the discovered learning behavior also transfers between grid world and (unseen) classic control environments to some extent. This generalization is still quite limited and warrants future investigation with larger meta-training distributions and neural networks of higher capacity.

4.8 Follow-up work

Generalisation in in-context meta-RL has gained significant more interest since the publication of SymLA. For instance, Tang and Ha [2021] established a relationship of VSML and SymLA-like systems to the attention mechanism in Transformers and demonstrating the benefits of permutation invariance in Reinforcement Learning. Other works scaled up the environment distributions to increase generalization in meta-RL [Lu et al., 2023; Team et al., 2023; Raparthy et al., 2024], a topic we will further discuss in the next Chapter 5 and Chapter 6.

Chapter 5

GPICL: How and when general-purpose in-context learning emerges in transformers

Keywords *in-context learning, transformers, fast weight programmers, emergence, algorithmic transitions*
Article *Kirsch et al. [2022b]*

In the previous two works (Chapter 3 & 4), we have shown that LSTMs can learn in-context both in supervised and reinforcement learning. The generalization capabilities were aided by parameter-sharing between multiple LSTM instantiations. Can other neural architectures, such as Transformers, also be trained to perform generalizable in-context learning? In this chapter, we show that with the right data distribution and training regime, Transformers and other black-box models can be meta-trained to act as general-purpose in-context learners.

5.1 Introduction

Modern machine learning requires system designers to specify aspects of the learning pipeline, such as losses, architectures, and optimizers. Meta-learning is the process of automatically *discovering new learning algorithms* instead of designing them manually [Schmidhuber, 1987]. An important quality of human-engineered learning algorithms, such as backpropagation and gradient descent, is their applicability to a wide range of tasks or environments. For learning-to-learn to exceed those capabilities, the meta-learned learning algorithms must be

similarily *general-purpose*. Recently, there has been significant progress toward this goal [Kirsch et al., 2020b; Oh et al., 2020]. The improved generality of the discovered learning algorithms has been achieved by introducing inductive bias, such as by bottlenecking the architecture or by hiding information, which encourage learning over memorization. Methods include restricting learning rules to use gradients [Metz et al., 2019b; Kirsch et al., 2020b; Oh et al., 2020], symbolic graphs [Real et al., 2020; Co-Reyes et al., 2021], or parameter sharing [Kirsch and Schmidhuber, 2021; Kirsch et al., 2022a].

While enabling generalization, these inductive biases come at the cost of increasing the effort to design these systems and potentially restrict the space of discoverable learning algorithms. Instead, we seek to explore general-purpose meta-learning systems with *minimal inductive bias*. Good candidates for this are black-box sequence-models as meta-learners such as LSTMs [Hochreiter et al., 2001; Wang et al., 2016; Duan et al., 2016] or (linear) Transformers [Schmidhuber, 1992b; Vaswani et al., 2017]. These memory-based or in-context learners take in training data and produce test-set predictions without any explicit definition of an inference model, training loss, or optimization algorithm. With recent advances of in-context learning in large language models [Brown et al., 2020], neural networks can already learn many concepts from demonstrations. What are the necessary conditions such that those models can learn from a wide range of demonstrations? To what extent can we elicit in-context learning that generalizes to a wider range of problems, in a similar way how learning via backpropagation and gradient descent can generalize?

In this work, we investigate how such in-context meta-learners can be trained to (meta-)generalize and learn on significantly different datasets than used during meta-training. For this we propose a Transformer-based *General-Purpose In-Context Learner* (GPICL) which is described with an associated meta-training task distribution in Section 5.3. In Section 5.4.1 we characterize algorithmic transitions—induced by scaling the number of tasks or the model size used for meta-training—between memorization, task identification, and general learning-to-learn. We further show in Section 5.4.2 that the capabilities of meta-trained algorithms are bottlenecked by their accessible state size (memory) determining the next prediction (such as the hidden state size in a recurrent network), unlike standard models which are thought to be bottlenecked by parameter count. Finally, in Section 5.4.3, we propose practical interventions that improve the meta-training of general purpose learning algorithms. Additional related work can be found in Section 5.5.

5.2 Background

What is a (supervised) learning algorithm? In this work, we focus on the setting of meta-learning supervised in-context learning algorithms. Consider a mapping

$$\left(\{x_i, y_i\}_{i=1}^{N_D}, x'\right) \mapsto y' \tag{5.1}$$

from the training (support) set $D = \{x_i, y_i\}_{i=1}^{N_D}$ and a query input x' to the query's prediction y' where $x_i, x' \in \mathbb{R}^{N_x}$, $y_i, y' \in \mathbb{R}^{N_y}$ and $N_D, N_x, N_y \in \mathbb{N}^+$. The subset of these functions that qualify as learning algorithms are those that improve their predictions y' given an increasingly larger training set D. Meta-learning then corresponds to finding these functions via meta-optimization. As in other black-box meta-learning models, we use a neural network to represent such functions. Such in-context learning is different from inner gradient-based meta-learning (such as MAML [Finn et al., 2017]) in that no explicit gradients are computed at meta-test time. All required mechanisms for learning are implicitly encoded in the black-box neural network.

What is a general-purpose learning algorithm? A learning algorithm can be considered general-purpose if it learns on a wide range of possible tasks D and their respective related queries x', y'. In this work, we are interested in strong generalization across entirely different datasets such as MNIST, Fashion MNIST, and CIFAR10. Human-engineered learning algorithms such as gradient-descent on a suitable loss function can be considered general-purpose learning algorithms that can be applied to any of these datasets (where the gradient is obtained via backpropagation or other means). Meta-learners often don't generalize that well at meta-test time when we have an entirely new dataset that we want to learn on. We set out to investigate under which conditions in-context learning generalizes well. In comparison to in-context learning, gradient-based methods like MAML hard-code the human-

Learning	Generalizing	Algorithm Description
✗	✗	Task memorization
✓	✗	Task identification
✗	✓	Zero-shot generalization
✓	✓	General-purpose learning algorithm

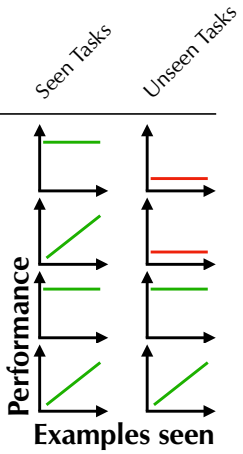


Table 5.1: An algorithm encoded in a neural network can be classified along two different dimensions: To what extent it *learns* and to what extent it *generalizes*.

engineered learning algorithm of gradient descent and inherit its generalization properties.

5.3 General-Purpose In-Context Learning

Due to the small number of inductive biases in black-box models, we can only expect (meta-)generalization when meta-training with an appropriately broad data distribution. Thus, changes in the data distribution affect whether and how a model meta-learns and meta-generalizes. We classify algorithms along two different dimensions: To what extent it learns (improving predictions given increasingly larger training sets provided at inference time), and to what extent it generalizes (performs well on instances, tasks, or datasets not seen before). Algorithms can then be categorized as in Table 5.1. In task memorization, the model immediately performs well on seen tasks but does not generalize. In task identification, the model identifies the task and gets better on it at inference time as it sees more examples but can only do so on tasks very similar to what it was trained on. In zero-shot generalization, the model immediately generalizes to unseen tasks, without observing examples. Finally, a general-purpose learning algorithm improves as it observes more examples both on seen and significantly different unseen tasks. We demonstrate algorithmic transitions occurring between these learning modalities, and empirically investigate these.

5.3.1 Generating tasks for learning-to-learn

Neural networks are known to require datasets of significant size to effectively generalize. While in standard supervised learning large quantities of data are common, meta-learning algorithms may require a similar number of distinct *tasks* in order to learn and generalize. Unfortunately, the number of commonly available tasks is orders of magnitudes smaller compared to the datapoints in each task.

Previous work has side-stepped this issue with architectural or algorithmic structure built into the learning algorithm, in effect drastically reducing the number of tasks required. For example, in Kirsch and Schmidhuber [2021]; Kirsch et al. [2022a], the authors included symmetries into the black-box model in the form of input and output permutation invariances. An alternative to this is the generation of new tasks [Schmidhuber, 2013; Clune, 2019; Such et al., 2020; Parker-Holder et al., 2022a]. Unfortunately, it is not easy to generate a wide range of tasks that are both diverse and contain structure as found in the real world.

Algorithm 9 Meta-Training for General-Purpose In-Context Learning (GPICL) via Augmentation

Require: Dataset $\bar{D} = \{\bar{x}_i, \bar{y}_i\}$, Number of tasks $K \in \mathbb{N}^+$

Define $p(D)$ by augmenting $\bar{D}$, here by:

$\{A_{ij}^{(k)}\}_{k=1}^{K} \sim \mathcal{N}(0, \frac{1}{N_x})$ ▷ Sample input projections

$\{\rho^{(k)}\}_{k=1}^{K} \sim p(\rho)$ ▷ Sample output permutations

$D^{(k)} = \{A^{(k)}\bar{x}_i, \rho^{(k)}(\bar{y}_i)\}$

$p(D) := \text{Uniform}[\{D^{(k)}\}_{k=1}^{K}]$

Meta-Training on $p(D)$

while not converged **do**

$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$ ▷ Equation 5.2

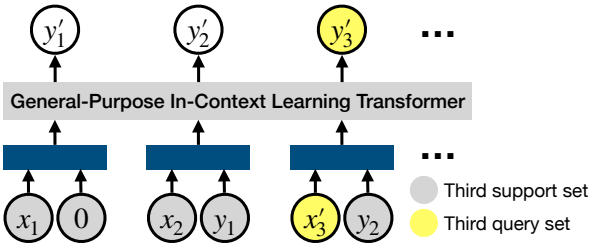


Figure 5.1: Our *General-Purpose In-Context Learner* (GPICL) is based on the vanilla Transformer which is trained to make predictions for queries x' given any prefix of a dataset $D := \{x_i, y_i\}_{i=1}^{N_D}$ as in Equation 5.2.

In this work, we take an intermediate step by augmenting existing datasets, in effect increasing the breadth of the task distribution based on existing task regularities. We generate a large number of tasks by taking existing supervised learning datasets, randomly projecting their inputs and permuting their classes (e.g. all inputs of class 3 are now class 1 and vice versa). While the random projection removes spatial structure from the inputs, this structure is not believed to be central to the task (for instance, the performance of SGD-trained fully connected networks is invariant to projection by a random orthogonal matrix [Wadia et al., 2021], see Chapter 4). Task augmentation allows us to investigate fundamental questions about learning-to-learn in the regime of many tasks without relying on huge amounts of existing tasks or elaborate schemes to generate those.

A task or dataset D is then defined by its corresponding base dataset $\bar{D} = \{\bar{x}_i, \bar{y}_i\}$, (linear) projection $A \in \mathbb{R}^{N_x \times N_x}$, with $A_{ij} \sim \mathcal{N}\left(0, \frac{1}{N_x}\right)$, and output permutation ρ, $D = \{A\bar{x}_i, \rho(\bar{y}_i)\}$. Unless noted otherwise, the distribution over output permutations $p(\rho)$ is uniform.

5.3.2 Meta-learning and meta-testing

Meta-learning Given those generated tasks, we then meta-train jointly on a mini-batch sampled from the whole distribution. First, we sample datasets D from the augmented task distribution $p(D)$ and then take a random batch $D_{1:N_D}$ from the training set. Second, we minimize $J(\theta)$, the sum of losses on the query prediction after observing any prefix $D_{1:j-1}$

$$J(\theta) = \mathbb{E}_{D\sim p(D)}\left[\sum_{j=1}^{N_D} l(f_\theta(D_{1:j-1}, x_j), y_j)\right], \tag{5.2}$$

where in the classification setting, l is the cross entropy loss between the label y_j and prediction $y' = f_\theta(D_{1:j-1}, x_j)$, f_θ is a neural network mapping to predictions y' as in Equation 5.1. During meta-training, we take gradient steps in $J(\theta)$ by backpropagation and Adam [Kingma and Ba, 2014]. To investigate the effect of the data distribution, we train on various numbers of tasks (Algorithm 9). Finally, we need to choose a black-box model for the function f_θ. We use a vanilla Transformer [Vaswani et al., 2017] with learned positional embeddings, visualized in Figure 5.1. We call it the *General-Purpose In-Context Learner* (GPICL). Each token corresponds to the concatenation of a transformed input x_i and one-hot encoded label y_{i-1}. The model predicts the corresponding logits $y' = y_i$ for the current input $x' = x_i$. When querying for the first x_1, no label for the previous input is available, so we feed a zero vector.

Meta-testing At meta-test time, no gradient-based learning is used. Instead, we simply obtain a prediction y' by evaluating the neural network f_θ on a dataset D and query point x'. The dataset D is either derived from the same base dataset (eg MNIST after meta-training on MNIST) or it is derived from a different dataset (eg Fashion MNIST or CIFAR10). In both cases a seen or unseen random projection is used. Datapoints are taken only from the respective test split of the base dataset.

5.4 Experiments on the emergence of general learning-to-learn

Multi-task training with standard classifiers Given a task distribution of many different classification tasks, we first ask under what conditions we expect "learning-to-learn" to emerge. We train a single model across many tasks where

each task corresponds to a random transformation of the MNIST dataset, but where the MLP only receives a single datapoint instead of a whole sequence as input. This corresponds to $N_D = 1$ in Equation 5.2. We would expect such a non-sequential classifier to be able to correctly predict for more tasks as its number of parameters increases. When plotting the network capacity against the number of tasks, we indeed observe a linear boundary where an increasing number of tasks can be fit the larger the network (Figure 5.2a). This is consistent with results in Collins et al. [2016], which found that a constant number of bits about the data distribution can be stored per model parameter, across a variety of model architectures and scales.

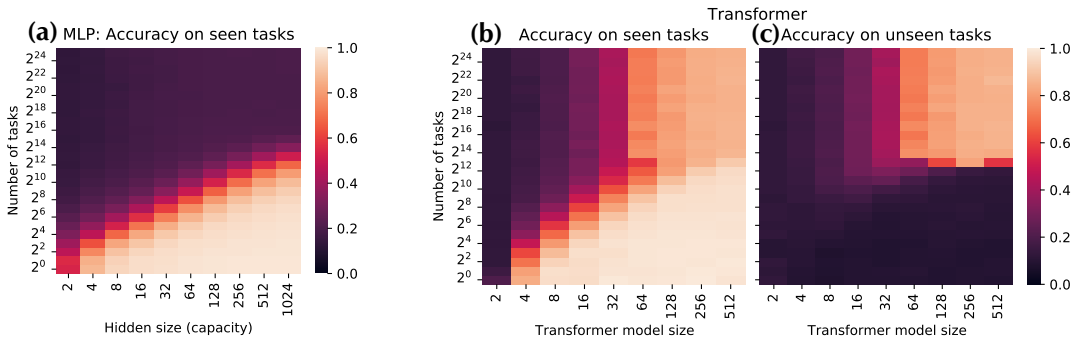


Figure 5.2: **GPICL is able to generalize to unseen tasks.** Each cell is a separate meta-training run. **(a)** An MLP classifier trained in a multi-task fashion across various numbers of tasks (generated based on MNIST) and network sizes is able to fit linearly more tasks, the larger its capacity. **(b)** A sequence model (here the GPICL Transformer) that observes a dataset D of inputs and labels transitions into generalizing to an seemingly unbounded number of tasks with an increase in model size. This is achieved by switching from a memorization solution to a learning solution that **(c)** generalizes to unseen tasks. This generalization does not occur with the MLP.

Learning-to-learn with large sequential models and data In contrast to the MLP classifier, a sequence model that observes multiple observations and their labels from the same task, could exceed that linear performance improvement by learning at inference time. Indeed, we observe that when switching to a Transformer that can observe a sequence of datapoints before making a prediction about the query, more tasks can be simultaneously fit (Figure 5.2b). At a certain model size and number of tasks, the model undergoes a transition, allowing to generalize to a seemingly unbounded number of tasks. We hypothesize that this is due to switching the prediction strategy from memorization to learning-to-

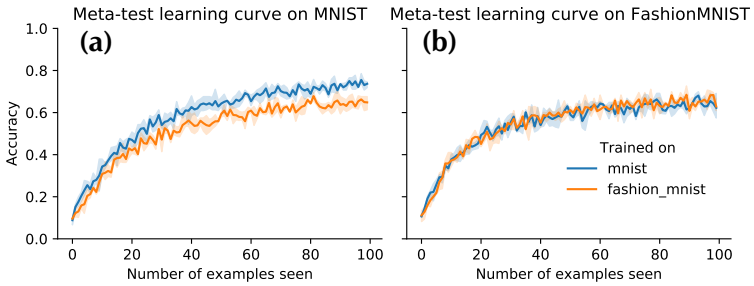


Figure 5.3: **GPICL learns from examples at test time, and generalizes to unseen tasks and datasets.** We meta-trained the Transformer on a set of tasks defined by random transformations of either MNIST (blue) or FashionMNIST (orange). We then meta-test on unseen tasks, and seen (ab) or unseen (ba) datasets. The plot shows the accuracy averaged across multiple runs at each inner step, with shading indicating 95% confidence intervals. The increase in performance at each step suggests we have learned a learning algorithm.

learn. Further, when (meta-)testing the same trained models from the previous experiment on an unseen task (new random transformation of MNIST), they generalize only in the regime of large numbers of tasks and model size (Figure 5.2c). As an in-context learner, meta-testing does not involve any gradient updates but only running the model in forward mode.

Insight 1: It is possible to learn-to-learn with black-box models Effective learning algorithms can be realized in-context using black-box models with few inductive biases, given sufficient meta-training task diversity and large enough model sizes. To transition to the learning-to-learn regime, we needed at least $2^{13} = 8192$ tasks.

In the following, we study learning-to-learn from the perspective of the *data distribution*, the *architecture*, and the *optimization dynamics*. For the data distribution, we look at how the data diversity affects the emergence and transitions of learning-to-learn, generalization, and memorization. For architecture, we analyze the role of the model and state size in various architectures. Finally, we observe challenges in meta-optimization and demonstrate how memorization followed by generalization is an important mechanism that can be facilitated by biasing the data distribution.

5.4.1 Large data: Generalization and algorithmic transitions

Simple data augmentations lead to the emergence of learning-to-learn To verify whether the observed generalizing solutions actually implement learning algorithms (as opposed to e.g. zero-shot generalization), we analyze the meta-test time behavior. We plot the accuracy for a given query point for varying numbers of examples in Figure 5.3. As is typical for learning algorithms, the performance improves when given more examples (inputs and labels).

Generalization Naturally, the question arises as to what extent these learning algorithms are general. While we have seen generalization to unseen tasks consisting of novel projections of the same dataset, do the learned algorithms also generalize to unseen datasets? In Figure 5.3 we observe strong out-of-distribution performance on Fashion MNIST after having trained on MNIST (b, blue), and there is no generalization gap compared to directly training on Fashion MNIST (b, orange). Similarly, when meta training on Fashion MNIST and meta testing on MNIST (a, orange) we observe that the learning algorithm generalizes, albeit with a larger generalization gap.

Comparison to other methods Other datasets and baselines are shown in Table 5.2. We aim to validate whether methods with less inductive bias (such as our GPICL), can compete with methods that include more biases suitable to learning-to-learn. This includes stochastic gradient descent (SGD), updating the parameters online after observing each datapoint. MAML [Finn et al., 2017] proceeds like SGD, but uses a meta-learned neural network initialization. Both methods that rely on backpropagation and gradient descent learn more slowly than our Transformer. In the case of MAML, this may be due to the main mechanism being feature reuse [Raghu et al., 2020] which is less useful when training across our wider task distribution. For in-context learners (methods that do not hard-code gradient descent at meta-test time), we test VSML [Kirsch and Schmidhuber, 2021] that discovered learning algorithms significantly generalizing between tasks. Our GPICL comes surprisingly close to VSML without requiring the associated inductive bias. GPICL generalizes to many datasets, even those that consist of random input-label pairs. We also observe that learning CIFAR10 and SVHN from only 99 examples with a general-purpose learning algorithm is difficult, which we address in Section 5.4.4. Training and testing with longer context lengths improves the final predictions (Appendix D.2). Using LSTM-based in-context learners performs worse, which we further discuss in Section 5.4.2

Table 5.2: **Meta-test generalization to various (unseen) datasets** after meta-training on augmented MNIST and seeing 99 examples, predicting the 100th. We report the mean across 3 meta-training seeds, 16 sequences from each task, 16 tasks sampled from each base dataset. GPICL is competitive to other approaches that require more inductive bias.

Method	Inductive bias	MNIST	Fashion MNIST	KMNIST	Random	CIFAR10	SVHN
SGD	Backprop, SGD	70.31%	50.78%	37.89%	100.00%	14.84%	10.16%
MAML	Backprop, SGD	53.71%	48.44%	36.33%	99.80%	17.38%	11.33%
VSML	In-context, param sharing	79.04%	68.49%	54.69%	100.00%	24.09%	17.45%
LSTM	In-context, black-box	25.39%	28.12%	18.10%	58.72%	12.11%	11.07%
GPICL (ours)	In-context, black-box	73.70%	62.24%	53.39%	100.00%	19.40%	14.58%

among other alternative architectures.

Insight 2: Simple data augmentations are effective for learning-to-learn The generality of the discovered learning algorithm can be controlled via the data distribution. Even when large task distributions are not (yet) naturally available, simple augmentations are effective.

Transitioning from memorization to task identification to general learning-to-learn When do the learned models correspond to memorizing, learning, and generalizing solutions? In Figure 5.4, we meta-train across varying numbers of tasks, with each point on the x-axis corresponding to multiple separate meta-training runs. We plot the accuracy difference between the last and first prediction (how much is learned at meta-test time) for a seen task, an unseen task, and an unseen task with a different base dataset. We observe three phases: In the first phase, the model memorizes all tasks, resulting in no within-sequence performance improvement. In the second phase, it

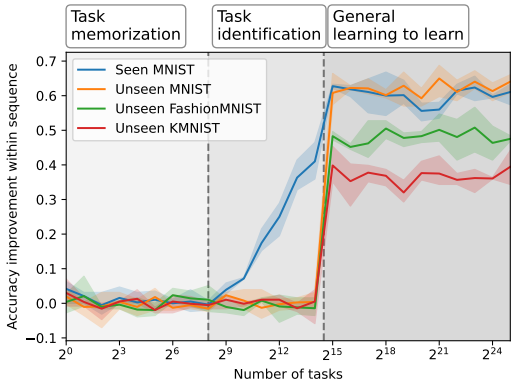


Figure 5.4: **Transformers exhibit three different phases in terms of meta-learned behavior.** (1) When training on a small number of tasks, tasks are memorized. (2) Tasks from the training distribution are identified, which is evident as a within-sequence increase of performance. (3) When training across many tasks, we discover a learning algorithm that generalizes to unseen tasks and unseen datasets.

memorizes and learns to identify tasks, resulting in a within-sequence improvement confined to seen task instances. In the final and third phase, we observe a more general learning-to-learn, a performance improvement for unseen tasks, even different base datasets (here FashionMNIST). This phenomenon applies to various other meta-training and meta-testing datasets. The corresponding experiments can be found in Appendix D.6. In Appendix D.3 we also investigate the behavior of the last transition.

Insight 3: The meta-learned behavior has algorithmic transitions When increasing the number of tasks, the meta-learned behavior transitions from task memorization, to task identification, to general learning-to-learn.

5.4.2 Architecture: Large memory (state) is crucial for learning

In the previous experiments we observed that given sufficient task diversity and model size, Transformers can learn general-purpose learning algorithms. This raises the question how essential the Transformer architecture is and whether other black-box models could be used. We hypothesize that for learning-to-learn the size of the memory at meta-test time (or state more generally) is particularly important in order to be able to store learning progress. Through self-attention, Transformers have a particularly large state. We test this by training several architectures with various state sizes in our meta-learning setting. In Figure 5.5a, we observe that when we vary the hyper-parameters which most influence the state size, we observe that for a specific state size we obtain similar performance of the discovered learning algorithm across architectures. In contrast, these architectures have markedly different numbers of parameters (Figure 5.5b).

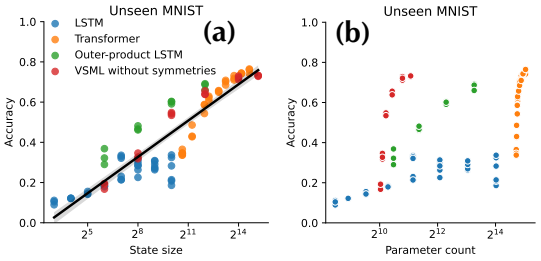


Figure 5.5: **The state size (accessible memory) of an architecture most strongly predicts its performance as a general-purpose learning algorithm.** **(a)** A large state is crucial for learning-to-learn to emerge. **(b)** The parameter count correlates less well with learning capabilities.

What corresponds to state (memory) in various architectures? Memory N_S in the context of recurrent neural networks corresponds to the hidden state or context vector of size N_H, thus $N_S \in \mathcal{O}(N_H)$. More generally, we can describe the state as the information bottleneck that the sequence has to pass through before making predictions. In the context of learning-to-learn, this state has to hold information about everything that has been learned so far. Standard learning algorithms such as neural networks trained via SGD would have a state that corresponds to the neural network parameters, iteratively updated via SGD. In transformers, self-attention allows for a particularly large state of $N_S \in \mathcal{O}(N_K N_L N_T)$ where N_K is the size of key, value, and query, N_L is the number of layers, and N_T is the length of the sequence. In addition to Figure 5.5, Figure D.6 show meta-test performance on more tasks and datasets.

Insight 4: Large state is more crucial than parameter count This suggests that the model size in terms of parameter count plays a smaller role in the setting of learning-to-learn and Transformers have benefited in particular from an increase in state size by self-attention. Beyond learning-to-learn, this likely applies to other tasks that rely on storing large amounts of sequence-specific information.

5.4.3 Challenges in meta-optimization

Meta-optimization is known to be challenging. Meta gradients [Finn et al., 2017; Xu et al., 2018; Bechtle et al., 2021] and works with parameter sharing or weight updates in their architecture [Kirsch and Schmidhuber, 2021; Pedersen and Risi, 2021; Risi, 2021] observed various difficulties: Slower convergence, local minima, unstable training, or loss plateaus at the beginning of training (see Appendix Figure D.14). We show that some of these problems also occur with black-box models and propose effective interventions.

Loss plateaus when meta-learning with black-box models By training across a large number of randomly transformed tasks, memorizing any task-specific information is difficult. Instead, the model is forced to find solutions that are directly learning. We observe that this results in (meta-)loss plateaus during meta-training where the loss only decreases slightly for long periods of time (Figure 5.6a). Only after a large number of steps (here around 35 thousand) does a drop in loss occur. In the loss plateau, the generalization loss increases on unseen tasks from both the same and a different base dataset (Figure 5.6b). This suggests that being able to first memorize slightly enables the following

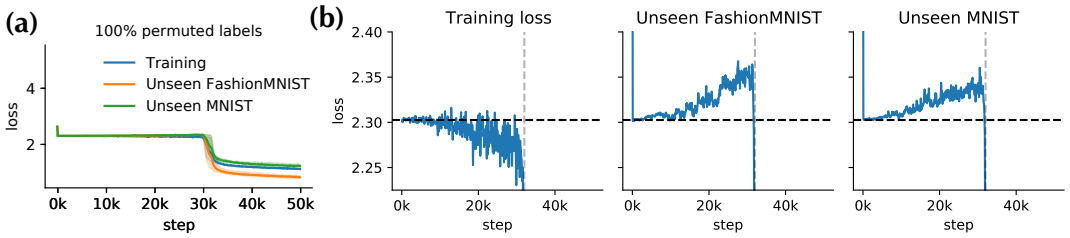


Figure 5.6: **Meta-training dynamics often involve an extended period where GPICL's performance is stuck on a plateau.** **(a)** Meta-loss vs. meta-training step, for a uniform distribution over meta-training tasks. Training tasks are generated by random transformations of FashionMNIST. **(b)** A zoomed in view of the plateau. The loss only decreases slightly and the model memorize small biases in the training data (decreasing generalization) before the loss drops sharply.

learning-to-learn phase. Furthermore, we observe that all gradients have a very small norm with exception of the last layer (Appendix Figure D.10).

Intervention 1: Increasing the batch size High variance gradients appear to be one reason training trajectories become trapped on the loss plateau (see Appendix Figures D.8, D.9). This suggests increasing the meta-batch size as a straightforward solution. When plotting various batch sizes against numbers of tasks we obtain three kinds of solutions at the end of meta-training (Figure 5.7a): (1) Solutions that generalize and learn, (2) Solutions that memorize, and (3) Solutions that are still in the loss plateau (due to maximum of 50 thousand optimization steps). The larger the batch size, the more tasks we can train on without getting stuck in a loss plateau. When plotting the length of the loss plateau against the task batch size (Figure 5.7b) we observe a power-law relationship with increasing batch sizes decreasing the plateau length. At the same time, the batch size also increases the number of total tasks seen in the plateau (Appendix Figure D.11). Thus, this intervention relies on parallelizability. An increase in the number of tasks also increases the plateau length (Figure 5.7c), possibly due to a larger number of tasks inhibiting the initial memorization phase.

Intervention 2: Changes in the meta-optimizer Given that many gradients in the loss plateau have very small norm, Adam would rescale those element-wise, potentially alleviating the issue. In practice, we observe that the gradients are so small that the ϵ in Adam's gradient-rescaling denominator (for numerical stability) limits the up-scaling of small gradients. Using smaller ϵ results in more than halving the plateau length. Alternatively, discarding the magnitude of the

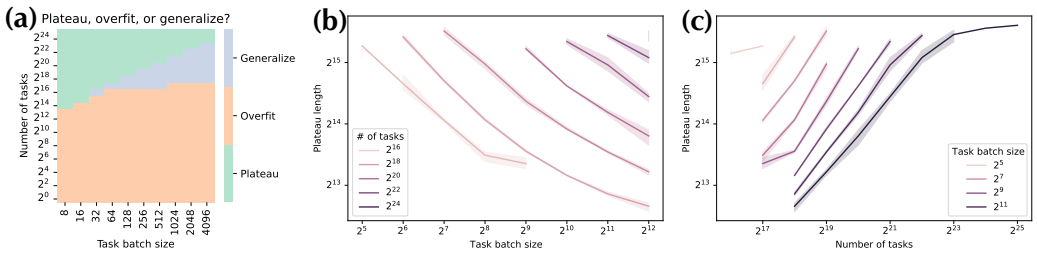


Figure 5.7: **Whether GPICL memorizes, generalizes, or remains trapped on a meta-loss plateau depends on the number of meta-training tasks, and the meta-training batch size. (a)** A phase diagram showing GPICL's behavior at the end of meta-training (50k steps). Solutions either memorize, generalize and learn, or remain in the loss plateau. With additional training steps, configurations in the plateau might eventually transition to memorization or generalization. Generalization only occurs with large enough batch sizes and sufficient, but not too many, tasks. **(b)** This behavior is explained by the plateau length decreasing with the increasing batch sizes (reducing the noise contribution), and **(c)** increasing with larger numbers of tasks.

gradient entirely by applying the sign operator to an exponential moving average of the gradient (replacing Adam's approximate magnitude normalization with direct magnitude normalization) has a similar effect while also increasing the numerical stability over Adam with small ϵ (Appendix Figure D.12).

Intervention 3: Biasing the data distribution / Curricula GPICL mainly relies on the data distribution for learning-to-learn. This enables a different kind of intervention: Biasing the data distribution. The approach is inspired by the observation that before leaving the loss plateau the model memorizes biases in the data. Instead of sampling label permutations uniformly, we bias towards a specific permutation by using a fixed permutation for a fraction of each batch. This completely eliminates the loss plateau, enabling a smooth path from memorizing to learning (Figure 5.8). Surpris-

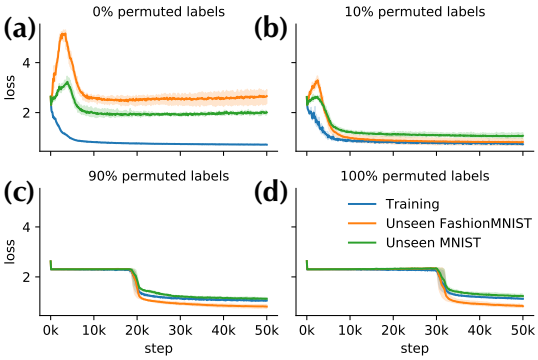


Figure 5.8: **Biasing the training distribution is an effective intervention which prevents a meta-loss plateau.** A uniform distribution over tasks leads to a long plateau **(d)**, while increasing the training fraction that corresponds to a single task reduces the plateau **(abc)**.

ingly, even when heavily biasing the distribution, memorization is followed by generalization. Memorization is observed as the initial increase in loss on unseen tasks in Figure 5.8ab. This biased data distribution can be viewed as a curriculum, solving an easier problem first that enables the subsequent harder learning-to-learn. Further investigation is required to understand how this transition occurs. This may be connected to grokking [Power et al., 2022] which we investigate in Appendix D.6. We hypothesize that many natural data distributions—including language—contain such sub-tasks that are easy to memorize followed by generalization.

5.4.4 Domain-specific and general-purpose learning

We demonstrated the feasibility of meta-learning in-context learning algorithms that are general-purpose. An even more useful learning algorithm would be capable of both generalizing, as well as leveraging domain-specific information for learning when it is available. This would allow for considerably more efficient in-context learning, scaling to more difficult datasets without very long input sequences. Toward this goal, we investigate a simple scheme that leverages pre-trained neural networks as features to learn upon. This could be from an unsupervised learner or a frozen large language model [Radford et al., 2021; Tsimpoukelli et al., 2021]. Here, we first project the inputs $\bar{x}_i$ of a base-dataset $\bar{D}$ into some latent space using a pre-trained network, and then proceed with meta-training and meta-testing as before, randomly projecting these alternative features. For the pre-trained network, we use a ResNet trained on ImageNet and remove its final layer. In Figure 5.9 we have meta-

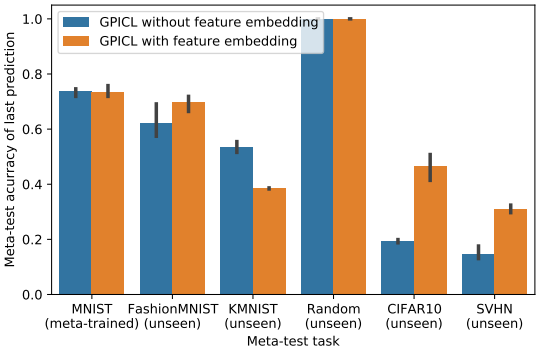


Figure 5.9: **Using pre-trained networks allows leveraging domain-specific knowledge while still generalizing to other datasets** GPICL is meta-trained on MNIST either with the randomly transformed raw inputs or randomly transformed pre-trained features. Pre-training helps to accelerate meta-test-time in-context learning on datasets that have a matching domain, such as CIFAR10. With only 100 examples, the learning algorithm can achieve about 45% accuracy on CIFAR10. The learning algorithms still generalize to a wide range of datasets. Error bars are 95% confidence intervals of the mean across meta-training runs.

trained GPICL on MNIST either with the randomly transformed raw inputs or randomly transformed embedded features. At meta-test-time the learning algorithm generalizes to a wide range of datasets, measured by the meta-test accuracy of the 100th example. At the same time, the pre-trained ImageNet helps to accelerate learning on datasets that have a matching domain, such as CIFAR10. We observe that with only 100 examples, the learning algorithm meta-trained on MNIST, can achieve about 45% accuracy on CIFAR10. In Appendix D.6 we demonstrate that CLIP [Radford et al., 2021] embeddings can further improve learning efficiency.

5.5 Related work

Meta-learning: Inductive biases and general-purpose learning algorithms Meta-learning approaches exist with a wide range of inductive biases, usually inspired by existing human-engineered learning algorithms. Some methods prewire the entire learning algorithm [Finn et al., 2017], pre-wire backpropagation and the structure of a gradient-based optimizer [Andrychowicz et al., 2016; Metz et al., 2019b, 2020a], or learn the loss function [Houthooft et al., 2018; Kirsch et al., 2020b; Bechtle et al., 2021].

Many methods search over hyper-parameters that alter existing learning algorithms [Xu et al., 2018; Metz et al., 2020b; Chen et al., 2022]. Fast weight programmers or hyper-networks update the weights of the same or another neural network [Schmidhuber, 1992b, 1993a; Ha et al., 2017; Irie et al., 2021b; Sandler et al., 2021; Kirsch and Schmidhuber, 2022b; Zhmoginov et al., 2022], frequently with various symmetries. There has been growing interest in meta-learning more general-purpose learning algorithms. Such learning algorithms aim to be general and reusable like other human-engineered algorithms (e.g. gradient descent). The improved generality of the discovered learning algorithm has been achieved by introducing inductive bias, such as by bottlenecking the architecture or by hiding information, encouraging learning over memorization. Methods include enforcing learning rules to use gradients [Metz et al., 2019b; Kirsch et al., 2020b; Oh et al., 2020], symbolic graphs [Real et al., 2020; Co-Reyes et al., 2021], parameter sharing and symmetries [Kirsch and Schmidhuber, 2021; Kirsch et al., 2022a], or adopting evolutionary inductive biases [Lange et al., 2023; Li et al., 2023]. Parameter sharing and symmetries have additionally been discussed in the context of self-organization [Tang and Ha, 2021; Risi, 2021; Pedersen and Risi, 2022].

In-context learning with black-box models Black-box neural networks can learn-to-learn purely in their activations (in-context) with little architectural and algorithmic bias [Hochreiter et al., 2001; Wang et al., 2016; Duan et al., 2016; Santoro et al., 2016; Mishra et al., 2018; Garnelo et al., 2018]. This requires a feedback or demonstration signal in the inputs that allows for learning such as the reward in reinforcement learning or label in supervised learning [Schmidhuber, 1993b]. While a frequently used architecture is the LSTM [Hochreiter and Schmidhuber, 1997b; Gers et al., 2000a], this mechanism has also seen substantial recent attention in Transformer models [Brown et al., 2020; Chan et al., 2022] under the name of in-context learning. In large language models (LLMs) demonstrations of a task in the input help solving language-based tasks at inference (meta-test) time [Brown et al., 2020]. This few-shot learning ability has been attributed to the data-distributional properties of text corpora [Chan et al., 2022]. In-context learning has also been interpreted from a Bayesian inference perspective [Ortega et al., 2019; Mikulik et al., 2020; Nguyen and Grover, 2022; Müller et al., 2022]. Our method GPICL is in the class of these black-box in-context learners. The number of model parameters has been at the core of scaling up LLMs to unlock greater capabilities and have been formulated in scaling laws [Kaplan et al., 2020; Hoffmann et al., 2022]. Our empirical study suggests that for learning-to-learn, the amount of memory (model state) is even more predictive of in-context learning capabilities than parameter count.

General-purpose in-context learning While in-context learning has been demonstrated with black-box models, little investigation of general-purpose meta-learning with these models has been undertaken. Generalization in LLMs has previously been studied in regard to reasoning and systematicity [Csordás et al., 2021; Delétang et al., 2022; Wei et al., 2022; Zhou et al., 2022; Anil et al., 2022] but not in their ability to learn new tasks in-context. In this work we focus on meta-generalization instead, the extent to which in-context learning algorithms generalize. In contrast to previous methods, GPICL implements *general-purpose* learning algorithms. Independently, Garg et al. [2022] recently studied generalization on synthetic functions compared to our augmented datasets. VSML [Kirsch and Schmidhuber, 2021] also implements in-context learning with black-box LSTMs, but makes use of parameter-sharing to aid generalization. PFNs [Müller et al., 2022] demonstrated learning to learn on small tabular datasets when meta-training on synthetically generated problems. Experiments on more complex classification settings such as Omniglot [Lake et al., 2011] relied on fine-tuning. In comparison, our method investigated meta-generalization of learning algorithms directly to datasets such as MNIST, Fashion MNIST, and

CIFAR10 while studying fundamental questions about the conditions necessary for such generalization. TabPFNs [Hollmann et al., 2022] extend PFNs to larger tabular datasets.

5.6 Limitations

An important subject of future work is the exploration of task generation beyond random projections, such as augmentation techniques for LLM training corpora or generation of tasks from scratch. A current limitation is the applicability of the discovered learning algorithms to arbitrary input and output sizes beyond random projections. Appropriate tokenization to unified representations may solve this [Chowdhery et al., 2022; Zhang et al., 2023b].

Furthermore, learning algorithms often process millions of inputs before outputting the final model. In the current black-box setting, this is still difficult to achieve and it requires new advances for in context length of sequence models. Recurrency-based models may suffer from accumulating errors, whereas Transformer's computational complexity grows quadratically in sequence length. Specifically, our Transformer-based model has a runtime complexity of $O(L(n^2d + nd^2))$ where L is the number of layers, d is the token / KQV size, and n is the number of tokens. This applies to both (meta-)training and (meta-)testing.

5.7 Conclusion

By generating tasks from existing datasets, we demonstrated that black-box models such as Transformers can meta-learn general-purpose in-context learning algorithms (GPICL). We observed that learning-to-learn arises in the regime of large models and large numbers of tasks with several transitions from task memorization, to task identification, to general learning. The size of the memory or model state significantly determines how well any architecture can learn how to learn across various neural network architectures. We identified difficulties in meta-optimization and proposed interventions in terms of optimizers, hyper-parameters, and a biased data distribution acting as a curriculum. We demonstrated that in-context learning algorithms can also be trained to combine domain-specific learning and general-purpose learning. We believe our findings open up new possibilities of data-driven general-purpose meta-learning with minimal inductive bias, including generalization improvements of in-context

learning in large language models (LLMs).

Chapter 6

GLAs: Towards black-box & general-purpose in-context learning agents

Keywords *in-context learning, transformers, supervised reinforcement learning*
Article *Kirsch et al. [2023] (preprint 2023)*

How can GPICL (Chapter 5) be extended to meta-reinforcement learning (meta-RL)? In this work, we investigate Transformer-based Generally Learning Agents (GLAs) that learn-to-reinforcement-learn purely via in-context learning.

6.1 Introduction

Improvements in Reinforcement Learning (RL) algorithms are mainly driven by the research and engineering of humans. Meta-learning instead automates this process [Schmidhuber, 1987; Parker-Holder et al., 2022b] to discover novel RL algorithms with little human intervention. A key property of human-engineered learning algorithms is their applicability to a wide range of RL problems. To replace such algorithms with automatically discovered ones, those need to be equally general-purpose [Kirsch et al., 2020b; Oh et al., 2020; Team et al., 2023].

One method of achieving generalization is to integrate inductive bias into the agent architecture or learning algorithm, such as the use of gradient descent [Kirsch et al., 2020b; Oh et al., 2020]. At the same time, this may limit

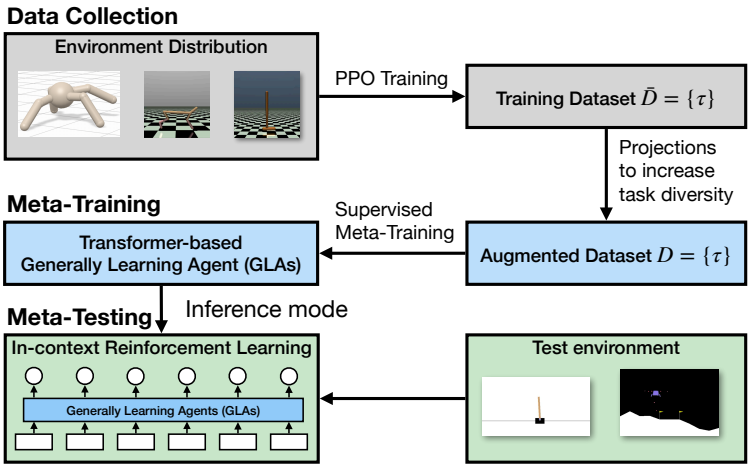


Figure 6.1: **Our Generally Learning Agents (GLAs) are meta-trained on augmented RL datasets via supervised learning.** First, one or more datasets of improving policies are collected using PPO. Next, these datasets are augmented with random observation and action projections to create a large diversity of tasks (environments). A Transformer is then trained to distill the (sped-up) learning process into a single in-context RL agent. Finally, at meta-test time, we can take an environment from a different domain (different actuators, observations, dynamics, and dimensionalities) and learn from rewards purely in-context without explicit hand-crafted RL algorithms or explicit gradient descent.

the discoverable learning algorithms. An alternative to this is to embed the entire learning algorithm into the black-box neural network such that the network learns-to-learn by in-context learning [Schmidhuber, 1993b; Hochreiter et al., 2001; Duan et al., 2016; Wang et al., 2016; Brown et al., 2020; Kirsch et al., 2022a,b]. This requires that the entire learning algorithm be meta-learned from scratch, such as (re-)discovering the principle of learning by gradient descent [Kirsch and Schmidhuber, 2021; Von Oswald et al., 2023; Akyürek et al., 2022] and credit assignment from rewards. In supervised learning, Large Language Models (LLMs) have led to strong in-context learning capabilities [Brown et al., 2020]. Scaling laws were discovered that model the increase in predictive performance and capabilities with more training data and model parameters [Kaplan et al., 2020]. Previous work suggested that these are also important drivers for the generality of in-context learning capabilities (Chapter 5 [Kirsch et al., 2022b]). In reinforcement learning, generalization of in-context learners has been limited.

Inspired by these works, we propose a path towards future agents that will be

able to learn-how-to-learn in-context across a wide range of environments. To achieve such generalization, a broad task distribution of diverse and challenging environments will be needed during meta-training. Our Generally Learning Agents (GLAs, Figure 6.1) are an important first step in this direction. We propose a framework where agents are able to learn new tasks in-context, without requiring pre-designed learning algorithms. Our GLAs are meta-trained using supervised learning techniques [Laskin et al., 2022; Liu and Abbeel, 2023; Lee et al., 2023] on a dataset of experiences generated from PPO agents. We empirically show that by sub-sampling the generated data, the discovered learning algorithms are not limited to imitating the PPO learning algorithm but may discover qualitatively different learning algorithms. We add augmentations to this dataset in the form of random projections to the observations and actions, which generates sufficient diversity to result in fairly general RL algorithms to be encoded into the neural network weights. We demonstrate that our GLAs are a significant step towards general-purpose cross-domain in-context learners by meta-testing them on different robotic control problems that were not seen during meta-training.

6.2 Meta-learning general-purpose in-context learning agents

In-context RL agents A reinforcement learning (RL) algorithm is a mapping $f : \tau \mapsto \theta$ from agent experience $\tau := (s_0, a_0, r_0, d_0, s_1, \ldots, s_L, a_L, r_L, d_L)$ to a policy $\hat{\pi}(a|s, \theta)$ with index $i \in \{0, \ldots, L\}$, observations s_i, actions a_i, rewards r_i, and terminations d_i.[1] We refer to those functions f as learning algorithms, where the expected return $\mathbb{E}_{\hat{\pi}}[R]$ is larger than the returns found in τ and tends to increase as new experiences are added to τ. Instead of modeling the learning algorithm f and policy $\hat{\pi}$ separately, we may also combine those to an in-context learning policy $\pi(a|s, \tau)$. Optimizing for π to discover learning algorithms then corresponds to meta-learning. We usually parameterize π using neural networks such as LSTMs [Hochreiter and Schmidhuber, 1997b; Gers et al., 2000a], Transformers [Vaswani et al., 2017], or linear Transformers [Schmidhuber, 1992b; Katharopoulos et al., 2020; Schlag et al., 2021a] due to the sequential nature of τ.

[1] Here we assume that π acts in an MDP such that s is a sufficient state representation, but this can be extended to POMDPs by providing a representation of multiple previous observations instead.

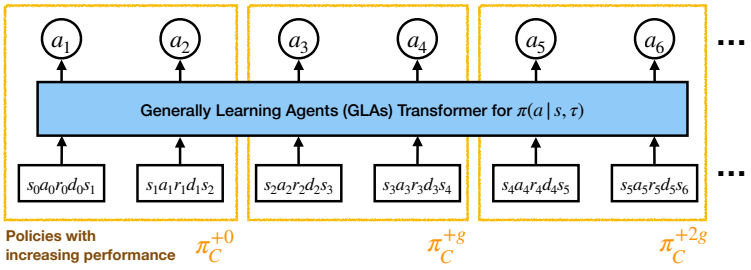


Figure 6.2: **Our RL agents reinforcement-learn purely in-context.** A Transformer is used to condition on previously observed environment transitions $(s_i, a_i, r_i, d_i, s_{i+1})$ and predicts the next action. Meta-Training is done on sequences of transitions generated from policies with increasing performance.

Meta-learning via supervised learning Meta-optimization often involves standard RL techniques [Wang et al., 2016; Duan et al., 2016] by directly maximizing the average return over multiple episodes (together referred to as the lifetime of the agent). Recently, supervised learning has been very successful in language modeling [Brown et al., 2020] but also has shown great promise in reinforcement learning [Schmidhuber, 2019; Chen et al., 2021; Reed et al., 2022].

In this work, we use supervised learning techniques for meta learning π and demonstrate their potential to be used effectively for training across a broad environment distribution to discover novel generalizing RL algorithms. To do so, we collect sequences of transitions $\tau := (x_0, \dots, x_L)$ with $x_i := (s_i, a_i, r_i, d_i, s_{i+1})$ that correspond to agent behavior with improving performance. Here, we generate those by running PPO [Schulman et al., 2017] on the meta-training environments. We then auto-regressively model the action distribution $p(a_i|s_i, \tau_{:i-1})$ given the previous transitions $\tau_{:i-1}$. To go beyond algorithm cloning/distillation [Kirsch and Schmidhuber, 2021; Laskin et al., 2022] and exceed the performance of the human-engineered learning algorithm that generated the data, we sub-sample the data: Given a subset of the data $\tau^0_{j:k}$ where $j < k$ generated from collection policy π^0_C, we predict actions in $\tau^{+g}_{l:m}$ where $l < m$ and $l \gg k$ that are generated from the policy π^{+g}_C that was updated by PPO for g iterations. We refer to g as the 'gap' between $\tau_{j:k}$ and $\tau_{l:m}$. We expect that as we increase the gap g, we meta-learn RL algorithms that learn more quickly. In summary, our supervised learning objective that we maximize using gradient ascent is

$$J(\theta) = -\sum_{i=l}^{m} D[p(a_i)|\pi_\theta(a_i|s_i, \tau^0_{j:k}, \tau^{+g}_{l:i-1}, \eta_i)] \tag{6.1}$$

where D is a divergence, here the sum of the reverse and forward KL. The objec-

tive as denoted here, only models a single gap g, but in practice we condition on multiple gaps, i.e. $\pi_\theta(a_i|s_i, \tau^0, \tau^{+g}, \tau^{+2g}, \ldots, \tau^{+ng}_{l:i-1}, \eta_i)$. We use a Transformer with parameters θ to model π_θ, depicted in Figure 6.2. Additionally, we may condition on auxiliary information η_i that describes the amount of improvement that is expected. Options for η_i are the gap g, indicating how many PPO updates to distill into π, or the policy performance (return) at index i, or the location l within the lifetime $l \in 0, \ldots, L$ of the agent.

This supervised learning scheme allows us to perform efficient meta-training on offline data with a stationary objective and without the need for collecting additional data. Because the whole data sequence is known in advance, without intermediate environment interactions, this allows for efficient training of Transformers to model π. Meta-training is summarized in Algorithm 10.

Training on broad task distributions If we have already solved the environments in the meta-training distribution with PPO, why is meta-learning a novel learning algorithm still useful? We hypothesize that meta-training across a sufficiently broad task distribution allows us to discover novel in-context RL algorithms that can be re-used on many unseen RL problems that we typically care about. As a proof of concept, in this work we (1) train on various diverse continuous control problems and (2) augment the supervised training dataset with random linear projections in its observations and actions to generate sufficient diversity. For the augmentations, we adopt the randomization methodology from previous work on supervised in-context learning [Kirsch et al., 2022b]. We linearly project observations to 64 dimensions, and actions to 16 dimensions. This also enables to meta-test our GLAs across domains where actuators and observations are of varying dimensionalities. We demonstrate that this results in increasingly generalizable in-context learning algorithms for RL.

Meta-Testing During meta testing we simply auto-regressively evaluate the Transformer starting with an empty history $\tau \leftarrow ()$, growing τ with each experienced transition for K environment interactions. This implements in-context RL and is described in Algorithm 11.

Algorithm 10 Supervised Meta-Training for GLAs

procedure Train(E) ▷ Meta-train on a set of MDPs E
  $\bar{D} \leftarrow \{\tau_e\}$ ▷ Collect a dataset of trajectories with increasing performance using PPO on environments $e \in E$
  $\theta \leftarrow$ random parameters ▷ Randomly initialize learning agent π
  while not converged **do**
    $D \leftarrow \text{augment}(\bar{D})$ ▷ Augment D; here using random projections on s_i and a_i
    $B \leftarrow \text{sample}(D)$ ▷ Sub-sample transitions from D
    $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta; B)$ ▷ Update learning agent π_θ via SGD on Equation 6.1
return Generally Learning Agent π_θ

Algorithm 11 Meta-Testing for GLAs

procedure Test(e, π_θ) ▷ Meta-test on a new MDP e with a generally learning agent π_θ
  $\tau \leftarrow ()$ ▷ Initialize empty history
  for $k \leftarrow 1$ to K **do**
    Use policy $\pi_\theta(a|s, \tau)$ to obtain a new transition $\xi = (s, a, r, d, s')$ from environment e
    $\tau_k \leftarrow \xi$ ▷ Update history

6.3 Experiments

Supervised learning discovers learning agents on single tasks To begin, we demonstrate that our supervised meta-RL algorithm can discover in-context learning policies that encode a learning algorithm specific to a task. Figure 6.3 shows how the mean return increases at meta-test time on a simple grid world and in continuous control. The grid world consists of a 3x3 grid with directional movement actions and a goal position at a fixed location. For continuous control, we meta-train on the Ant-v4 MuJoCo environment. We observe that the initial test return is already larger than a random policy on Ant-v4 – suggesting that the learned learning algorithm leverages task-specific knowledge.

In-context Learning can be sped-up when the gap is increased How can the speed of learning be controlled? In Figure 6.3 (left) we demonstrate how an increased gap g can speed up learning at meta-test time. We also test the limiting

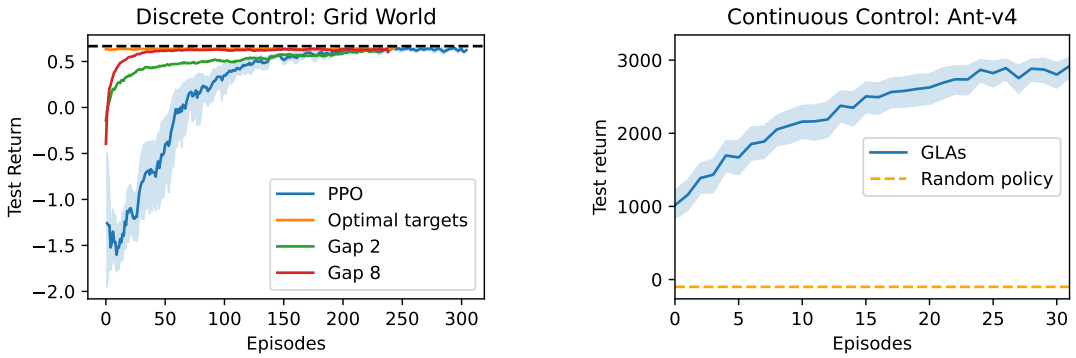


Figure 6.3: **Supervised learning discovers in-context learning agents on single tasks with controllable efficiency.** In both grid worlds and continuous control the mean return increases in-context with more environment interactions when running the GLAs Transformer. The rate of learning can be controlled by the gap g used during meta-training. On Ant-v4, the initial agent is already better than a random policy, suggesting that the learned in-context RL algorithm leverages task-specific knowledge. Shading indicates 95% confidence intervals with 64 meta-test training seeds.

case of a maximally large gap g that corresponds to the action targets taken by the optimal policy in the dataset. We find that in the case of a single task, the network simply learns the optimal policy.

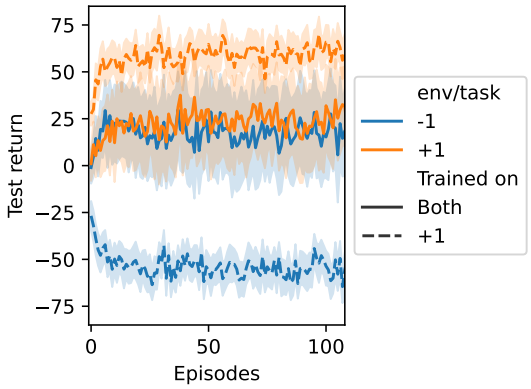


Figure 6.4: **In-context learning agents trained via supervised-learning can adapt to task changes.** On the standard Ant-Dir meta-learning benchmark the agent adapts when having seen both tasks during training, but does not learn in-context on an unseen task. Shading indicates 95% confidence intervals with 64 meta-test training seeds.

Standard meta-learning benchmarks (simple task distributions) How does the meta-test behavior change when we move from single tasks to task distributions? Here we begin by using a standard meta-learning task distribution, the Ant-Dir environment [Finn et al., 2017] that involves the forward and backward task. The task is not part of the policy inputs, and thus has to be inferred from observations and rewards. In the Ant-Dir environment (Figure 6.4), we observe

that training on a single task only allows for learning on that particular task, but does not generalize to the task of moving into the other direction. Conversely, meta-training on both environments allows for in-context learning in either task. We observe that for some meta-test training seeds, the task is incorrectly recognized, resulting in larger confidence intervals. The policy then follows the incorrect task. We hypothesize that this is a difficulty with the supervised meta-training objective on such meta-task distributions where the task is determined early in meta-test training and future actions simply condition on states seen during meta-training, independent of the task rewards. This may be alleviated by broader task distributions.

Generalization fails to significantly different tasks and environments We have motivated this work with the goal of in-context learning agents that generalize to a wide range of environments, across domains. Given this rather simple training distribution, we would not expect the agent to generalize to a different environment such as Cartpole or Reacher. To do so, we need to scale the diversity of the data distribution.

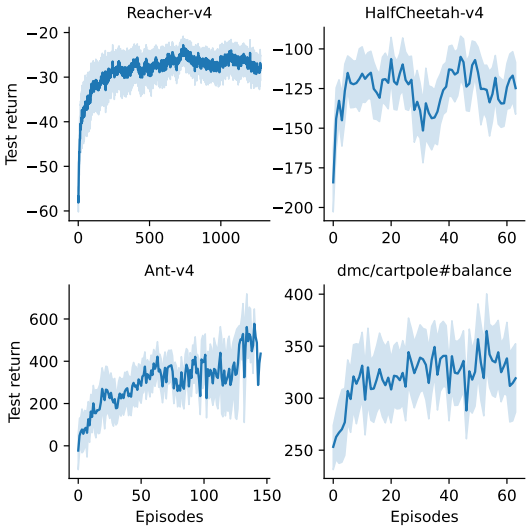


Figure 6.5: **GLAs generalize to novel domains via in-context RL.** After meta-training on augmented Ant-v4, the agent can learn in-context on Reacher-v4, HalfCheetah-v4, and DeepMind-Control Cartpole. Shading indicates 95% confidence intervals with 32 meta-test training seeds.

Scaling the task distribution enables cross-domain generalization Next, we augment the dataset by randomly projecting observations and actions, as described in Section 6.2. Here, we meta-train on the augmented Ant-v4 environment. This makes it impossible to directly encode the optimal policy and forces GLAs to learn in-context. Instead of just meta-testing the in-context learning agent on the same Ant-v4 environment or variations thereof, we also apply it to entirely different domains in Figure 6.5. We observe, that the agent to some

extent implements a learning algorithm that applies across domains. It performs in-context learning not just on the seen Ant-v4 environment (with an unseen random projection), but also generalizes its learning algorithm to the Reacher-v4, HalfCheetah-v4, and DeepMind-Control Cartpole environment. The resulting performance is still sub-optimal compared to PPO (Figure 6.6), but there is visible improvement (test-time learning) on tasks with different actuators, task dynamics, and observations. Meta-training on a much broader task-distribution of many continuous control tasks combined with augmentations as proposed here may significantly improve these results.

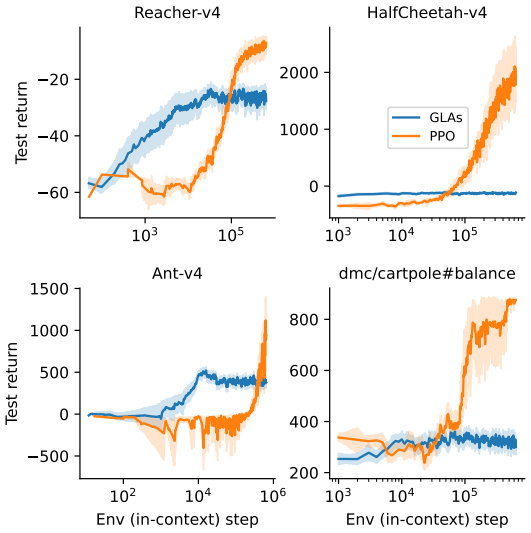


Figure 6.6: **High sample efficiency but early convergence.** The RL algorithm discovered by GLAs tends to be more sample-efficient than PPO, but converges to significantly lower performance. We expect that training on a broader task distribution in conjunction with larger models and longer context length would improve these results. Shading indicates 95% confidence intervals with 32 meta-test training seeds.

6.4 Conclusion

Reinforcement Learning (RL) research has produced a wide range of methods for learning from rewards. In this work, we instead searched for novel RL algorithms purely by conditioning on previous experience from the environment, without any explicit optimization at meta-test time. Compared to previous work in memory-based and in-context meta-RL, we have shown that given a sufficiently rich environment distribution, the discovered RL algorithms start generalizing across domains, moving us closer to automating RL research through meta-learning. To achieve this, we collected an offline dataset of agent experience with improving performance and then augmented the dataset with random projections in observation and action space. When meta-trained on these environments using supervised learning, the resulting RL algorithms encoded in our Generally Learning Agents (GLAs) generalize to robotic control problems

that are significantly different from training, such as meta-training on Ant and learning in-context on the Reacher and Cartpole environments. At this point, the discovered learning algorithms converge early and result in suboptimal performance.

We believe our approach provides a foundation for new large models, trained across an extremely diverse set of RL environments, to enable efficient learning from feedback far beyond our current RL algorithms. Based on our initial experiments in this work, we plan to further improve generalization, robustness, and performance at meta-test time. Further broadening the task distribution with generated, real, and augmented environments will improve the (learning) capabilities of our GLAs agents. Post-training of existing large language models (LLMs) instead of training from scratch may be a promising direction. Finally, developing sequence models that efficiently can process and compress hundreds of thousands of environment transitions [e.g. Schlag et al., 2021a; Gu et al., 2021; Lu et al., 2023] will be important to improve the efficiency and expressivity of both learning and acting at meta-test time. The quadratic computational complexity of Transformers in sequence length remains a challenge for both meta-training and meta-testing with large numbers of environment transitions.

Chapter 7

FME: Eliminating meta-optimization and recursive self-improvement

Keywords *in-context learning, recursive self-improvement, self-referential, meta-optimization*
Article *Kirsch and Schmidhuber [2022b] (preprint 2022)*

As we have seen in the last few chapters, meta-learning automates the search for learning algorithms. At the same time, it creates a dependency on human engineering on the meta-level, where meta-learning algorithms need to be designed. To avoid this, in this chapter we investigate systems that recursively self-improve in a positive feedback loop, where every improvement of the system to itself can lead to further, possibly even faster, improvements. We discuss methods to do so without the need for explicit meta-optimization, reducing our reliance on human engineering.

7.1 Introduction

Machine learning is the process of deriving models and behavior from data or environment interaction using human-engineered learning algorithms. Meta learning takes this process to the meta-level: Its goal is to derive the learning algorithms themselves automatically as well [Schmidhuber, 1987; Hochreiter et al., 2001; Wang et al., 2016; Duan et al., 2016; Finn et al., 2017; Flennerhag et al., 2020; Kirsch et al., 2020b; Kirsch and Schmidhuber, 2021]. Unfortunately, this usually creates a dependency on human engineering on the meta-level, where researchers now have to design meta learning algorithms. In this chapter, we

investigate recursively self-improving meta-learning systems (RSI) [Schmidhuber, 2009; Schmidhuber et al., 1997] that do so without the need for explicit meta-optimization, reducing our reliance on human engineering. Starting from a simple initialization, every improvement of the system to itself can lead to further improvements in a positive feedback loop, potentially accelerating improvements over time. Ideally, this process would be *open-ended*, i.e. produce better and better machine learning systems indefinitely without human intervention.

We discuss different possible substrates for such RSIs, such as memory-based in-context learners and a fully self-referential neural architecture [Schmidhuber, 1993b]. In the latter, all variables of the system are under the control of the system itself, including its own weights. We identify necessary conditions for such self-referential architectures and discuss possible implementations.

A major challenge of such systems is to ensure that changes lead to actual improvement, rather than degradation of capabilities. Because the improvement operator itself is being modified, it is difficult to guarantee that changes will lead to better performance in the future. While in principle changes could be proven to be beneficial [Schmidhuber, 2009], finding such proofs is difficult in practice, especially when the system is implemented by a neural network. Instead, we focus on empirical methods to ensure that self-modifications lead to long-term improvement. At the same time, we argue that the meta-heuristic that ensures these long-term improvements should be as simple as possible, to avoid irreversible inductive bias in the entire system that can not be self-modified later. For example, were we to implement a standard evolutionary algorithm [e.g. Wierstra et al., 2008; Salimans et al., 2017] to ensure this improvement, this algorithm would be always part of the system, sitting at the meta-level, and could not be changed. To that end, we propose fitness monotonic execution (FME), a simple approach to avoid explicit meta-optimization. We empirically demonstrate this in the context of a neural network that self-modifies to solve bandit and classic control tasks, improves its self-modifications, and learns how to learn purely by assigning more computational resources to better performing solutions.

Finally, we discuss how pre-trained models and human-designed learning algorithms such as Large Language Models (LLMs) and backpropagation are great candidates to bootstrap a process of recursive self-improvement with minimal human intervention.

7.2 What is needed for recursive self-improvement (RSI)?

A recursively self-improving system is a system that can improve itself, and these improvements lead to further improvements in a positive feedback loop. To build such a system, we first need to choose a substrate and initialization that is general enough to allow for arbitrary self-modifications. We can frame this as choosing a reference Turing machine and its initial program, ensuring that the substrate that we pick is Turing complete in its self-modifications.

7.2.1 Partially or fully self-referential architectures

In-context learners are partially self-referential Let's first consider a neural implementation of such a system. As we have seen in earlier chapters (e.g. Chapter 3), neural networks that update their weights or memory (in-context learners) are equivalent in their expressive power. Nevertheless, in both cases there exist free variables (usually weights) that are not updated by the dynamics of the system itself, but by a human-engineered (meta-) learning algorithm or initialization. Would such a system be sufficient to generate an RSI or would we require full self-reference where all variables (weights and activations) are modifiable by the system itself [e.g. Schmidhuber, 1993b]?

Fully self-referential architectures While activations and memory in a neural network change dynamically, the weights of a neural network are usually updated by a fixed human-engineered learning algorithm. It thus may seem intuitive to reason that a neural network may recursively self-improve if it was capable of modifying its own weights as well. One previously suggested way of achieving this is to bring both the activations and weights in a neural network under the control of the network itself [Schmidhuber, 1993b]. This is referred to as a self-referential neural architecture. Compare this to a conventional neural network where there is a subset of variables (called the weights or parameters) that are only updated by a fixed learning algorithm (such as backpropagation). This entails that part of the (meta-)learning behavior is fixed and needs to be defined by the researcher. In contrast, fully self-referential architectures control all variables. This includes activations (conventionally updated by the neural network itself), weights (conventionally updated by a learning algorithm), meta weights, etc.

Notation In the remainder of this chapter, we denote external inputs to the neural network as $x \in \mathbb{R}^{N_x}$ (such as observations and rewards in Reinforcement Learning, or error signals in supervised learning), outputs as $y \in \mathbb{R}^{N_y}$ (e.g. actions in Reinforcement Learning), and the parameters of the neural network as θ. Further, we denote time-varying variables (memory) as $h \in \mathbb{R}^{N_h}$ (such as the hidden state of an RNN). We summarize all variables in a neural network as $\phi = \{\theta, h, y\}$.

A self-referential architecture $\phi \leftarrow g_\phi(x)$ is described by function g that may update all the variables $\phi = \{\theta, h, y\}$. The network controls all of its variables in the sense that any elements of ϕ can be changed by the network itself.

Is full self-reference necessary for RSI? We have previously reasoned, that a recursively self-improving system may benefit from controlling all of its variables, including the neural network weights. Next, we show that such self-referential architectures do not have a fundamental representational advantage over memory-based architectures (in-context learners) when the free (initial) variables are meta-optimized using a human engineered learning algorithm.

We defined self-referential architectures $\phi \leftarrow g_\phi(x)$ as those that can update all their variables ϕ. For notational purposes, we can express y as the explicit output $\phi, y \leftarrow g_\phi(x)$. Compare this to a memory architecture such as a recurrent neural network $h, y \leftarrow f_\theta(h, x)$ parameterized by θ where h corresponds to its hidden state (memory). Can the self-referential architecture represent any functions that the memory architecture can not? A commonly used intuition [Schmidhuber, 1993b] is that self-referential architectures are self-modifying, in that they change their own weights, affecting not only their outputs and current weights but also future weight changes through g_ϕ. These architectures can thus not only learn, but also meta-learn, meta-meta-learn, etc. While memory architectures do not update their weights, they can also be self-modifying. Changes in memory h affect the output directly, but also the effective function f_θ by modifying its input h, in turn determining future changes to h. More generally, it directly follows from the Turing completeness of RNNs [Siegelmann and Sontag, 1991] that we can use in-context learners (memory-based architectures) to simulate self-referential systems:

Observation 7.2.1. For any self-referential architecture $\phi, y \leftarrow g_\phi(x)$ and some initial ϕ_0 we can find a memory architecture $h, y \leftarrow f_\theta(h, x)$, θ, and initial h_0 such that for any sequence of $x_{1:T}$ we have $\hat{f}(x_{1:T}) = \hat{g}(x_{1:T})$ where $\hat{f}$ and $\hat{g}$ are the unrolls returning $y_{1:T}$ of f and g respectively.

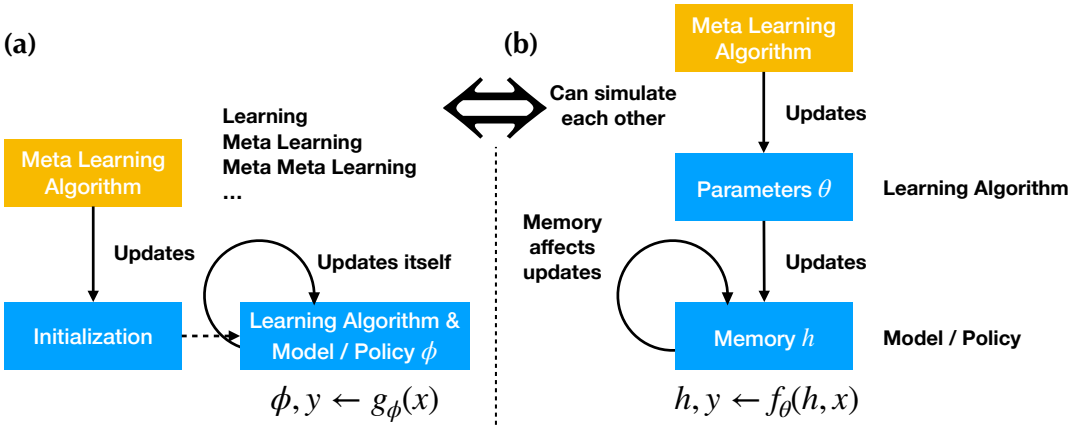


Figure 7.1: **Self-referential (a) and memory-based (b) architectures (in-context learners) can represent the same function class and both require meta optimization.** Self-referential networks can directly update all of their weights, whereas memory-based networks can self-modify through memory updates. The former require meta optimization of the weight initialization ϕ_0, the latter require the initial memory h_0 and weights θ.

Proof sketch. From the Turing completeness of RNNs [Siegelmann and Sontag, 1991], it follows that we can construct an RNN simulator f_θ (with sufficiently large parameterization θ) that stores ϕ in h and sets θ, h_0 such that at each step $t \in \mathbb{N}$ it performs the same computation as g_ϕ by taking ϕ_t from h_t, computing ϕ_{t+1}, and storing it in h_{t+1}. □

Furthermore, any memory-based architecture can be represented by a self-referential architecture where a subset of variables is updated by the identity function. In conclusion, the function class that can be represented by self-referential architectures is equivalent to memory architectures, given sufficiently rich parameterizations. This is visualized in Figure 7.1. For both memory architectures and self-referential architectures the same question arises: How do we set the free variables θ, h_0, or ϕ_0 in the absence of direct meta optimization?

7.2.2 Substrates for RSI

Given our previous analysis of in-context learners and self-referential architectures, we can now discuss possible substrates and initializations for building a recursively self-improving system.

Universal initialization Our first option is to pick any Turing complete in-context learners, such as an LSTM [Hochreiter and Schmidhuber, 1997b] or an auto-regressive transformer [Schuurmans et al., 2024], and initialize its non-modifiable variables (weights θ) to be universal in the sense that any behavior and learning is expressible in changing h. This would require us constructing such a universal initialization, which may be non-trivial. We leave this for future work.

Large language model initialization A useful initialization may also be a large language model (LLM). Assuming that an LLM is a universal computer [Schuurmans et al., 2024], in principle context updates are sufficient to cover all possible solutions and learning algorithms. We discuss this in more detail later in Section 7.6 and in Chapter 8.

Random initialization, but modifiable Finally, we may choose a fully self-referential architecture and initialize all its variables randomly. Due to the full self-reference, all variables are modifiable by the system itself, potentially reverting any initial inductive bias. This is the option that we explore in the remainder of this chapter.

7.2.3 How to construct a fully self-referential architecture

In this section, we discuss necessary conditions to construct a fully self-referential architecture and possible implementations.

Necessary Conditions A self-referential architecture $\phi \leftarrow g_\phi(x)$ is described by a connected computational graph for the function g that has variables $\phi = \{\theta, h, y\}$ (one node per scalar). The network controls all of its variables in the sense that any element(s) of ϕ can be changed by the network itself. This blurs the distinction between activations (memory) h and weights θ. Computational graphs that fulfill this definition are required to have a certain structure. At least some variables need to be reused in multiple operations (node out-degree > 1). We refer to this as variable sharing, a generalization of weight sharing [Fukushima, 1979] extending beyond classical weights.

As an illustrating example, consider a square dense weight matrix. It consists of N^2 weights and N activations. While the activations are time-varying, the weights are source nodes in the computational graph and cannot be directly updated by actions of the network itself. To change that, we need to derive N^2

variables from N time-varying variables. This can only be done by reusing some of the same N time-varying variables in multiple operations, generating the N^2 variables.
Observation 7.2.2. Variable sharing in self-referential systems. Assuming a connected computational graph, an architecture that updates all its variables $\phi \in \mathbb{R}^{N_\phi}$ in iteration t needs to reuse elements of ϕ multiple times in the computational graph to generate $\phi_{t+1} \in \mathbb{R}^{N_\phi}$ from $\phi_t \in \mathbb{R}^{N_\phi}$. Importantly, the connectivity constraint applies to the subgraph whose nodes represent entries of ϕ, and not to the graph induced by the additional inputs x.

Proof sketch. By the pigeonhole principle, as there are no more elements in ϕ_t than there are in ϕ_{t+1}, any operation generating an element in ϕ_{t+1} that makes use of more than one element in ϕ_t needs to reuse an element already in use by a different operation. □

Implementations Under the previous constraints, various implementations for self-referential neural architectures are conceivable. Schmidhuber [1993b] assigns an address to each weight such that the network outputs can be used to attend to weights and both read and write their values. Instead of updating one weight at a time, the fast weight programmers (FWPs) of 1992-93 [Schmidhuber, 1992d, 1993c] are networks that learn to generate key and value patterns to rapidly change many fast weights simultaneously (although not all FWPs are fully self-referential). Outer products between activations (a type of variable sharing) are used to derive $M * N$ variables, $M, N \in \mathbb{N}$, from $M + N$ variables. This allows for updating all the weights of a neural network layer by its own activations [Irie et al., 2021a]. Alternatively, a coordinate-wise mechanism may generate all updates continuously as a function of the weight address [D'Ambrosio and Stanley, 2007]. Other works have used multiple RNNs with shared weights and messaging passing between those to increase the number of time-varying variables h arbitrarily while keeping the number of parameters θ constant [Rosa et al., 2019b; Kirsch and Schmidhuber, 2021]. Such systems can be made self-referential by using a subset of h to update parameters θ.

7.3 Method: Fitness Monotonic Execution

In the previous section, we discussed possible substrates for building a recursively self-improving system. We concluded that both memory-based architectures (in-context learners) and fully self-referential architectures are sufficient

from a representational perspective, but identified that an appropriate initialization of the free (initial) variables is required.

Next, we propose a method, *fitness monotonic execution* (FME), that avoids explicit meta optimization of these free variables. Instead of modifying ϕ directly using a human-engineered learning algorithm, we simply select between different configurations of ϕ that are generated using self-modifications. In particular, through interactions with the environment we continuously add new solutions to a set of $\Phi = \{\phi_i\}$. Computation time is distributed across solutions monotonic in their performance, i.e. better performing solutions are executed longer (or are selected for execution more frequently). This can be formalized as a pmf $p(\phi)$ that assigns each solution $\phi \in \Phi$ a probability for being executed at any given time-step based on its average reward $\frac{R(\phi)}{\Delta t}$ relative to other solutions (where Δt is the solution's total lifetime). This can be further normalized by the frequency of solutions (e.g. the performance of solutions determines the probability, not their identity) to prevent many bad performing solutions to dominate over few good ones. As a special case, p may put all probability mass on the current best solution, greedily selecting for improvement. See Algorithm 12 and Figure 7.2 for a full description.

As fitness monotonic execution does not prescribe in a fixed scheme how any solutions are modified – solutions determine this themselves – fully self-referential architectures are well suited. Compared to conventional meta-optimizers, if any variables were not self-modified when employing FME, their value would never be changed and remain fixed to their initial value.

Algorithm 12 Fitness monotonic execution

Require: Initial solution(s) $\Phi = \{\phi_i\}$, self-referential architecture g_ϕ, probability $p(\phi)$, an RL environment E

while forever **do**

- $\phi \sim p(\phi)$ where $\phi \in \Phi$ ▷ Sample next solution to execute, monotonic in its performance
- $\phi, y_{1:L} \leftarrow g_\phi^L(x_{1:L})$ ▷ Execute g for L steps with $x_{1:L}$ from the environment E including a feedback signal
- $\Phi \leftarrow \Phi \cup \{\phi\}$ ▷ Add new ϕ to Φ

Least-recently-used Buffer To limit the number of solutions we need to store, we implement fitness monotonic execution with a least-recently-used (LRU)

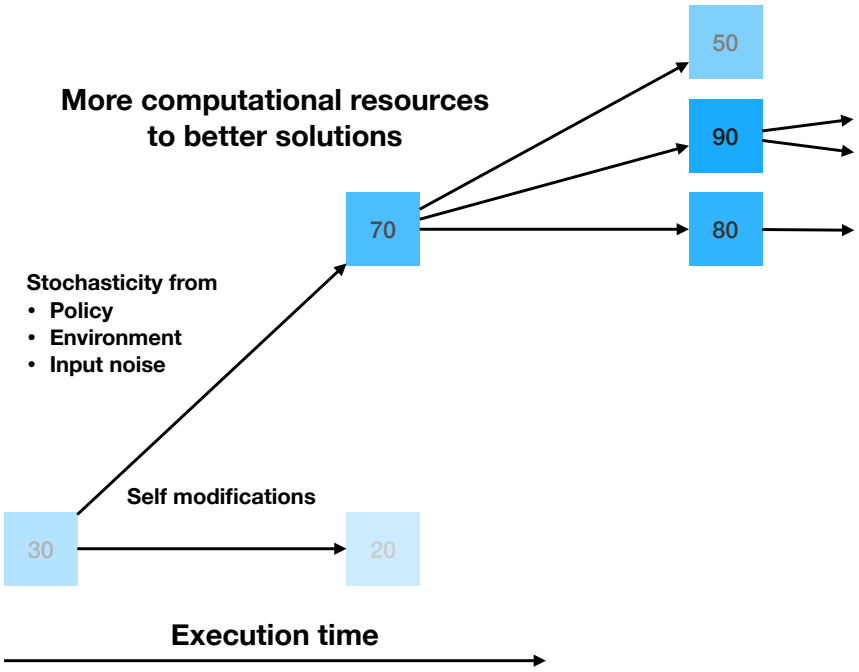


Figure 7.2: **In fitness monotonic execution (FME) the system self-modifies from a random initialization. Better performing solutions are executed longer (or selected more frequently).** In effect, solutions self-modify their behavior, learning, meta learning, meta meta learning, etc without a fixed scheme for meta optimization prescribing how parameters are updated.

buffer. It consists of m buckets where each bucket holds recent solutions in a specific performance range. Solutions from buckets with higher performance are sampled exponentially more frequently.

Outer-product-based Architecture For the self-referential neural network architecture, we chose an outer-product mechanism adapted from prior work [Irie et al., 2021a]. By applying a weight matrix $W_{t-1} \in \mathbb{R}^{N_x \times (N_y+2N_x+4)}$ to some input $x_t \in \mathbb{R}^{N_x}$ we generate the output $y_t \in \mathbb{R}^{N_y}$, key $k_t \in \mathbb{R}^{N_x}$, query $q_t \in \mathbb{R}^{N_x}$, and a learning rate $\beta_t \in \mathbb{R}^4$. Using an outer-product, the key and query generate an update to the weight matrix W_{t-1}, obtaining W_t:

$$y_t, k_t, q_t, \beta_t = W_{t-1}\psi(x_t) \tag{7.1}$$
$$\bar{v}_t = W_{t-1}\psi(k_t) \tag{7.2}$$
$$v_t = W_{t-1}\psi(q_t) \tag{7.3}$$
$$W_t = W_{t-1} + \sigma(\beta_t)(\psi(v_t) - \psi(\bar{v}_t)) \otimes \psi(k_t) \tag{7.4}$$

where ψ is the tanh activation, σ is the sigmoid function, and $\otimes$ is the outer product. The learning rate $\beta_t \in \mathbb{R}^4$ controls the rate of update to the four parts generating y, k, q, β. We stack multiple such self-referential layers.

7.4 Experiments

We empirically investigate several questions: Firstly, starting with a randomly initialized solution, can the network modify itself to solve a bandit task? We compare this to a hill climbing strategy where ϕ are modified by $\phi_{t+1} = \phi_t + \epsilon$ with variance-tuned Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$ and selection is equivalent to FME. Secondly, how do self-modifications compare when solving a markov decision process? Thirdly, given a bandit task that is non-stationary, can the network learn to modify itself based on the reward it receives as input? Refer to Section E.1 for implementation details.

Learning a Bandit Policy The first question we investigate is whether a randomly initialized self-referential architecture is capable of making self-modifications that lead to a useful policy for a given task. We test this on a simple 2-armed bandit where one arm gives payouts (rewards) of 1 and the other 0. From Figure 7.3 (left) we observe that after around 40 self-modifications and selections a solution is found that always selects the arm with a higher payoff. We compare this to hill climbing with a variance-tuned Gaussian noise on

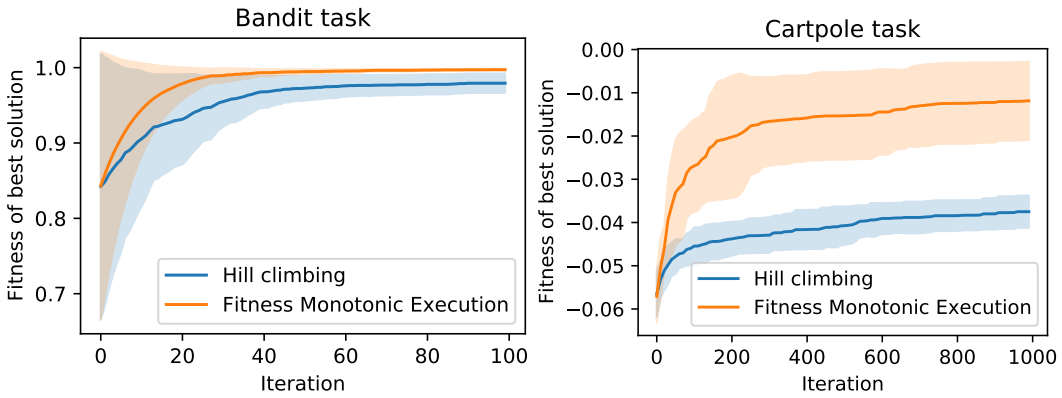


Figure 7.3: **Self-modifications lead to policy improvement and improve future improvements.** A randomly initialized self-referential architecture makes modifications to itself to solve a two-armed bandit problem (left). On a Cartpole task (right) the self-modifications not only directly improve the policy, but also improve future improvements, resulting in faster learning compared to hill climbing. The found policy balances the pole for about 100 steps. Standard deviations are shown for 5 seeds.

the network parameters. Hill-climbing is a natural baseline, as it proposes new solutions using a fixed scheme (instead of self-modifications) and then chooses the best solution. Thus, it validates the usefulness of self-modifications. To match this baseline closely to FME, we simply replace self-modifications with fixed variance-tuned Gaussian noise and keep the selection scheme identical. We observe that in this simple environment, fitness monotonic execution is as effective as hill climbing to find an optimal solution to this bandit problem.

Cartpole Next, we increase the difficulty of the policy to be found by running a self-referential network on the Cartpole task (Figure 7.3, right). We observe that reaching a good performing policy takes significantly more self-modifications and selections. At the same time, a simple hill climbing strategy (with tuned noise) fails at improving the policy at the same rate as the self-modifying architecture. This suggests that we are not only selecting for good policies but also strategies for self-modification that lead to policy improvement in the future.

Meta Learning a Bandit Learning Algorithm Given a non-stationary task, a good policy can not exhibit a fixed behavior but must adapt to changing rewards (learn). We test the capabilities of fitness monotonic execution to adapt

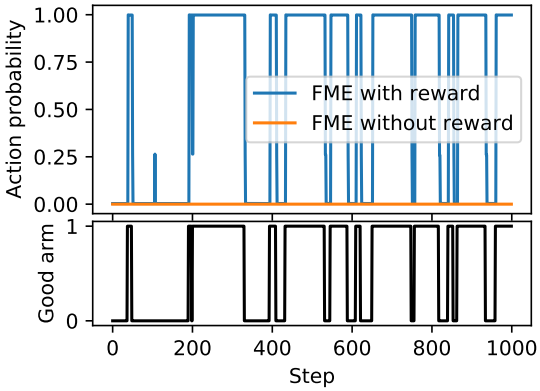


Figure 7.4: **Learning to learn (to learn?)** Given the reward as input, self-modifications enable adaptation to swapping of the good arm in a two-armed bandit.

to a changing bandit task. In Figure 7.4 we swap the good and bad arm at random intervals. We further feed the reward as an input to the policy such that it can adapt its behavior based on the reward. We observe that fitness monotonic execution leads to self-modifying policies that change their action (learn) in response to the reward they previously received. In contrast, if this reward is not fed as an input, the policy fails to adapt.

7.5 Related work

In-context Learning and Fast Weight Programmers In-context learning [Hochreiter et al., 2001; Wang et al., 2016; Duan et al., 2016; Brown et al., 2020] (or memory-based meta learning) describes neural networks that learn-to-learn purely within their activations. This is enabled through the inclusion of a feedback signal or demonstration in the network inputs [Schmidhuber, 1993b]. Previous work has demonstrated the capabilities of neural networks to encode human-engineered learning algorithms such as backpropagation [Linnainmaa, 1970] purely in the forward-pass of neural networks or to discover more efficient learning algorithms from scratch [Kirsch and Schmidhuber, 2021]. Closely related to in-context learning are fast weight programmers (FWPs) that learn to update weights explicitly [Schmidhuber, 1992d, 1993c; Miconi et al., 2018; Schlag et al., 2021b]. The principles that connect in-context learning with learned weight updates are parameter sharing and multiplicative interactions [Kirsch and Schmidhuber, 2021; Kirsch et al., 2022a]. Almost all in-context learners to date are not fully self-referential and require (meta-)parameters to be explicitly meta-optimized by known learning algorithms.

Self-referential Meta Learning and Recursive Self-Improvement Recursively self-improving systems enhance their design, which in turn accelerates their improvements in a positive feedback loop. The Gödel machine is such a system which is based on provably optimal self-modifications [Schmidhuber, 2009]. In this work, modifications must not be proven to be optimal, but are instead computationally prioritized based on their performance. For a system to change all its properties, self-referential meta learning makes all variables (weights and activations) time-varying and modifiable to the neural network itself [Schmidhuber, 1993b; Irie et al., 2021a; Flennerhag et al., 2021]. As we have discussed in this work, both self-referential neural networks as well as in-context learners can in principle implement learning, meta learning, meta meta learning, etc in their neural network dynamics. This still requires explicit meta optimization of some (initial) parameters. In contrast, we introduced fitness monotonic execution (FME) to allow self-modifications to govern (meta-)learning without explicit meta optimization. A related algorithm to this is the success story algorithm (SSA) [Schmidhuber, 1994a; Wiering and Schmidhuber, 1996b; Schmidhuber et al., 1997]. In this algorithm, regular checkpoints of the learner are created and self-modifications are reverted when the average reward decreases. Different from our approach, this process is sequential and keeps only a single history of self-modifications. This potentially limits parallelizability on modern hardware and results in slower exploration.

Portfolio Algorithms Portfolio algorithms [Huberman et al., 1997] are algorithmically related to FME, although view the selection of the algorithm/solution as the 'meta learning' problem [Gagliolo and Schmidhuber, 2011]. This is different from FME where the meta learning can happen within the algorithms themselves and the aim is to put minimal assumptions/biases into the execution of solutions (portfolio schedule).

7.6 Discussion

Limitations This chapter represents an initial discussion of recursively self-improving systems that do not rely on fixed human-engineered meta optimization. The empirical evaluation is still minimalistic at this time but should be a good starting point for larger experiments and improvements. Our intention is to incite further interest in this research direction. Whether FME qualifies as a (minimalistic, gradient-free) meta-optimizer remains an open semantic question. Conventional optimizers prescribe how solutions are adjusted in response

to an external feedback signal; by contrast, the modifications produced by FME are self-generated by the system, thereby reducing the designer-imposed bias typical of conventional meta-optimizers.

Alternative initializations While the initial $\Phi = \{\phi_i\}$ can be set randomly, initial (meta-)learning progress may be slow. Instead, human-engineered meta-optimization can be used initially and then switched to fitness monotonic execution. Alternatively, we may also initialize our self-referential meta-learner with a human-engineered learning algorithm such as backpropagation [Kirsch and Schmidhuber, 2021]. This may speed up the initial learning process and allow the system to bootstrap itself to a more efficient learning algorithm. The self-referentialness of the system may allow it to reverse any initial inductive bias that was introduced by the human-engineered initialization.

Recursively self-improving LLM-based systems A useful initialization to bootstrap recursive self-improvement may also be a large language model (LLM) (Figure 7.5). The model can self-improve in various ways. In the simplest case, the LLM may self-modify by running the model in inference mode, updating its context as it goes. This context can be used to accumulate knowledge and skills. Assuming that an LLM is a universal computer [Schuurmans et al., 2024], in principle context updates are sufficient to cover all possible solutions and learning algorithms. Beyond this, the LLM could also generate code for new tools that when called can update the context, change the code (control flow) that defines how the LLM is called, or train new models that when called inject results into the context. Finally, such a self-improving LLM could even be entirely self-referential where the LLM writes machine learning code that produces updates to its own parameters or replacement. We further discuss this in the next Chapter 8 on automating AI research with AI Scientists. Indeed, since first writing this chapter, researchers have taken the first steps to prompt LLMs to update LLM prompts (context) [Fernando et al., 2023] or write code to improve their own functioning [Zelikman et al., 2023]. While these approaches are still limited in their expressivity and are not yet leading to fully open-ended recursive self-improvement, these are important first steps in this direction. One significant limitation of Zelikman et al. [2023] is that compared to FME, there is no guarantee that self-modifications are actually maximizing the return. Improvements purely rely on the in-context learning behavior of the LLM.

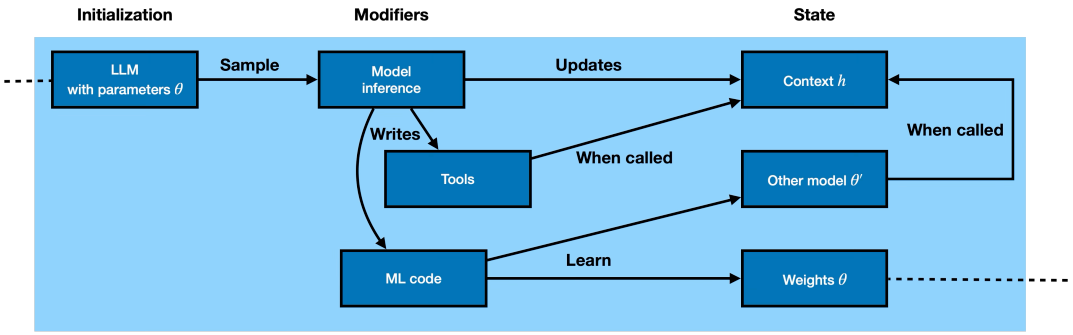


Figure 7.5: **LLMs may enable various kinds of different self-referential self-improving systems.**

Self-modifying Architecture We described a meta learner that can self-modify all its variables including those that define the self-modifications, but its architecture is still hard-coded. While many architectures are computationally universal [Siegelmann and Sontag, 1991], modifications may still be useful for efficiency reasons [Miller et al., 1989; Elsken et al., 2019]. In fitness monotonic execution, the self-modifications do not require differentiability. Thus, self-modifications can be extended to include architecture modifications $\phi, y, g \leftarrow g_\phi(x)$, such as adding or removing neurons and weights, changing operations, and (un-)sharing variables.

Communication between Solutions/Agents Solutions are executed entirely isolated in the present experiments. Through fitness monotonic execution, they compete for computational time. Related to approaches simulating artificial life [Langton, 1997], this may be extended to collaboration and other interactions through the addition of communication channels between solutions (agents).

Where does the reward come from? In this chapter, we assumed a fixed reward signal from the environment. For open-ended recursive self-improvement, this reward signal will need to be generated internally, such as by driving the system through artificial curiosity [Schmidhuber, 1991a], rewarding the learning progress about the environment. This could for instance be framed as automatically making scientific discoveries (Chapter 8) that lead to information gain about the world. A natural alternative formulation of reward is the acquisition of compute resources, naturally tying an agent's ability to obtain and use more computational resources to its capacity to self-improve. Furthermore, in

natural evolution, organisms that found or built their niche are not necessarily required to further increase their fitness. To build a similar fitness monotonic execution system, the increase in compute with more reward may flatten out at some point, encouraging exploration and diversity of solutions. This is related to minimal criterion research in the open-endedness community [Brant and Stanley, 2017].

7.7 Conclusion

In this chapter, we discussed recursively self-improving systems that reduce our reliance on human engineering to the largest extent possible. In particular this means avoiding the use of human engineered learning algorithms on the meta level. We showed that in order to construct systems that can change all their parameters (or variables more generally), functionality needs to be reused. This is done in the form of parameter (variable) sharing. We further demonstrated the representational equivalence between neural networks in-context learning with memory and self-referential architectures while highlighting the benefit of self-referential architectures in the absence of meta optimization. Then, we proposed fitness monotonic execution (FME), a simple approach to avoid explicit meta optimization and to ensure long-term improvement. A neural network self-modifies to solve bandit and classic control tasks, improves its self-modifications, and learns how to learn, purely by assigning more computational resources to better performing solutions. While this system self-improved in a self-referential way, it does not yet exhibit recursive self-improvement in an open-ended manner. Instead, it converges to a fixed solution, which given the limited environment complexity, is expected. We believe that this work opens up a new research direction towards self-referential and recursively self-improving systems on the path to Artificial Super Intelligence. Finally, we discussed future recursively self-improving systems based on large language models (LLMs).

Chapter 8

What's next: Leveraging LLMs to automate AI research

Keywords *LLMs, automated science, in-context learning, self-improvement*

Automating AI research with reasoning in natural language In this thesis, we have so far focused on meta-learning approaches that operate with neural representations to automate AI research, such as objective functions and in-context learning. An alternative to this is leveraging human abstractions, such as natural language and code. While meta-optimization of symbolic abstractions is challenging with conventional learning algorithms [Schmidhuber, 1987; Real et al., 2020], recent breakthroughs with large language models (LLMs) enable us to approach this problem in a novel way: LLMs have been pre-trained on a vast amount of human artifacts, such as scientific articles and code. In this way, LLMs likely already possess the same abstractions and reasoning processes that humans employ during scientific research, thus enabling them to *generate novel AI algorithms*.

In-context learning, a significant subject of this thesis, plays a crucial role in automating AI research with LLMs. Instead of using in-context learning directly as the 'discovered' learning algorithm, it can serve as the inner loop of a language-based automated scientist. In this way, language-based in-context learning creates another level of meta-learning – performing scientific research by writing code for experimentation, reasoning about hypotheses, and improving based on the results via in-context learning. This is achieved by storing previous insights in the context of the LLM. Thus, it *improves* its scientific artifacts throughout the

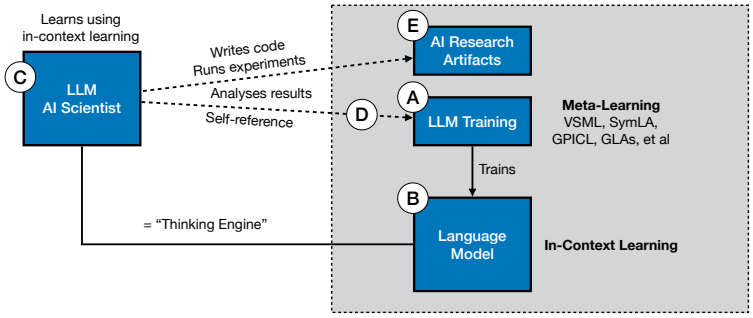


Figure 8.1: **In-context learning as the foundation for scientific reasoning.** The AI Scientist adds a complementary layer of learning on top of in-context learning. (A) (Meta-)training of LLMs results in general in-context learners with useful priors about the nature of research. (B) These LLMs then serve as the thinking engine for the (C) AI Scientist, which in-context learns from experimentation and observation. This, in turn, feeds into new hypotheses and experiments. (D) Such experiments may be self-referential, in that they improve the AI Scientist or its underlying LLM. (E) Or they may involve other AI research that does not directly improve the AI Scientist itself.

research process. Figure Figure 8.1 illustrates this concept.

The LLM-based AI Scientist Such a system could imitate the entire scientific process of humans, from generating hypotheses, designing experiments, running them, and analyzing the results, to writing a research report – not just in AI research but in any field of science. This is visualized in Figure 8.2. Such an AI Scientist closely mirrors the goals of meta-learning and automated machine learning, but with the added benefit of imitating human reasoning and processing, bootstrapping from the vast amount of human knowledge stored in LLMs and the scientific literature. Looking at automating AI research from this perspective, *artificial curiosity* [Schmidhuber, 1991a; Herrmann et al., 2022] now becomes a key component. What ideas, hypotheses, and experiments should the AI Scientist generate next? According to the principle of artificial curiosity, the AI Scientist should generate hypotheses and experiments that maximize its learning progress (information gain). Even without explicitly modeling such information gain, LLMs naturally encode principles of human curiosity [Zhang et al., 2023a].

How to build the AI Scientist? To develop an LLM-based AI Scientist capable of automating AI research, the model could undergo a training process that

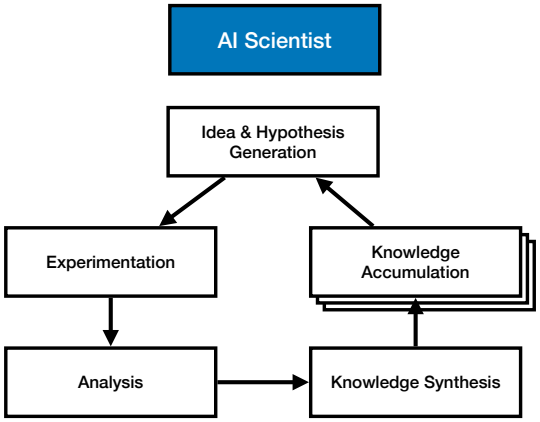


Figure 8.2: **The AI Scientist loop.** The AI Scientist generates hypotheses, designs experiments, runs them, and analyzes the results. It then writes a research report, updates its stored knowledge, and generates new hypotheses.

mirrors the development of large language models (LLMs), involving prompting, pre-training, fine-tuning, and reinforcement learning.

The initial phase would focus on prompting: existing LLMs can be directed to perform AI research tasks by providing them with structured prompts. This stage allows us to assess the model's baseline capabilities in analyzing and generating knowledge based on existing knowledge.

In the next phase, a pre-trained LLM could undergo fine-tuning on datasets containing historical research project information from large repositories. These datasets may include research processes documented in code repositories, meetings, text chats, LaTeX files, experimental logs, and experiment/project tracking software. This stage would help the model better understand the sequential nature of research, capturing the evolution of scientific ideas and experimental practices.

Finally, reinforcement learning (RL) could be used to optimize the model further. Reward metrics could be derived from experimental results, with possible metrics including standardized benchmarks (performance on a specific task), scientist self-generated benchmarks, or LLM-generated reviews of automated research reports or intermediate findings. An artificial curiosity based reward signal could also be employed, encouraging the model to explore hypotheses and experiments that maximize its learning progress.

The AI Scientist may initially rely on and learn from human interventions while

its capabilities are still limited, and over time shift toward open-ended research automation across fields of science.

Example use cases We envision an AI Scientist that can be utilized in various ways to enhance and automate research processes. For instance, given the experimental code and training metrics of a research project, the system can suggest code modifications to stabilize, accelerate, or climb these metrics, such as changes in the optimizer, hyperparameters, neural architecture, or regularizers. It can also provide automated suggestions for experimental design changes at both the conceptual and code levels, based on previous research projects in the training distribution. Additionally, the AI Scientist can generate new ideas and hypotheses based on prior research or the current project state, create experimental prototypes from project descriptions, and automatically scale research project prototypes to larger data and compute resources by generating code modifications informed by the evolution of trained-on research. Furthermore, it is capable of fully automating the end-to-end research process, from research question and hypothesis generation, through code writing and empirical evaluation, to the completion of a research report. Humans may intervene at any point in this process to steer the automated research toward areas of interest.

Recursive self-improvement with LLMs In our previous description of the AI Scientist, we have mostly focused on automating the field of AI research. Why is AI research the most important field of science to be automated by AI? The capabilities of the AI Scientist are directly driven by progress in AI research itself. Even more so, the AI Scientist may directly conduct research on its own code base and the foundation model it is built upon. Thus, initial progress on the AI Scientist may quickly lead to positive feedback loops where the AI Scientist's capabilities accelerate over time. This is a form of recursive self-improvement, as previously discussed in Chapter 7. There are a range of possible self-referential loops that the AI Scientist may engage in to improve itself:

1. **Scientist Code & Prompting (Scaffold)** The AI Scientist may improve its own code base and prompting strategies. It may directly call LLMs to generate such code and prompt updates, or it may develop new software tools or machine learning models that generate such updates.

2. **LLM Context (Knowledge)** A major mechanism through which the AI Scientist learns and improves is through the accumulation of knowledge and feeding it back in the context of the LLM (in-context learning). This may

be in the form of notes, research reports, code snippets, experiment logs, or any other artifacts that the AI Scientist generates during its research process. The accumulation of this knowledge may lead to an increasingly more generally capable AI Scientist.

3. **LLM Parameters** The AI Scientist may also directly research and implement model training algorithms that build an improved version of its underlying LLM.

How is the AI Scientist related to general-purpose meta-learning? The AI Scientist is a natural continuation of the research agenda on general-purpose meta-learning. The previous sections in this thesis have focused on search spaces where the learning algorithms are neural networks (such as discovering in-context learning algorithms). While in principle this search space is Turing-complete, operating in the same search space as human researchers enables re-using the abstractions that humans use, potentially speeding up discovery significantly. It also acts as a regularizer, where findings need to be compressed into natural language, code, and mathematical formalism, further improving generalization.

Initial empirical evidence in this space While this chapter presents a conceptual framework, there have been some concurrent empirical studies that offer preliminary evidence supporting the feasibility of this approach. ML-Agent-Bench and MLE-Bench [Huang et al., 2023; Chan et al., 2024] demonstrate the capabilities and limitations of current LLMs to follow the workflow of a machine learning engineer by writing and iterating on code. Other works investigate the automated development of new 'LLM Agents', i.e., control flows that call LLMs in arbitrary ways, based on LLMs that optimize graphs [Zhuge et al., 2024] or code [Hu et al., 2024]. Promptbreeder re-programs LLMs by changing their prompts [Fernando et al., 2023] while leaving the control flow fixed. Lu et al. [2024] prompted LLMs to build a proof-of-concept system that follows the scientific method, writing a research report on its findings. In STOP, an evolutionary algorithm runs on itself to optimize its own description, using LLMs as the mutation operator [Zelikman et al., 2023].

Chapter 9

Conclusion

In this thesis, we set out to **automate AI research itself**. We began from the premise that the essential ingredient of Artificial General Intelligence (and the precondition for Artificial Super Intelligence) is not the breadth of capabilities, but the capacity for continual self-improvement, including improvement of the learning algorithm. If AI can reliably improve its own methods, the open-ended process of discovery that is still driven by humans today, could be entirely automated tomorrow.

Our contributions move this goal from vision toward practice. We identified that current meta-learning approaches excel at learning on similar problems with few demonstrations or trials but lack the generalization capabilities to be truly useful as replacements for human-developed learning algorithms. To this end, we have made significant contributions to the emerging field of **general-purpose meta-learning**. Through our work on *MetaGenRL*, *VSML*, *SymLA*, *GPICL*, and *GLAs*, we have demonstrated the effectiveness of meta-learning in generating reinforcement learning algorithms and novel general-purpose learning algorithms for supervised learning. *MetaGenRL* [Kirsch et al., 2020b, Chapter 2] is a gradient-based off-policy meta-reinforcement learning algorithm that leverages a population of DDPG-like agents to meta-learn general objective functions. Unlike related methods, the meta-learned objective functions not only generalize within narrow task distributions but also show similar performance on entirely different tasks while markedly outperforming REINFORCE and PPO. *VSML* [Kirsch and Schmidhuber, 2021, Chapter 3] is a simple principle of weight sharing and sparsity for meta-learning powerful in-context learning algorithms (LAs). Using *learning algorithm cloning*, VSML can learn to implement the backpropagation

algorithm and its parameter updates encoded implicitly in the recurrent dynamics. Furthermore, VSML can meta-learn from scratch supervised LAs that do not explicitly rely on gradient computation and that *generalize to unseen datasets*. In SymLA [Kirsch et al., 2022a, Chapter 4], we identify symmetries that exist in backpropagation-based methods for meta-RL but are missing from black-box methods. By integrating these symmetries into neural networks, SymLA is less prone to overfitting compared to standard in-context learners and demonstrates first signs of generalization across domains in in-context meta-RL. Furthermore, we explore how the task distribution, memory capacity, and other model and training properties affect the in-context LA generalization of Transformers in supervised [GPICL Kirsch et al., 2022b, Chapter 5] learning and explore generalization in reinforcement learning [GLAs Kirsch et al., 2023, Chapter 6].

At the same time, even strong meta-learners have historically depended on human-designed outer objectives and training loops. We proposed a research direction that seeks to **reduce these hard-coded inductive biases even further by allowing a neural network to self-modify** while distributing more computational resources to better-performing solutions. In Chapter 7 [Kirsch and Schmidhuber, 2022b], we investigated fundamental questions about such recursive self-improvement and how to minimize human engineering in such systems. In addition to neural self-modification, we discussed a range of alternative substrates, such as self-modifying code and knowledge. We posit that such approaches form the basis for future open-ended processes of self-improvement on the path to Artificial Super Intelligence (ASI) [Morris et al., 2023].

In Chapter 8, we discussed the concept of a meta-learning system referred to as the **'AI Scientist' that automates AI research by reasoning in natural language and using the scientific method**. It operates in the same language-based search space used by human researchers, proposing hypotheses, writing and editing code, designing experiments, interpreting results, and generating reports. Starting with partial autonomy and human oversight, such systems are poised to close the research loop end-to-end as their competence and reliability grow. The concept of recursive self-improvement, particularly when the AI Scientist performs research on itself, suggests that the AI Scientist could rapidly enhance its own capabilities, creating a positive feedback loop. AI Scientist aligns with the broader agenda of general-purpose meta-learning as discussed in this thesis, leveraging human-like search spaces to accelerate discovery.

Looking forward, two ingredients appear key for a **fully automated research process, and ultimately for achieving ASI**. First, systems must be able to inter-

act with the real world: initially through humans, digital actuators, and simulators, and over time through robotics and other interfaces. In addition, general world models [e.g. Bruce et al., 2024] that capture not only task-specific regularities but broad structure across diverse phenomena may be trained and used as proxies for the real world. Second, we need to scale up recursively self-improving systems such as these discussed in Chapter 7 and Chapter 8, minimizing fixed inductive biases and human intervention. We need guardrails that ensure such systems continue to self-improve in an open-ended manner, while making sure these biases are not limiting automated discovery. In practice, an effective human–AI collaboration loop (one that becomes rarer, lower-friction, and increasingly supervisory) is a crucial stepping stone.

While our focus has been AI research, we expect **the same mechanisms to extend to other sciences**. Systems that automate research such as the AI Scientist (Chapter 8) could be leveraged in fields such as biology, materials science, and the social sciences. In this sense, automating AI research is both a proving ground and a catalyst: the better we automate the process of discovery itself, the closer we move to systems whose defining capability is sustained recursive self-improvement; the hallmark of AGI and the precondition for ASI.

As Jürgen Schmidhuber envisioned, **once we build an AI that can improve itself better than we can, we may retire** [Schmidhuber, 2014]. Until then, the work continues – toward systems that learn to do our research for us, and toward the open-ended future of automated discovery that follows.

Appendix A

Appendix on MetaGenRL

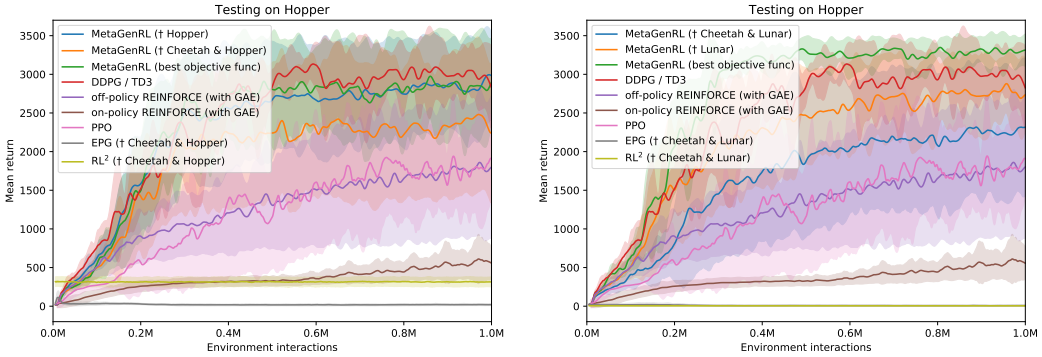
A.1 Additional results

A.1.1 All training and test regimes

In the main text, we have shown several combinations of meta-training, and testing environments. We will now show results for all combinations, including the respective human engineered baselines.

Hopper On *Hopper* (Figure A.1) we find that MetaGenRL works well, both in terms of generalization to previously seen environments, and to unseen environments. The PPO, REINFORCE, RL^2, and EPG baselines are outperformed significantly. Regarding RL^2 we observe that it is only able to obtain reward when *Hopper* was included during meta-training, although its performance is generally poor. Regarding EPG, we observe some learning progress during meta-testing on *Hopper* after meta-training on *Cheetah* and *Hopper* (Figure A.1a), although it drops back down quickly as test-time training proceeds. In contrast, when meta-testing on *Hopper* after meta-training on *Cheetah* and *Lunar* (Figure A.1b) no test-time training progress is observed at all.

Cheetah Similar results are observed in Figure A.2 for *Cheetah*, where MetaGenRL outperforms PPO and REINFORCE significantly. On the other hand, it can be seen that DDPG notably outperforms MetaGenRL on this environment. It will be interesting to further study these differences in the future to improve the expressibility of our approach. Regarding RL^2 and EPG only within distribution

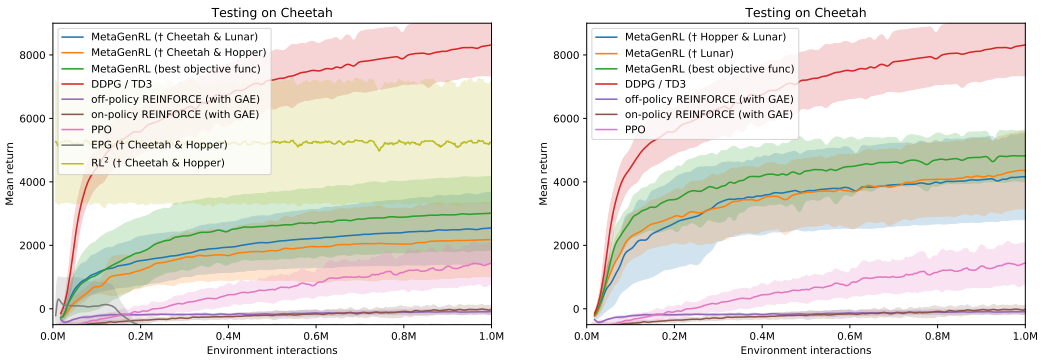


(a) Within distribution generalization. *(b)* Out of distribution generalization.

Figure A.1: Comparing the test-time training behavior of the meta-learned objective functions by MetaGenRL to other (meta) reinforcement learning algorithms on *Hopper*. We consider within distribution testing (a), and out of distribution testing (b) by varying the meta-training environments (denoted by †) for the meta-RL approaches. All runs are shown with mean and standard deviation computed over multiple random seeds (MetaGenRL: 6 meta-train $\times$ 2 meta-test seeds, RL^2: 6 meta-train $\times$ 2 meta-test seeds, EPG: 3 meta-train $\times$ 2 meta-test seeds, and 6 seeds for all others).

generalization results are available due to *Cheetah* having larger observations and / or action spaces compared to *Hopper* and *Lunar*. We observe that RL^2 performs similar to our earlier findings on *Hopper* but significantly improves in terms of within-distribution generalization (likely due to greater overfitting, as was consistently observed for other splits). EPG shows initially more promise on within distribution generalization (Figure A.2a), but ends up like before.

Lunar On *Lunar* (Figure A.3) we find that MetaGenRL is only marginally better compared to the REINFORCE and PPO baselines in terms of within distribution generalization and worse in terms of out of distribution generalization. Analyzing this result reveals that although many of the runs train rather well, some get stuck during the early stages of training without or only delayed recovering. These outliers lead to a seemingly very large variance for MetaGenRL in Figure A.3b. We will provide a more detailed analysis of this result in Section A.1.2. If we focus on the best performing objective function then we observe competitive performance to DDPG (Figure A.3a). Nonetheless, we notice that the objective function trained on *Hopper* generalizes worse to *Lunar*, despite our earlier result that objective functions trained on *Lunar* do in fact general-



(a) Within distribution generalization. *(b)* Out of distribution generalization.

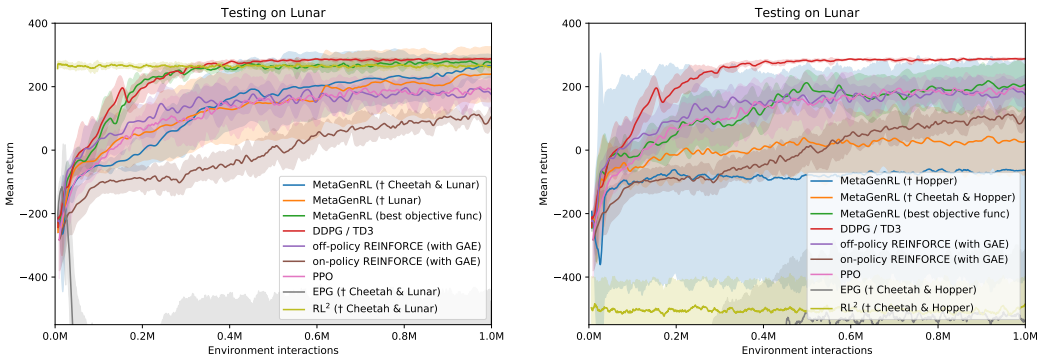
Figure A.2: Comparing the test-time training behavior of the meta-learned objective functions by MetaGenRL to other (meta) reinforcement learning algorithms on *Cheetah*. We consider within distribution testing (a), and out of distribution testing (b) by varying the meta-training environments (denoted by †) for the meta-RL approaches. All runs are shown with mean and standard deviation computed over multiple random seeds (MetaGenRL: 6 meta-train $\times$ 2 meta-test seeds, RL^2: 6 meta-train $\times$ 2 meta-test seeds, EPG: 3 meta-train $\times$ 2 meta-test seeds, and 6 seeds for all others).

ize well to *Hopper*. MetaGenRL is still able to outperform both RL^2 and EPG in terms of out of distribution generalization. We do note that EPG is able to meta-learn objective functions that are able to improve to some extent during test time.

Comparing final scores An overview of the final scores that were obtained for MetaGenRL in comparison to the human engineered baselines is shown in Table A.1. It can be seen that MetaGenRL outperforms PPO and off-/on-policy REINFORCE in most configurations while DDPG with TD3 tricks remains stronger on two of the three environments. Note that DDPG is currently not among the representable algorithms by MetaGenRL.

A.1.2 Stability of learned objective functions

In the results presented in Figure A.3 on *Lunar* we observed a seemingly large variance for MetaGenRL that was due to outliers. Indeed, when analyzing the individual runs meta-trained on *Lunar* and tested on *Lunar* we found that that one of the runs converged to a local optimum early on during training and was



(a) Within distribution generalization. *(b)* Out of distribution generalization.

Figure A.3: Comparing the test-time training behavior of the meta-learned objective functions by MetaGenRL to other (meta) reinforcement learning algorithms on *Lunar*. We consider within distribution testing (a), and out of distribution testing (b) by varying the meta-training environments (denoted by †) for the meta-RL approaches. All runs are shown with mean and standard deviation computed over multiple random seeds (MetaGenRL: 6 meta-train $\times$ 2 meta-test seeds, RL^2: 6 meta-train $\times$ 2 meta-test seeds, EPG: 3 meta-train $\times$ 2 meta-test seeds, and 6 seeds for all others).

unable to recover from this afterwards. On the other hand, we also observed that runs can be 'stuck' for a long time to then make very fast learning progress. It suggests that the objective function may sometimes experience difficulties in providing meaningful updates to the policy parameters during the early stages of training.

We have further analyzed this issue by evaluating one of the objective functions at several intervals throughout meta-training in Figure A.4. From the meta-training curve (bottom) it can be seen that meta-training in *Lunar* converges very early. This means that from then on, updates to the objective function will be based on mostly converged policies. As the test-time plots show, these additional updates appear to negatively affect test-time performance. We hypothesize that the objective function essentially 'forgets' about the early stages of training a randomly initialized agent, by only incorporating information about good performing agents. A possible solution to this problem would be to keep older policies in the meta-training agent population or use early stopping.

Finally, if we exclude four random seeds (of 12), we indeed find a significant reduction in the variance (and increase in the mean) of the results observed for

Table A.1: Agent mean return across multiple seeds (MetaGenRL: 6 meta-train $\times$ 2 meta-test seeds, and 6 seeds for all others) for meta-test training on previously seen environments (cyan) and on unseen (different) environments (brown) compared to human engineered baselines.

	Training (below) / Test (right)	Cheetah	Hopper	Lunar
MetaGenRL (20 agents)	**Cheetah & Hopper**	2185	2433	18
	Cheetah & Lunar	2551	2363	258
	Hopper & Lunar	4160	**2966**	146
	Hopper	3646	2937	-62
	Lunar	4366	2717	244
MetaGenRL (40 agents)	**Lunar & Hopper & Walker & Ant**	3106	2869	201
	Cheetah & Lunar & Walker & Ant	3331	2452	-71
	Cheetah & Hopper & Walker & Ant	2541	2345	-148
PPO	-	1455	1894	187
DDPG / TD3	-	**8315**	2718	**288**
off-policy REINFORCE (GAE)	-	-88	1804	168
on-policy REINFORCE (GAE)	-	38	565	120

Figure A.4: Meta-training with 20 agents on *LunarLander*. We meta-test the objective function at different stages in training on the same environment.

MetaGenRL (see Figure A.5).

A.1.3 Ablation of agent population size and unique environments

In our experiments we have used a population of 20 agents during meta-training to ensure diversity in the conditions under which the objective function needs to optimize. The size of this population is a crucial parameter for a stable meta-optimization. Indeed, in Figure A.6 it can be seen that meta-training becomes increasingly unstable as the number of agents in the population decreases.

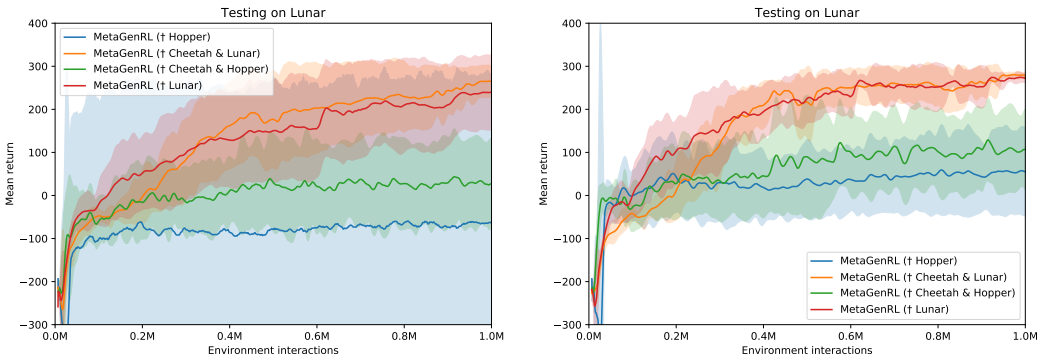


Figure A.5: The left plot shows all 12 random seeds on the meta-test environment *Lunar* while the right has the 4 worst random seeds removed. The variance is now reduced significantly.

Using a similar argument, one would expect to gain from increasing the number of distinct environments (or agents) during meta-training. In order to verify this, we have evaluated two additional settings: Meta-training on *Cheetah & Lunar & Walker & Ant* with 20 and 40 agents respectively. Figure A.7 shows the result of meta-testing on *Hopper* for these experiments (also see the final results reported for 40 agents in Table A.1). Unexpectedly, we find that increasing the number of distinct environments does not yield a significant improvement and, in fact, sometimes even decrease performance. One possibility is that this is due to the simple form of the objective function under consideration, which has no access to the environment observations to efficiently distinguish between them. Another possibility is that MetaGenRL's hyperparameters require additional tuning in order to be compatible with these setups.

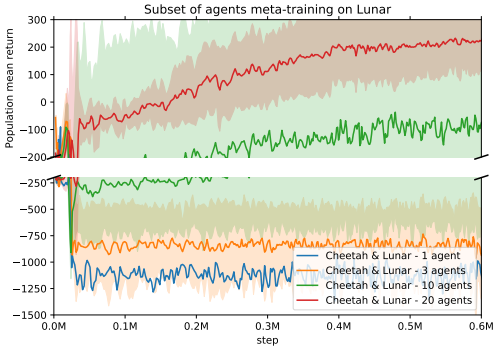


Figure A.6: Stable meta-training requires a large population size of at least 20 agents. Meta-training performance is shown for a single run with the mean and standard deviation across the agent population.

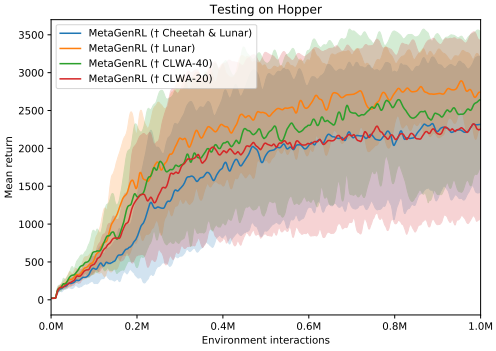


Figure A.7: Meta-training on *Cheetah*, *Lunar*, *Walker*, and *Ant* with 20 or 40 agents; meta-testing on the out-of-distribution *Hopper* environment. We compare to previous VSML configurations.

A.2 Experiment details

In the following we describe all experimental details regarding the architectures used, meta-training, hyperparameters, and baselines. The code to reproduce our experiments is available at `http://louiskirsch.com/code/metagenrl`.

A.2.1 Neural objective function architecture

Neural Architecture In this work we use an LSTM to implement the objective function (Figure 2.2). The LSTM runs backwards in time over the state, action, and reward tuples that were encountered during the trajectory τ under consideration. At each step t the LSTM receives as input the reward r_t, value estimates of the current and previous state V_t, V_{t+1}, the current timestep t and finally the action that was taken at the current timestep a_t in addition to the action as determined by the current policy $\pi_\phi(s_t)$. The actions are first processed by one dimensional convolutional layers striding over the action dimension followed by a reduction to the mean. This allows for different action sizes between environments. Let $A^{(B)} \in \mathbb{R}^{1\times D}$ be the action from the replay buffer, $A^{(\pi)} \in \mathbb{R}^{1\times D}$ be the action predicted by the policy, and $W \in \mathbb{R}^{2\times N}$ a learnable matrix corresponding to N outgoing units, then the actions are transformed by

$$\frac{1}{D}\sum_{i=1}^{D}([A^{(B)}, A^{(\pi)}]^T W)_i, \tag{A.1}$$

where $[a, b]$ is a concatenation of a and b along the first axis. This corresponds to a convolution with kernel size 1 and stride 1. Further transformations with nonlinearities can be added after applying W, if necessary. We found it helpful (but not strictly necessary) to use ReLU activations for half of the units and square activations for the other half.

At each time-step the LSTM outputs a scalar value l_t (bounded between $-\eta$ and η using a scaled tanh activation), which are summed to obtain the value of the neural objective function. Differentiating this value with respect to the policy parameters ϕ then yields gradients that can be used to improve π_ϕ. We only allow gradients to flow backwards through $\pi_\phi(s_t)$ to ϕ. This implementation is closely related to the functional form of a REINFORCE [Williams, 1992] estimator using the generalized advantage estimation [Schulman et al., 2015b].

All feed-forward networks (critic and policy) use ReLU activations and layer normalization [Ba et al., 2016b]. The LSTM uses tanh activations for cell and hidden

state transformations, sigmoid activations for the gates. The input time t is normalized between 0 at the beginning of the episode and 1 at the final transition. Any other hyper-parameters can be seen in Table A.2.

Extensibility The expressability of the objective function can be further increased through several means. One possibility is to add the entire sequence of state observations $o_{1:T}$ to its inputs, or by introducing a bi-directional LSTM. Secondly, additional information about the policy (such as the hidden state of a recurrent policy) can be provided to L. Although not explored in this work, this would in principle allow one to learn an objective that encourages certain representations to emerge, e.g. a predictive representation about future observations, akin to a world model [Schmidhuber, 1990; Ha and Schmidhuber, 2018; Racanière et al., 2017]. In turn, these could create pressure to adapt the policy's actions to explore unknown dynamics in the environment [Schmidhuber, 1991c, 1990; Houthooft et al., 2016; Pathak et al., 2017].

A.2.2 Meta training

Annealing with DDPG At the beginning of meta-training (learning L_α), the objective function is randomly initialized and thus does not make sensible updates to the policies. This can lead to irreversibly breaking the policies early during training. Our current implementation circumvents this issue by linearly annealing $\nabla_\phi L_\alpha$ the first 10k timesteps ($\sim 2\%$ of all timesteps) with DDPG $\nabla_\phi Q_\theta(s_t, \pi_\phi(s_t))$. Preliminary experiments suggested that an exponential learning rate schedule on the gradient of $\nabla_\phi L_\alpha$ for the first 10k steps can replace the annealing with DDPG. The learning rate anneals exponentially between a learning rate of zero and 1e-3. However, in some rare cases this may still lead to unsuccessful training runs, and thus we have omitted this approach from the present work.

Standard training During training, the critic is updated twice as many times as the policy and objective function, similar to TD3 [Fujimoto et al., 2018]. One gradient update with data sampled from the replay buffer is applied for every timestep collected from the environment. The gradient with respect to ϕ in Equation 2.6 is combined with ϕ using a fixed learning rate in the standard way, all other parameter updates use Adam [Kingma and Ba, 2014] with the default parameters. Any other hyper-parameters can be seen in Table A.2 and Table A.3.

Using additional gradient steps In our experiments (Figure 2.5.2) we analyzed the effect of applying *multiple* gradient updates to the policy using L_α before applying the critic to compute second-order gradients with respect to the objective function parameters. For two updates, this gives

$$\begin{aligned} \nabla_\alpha Q_\theta(s_t, \pi_{\phi^\dagger}(s_t)) \text{ with } \phi^\dagger = \phi' - \nabla_{\phi'} L_\alpha(\tau_1, x(\phi'), V) \\ \text{and } \phi' = \phi - \nabla_\phi L_\alpha(\tau_2, x(\phi), V) \end{aligned} \tag{A.2}$$

and can be extended to more than two correspondingly. Additionally, we use disjoint mini batches of data τ: τ_1, τ_2. When updating the policy using $\nabla_\phi L_\alpha$ we continue to use only a single gradient step.

A.2.3 Baselines

RL^2 The implementation for RL^2 mimics the paper by Duan et al. [Duan et al., 2016]. However, we were unable to achieve good results with TRPO [Schulman et al., 2015a] on the MuJoCo environments and thus used PPO [Schulman et al., 2017] instead. The PPO hyperparameters and implementation are taken from rllib [Liang et al., 2018]. Our implementation uses an LSTM with 64 units and does not reset the state of the LSTM for two episodes in sequence. Resetting after additional episodes were given did not improve training results. Different action and observation dimensionalities across environments were handled by using an environment wrapper that pads both with zeros appropriately.

EPG We use the official EPG code base `https://github.com/openai/EPG` from the original paper [Houthooft et al., 2018]. The hyperparameters are taken from the paper, $V = 64$ noise vectors, an update frequency of $M = 64$, and 128 updates for every inner loop, resulting in an inner loop length of 8196 steps. During meta-test training, we run with the same update frequency for a total of 1 million steps.

PPO & On-Policy REINFORCE with GAE We use the tuned implementations from `https://spinningup.openai.com/en/latest/spinningup/bench.html` which include a GAE [Schulman et al., 2015b] baseline.

Off-Policy Reinforce with GAE The implementation is equivalent to VSML except that the objective function is fixed to be the REINFORCE estimator with a GAE [Schulman et al., 2015b] baseline. Thus, experience is sampled from a replay buffer. We have also experimented with an importance weighted unbiased estimator but this resulted in poor performance.

Table A.2: Architecture hyperparameters

Parameter	Value
Critic number of layers	3
Critic number of units	350
Policy number of layers	3
Policy number of units	350
Objective func LSTM units	32
Objective func action conv layers	3
Objective func action conv filters	32
Error bound η	1000

Table A.3: Training hyperparameters

Parameter	Value
Truncated episode length	20
Global norm gradient clipping	1.0
Critic learning rate λ_1	1e-3
Policy learning rate λ_2	1e-3
Second order learning rate λ_3	1e-3
Obj. func. learning rate λ_4	1e-3
Critic noise	0.2
Critic noise clip	0.5
Target network update speed	0.005
Discount factor	0.99
Batch size	100
Random exploration timesteps	10000
Policy gaussian noise std	0.1
Timesteps per agent	1M

DDPG Our implementation is based on `https://spinningup.openai.com/en/latest/spinningup/bench.html` and uses the same TD3 tricks [Fujimoto et al., 2018] and hyperparameters (where applicable) that VSML uses.

Appendix B

Appendix on VSML

B.1 Derivations

Theorem B.1.1. *The weight matrices W and C used to compute VSML RNNs from Equation 3.4 can be expressed as a standard RNN with weight matrix $\tilde{W}$ (Equation 3.5) such that*

$$s_{abj} \leftarrow \sigma(\sum_i s_{abi} W_{ij} + \sum_{c,i} s_{cai} C_{ij}) \tag{B.1}$$

$$= \sigma(\sum_{c,d,i} s_{cdi} \tilde{W}_{cdiabj}). \tag{B.2}$$

The weight matrix $\tilde{W}$ has entries of zero and shared entries given by Equation 3.6.

$$\tilde{W}_{cdiabj} = \begin{cases} C_{ij}, & \text{if } d = a \wedge (d \neq b \vee c \neq a). \\ W_{ij}, & \text{if } d \neq a \wedge d = b \wedge c = a. \\ C_{ij} + W_{ij}, & \text{if } d = a \wedge d = b \wedge c = a. \\ 0, & \text{otherwise.} \end{cases} \tag{3.6 revisited}$$

Proof. We rearrange $\tilde{W}$ into two separate weight matrices

$$\sum_{c,d,i} s_{cdi} \tilde{W}_{cdiabj} \tag{B.3}$$

$$= \sum_{c,d,i} s_{cdi} A_{cdiabj} + \sum_{c,d,i} s_{cdi} (\tilde{W} - A)_{cdiabj}. \tag{B.4}$$

Then assuming $A_{cdiabj} = (d \equiv b)(c \equiv a)W_{ij}$, where $x \equiv y$ equals 1 iff x and y are equal and 0 otherwise, it holds that

$$\sum_{c,d,i} s_{cdi} A_{cdiabj} = \sum_{i} s_{abi} W_{ij}. \tag{B.5}$$

Further, assuming $(\tilde{W} - A)_{cdiabj} = (d \equiv a)C_{ij}$ we obtain

$$\sum_{c,d,i} s_{cdi} (\tilde{W} - A)_{cdiabj} = \sum_{c,i} s_{cai} C_{ij}. \tag{B.6}$$

Finally, solving both conditions for $\tilde{W}$ gives

$$\tilde{W}_{cdiabj} = (d \equiv a)C_{ij} + (d \equiv b)(c \equiv a)W_{ij}, \tag{B.7}$$

which we rewrite in tabular notation:

$$\tilde{W}_{cdiabj} = \begin{cases} C_{ij}, & \text{if } d = a \wedge (d \neq b \vee c \neq a). \\ W_{ij}, & \text{if } d \neq a \wedge d = b \wedge c = a. \\ C_{ij} + W_{ij}, & \text{if } d = a \wedge d = b \wedge c = a. \\ 0, & \text{otherwise.} \end{cases} \tag{B.8}$$

Thus, Equation B.1 holds and any weight matrices W and C can be expressed by a single weight matrix $\tilde{W}$. □

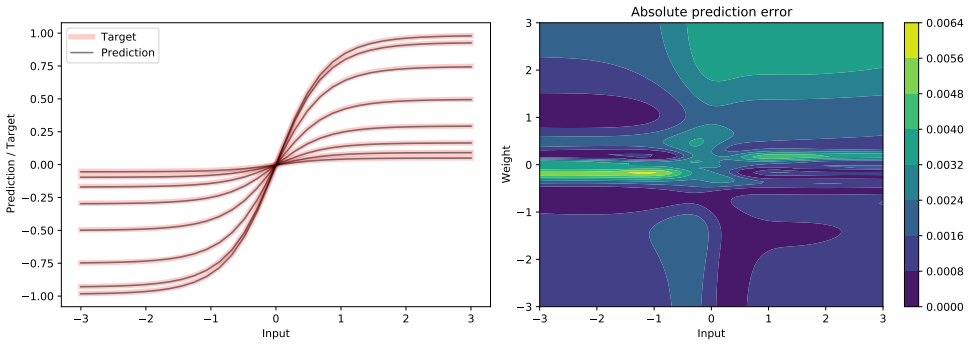


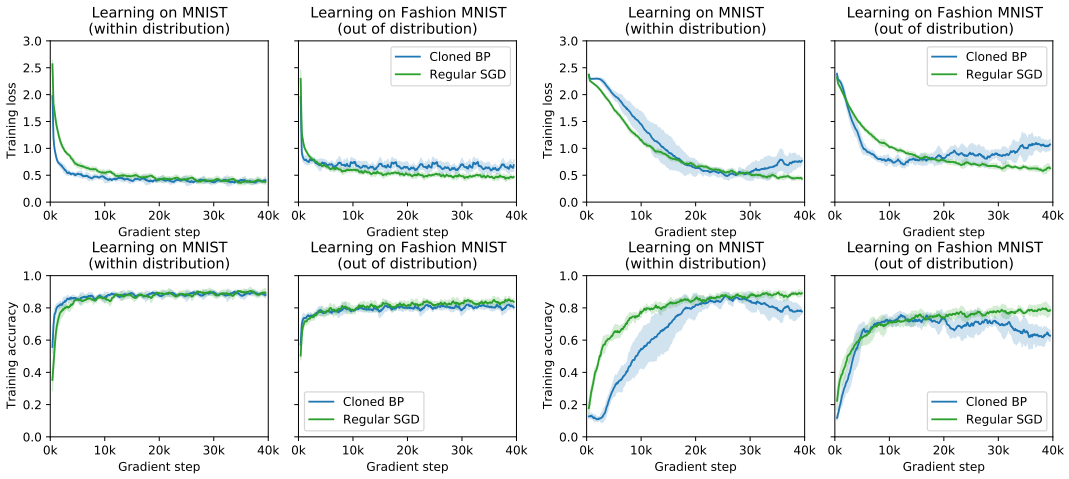
Figure B.1: We are optimizing VSML RNNs to implement neural forward computation such that for different inputs and weights a tanh-activated multiplicative interaction is produced (left), with different lines for different w. These neural dynamics are not exactly matched everywhere (right), but the error is relatively small.

B.2 Additional experiments

B.2.1 Learning algorithm cloning

VSML RNNs can implement neural forward computation In this experiment, we optimize the VSML RNN to compute $y = tanh(x)w$. Figure B.1 (left) shows how for different inputs x and weights w the LSTM produces the correct target value, including the multiplicative interaction. The heat-map (right) shows that low prediction errors are produced but the target dynamics are not perfectly matched. We repeat these LSTMs in line with Equation 3.7 to obtain an 'emergent' neural network.

Learning Algorithm Cloning Curriculum In principle, backpropagation can be simply cloned on random data such that forward computation implements multiplicative activation-weight interaction and backward computation passes an error signal back given previous forward activations. If the previous forward activations are fed as an input one could stack VSML RNNs that implement these two operations to mimic arbitrarily deep NNs. By purely training on random data and unrolling for one step, we can successfully learn on MNIST and Fashion MNIST in the shallow setting. For deeper models, in practice, cloning errors accumulate and input and state distributions shift. To achieve learning in deeper networks we have used a curriculum on random and MNIST data. We first match the forward activations, backward errors, and weight updates for a



(a) Shallow network arrangement. *(b)* Deep (2 layer) network arrangement.

Figure B.2: Additional experiments with VSML RNNs implementing backpropagation. Standard deviations are over 6 seeds.

shallow network. Next, we use a deep network and provide intermediate errors by a ground truth network. Finally, we remove intermediate errors and use the RNN's intermediate predictions that are now close to the ground truth. The final VSML RNN can be used to train both shallow (Figure B.2a) and deep configurations (Figure B.2b).

B.2.2 Meta learning from scratch

Meta testing learning curves & sample efficiency In Figure 3.8 we only showed accuracies after 2k steps. Figure B.3 provides the entire meta test training trajectories for a subset of all configurations. Furthermore, in Figure B.4 we show the cumulative accuracy on the first 100 examples. From both figures, it is evident that learning at the beginning is accelerated compared to SGD with Adam. Also compare with our introspection from Section 3.5.

Ablation: Projection augmentations In the main text (Figure 3.8) we have randomly projected inputs during VSML meta training. When not randomly projecting inputs (Figure B.5), generalization of VSML is slightly reduced. In Figure B.6 we have enabled these augmentations for all methods, including the baselines. While VSML benefits from the augmentations, the in-context RNN, Hebbian fast weights, and external memory baselines do not increase their generalization significantly with those enabled. In Figure B.7 we show meta test training curves

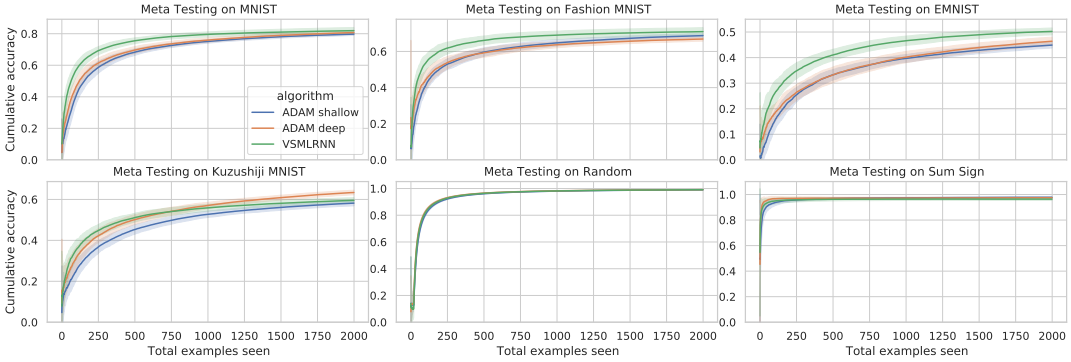


Figure B.3: Meta testing learning curves. All 6 meta test tasks are unseen. VSML RNN has been meta trained on MNIST, Fashion MNIST, EMNIST, KMNIST, and Random, excluding the respective dataset that is being meta tested on. Standard deviations are over 32 seeds.

for both the augmented as well as non-augmented case.

Introspect longer meta test training run Similarly to Figure 3.9, we look at how VSML RNNs learned to learn after meta-training on the MNIST dataset. In this case, we meta-test for 100 steps by sampling from the full MNIST dataset in Figure B.8 without repeating digits. Compared to the previous setup, learning is slower as there is a larger variety of possible inputs. Nevertheless, we observe that VSML RNNs still associate inputs with their label rather quickly compared to SGD.

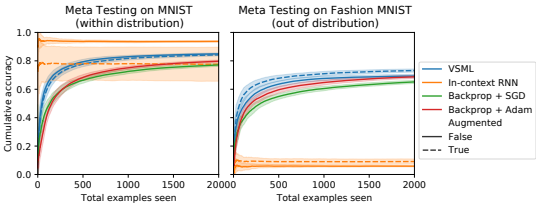


Figure B.7: On the MNIST meta training example from Figure 3.6 we plot the effect of adding the random projection augmentation to VSML and the in-context RNN. The Fashion MNIST performance (out of distribution) is slightly improved for VSML while the effect on the in-context RNN is limited.

Omniglot In this work, we have focused on the objective of meta-learning a general-purpose learning algorithm. Differently from most contemporary meta-learning approaches, we tested the discovered learning algorithm on significantly different datasets to assess its generalization capabilities. These generalization capabilities may affect the performance on standard few-shot benchmarks, such as Omniglot. In this section, we assess how VSML performs on those datasets where the tasks at meta-test time are similar to those during meta-

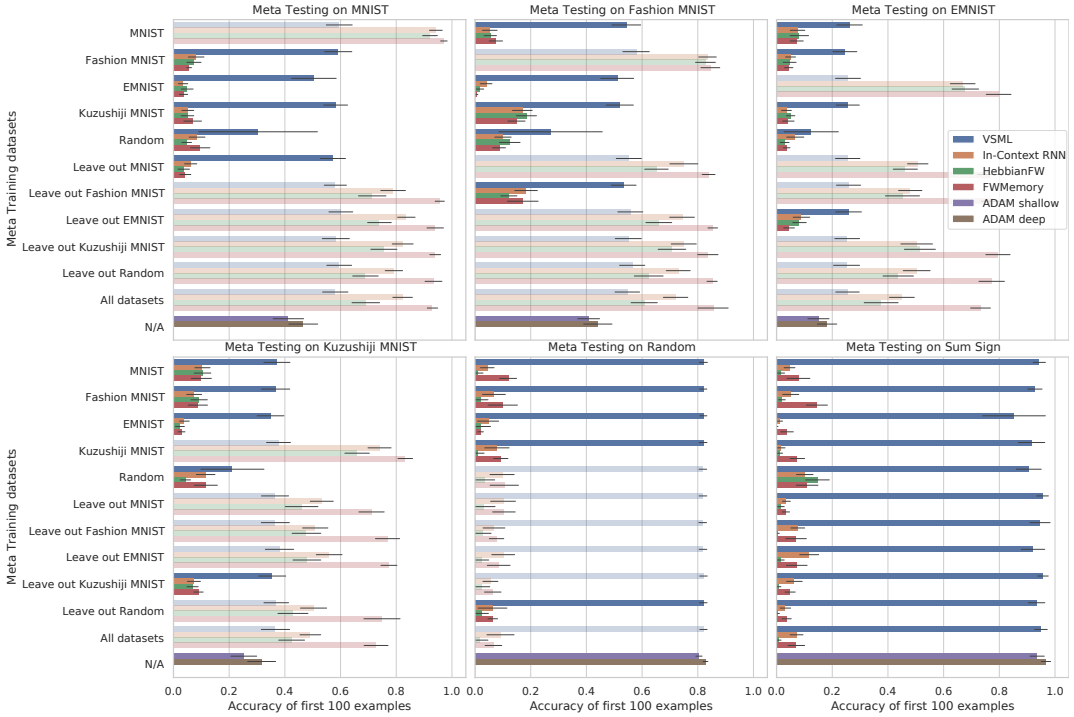


Figure B.4: Online learning on various datasets. Cumulative accuracy in % after having seen **100 training examples** evaluated after each prediction starting with random states (VSML, in-context RNN, HebbianFW, FWMemory) or random parameters (SGD). Standard deviations are over 32 meta test training runs. Meta testing is done on the official test set of each dataset. Meta training is on subsets of datasets excluding the Sum Sign dataset. Unseen tasks, most relevant from a general-purpose LA perspective, are opaque.

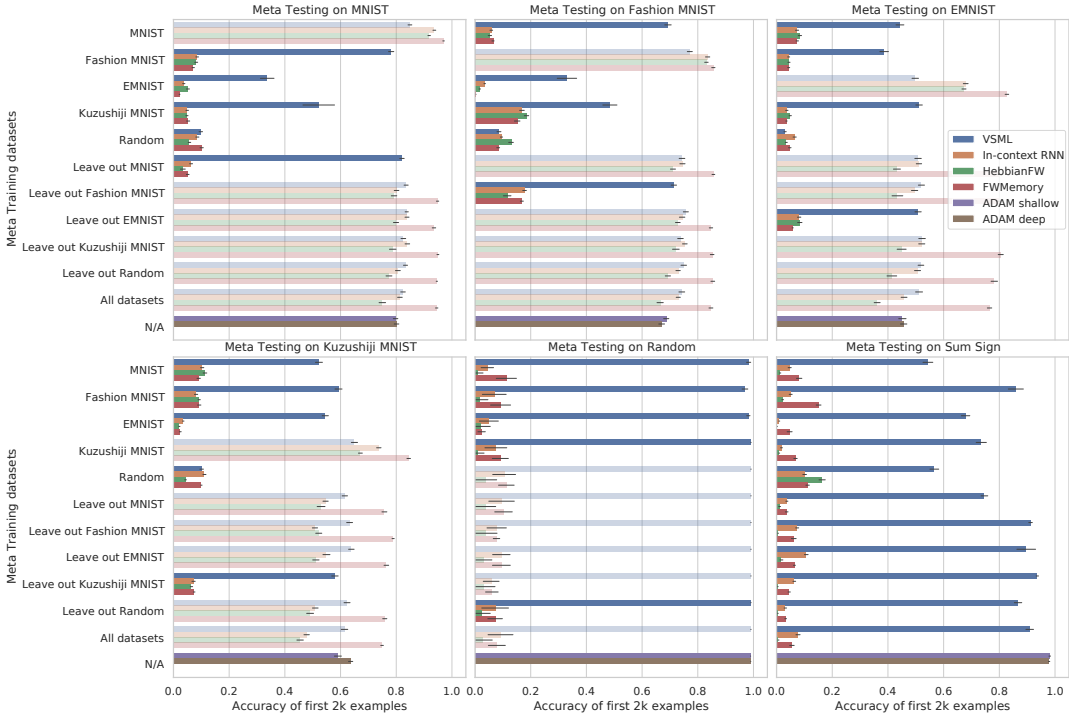


Figure B.5: Same as figure Figure 3.8 and Figure B.4 but with accuracies after having seen 2k training examples and **no random projections for all methods** during meta training.

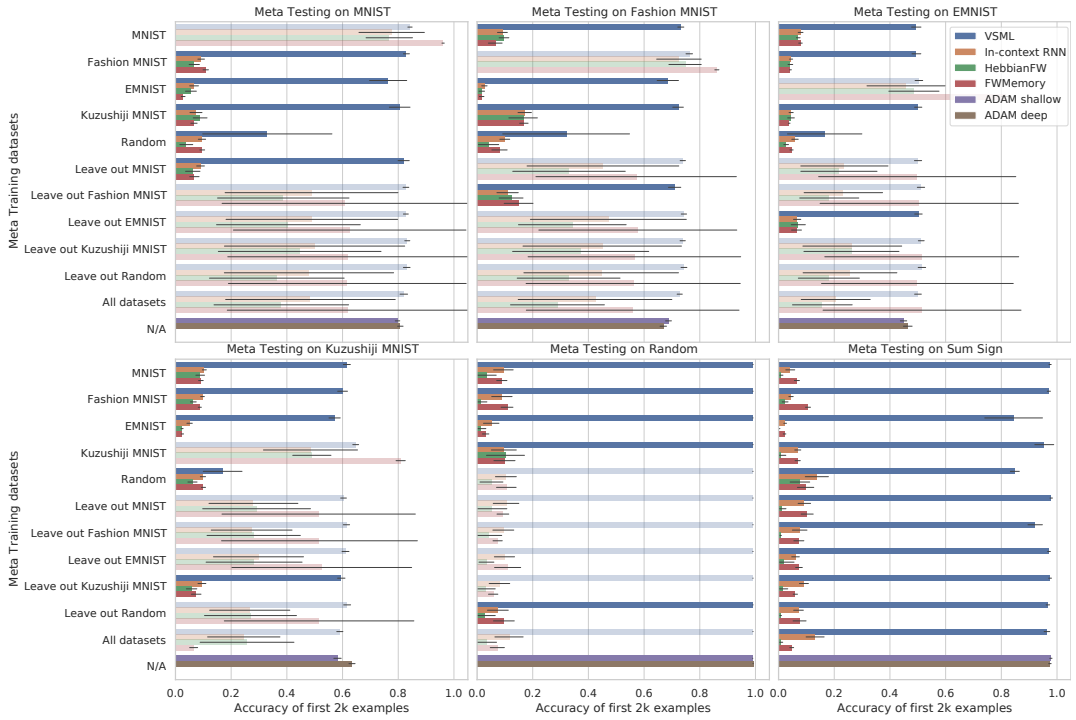


Figure B.6: Same as figure Figure 3.8 and Figure B.4 but with accuracies after having seen 2k training examples and **random projections for all methods including baselines** during meta training.

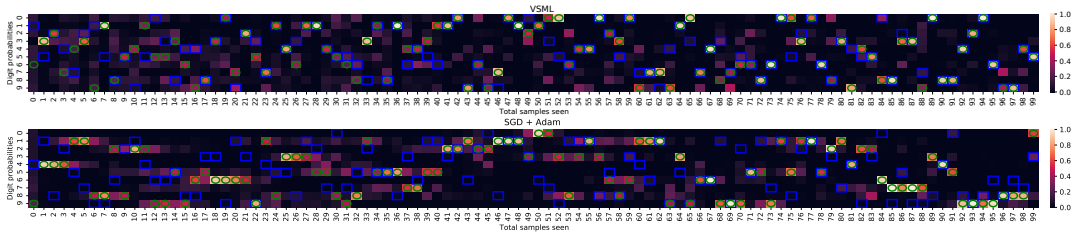


Figure B.8: Introspection of how output probabilities change after observing an input and its error at the output units when meta testing **on the full MNIST dataset**. We highlight the input class □ as well as the predicted class ◯ for 100 examples in sequence. The top plot shows the VSML RNN quickly associating the input images with the right label, generalizing to future inputs. The bottom plot shows the same dataset processed by SGD with Adam which learns significantly slower by following the gradient.

training.

On Omniglot, our experimental setting corresponds to the common 5-way, 1-shot setting [Miconi et al., 2018]: In each episode, we select 5 random classes, sample 1 instance each, and show it to the network with the label and prediction error. Then, we sample a new random test instance from one of the 5 classes and meta-train to minimize the cross-entropy on that example. At meta-test time we use unseen alphabets (classes) from the test set and report the accuracy of the test instance across 100 episodes.

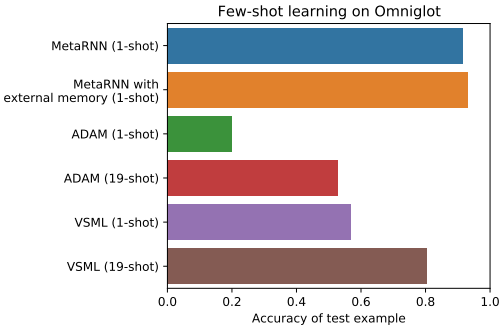


Figure B.9: VSML on the Omniglot dataset.

The results (Figure B.9) nicely demonstrate how common baselines such as the in-context RNN [Hochreiter et al., 2001; Duan et al., 2016; Wang et al., 2016] or a in-context RNN with external memory [Schlag et al., 2021b] work well in an Omniglot setting, yet fail when the gap increases between meta train and meta test, thus requiring stronger generalization (Figure 3.6, Figure 3.8). In contrast, VSML generalizes well to unseen datasets, e.g. Fashion MNIST, although it learns more slowly on Omniglot. Finally, these new results demonstrate how VSML learns significantly faster on Omniglot compared to SGD with Adam, thus highlighting the benefits of the meta-learning approach adopted in this work.

Short horizon bias In this work, we have observed that VSML can be significantly more sample efficient compared to backpropagation with gradient descent, in particular for the first few examples. The longer we unroll the VSML RNNs, the smaller this gap becomes. In Figure B.10 we run VSML for $12,000$ examples ($24,000$ RNN ticks). From this plot, it is evident that at some point gradient de-

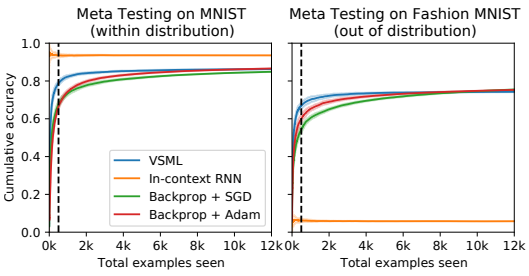


Figure B.10: Short horizon bias.

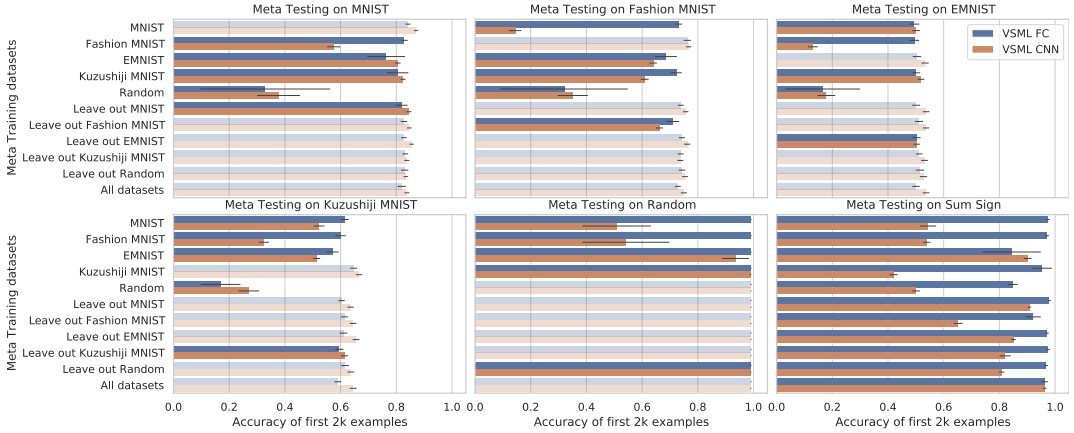


Figure B.11: Convolutions are competitive to the standard fully connected setup.

scent overtakes VSML in terms of learning progress. We call this phenomenon the *short horizon bias*, where meta-test training is fast in the beginning but flattens out at some horizon. In the current version of VSML we only meta-optimize the RNN for 500 examples (marked by the vertical dashed line) starting with a random initialization, not explicitly optimizing learning beyond that point, resulting in this bias. In future work, we will investigate methods to circumvent this bias, for example by resuming from previous states (learning progress) similar to a persistent population in previous meta-learning work [Kirsch et al., 2020b].

Convolutional Neural Networks VSML's sub-RNNs can not only be arranged to fully connected layers, but also convolutions. For this experiment, we have implemented a convolutional neural network (CNN) version of VSML. This is done by replacing each weight in the kernel with a multi-dimensional RNN state and replacing the kernel multiplications with VSML sub-RNNs. We used a convolutional layer with kernel size 3, stride 2, and 8 channels, followed by a dense layer. On our existing datasets, it performs similarly to the fully connected architecture, as can be seen in Figure B.11.

We also applied our CNN variant to CIFAR10. Note that in this work we are interested in the online learning setting (similar to the one of in-context RNNs). This is a challenging problem on which gradient descent with backpropagation also struggles. Many consecutive examples ($> 10^5$ steps) are required for learning. Online performance is generally lower than in the batched setting, which we do not explore here. When meta-training on CIFAR10 (Figure B.12) we observe that meta-test time learning on CIFAR is initially faster compared to SGD

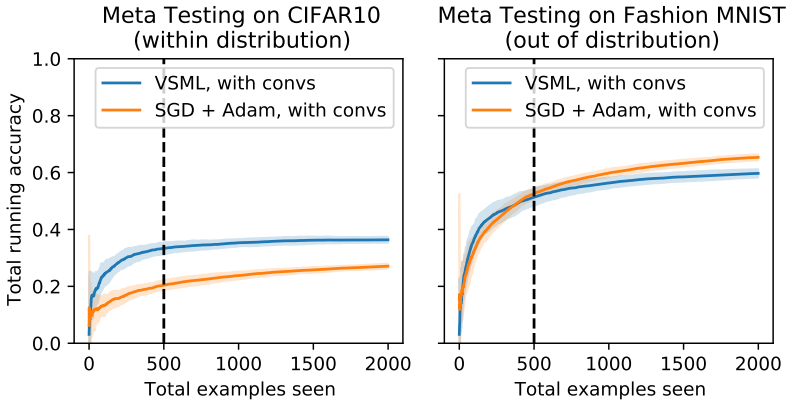


Figure B.12: Meta Training on CIFAR10 with a CNN version of VSML.

while still generalizing to Fashion MNIST. On the other hand, with a sufficiently large meta-training distribution, we would hope to see a similar generalization to CIFAR10 when CIFAR10 is unseen. As is visible in both plots, the learning speed decreases at some point. This is probably due to the current short-horizon bias as discussed in the previous paragraph. Future improvements are necessary to further scale VSML to harder learning problems.

B.3 Other training details

LSTM implementation We implement the VSML RNN using $A \cdot B$ LSTMs with forward and backward messages as described in Equation 3.7. Each LSTM ab at layer k is updated by

$$z_{ab}^{(k)}, h_{ab}^{(k)} \leftarrow f_{\text{LSTM}}(z_{ab}^{(k)}, h_{ab}^{(k)}, \overrightarrow{m}_a^{(k)}, \overleftarrow{m}_b^{(k)}). \tag{B.9}$$

The functions $f_{\overrightarrow{m}}$ and $f_{\overleftarrow{m}}$ are a linear projection to outputs of size $N' = 8$ and $N'' = 8$ respectively. The state size is given by $N = 64$ for LA cloning and $N = 16$ for meta-learning from scratch. $A^{(1)}$ and $B^{(K)}$ are fixed according to the input / output size of the data set, and others are freely chosen as described in the respective experiment. We found that averaging messages instead of summing them, $\overrightarrow{m}_b^{(k)} := \frac{1}{A^{(k-1)}} \sum_{a'} f_{\overrightarrow{m}}(s_{a'b}^{(k-1)})$ and $\overleftarrow{m}_a^{(k)} := \frac{1}{B^{(k+1)}} \sum_{b'} f_{\overleftarrow{m}}(s_{ab'}^{(k+1)})$, improves the stability of meta-training.

Source code is available at `http://louiskirsch.com/code/vsml`.

B.3.1 Learning algorithm cloning

General training remarks During the forward evaluation of layers $1, \ldots, K$ we freeze the LSTM state. During the backward pass, we only retain two state dimensions that correspond to the weight and the bias. We also zero all other LSTM input dimensions in $\overrightarrow{m}$ and $\overleftarrow{m}$ except those that encode the input x and the error e. We maintain a buffer of VSML RNN states from which we sample a batch during LA cloning and append one of the new states to the buffer. This ensures diversity across possible VSML RNN states during LA cloning.

Batching for VSML RNNs In Section 3.3.2 we optimize a VSML RNN to implement backpropagation. To stabilize learning at meta-test time, we run the RNN on multiple data points (batch size 64) and then average their states corresponding to w and b as an analogue to batching in standard gradient descent.

Stability during meta testing To prevent exploding states during meta-testing, we also clip the LSTM state between -4 and 4.

Bounded states in LSTMs In LSTMs the hidden state is bounded between $(-1, 1)$. For learning algorithm cloning, we would like to support weights and biases beyond this range. This can be circumvented by choosing a constant, here 4, by which we scale w and b down to store them in the context state. This is only relevant during learning algorithm cloning.

B.3.2 Meta learning from scratch

Hyperparameter search strategy The VSML hyperparameters were searched using wandb's [Biewald, 2020] Bayesian search during development. Parameters that lead to stable meta-learning on MNIST were chosen. The final parameters were not further tuned, and doing so may lead to additional performance gains. For the in-context RNN we picked parameters that matched VSML RNN as much as possible. For our SGD and SGD with Adam baselines, we performed a grid search on the learning rate on MNIST to find the best learning rates.

Meta Training Meta training is done across 128 GPUs using ES as proposed by OpenAI [Salimans et al., 2017] for a total of 10k steps. We use a population size of 1024, each population member is evaluated on one trajectory of 500 online examples. We use noise with a fixed standard deviation of 0.05. To apply the estimated gradient, we use Adam with a learning rate of 0.025 and betas

set to 0.9 and 0.999. We have run similar experiments (where GPU memory is sufficient) with distributed gradient descent on 8 GPUs, which led to less stable training but qualitatively similar results with appropriate early stopping and gradient clipping.

VSML RNN architecture Each sub-RNN has a state size of $N = 16$ and the messages are sized $N' = N'' = 8$. We only use a single layer between the input and prediction, thus A equals the flattened input image dimension and $B = 10$ for the predicted logits. The outputs are squashed between ± 100 using tanh. We run this layer two ticks per input. The states are initialized randomly from independent standard normals.

SGD baseline architecture and learning rate The deep SGD baseline uses a hidden layer of size 160, resulting in approximately $125k$ parameters on MNIST to match the number of state dimensions of the VSML RNN. We use a tanh activation function to match the LSTM setup. The tuned learning rate used for vanilla SGD is 10^{-2} and 10^{-3} for Adam.

in-context RNN baseline We use an LSTM hidden size of 16 and an input size of $|\text{image}| + |\text{error}|$ where $|\text{error}|$ corresponds to the output size. Inputs are padded to be equal size across all meta-training datasets. This results in about $100k$ to $150k$ parameters.

Hebbian fast weight baseline We compare to a Hebbian fast weight baseline as described in Miconi et al. [2018] where a single layer is adapted using learned synaptic plasticity. A single layer is adapted using Oja's rule by feeding the prediction errors and label as additional inputs.

Specialization through RNN coordinates In addition to the recurrent inputs and inputs from the interaction term, each sub-RNN can be fed its coordinates a, b, position in time, or position in the layer stack. This may allow for (1) specialization, akin to the specialization of biological neurons, and (2) for implicitly meta-learning neural architectures by suppressing outputs of sub-RNNs based on positional information. In our experiments, we have not yet observed any benefits of this approach and leave this to future work.

Meta learning batched LAs In our meta-learning from scratch experiments, we discovered online learning algorithms (similar to in-context RNNs [Hochreiter

et al., 2001; Wang et al., 2016; Duan et al., 2016]). We demonstrated high sample efficiency but the final performance trails the one of batched SGD training. In future experiments, we also want to investigate a batched variant. Every tick we could average a subset of each state s_{ab} across multiple parallel running VSML RNNs. This would allow for meta-learning batched LAs from scratch.

Optimizing final prediction error vs sum of all errors In our experiments we are interested in sample efficient learning, i.e., the model making good predictions as early as possible in training. This is encouraged by minimizing the sum of all prediction errors throughout training. If only good final performance is desired, optimizing solely final prediction error or a weighting of prediction errors is an interesting alternative to be investigated in the future.

Recursive replacement of weights Variable sharing in NNs by replacing each weight with an LSTM introduces new meta variables V_M. Those variables themselves may be replaced again by LSTMs, yielding a multi-level hierarchy with arbitrary depth. We leave the exploration of such hierarchies to future work.

Alternative sparse shared weight matrices In this work, we have focused on a version of VSML where the sparse shared weight matrix is defined by many RNNs that pass messages. Alternative ways of structuring variable sharing and sparsity may lead to different kinds of learning algorithms. Investigating these alternatives or even searching the space of variable sharing and sparsity patterns are interesting directions for future research.

Meta Testing algorithm Meta testing corresponds to unrolling the VSML RNNs. The learning algorithm is encoded purely in the recurrent dynamics. See Algorithm 13 for pseudo-code.

Algorithm 13 VSML: Meta Testing

Require: Dataset $D = \{(x_i, y_i)\}$, LSTM parameters V_M
  $V_L = \{s_{ab}^{(k)}\} \leftarrow$ initialize LSTM states $\quad \forall a, b, k$
  for $(x, y) \in \{(x_1, y_1), \ldots, (x_T, y_T)\} \subset D$ **do** ▷ Inner loop over T examples
    $\overrightarrow{m}_{a1}^{(1)} := x_a \quad \forall a$ ▷ Initialize from input image x
    for $k \in \{1, \ldots, K\}$ **do** ▷ Iterating over K layers
      $s_{ab}^{(k)} \leftarrow f_{RNN}(s_{ab}^{(k)}, \overrightarrow{m}_a^{(k)}, \overleftarrow{m}_b^{(k)}) \quad \forall a, b$ ▷ Equation 3.7
      $\overrightarrow{m}_b^{(k+1)} := \sum_{a'} f_{\overrightarrow{m}}(s_{a'b}^{(k)}) \quad \forall b$ ▷ Create forward message
      $\overleftarrow{m}_a^{(k-1)} := \sum_{b'} f_{\overleftarrow{m}}(s_{ab'}^{(k)}) \quad \forall a$ ▷ Create backward message
    $\hat{y}_a := \overrightarrow{m}_{a1}^{(K+1)} \quad \forall a$ ▷ Read output
    $e := \nabla_{\hat{y}} L(\hat{y}, y)$ ▷ Compute error at outputs using loss L
    $\overleftarrow{m}_{b1}^{(K)} := e_b \quad \forall b$ ▷ Input errors

B.4 Other relationships to previous work

B.4.1 VSML as distributed memory

Compared to other works with additional external memory mechanisms [Sun, 1991; Mozer and Das, 1993; Santoro et al., 2016; Mishra et al., 2018; Schlag et al., 2021b], VSML can also be viewed as having memory distributed across the network. The memory writing and reading mechanism implemented in the meta variables V_M is shared across the network.

B.4.2 Connection to modular learning

Our sub-LSTMs can also be framed as *modules* that have some shared meta variables V_M and distinct learned variables V_L. Previous works in modular learning [Shazeer et al., 2017; Rosenbaum et al., 2018; Kirsch et al., 2018] were motivated by learning experts with unique parameters that are conditionally selected to suit the current task or context. In contrast, VSML has recurrent modules that share the same parameters V_M to resemble a learning algorithm. There is no explicit conditional selection of modules, although it could emerge based on activations or be facilitated via additional attention mechanisms.

B.4.3 Connection to self-organization and complex systems

In self-organizing systems, global behavior emerges from the behavior of many local systems such as cellular automata [Codd, 2014] and their recent neural variants [Mordvintsev et al., 2020; Sudhakaran et al., 2021]. VSML can be seen as such a self-organizing system where many sub-RNNs induce the emergence of a global learning algorithm.

Appendix C

Appendix on SymLA

Algorithm 14 SymLA meta training

Require: Distribution over RL environment(s) $p(e)$

$\theta \leftarrow$ initialize LSTM parameters

while meta loss has not converged **do** ▷ Outer loop in parallel over envs

$e \sim p(e)$ and samples $\phi \sim \mathbb{N}(\phi|\theta, \Sigma)$

$\{h_{ab}\} \leftarrow$ initialize LSTM states $\forall a, b$

$o_1 \sim p(o_1)$ ▷ Initialize environment e

for $t \in \{1, \ldots, L\}$ **do** ▷ Inner loop over lifetime in environment e

$h_{ab} \leftarrow f_{\text{LSTM}}(h_{ab}, o_{t,a}, a_{t-1,b}, r_{t-1}, \overrightarrow{m}_b, \overleftarrow{m}_a) \quad \forall a, b$ ▷ Equation 4.9

$\overrightarrow{m}_b \leftarrow \sum_a f_{\overrightarrow{m}}(h_{ab}) \quad \forall b$ ▷ Create forward messages

$\overleftarrow{m}_a \leftarrow \sum_b f_{\overleftarrow{m}}(h_{ab}) \quad \forall a$ ▷ Create backward messages

$y \leftarrow \overrightarrow{m}_{\cdot 1}$ ▷ Read out action

$a_t \sim p(a_t; y)$ ▷ Sample action from distribution parameterized by y

Send action a_t to environment e, observe o_{t+1} and r_t

$\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}_{\phi \sim \mathcal{N}(\phi|\theta,\Sigma)}[\mathbb{E}_{e \sim p(e)}[\sum_{t=1}^{L} r_t^{(e)}(\phi)]]$ ▷ Update θ using evolution strategies (Equation 4.10)

C.1 Bandits from Wang et al. [2016]

In our experiments, we use bandits of varying difficulty from Wang et al. [2016]. Let p_1 be the probability of the first arm for a payout of $r = 1$, $r = 0$ otherwise, and p_2 the payout for the second arm. Then, we define the

- uniform independent bandit with $p_1 \sim U[0, 1]$ and $p_2 \sim U[0, 1]$,

Table C.1: A comparison between fixed reinforcement learning algorithms (REINFORCE), backpropagation-based meta RL (MAML, MetaGenRL, LPG), in-context learning RNNs, and our in-context learner with symmetries (SymLA). $\pi_\theta^{(s)}$ denotes a *stationary* policy that is updated at fixed intervals by backpropagation.

	REINFORCE	MetaGenRL / LPG	MAML	in-context RNN	SymLA (ours)
Meta variables	/	ϕ	Initial θ_0	θ	θ
Learned variables	θ	θ	θ	RNN state h	RNN states $h_{ab}^{(k)}$
Learning algorithm	fixed loss func L + Backprop	learned loss func L_ϕ + Backprop	fixed loss func L + Backprop	π_θ	π_θ
Policy	$\pi_\theta^{(s)}$	$\pi_\theta^{(s)}$	$\pi_\theta^{(s)}$	π_θ	π_θ
Black-box	✗	✗	✗	✓	✓
Symmetries in LA	✓	✓	✓	✗	✓

- uniform dependent bandit with $p_1 \sim U[0, 1]$ and $p_2 = 1 - p_1$,
- easy dependent bandit with $p_1 \sim U\{0.1, 0.9\}$ and $p_2 = 1 - p_1$,
- medium dependent bandit with $p_1 \sim U\{0.25, 0.75\}$ and $p_2 = 1 - p_1$,
- hard dependent bandit with $p_1 \sim U\{0.4, 0.6\}$ and $p_2 = 1 - p_1$.

C.2 Hyperparameters

C.2.1 SymLA architecture

We use a single recurrent layer, $K = 1$, with a message size of $\overleftarrow{M} = 8$ and $\overrightarrow{M} = 8$. To produce the next state h_{ab} according to Equation 4.9, we use parameter-shared LSTMs with a hidden size of $N = 16$ ($N = 64$ for bandits to match Wang et al. [2016]) and run the recurrent cell for 2 micro ticks.

C.2.2 Meta learning / outer loop

We estimate gradients ∇_θ using evolutionary strategies [Salimans et al., 2017] with 10 evaluations per population sample to estimate the fitness value (100 evaluations for bandits). Then, we apply those using Adam with a learning rate of $\alpha = 0.01$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$ ($\alpha = 0.2$ for bandits). We use a fixed noise standard deviation of $\sigma = 0.035$ ($\sigma = 0.2$ for bandits) and a population size of 512. Our inner loop has a length of $L = 500$ ($L = 100$ for bandits), concatenating multiple episodes. We meta-optimize for $4,000$ outer steps for

bandit experiments, and $20,000$ otherwise.

C.2.3 Generalisation to unseen environments

We apply a random linear transformation (Glorot normal) to environment observations, mapping those to a 16-dimensional vector.

C.3 Scalability and complexity

The computational complexity of the inner loop (and meta testing) is $O(N^2W)$ per environment step, where N is the hidden size of each RNN and W is the number of RNNs (number of parameters in a conventional neural network). N can generally be small, in most experiments $N = 16$ (see Appendix C.2). Space complexity is independent of the number of time-steps and is $O(N^2 + NW + MS)$, where $M = 8$ is the message size and S denotes the number of messages. The computational complexity of meta training highly depends on the chosen meta-optimizer. With ES, each outer optimization step has a complexity of $O(N^2WLPE)$, where $L = 500$ is the length of the evaluated lifetime, $P = 512$ is the size of the particle population (within range of the ES literature), and E is the number of evaluations per particle to estimate the average reward. Space complexity is generally low for gradient-free optimization such as ES; for meta-training it is $O(N^2+NW+MS)$, if the population is evaluated in sequence. Compared to the in-context RNNs (RL^2), SymLA is slower by a factor of N^2 (here $N = 16$) in both meta training and testing if the number of RNNs is chosen to equal the in-context RNN's parameters. In practice, the RNNs also increase the capacity such that fewer RNNs may also be sufficient.

C.4 Code snippet

```
import haiku as hk
import jax
import jax.numpy as jnp
import numpy as np
import optax


class SymlaLayer(hk.Module):

  def __init__(self, input_size: int, output_size: int, msg_size: int, hidden_size: int, micro_ticks: int):
    super().__init__()
    self.input_size = input_size
    self.output_size = output_size
    self.micro_ticks = micro_ticks
    self._lstm = hk.LSTM(hidden_size)
    self._fwd_messenger = hk.Linear(msg_size)
    self._bwd_messenger = hk.Linear(msg_size)
    self._tick = hk.vmap(hk.vmap(self._tick, (0, None, 0, None)), (0, 0, None, None))

  def _tick(self, lstm_state: hk.LSTMState, fwd_msg: jnp.ndarray, bwd_msg: jnp.ndarray, aux: jnp.ndarray):
    inp = jnp.concatenate([fwd_msg, bwd_msg, aux])
    out, lstm_state = self._lstm(inp, lstm_state)
    return out, lstm_state

  def create_state(self):
    lstm_state_shape = (2, self.input_size, self.output_size, self._lstm.hidden_size)
    lstm_state = jnp.zeros(lstm_state_shape)
    lstm_state = hk.LSTMState(hidden=lstm_state[0], cell=lstm_state[1])

    fwd_msg_shape = (self.output_size, self._fwd_messenger.output_size)
    fwd_msg = jnp.zeros(fwd_msg_shape)

    bwd_msg_shape = (self.input_size, self._bwd_messenger.output_size)
    bwd_msg = jnp.zeros(bwd_msg_shape)

    return lstm_state, fwd_msg, bwd_msg

  def __call__(self, state, inp: jnp.ndarray, inp_end: jnp.ndarray, aux: jnp.ndarray):
    lstm_state, fwd_msg, bwd_msg = state

    # Update state
    in_fwd_msg = jnp.concatenate([bwd_msg, inp[:, None]], axis=-1)
    in_bwd_msg = jnp.concatenate([fwd_msg, inp_end[:, None]], axis=-1)
    for _ in range(self.micro_ticks):
      out, lstm_state = self._tick(lstm_state, in_fwd_msg, in_bwd_msg, aux)

    # Update forward messages
    out_fwd_msg = self._fwd_messenger(out).mean(axis=0)
    # Update backward messages
    out_bwd_msg = self._bwd_messenger(out).mean(axis=1)

    # Read out logits for action
    logits = out_fwd_msg[:, 0]

    return logits, (lstm_state, out_fwd_msg, out_bwd_msg)


class SymlaModel(hk.Module):

  def __init__(self):
    super().__init__()
    self._layer = SymlaLayer(...)

  def __call__(self, env, env_state):
    prev_action = jnp.zeros(env.action_shape)
    state = self._layer.create_state()
    rng_ticks = jnp.array(hk.next_rng_keys(env.meta_episode_length))

    def scan_tick(carry, rng_tick):
      env_state, state, prev_action = carry
      rng_tick, rng_action = jax.random.split(rng_tick)

      # Obtain signals from environment
      obs = env.observation(env_state)
      reward = env.reward(env_state)
```

```
      done = env.is_terminal(env_state).astype(jnp.float32)

      # Tick layer
      inp = obs.flatten()
      aux = jnp.stack([reward, done])
      logits, new_state = self._layer(state, inp, prev_action, aux)

      # Create action
      action = jax.random.categorical(rng_action, logits)
      action = jax.nn.one_hot(action, logits.shape[-1])

      # Tick environment
      new_env_state = env.step(rng_tick, env_state, action)
      reward = env.reward(env_state)

      return (new_env_state, new_state, action), reward

    _, rewards = hk.scan(scan_tick, (env_state, state, prev_action), rng_ticks)
    loss = -jnp.mean(rewards)
    return loss, rewards


class Experiment:

  def __init__(self, noise_std: float, population_size: int, learning_rate: float):
    self._model = hk.transform(lambda *x: SymlaModel()(*x))
    self._optimizer = optax.adam(learning_rate)
    self._env = Env()
    self._population_size = population_size
    self._noise_std = noise_std
    self._update_func = jax.jit(self._update_func)

  def _es_eval(self, params, rng, env_state):
    # Extract shapes
    treedef = jax.tree_structure(params)
    shapes = jax.tree_map(lambda p: np.asarray(p.shape), params)

    # Random keys
    rng, param_rng = jax.random.split(rng)
    keys = jax.tree_unflatten(treedef, jax.random.split(param_rng, treedef.num_leaves))

    # Generate noise
    noise = jax.tree_multimap(jax.random.normal, keys, shapes)
    scaled_noise = jax.tree_map(lambda x: x * self._noise_std, noise)

    # Antithetic sampling
    params_pos = jax.tree_multimap(jnp.add, params, scaled_noise)
    params_neg = jax.tree_multimap(jnp.subtract, params, scaled_noise)

    # Evalute in environment
    loss_pos, rewards = self._model.apply(params_pos, rng, self._env, env_state)
    loss_neg, _ = self._model.apply(params_neg, rng, self._env, env_state)

    # Compute grads
    es_factor = (loss_pos - loss_neg) / (2 * self._noise_std ** 2)
    grads = jax.tree_map(lambda x: x * es_factor, scaled_noise)

    return grads

  def _update_func(self, params, opt_state, rng):
    rng, rng_update = jax.random.split(rng)
    grads = self._es_grads(params, rng_update)

    updates, opt_state = self._optimizer.update(grads, opt_state)
    params = optax.apply_updates(params, updates)

    return params, opt_state, rng

  def _es_grads(self, params, rng):
    rng_env_init, rng_eval = jax.random.split(rng)
    rng_env_init = jax.random.split(rng_env_init, self._population_size)
    rng_eval = jax.random.split(rng_eval, self._population_size)

    env_state = jax.vmap(self._env.initial_state)(rng_env_init)
    v_es_eval = jax.vmap(self._es_eval, in_axes=(None, 0, 0))
    grads = v_es_eval(params, rng_eval, env_state)
    grads = jax.tree_map(lambda x: jnp.mean(x, axis=0), grads)

    return grads
```

```
def train(self, seed: int, num_iterations: int):
  rng = jax.random.PRNGKey(seed)
  rng, rng_init = jax.random.split(rng)

  dummy_env_state = self._env.initial_state(rng_init)

  params = self._model.init(rng_init, self._env, dummy_env_state)
  opt_state = self._optimizer.init(params)

  for _ in range(num_iterations):
    params, opt_state, rng = self._update_func(params, opt_state, rng)
```

Appendix D

Appendix on GPICL

D.1 Summary of insights

Insight 1: It is possible to learn-to-learn with black-box models Effective in-context learning algorithms can be realized using black-box models with few inductive biases, given sufficient meta-training task diversity and large enough model sizes. To transition to the learning-to-learn regime, we needed at least $2^{13} = 8192$ tasks.

Insight 2: Simple data augmentations are effective for general learning-to-learn The generality of the discovered learning algorithm can be controlled via the data distribution. Even when large task distributions are not (yet) naturally available, simple augmentations that promote permutation and scale invariance are effective.

Insight 3: The meta-learned behavior has algorithmic transitions When increasing the number of tasks, the meta-learned behavior transitions from task memorization, to task identification, to general learning-to-learn.

Insight 4: Large state is more crucial than parameter count The specific inductive biases of each architecture matter to a smaller degree. The driving factor behind their ability to learn how to learn is the size of their state. Furthermore, this suggests that the model size in terms of numbers of parameters plays a smaller role in the setting of learning-to-learn and Transformers have benefited in particular from an increase in state size by self-attention. In non-meta-learning

sequence tasks parameter count is thought to be the performance bottleneck [Collins et al., 2016]. Beyond learning-to-learn, this likely applies to other tasks that rely on processing and storing large amounts of sequence-specific information.

D.2 Limitations

Varying input and output sizes Compared to many previous works in meta-learning [Andrychowicz et al., 2016; Finn et al., 2017; Kirsch and Schmidhuber, 2021], the discovered learning algorithms are only applicable to an arbitrary input and output size by using random projections. This may make it more difficult to apply the learning algorithm to a new, unseen problem. This problem also applies to Transformers applied to multiple tasks and modalities. Related work has solved this problem by tokenizing inputs to compatible, unified representations [Chowdhery et al., 2022]. We expect these techniques or others to be useful in the learning-to-learn context too.

Processing large datasets Learning algorithms often process millions of inputs before outputting the final model. In the black-box setting, this is still difficult to achieve. Recurrency-based models usually suffer from accumulating errors, whereas Transformers computational complexity grows quadratically in the sequence length. Additional work is required to build models capable of processing and being trained on long sequences. Alternatively, parallel processing, similar to batching in learning algorithms, may be a useful building block.

D.3 The transition to general learning-to-learn

In Figure 5.4 we observe a quick transition from task identification to generalizing learning-to-learn (the second dashed line) as a function of the number of tasks. Previously, Figure 5.2 (c) showed a similar transition from no learning to learning on unseen tasks. What happens during this transition and when do the found solutions correspond to memorizing (task memorization or seen task identification) vs generalizing solutions? To analyze the transition from task identification to general learning to learn, we perform multiple training runs with varying seeds and numbers of tasks on MNIST. This is shown in Figure D.1, reporting the final training loss. We find that the distribution is bi-modal. Solutions at the end of training are memorizing or generalizing. Memorization

cluster: The larger the number of tasks, the more difficult it is to memorize all of them with a fixed model capacity (or learn to identify each task). Generalization cluster: At a certain number of tasks (here 6000), a transition point is reached where optimization sometimes discovers a lower training loss that corresponds to a generalizing learning to learn solution. For larger numbers of tasks the solutions always settle in the generalizing cluster.

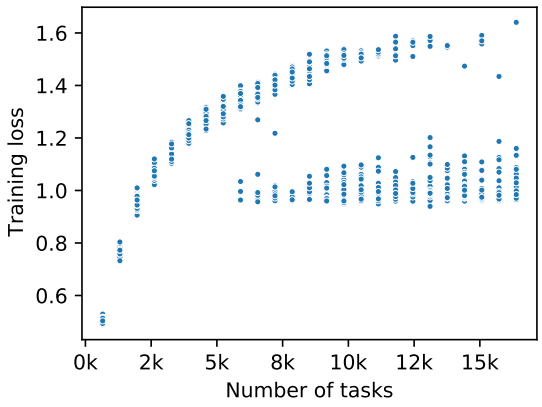


Figure D.1: **Solutions found by GPICL after meta-training are bi-modal, with a memorization and generalization mode.** Each point represents the training loss at the end of meta-training for runs with different seeds and for various numbers of tasks that include the transition boundary previously observed. Almost all solutions are either in a memorization cluster or in a generalization cluster.

D.4 Architectural details and hyperparameters

Transformer details By default, all Transformers have a key, value, and query size of 32, 8 heads, and 4 layers, and model size of $N_M = 256$. The model size defines the dimensionality of each token, and the MLP between layers scales this size up to a hidden representation of $4 \times N_M$ where N_M corresponds to the model size.

Outer-product LSTM We slightly modify an LSTM by replacing the context state with an outer-product update and inner-product read-out.

```
x_and_h = jnp.concatenate([inputs, prev_state.hidden], axis=-1)

gated = hk.Linear(8 * size * self.num_heads)(x_and_h)
gated = gated.reshape((batch_size, self.num_heads, 8 * size))
gated = checkpoint_name(gated, 'gated')

# i = input, g = cell_gate, f = forget_gate,
# q = query, o = output_gate
sizes = (3 * size, 3 * size, size, size)
indices = np.cumsum(sizes[:-1])
k1, k2, q, o = jnp.split(gated, indices, axis=-1)
scale = jax.nn.softplus(
    hk.get_parameter('key_scale', shape=(), dtype=k1.dtype,
                     init=jnp.zeros))
i, g, f = jnp.einsum('bhki,bhkj->kbhij',
                     jax.nn.tanh(split_axis(k1, (3, size))) * scale,
                     jax.nn.tanh(split_axis(k2, (3, size))))
```

```
f = jax.nn.sigmoid(f + 1)  # Forget bias
c = f * prev_state.cell + jax.nn.sigmoid(i) * g
read = jnp.einsum('bhij,bhi->bhj', c, q)
h = hk.Flatten()(jax.nn.sigmoid(o) * jnp.tanh(read))
```

VSML We use a version of VSML with a single layer and self-messages [Kirsch et al., 2022a] of size 8. Each LSTM has a hidden size of 16. For each LSTM update we use two micro-ticks. We train on 2^{25} tasks with a 90% biased permutation distribution. The task batch size is 8. All images are scaled to a size of $32 \times 32 \times 3$

VSML without symmetries Before activations are fed to a standard instantiation of VSML, all inputs are projected using a learnable linear projection. Logits are generated using another linear projection, followed by a softmax. We use a version of VSML with a single layer and self-messages [Kirsch et al., 2022a] of size 8. The LSTMs are on a grid of $k \times k$ LSTMs, where $k \in \{1, 2, 4, 8, 16, 24\}$. Each LSTM has a hidden size of 64. For each LSTM update we use two micro-ticks. We train on 2^{25} tasks with a 90% biased permutation distribution. The task batch size is 128. All images are scaled to a size of 14×14.

LSTM For the results in Table 5.2, we used a hidden size of 256 and 10^5 optimization steps. Larger hidden sizes were harder to optimize. We train on 2^{25} tasks with a 90% biased permutation distribution. The task batch size is 128. All images are scaled to a size of $32 \times 32 \times 3$

D.5 Experimental details

Most experiments can be run on a single GPU, some require 16 GPUs due to sequence length and large batch sizes, with sufficient GPU memory (around 16 GB each). Some experiments, such as Figure 5.2, require up to 1000 runs of that kind to produce the final heat-map.

Input normalization Each dataset is z-normalized by its mean and standard deviation across all examples and pixels.

Number of seeds and shading If not noted otherwise, line plots use 8 seeds for meta-training and at least 512 seeds for meta-testing. Shading indicates 95% confidence intervals.

Random dataset To test the meta-learned learning algorithms on a synthetically generated problem, we generate classification datasets of 10 datapoints where the input $x \in \mathbb{R}^{32\times32\times3}$ is drawn from a uniform distribution between 0 and 1. For each datapoint, labels y are drawn from a uniform categorical distribution of 10 classes.

Figure 5.2 The MLP has two hidden layers of varying size with relu activations. The Transformer has the default parameters as defined above.

Figure 5.3 We use a transformer model with a model size of 256. We train on 2^{25} tasks with a 90% biased permutation distribution. The task batch size is 128. All images are scaled to a size of $32 \times 32 \times 3$ Inputs are z-normalized across the dataset and all input dimensions.

Table 5.2 The SGD baseline was obtained by sweeping over learning rates from 10^{-4} to 0.5, optimizers SGD, Adam and Adam with weight decay, one or two layers, and hidden sizes of 32, 64, or 128 on MNIST. The best configuration (most sample efficient) corresponds to a learning rate of 10^{-3}, Adam, and no hidden layers. SGD performs updates online on each one out of the 100 data points. MAML is equivalent to SGD up to the difference that we meta-train the weight initialization according to Equation 5.2 where θ are the initial parameters of the classifier that is then updated using SGD at meta-test time. All black-box approaches do not use gradient descent at meta-test time. All meta-learning approaches where meta-trained and tuned via grid search on MNIST.

Figure 5.4 Input normalization is disabled.

Figure 5.5 The Transformer uses a task batch size of 512.

Figure 5.6 Trained on 2^{16} tasks generated from FashionMNIST with labels fully permuted.

Figure 5.7 Trained on 2^{16} tasks generated from FashionMNIST with labels fully permuted.

Figure 5.8 Trained on 2^{16} tasks generated from FashionMNIST with label permutations varied.

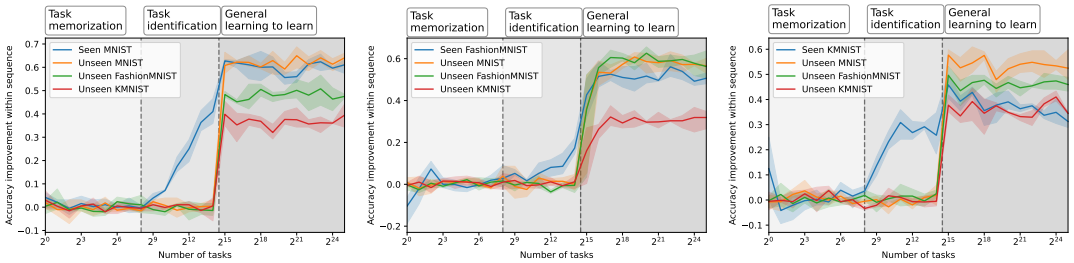


Figure D.2: **Transformers exhibit three different phases in terms of meta-learned behavior on various meta training datasets.** (1) When training on a small number of tasks, tasks are memorized. (2) Tasks from the training distribution are identified, which is evident as a within-sequence increase of performance. (3) When training across many tasks, we discover a learning algorithm that generalizes to unseen tasks and unseen datasets.

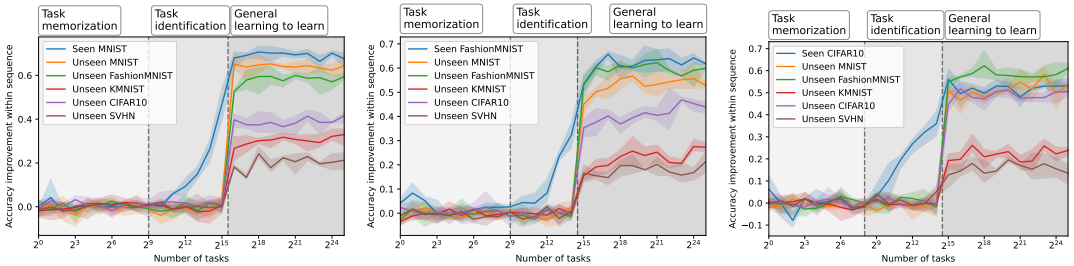


Figure D.3: **The algorithmic transitions also happen when using the embeddings from Section 5.4.4.** This enables faster learning on datasets such as CIFAR10 with only 100 training examples while still generalizing to various datasets.

Figure D.1 We trained a Transformer with model size 64 and 32 seeds for each number-of-tasks-configuration.

D.6 Additional experiments

Algorithmic transitions on other meta training datasets In Figure 5.2 and Figure 5.4 we observe a quick transition between task identification and general learning-to-learn as a function of the number of tasks. We show these transitions on more meta training datasets in Figure D.2. When using ImageNet embeddings as discussed in Section 5.4.4, we observe similar transitions also on CIFAR10 and other datasets as shown in Figure D.3.

Meta-test loss changes in algorithmic transitions We have observed algorithmic transitions across various datasets. In Section D.3 we observed that solutions found by GPICL after meta-training cluster into two groups of task memorization/identification and general learning-to-learn. As the number of tasks increases, more meta-training runs settle in the generalization cluster. A similar behavior can be observed for meta-test losses (on the final predicted example) in Figure D.4. There is a visible transition to a much lower meta-test loss at a certain number of tasks on MNIST and KMNIST. During this transition, separate meta-training runs cluster into two separate modes. Also compare with Figure D.2 and Figure D.3. On FashionMNIST, this transition appears to be significantly smoother but still changes its 'within sequence learning behavior' in three phases as in Figure D.2.

CLIP embeddings and mini-Imagenet In addition to the ImageNet embeddings from Section 5.4.4, we have also conducted experiments with CLIP [Radford et al., 2021] embeddings and mini-Imagenet. In these experiments (see Figure D.5), we first project inputs into a latent space with a pre-trained CLIP model (ViT-B-32 laion2b_s34b_b79k) and then proceed as before, randomly projecting these features, and training a GPICL Transformer on top. We add the mini-ImageNet dataset in these experiments and use a 10-way 10-shot setting to ensure the same number of classes across datasets and a similar sequence length to previous experiments. We observe strong and generalizable in-context learning when leveraging these pre-trained embeddings, without meta-training on unseen datasets.

Large State is Crucial for Learning We show that for learning-to-learn the size of the memory N_S at meta-test time (or state more generally) is particularly important in order to be able to store learning progress. We test this by training several architectures with various N_S in our meta-learning setting. In addition to Figure 5.5, Figure D.6 show meta-test performance on more tasks and datasets.

Sequence length In all experiments of the main chapter we have meta-trained on a sequence length (number of examples) of 100. This is a small training dataset compared to many human-engineered learning algorithms. In general, as long as the learning algorithm does not overfit the training data, more examples should increase the predictive performance. In Figure D.7 we investigate how our model scales to longer sequence lengths. We observe that the final accuracy

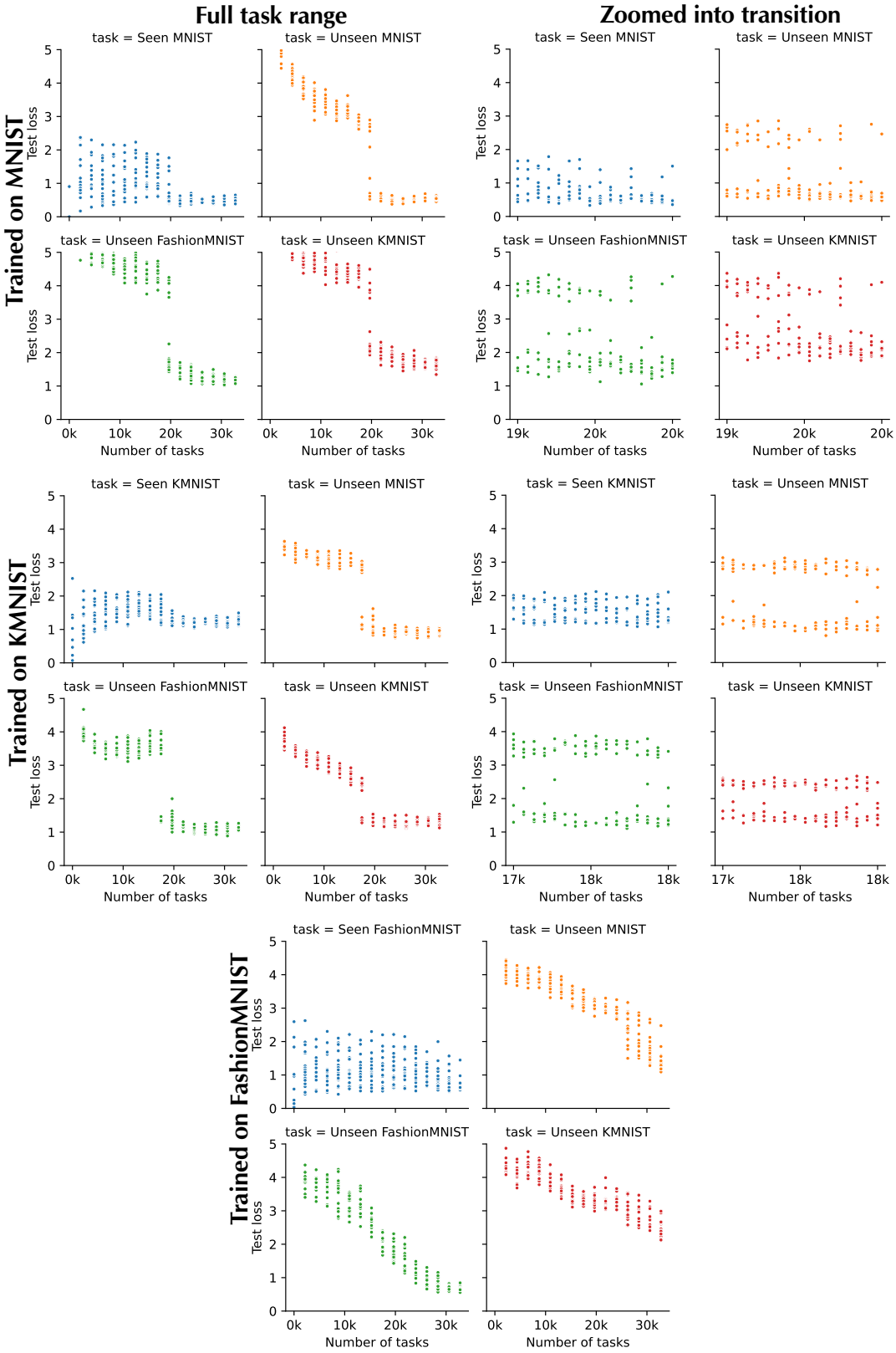


Figure D.4: **The meta-test loss transitions at a certain number of tasks.** Each point represents the meta-test loss on the final predicted example for meta-training runs with different seeds and for various numbers of tasks that include the transition boundary previously observed. There is a visible transition to a much lower meta-test loss at a certain number of tasks on MNIST and KMNIST. The right column zooms into the transition an shows how separate training runs cluster into two separate modes. On FashionMNIST, this transition appears to be significantly smoother.

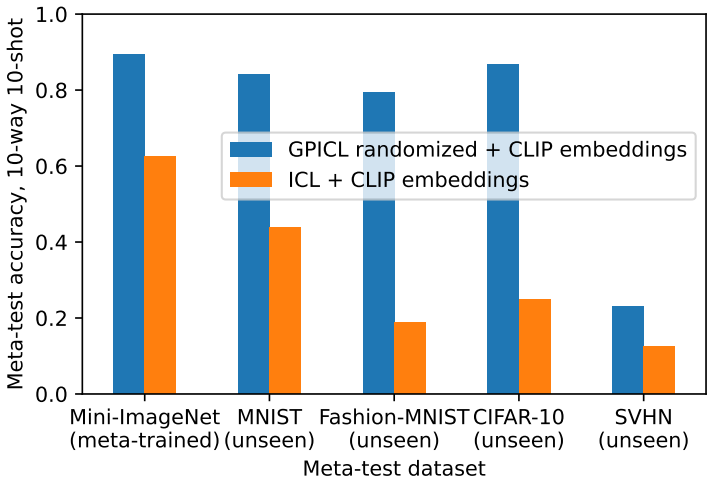


Figure D.5: **CLIP embeddings provide useful domain-specific knowledge that can be leveraged while still generalizing to other datasets** GPICL is meta-trained on mini-Imagenet either directly with CLIP embeddings or with randomly transformed embeddings. CLIP helps to accelerate meta-test-time in-context learning on many datasets, with the exception of SVHN. The learning algorithms still generalize to a wide range of datasets.

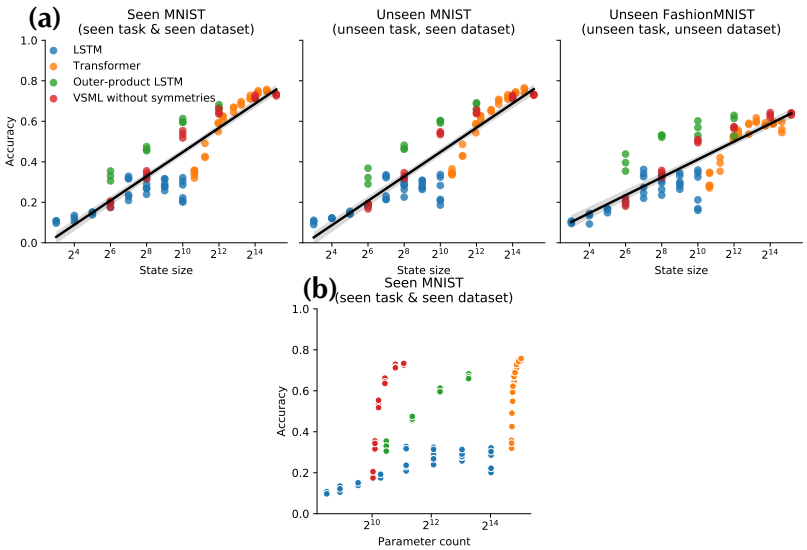


Figure D.6: **The state size (accessible memory) of an architecture most strongly predicts its performance as a general-purpose learning algorithm. (a)** A large state is crucial for learning-to-learn to emerge. **(b)** The parameter count correlates less well with learning capabilities.

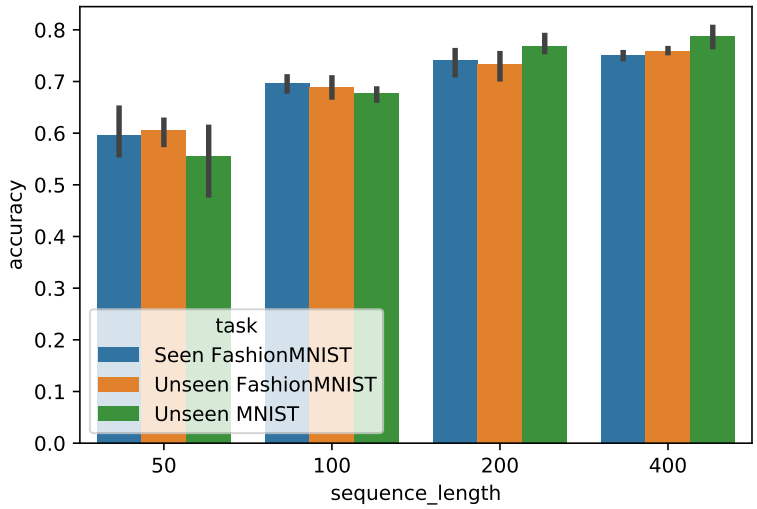


Figure D.7: Increasing the sequence length during meta-training and meta-testing improves the predictive performance of the final query in the sequence. Error bars indicate 95% confidence intervals.

of the last query in the sequence consistently increases with longer sequences. The generalization to longer sequences than those seen during meta-training is another important direction for future work.

Gradient and update statistics To better understand the properties of the loss plateau, we visualize different statistics of the gradients, optimizer, and updates. In Figure D.8, we track the exponential moving average statistics of Adam before the loss plateau and after (dashed vertical line). In Figure D.9 we investigate how gradients differ between settings with a plateau and settings with a biased distribution where the plateau is avoided. We plot the cosine similarity between consecutive optimization steps, the gradient L2-norm, and the similarity and norm of the weight updates after normalization with Adam. The statistics are plotted cumulatively or smoothed with a Gaussian filter for better readability. The gradient and update cosine similarity differ only marginally between cases with a plateau and cases without. We observe that the gradient L2-norm in the plateau is half as big as in the biased distribution case, although the updates that Adam applies are going towards zero. This also results in not moving far from parameter initialization when in the plateau. We hypothesize this has to do with varying gradient norms when looking at individual parameter tensors (Figure D.10). We observe that the gradients have a small norm for most tensors, except for the last layer.

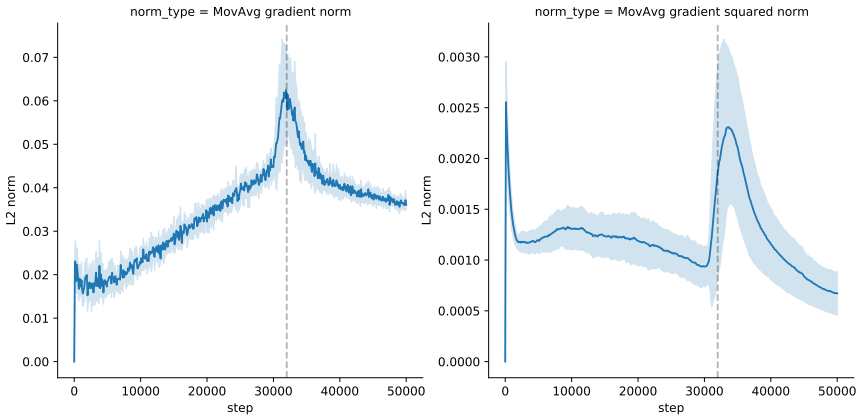


Figure D.8: L2-norms of the gradient and squared gradient exponential moving average in Adam. The dashed line corresponds to the loss drop at the end of the loss plateau.

Batch size and number of tasks influence on plateau length Instead of looking at the plateau length in terms of the number of steps (Figure 5.7), we may also be concerned with the total number of tasks seen within the plateau. This is relevant in particular when the task batch is not processed fully in parallel but gradients are accumulated. Figure D.11 shows the same figure but with the number of tasks in the plateau on the y-axis instead. It can be observed that larger batch-sizes actually increase the data requirement to leave the plateau, despite decreasing the plateau in terms of the number of optimization steps. Similarly, a larger task training distribution requires a larger number of tasks to be seen within the plateau.

Adjusting Adam's ϵ or changing the optimizer As discussed in the main chapter and visualized in Figure D.12b, decreasing ϵ significantly shortens the plateau. This is due to the rescaling of very small gradient magnitudes being limited by ϵ. At the same time it incurs some instability. Directly normalizing the gradient by applying the sign function element-wise (Figure D.12a) to the exponential gradient average shortens the plateau even further.

When memorization happens, can we elicit grokking? In Figure 5.7a we have seen that an insufficiently large task distribution can lead to memorization instead of general learning-to-learn. At the same time, Figure 5.8 showed that biasing the data distribution is helpful to avoid loss plateaus. Power et al. [2022] observed a phenomenon which they called "grokking" in which even after hav-

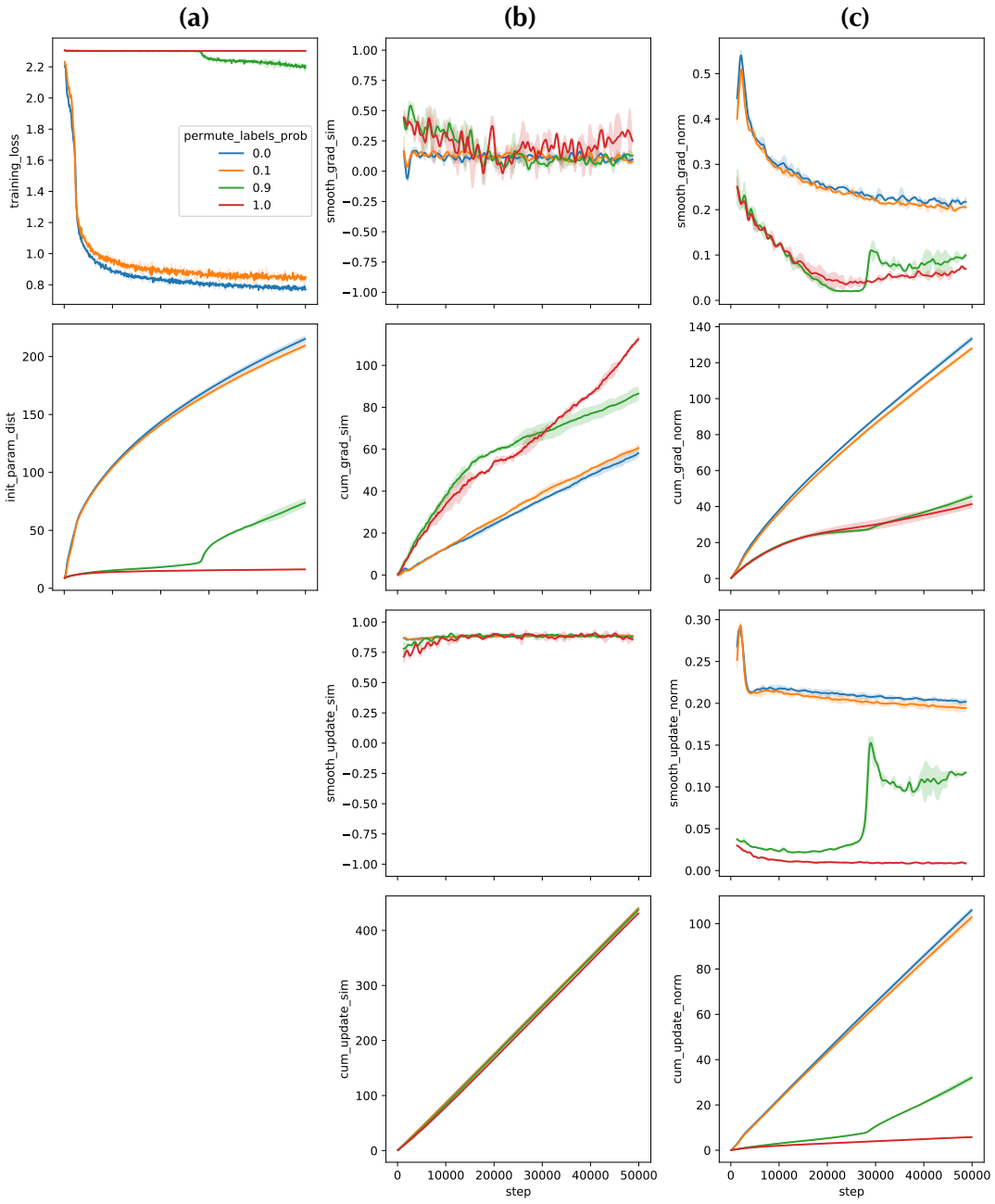


Figure D.9: Gradient and Adam update statistics for differently biased data distributions. **(a)** Plateaus in the loss are influenced by the bias in the data distribution. Plateaus result in moving away slowly from the parameter initialization. **(b)** The cosine similarity of both gradients and updates in consecutive steps is only marginally different with or without a loss plateau. **(c)** While the gradient norm is about half as big when a plateau exists, the updates are going towards zero.

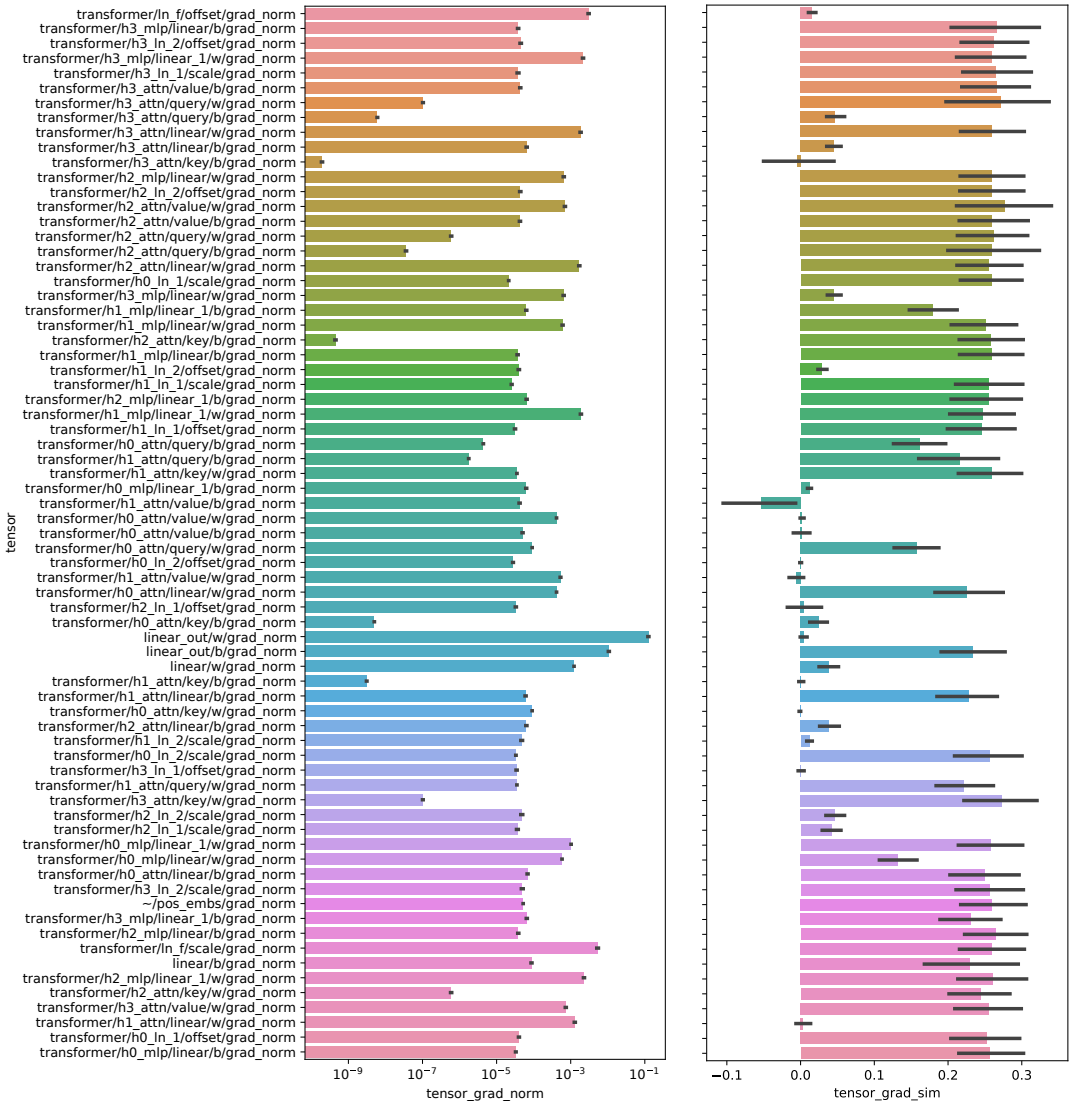


Figure D.10: Gradient L2 norms (left) and gradient cosine similarity for consecutive optimization steps (right) for different parameter tensors. The last (output) layer has the largest gradients. Most other gradients are small.

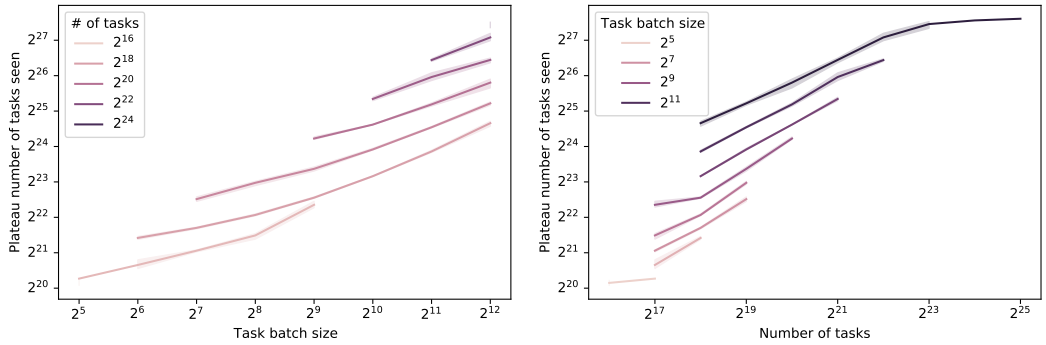


Figure D.11: Instead of plotting the loss plateau length in terms of optimization steps, we look at the total number of tasks seen within the plateau as a function of the task batch size and the number of tasks in the training distribution. An increase in the task batch size leads to more tasks to be processed to leave the plateau.

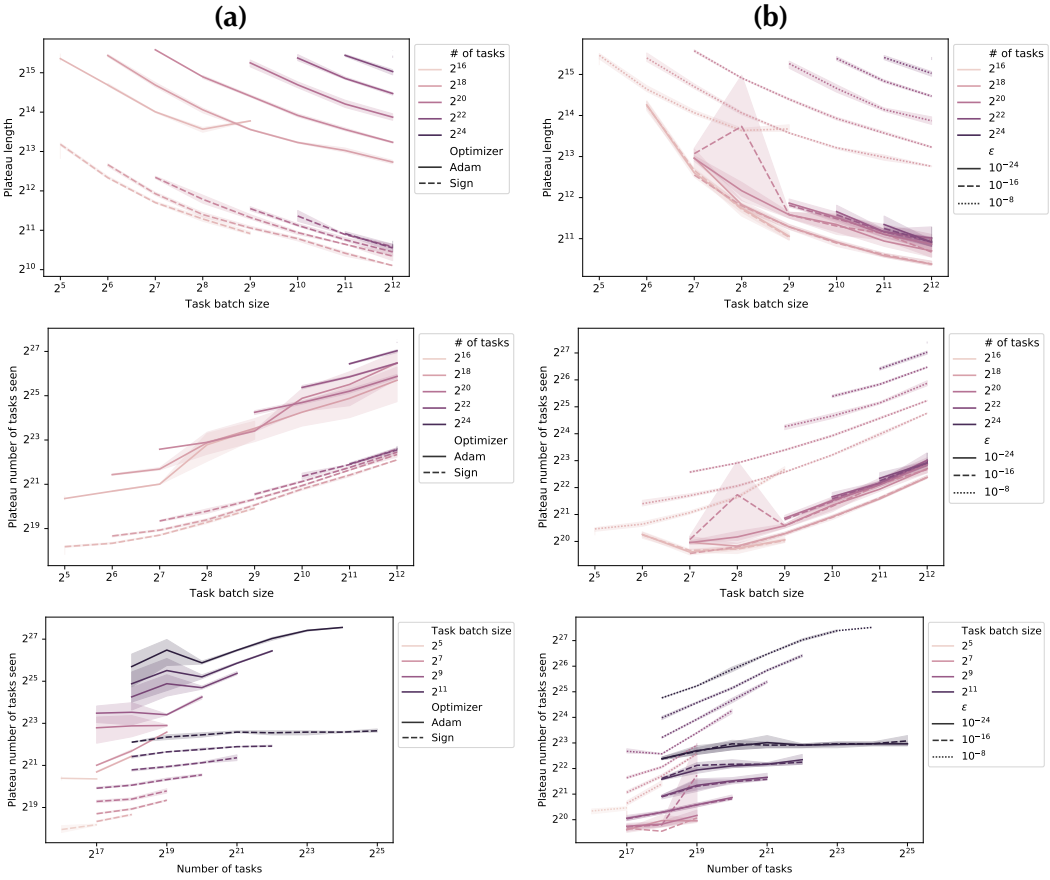


Figure D.12: **(a)** When replacing Adam with a sign normalization of the gradient or **(b)** reducing ϵ the plateau length is significantly shorter.

ing converged in terms of training loss, test loss may suddenly decrease. Large amounts of regularization, like weight decay with a coefficient of 1.0 were found to facilitate this behavior. Is grokking connected to the optimization behavior we observe, and if so, do similar interventions help in our setting? We look in particular at the boundary of memorization and generalization ($2^{14} = 16384$) where doubling the number of tasks a few more times would lead to generalization. Figure D.13 shows three task settings, $2^{10}, 2^{14}, 2^{16}$, and three different weight decay coefficients, $0.01, 0.1, 1.0$. The setting of 2^{16} tasks shows generalization by default and only serves as a baseline for the weight decay coefficient analysis. In the cases of memorization due to too few tasks, we have not been able to produce grokking behavior.

Optimization difficulties in VSML Previous work has observed several optimization difficulties: Slower convergence, local minima, unstable training, or loss plateaus at the beginning of training. Figure D.14 shows some of these difficulties in the context of VSML [Kirsch and Schmidhuber, 2021]. Because VSML has permutation invariance and parameter sharing built into the architecture as an inductive bias, changing the number of tasks has only a small effect. We observe that in particular deeper architectures make meta-optimization more difficult.

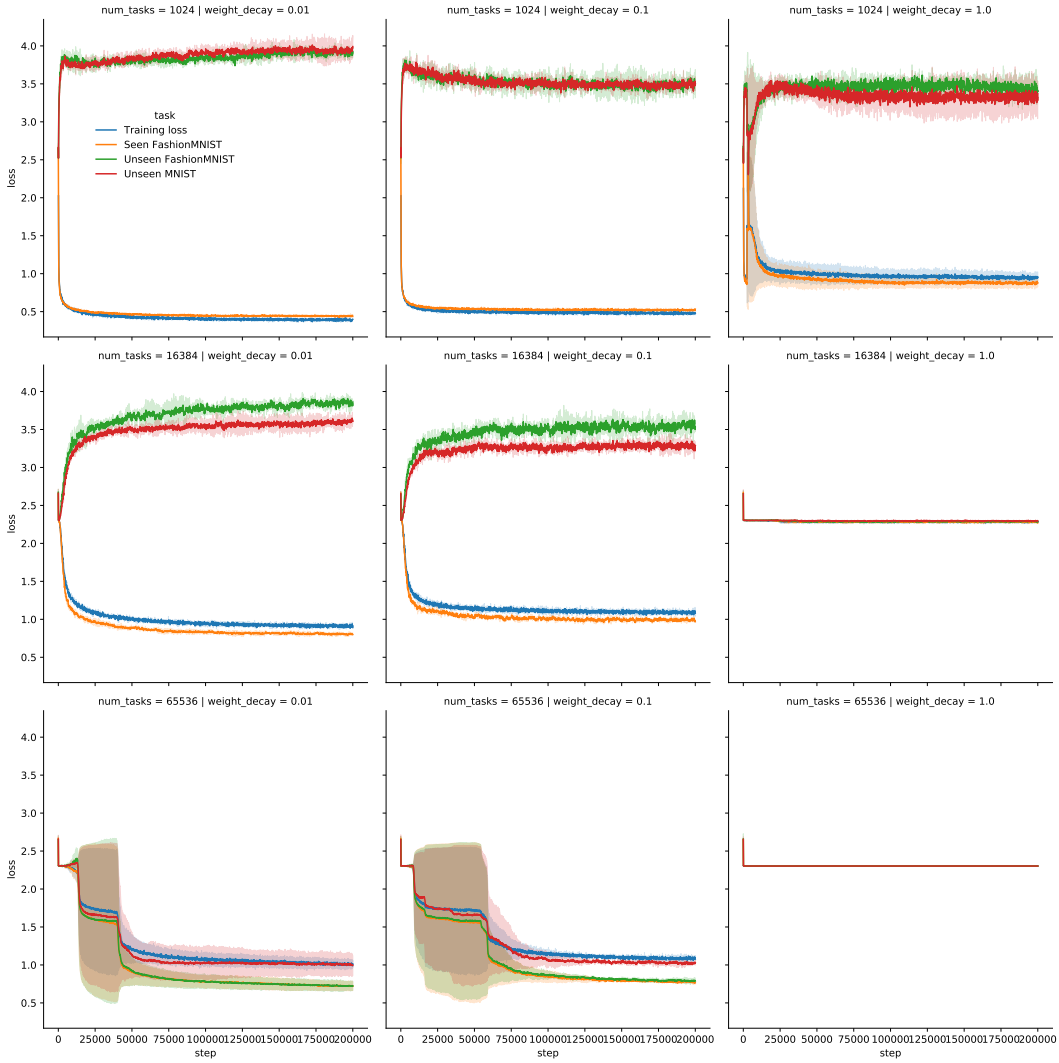


Figure D.13: We investigate whether grokking as defined in Power et al. [2022] can be produced when we observe memorization on a smaller numbers of tasks. This would correspond to the test loss decreasing long after the training loss has converged. We have not been able to elicit this behavior when looking at different numbers of tasks and weight decay coefficients.

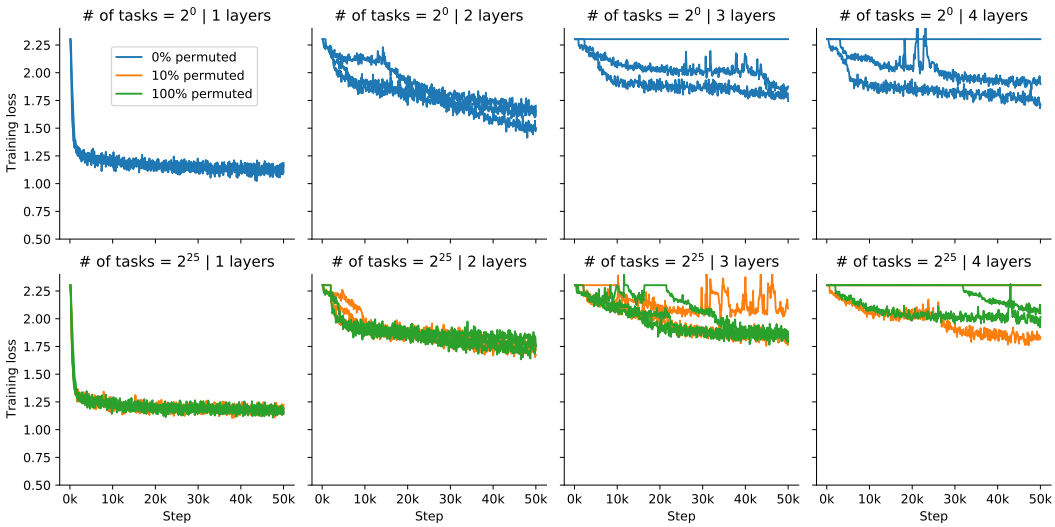


Figure D.14: Loss plateaus and slow convergence with deeper variants of VSML.

Appendix E

Appendix on FME

E.1 Implementation details

Least-recently-used Buffer We initialize a least-recently-used (LRU) buffer with a single randomly initialized neural network. The $m = 100$ buckets evenly cover the entire current performance range and each holds the 100 most recent solutions. Solutions from buckets with higher performance are sampled exponentially more frequently. We use an exponential base of e^{20}. All layers are initialized from a truncated normal with a standard deviation of $\sigma = \frac{1}{\sqrt{N_x}}$. When selected, a solution is executed for $L = 1000$ steps.

Architecture We stack three self-referential layers with 32 hidden units.

Sources of randomness To create a temporal tree of self-modifying solutions, randomness must be injected into the system. This randomness originates from the policy action sampling, non-deterministic environment steps, and potential external noise injection as an input to the policy. We found external noise injection not to improve the agent's performance when sufficient randomness originates from the policy and environment.

Appendix F

List of contributions

MetaGenRL, ICLR 2020
Meta-learning RL algorithms that generalize to vastly different environments.
Louis Kirsch, Sjoerd van Steenkiste, and Juergen Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. In *International Conference on Learning Representations*, 2020b. 2019 arXiv preprint arXiv:1910.04098

Invited speaker, NeurIPS 2020 Meta Learning Workshop
Position on meta-learning learning algorithms that generalize.
Louis Kirsch. Invited talk: General meta learning. Meta Learning Workshop at Advances in Neural Information Processing Systems, 2020

VSML, NeurIPS 2021
Encode backpropagation in RNN dynamics & meta-learn general-purpose supervised in-context learners from scratch that do not require hardcoded backpropagation.
Louis Kirsch and Jürgen Schmidhuber. Meta learning backpropagation and improving it. *Advances in Neural Information Processing Systems*, 34, 2021. 2020 arXiv preprint arXiv:2012.14905

SymLA, AAAI 2022
How symmetries can help generalization in meta-RL.
Louis Kirsch, Sebastian Flennerhag, Hado van Hasselt, Abram Friesen, Junhyuk Oh, and Yutian Chen. Introducing symmetries to black box meta reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 7202–7210, 2022a. 2021 arXiv preprint arXiv:2109.10781

Invited speaker, ICML 2022 Decision-aware RL workshop

Louis Kirsch. Invited talk: General-purpose meta learning. Decision-awareness in Reinforcement Learning Workshop at ICML, 2022

Self-referential learning, AutoML Conf Workshop
We investigate methods for self-referential meta-learning that do not require explicit meta-optimization to reduce our reliance on human engineering.
Louis Kirsch and Jürgen Schmidhuber. Eliminating meta optimization through self-referential meta learning. *arXiv preprint arXiv:2212.14392 and First Conference on Automated Machine Learning (Workshop)*, 2022a

GPICL, Meta Learning Workshop
We demonstrate that Transformers and other black-box models can exhibit in-context learning that generalizes to significantly different datasets while undergoing multiple transitions in terms of their learning behavior.
Louis Kirsch, James Harrison, Jascha Sohl-Dickstein, and Luke Metz. General-purpose in-context learning by meta-learning transformers. *arXiv preprint arXiv:2212.04458 and Workshop on Meta-Learning at NeurIPS*, 2022b

GLAs, Foundations Models for Decision Making Workshop
We meta-train in-context learning RL agents that generalize across domains (with different actuators, observations, dynamics, and dimensionalities) using supervised learning.
Louis Kirsch, James Harrison, Daniel Freeman, Jascha Sohl-Dickstein, and Jürgen Schmidhuber. Towards general-purpose in-context learning agents. *Foundation Models for Decision Making Workshop at NeurIPS*, 2023

CSCS (Supercomputer) grants

- Louis Kirsch, Michael Wand, and Jürgen Schmiduber. Learning learning algorithms. Swiss National Supercomputing Centre (CSCS) Project, 2019
- Louis Kirsch and Jürgen Schmiduber. Learning to replace human-engineered reinforcement learning algorithms. Swiss National Supercomputing Centre (CSCS) Project, 2020
- Louis Kirsch and Jürgen Schmiduber. Meta learning general purpose reinforcement learning algorithms. Swiss National Supercomputing Centre (CSCS) Project, 2021
- Aditya Ramesh, Louis Kirsch, and Jürgen Schmiduber. Improving curiosity-driven exploration for reinforcement learning. Swiss National Supercomputing Centre (CSCS) Project, 2022a

Workshop organization

- Louis Kirsch, Ignasi Clavera, Kate Rakelly, Chelsea Finn, Jane Wang, and Jeff Clune. Beyond 'tabula rasa' in reinforcement learning: agents that remember, adapt, and generalize. International Conference on Learning Representations, 2020a

- Feryal Behbahani, Khimya Khetarpal, Louis Kirsch, Rose Wang, Annie Xie, Adam White, and Doina Precup. A roadmap to never-ending rl. International Conference on Learning Representations, 2021

Collaborations

- Francesco Faccio, Louis Kirsch, and Jürgen Schmidhuber. Parameter-based value functions. In *International Conference on Learning Representations*, 2021a
- Aditya Ramesh, Louis Kirsch, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Exploring through random curiosity with general value functions. *Advances in Neural Information Processing Systems*, 2022b
- Luisa Zintgraf, Zita Marinho, Iurii Kemaev, Louis Kirsch, Junhyuk Oh, and Tom Schaul. Rl2x: Reinforcement learning to explore. In *The Multi-disciplinary Conference on Reinforcement Learning and Decision Making*, 2022
- Kenny John Young, Aditya Ramesh, Louis Kirsch, and Jürgen Schmidhuber. The benefits of model-based generalization in reinforcement learning. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 40254–40276. PMLR, 23–29 Jul 2023
- Vincent Herrmann, Louis Kirsch, and Jürgen Schmidhuber. Learning one abstract bit at a time through self-invented experiments encoded as neural networks. *arXiv preprint arXiv:2212.14374*, 2022
- Francesco Faccio, Vincent Herrmann, Aditya Ramesh, Louis Kirsch, and Jürgen Schmidhuber. Goal-conditioned generators of deep policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2023
- Mingchen Zhuge, Haozhe Liu, Francesco Faccio, Dylan R Ashley, Róbert Csordás, Anand Gopalakrishnan, Abdullah Hamdi, Hasan Abed Al Kader Hammoud, Vincent Herrmann, Kazuki Irie, et al. Mindstorms in natural language-based societies of mind. *arXiv preprint arXiv:2305.17066*, 2023
- Samuel Schmidgall, Jascha Achterberg, Thomas Miconi, Louis Kirsch, Rojin Ziaei, S Hajiseyedrazi, and Jason Eshraghian. Brain-inspired learning in artificial neural networks: a review. *arXiv preprint arXiv:2305.11252*, 2023
- Aleksandar Stanić, Dylan Ashley, Oleg Serikov, Louis Kirsch, Francesco Faccio, Jürgen Schmidhuber, Thomas Hofmann, and Imanol Schlag. The languini kitchen: Enabling language modelling research at different scales of compute. *arXiv preprint arXiv:2309.11197*, 2023
- Matthew Jackson, Chris Lu, Louis Kirsch, Robert Lange, Shimon Whiteson, and Jakob Foerster. Discovering temporally-aware reinforcement learning algorithms. *Advances in Neural Information Processing Systems*, 2023
- Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024

- Akarsh Kumar, Chris Lu, Louis Kirsch, Yujin Tang, Kenneth O Stanley, Phillip Isola, and David Ha. Automating the search for artificial life with foundation models. *arXiv preprint arXiv:2412.17799*, 2024

Bibliography

Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. *arXiv preprint arXiv:2211.15661*, 2022.

Ferran Alet, Martin F. Schneider, Tomas Lozano-Perez, and Leslie Pack Kaelbling. Meta-learning curiosity algorithms. In *International Conference on Learning Representations*, 2020.

S. Amari. A theory of adaptive pattern classifiers. *IEEE Trans. EC*, 16(3): 299–307, 1967.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

Cem Anil, Yuhuai Wu, Anders Johan Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Venkatesh Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

Jimmy Ba, Geoffrey Hinton, Volodymyr Mnih, Joel Z. Leibo, and Catalin Ionescu. Using Fast Weights to Attend to the Recent Past. In *Advances in Neural Information Processing Systems*, pages 4331–4339, 2016a.

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

Sarah Bechtle, Artem Molchanov, Yevgen Chebotar, Edward Grefenstette, Ludovic Righetti, Gaurav Sukhatme, and Franziska Meier. Meta learning via

learned loss. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 4161–4168. IEEE, 2021.

Feryal Behbahani, Khimya Khetarpal, Louis Kirsch, Rose Wang, Annie Xie, Adam White, and Doina Precup. A roadmap to never-ending rl. International Conference on Learning Representations, 2021.

Samy Bengio, Yoshua Bengio, Jocelyn Cloutier, and Jan Gecsei. On the optimization of a synaptic learning rule. In *Preprints Conf. Optimality in Artificial and Biological Neural Networks*, volume 2, 1992.

Y Bengio, S Bengio, and J Cloutier. Learning a synaptic learning rule. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume 2, pages 969–vol. IEEE, 1991.

Lukas Biewald. Experiment Tracking with Weights and Biases, 2020. URL `https://www.wandb.com/`.

Jonathan C Brant and Kenneth O Stanley. Minimal criterion coevolution: a new approach to open-ended search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 67–74, 2017.

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

Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

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.

Jake Bruce, Michael D Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, et al. Genie: Generative interactive environments. In *Forty-first International Conference on Machine Learning*, 2024.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al.

Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024.

Stephanie CY Chan, Adam Santoro, Andrew K Lampinen, Jane X Wang, Aaditya Singh, Pierre H Richemond, Jay McClelland, and Felix Hill. Data distributional properties drive emergent in-context learning in transformers. *arXiv preprint arXiv:2205.05055*, 2022.

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*, 34:15084–15097, 2021.

Yutian Chen, Xingyou Song, Chansoo Lee, Zi Wang, Richard Zhang, David Dohan, Kazuya Kawakami, Greg Kochanski, Arnaud Doucet, Marc'aurelio Ranzato, et al. Towards learning universal hyperparameter optimizers with transformers. *Advances in Neural Information Processing Systems*, 35: 32053–32068, 2022.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 6 2014.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, and David Ha. Deep Learning for Classical Japanese Literature, 2018.

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

John D Co-Reyes, Yingjie Miao, Daiyi Peng, Esteban Real, Quoc V Le, Sergey Levine, Honglak Lee, and Aleksandra Faust. Evolving reinforcement learning algorithms. In *International Conference on Learning Representations*, 2021.

Edgar F Codd. *Cellular automata*. Academic press, 2014.

Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 international joint conference on neural networks (IJCNN)*, pages 2921–2926. IEEE, 2017.

Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capacity and trainability in recurrent neural networks. *arXiv preprint arXiv:1611.09913*, 2016.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. The devil is in the detail: Simple tricks improve systematic generalization of transformers. In *EMNLP*, 2021.

David B D'Ambrosio and Kenneth O Stanley. A novel generative encoding for exploiting neural network sensor and output geometry. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 974–981, 2007.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Marcus Hutter, Shane Legg, and Pedro A Ortega. Neural networks and the chomsky hierarchy. *arXiv preprint arXiv:2207.02098*, 2022.

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1): 1997–2017, 2019.

Francesco Faccio, Louis Kirsch, and Jürgen Schmidhuber. Parameter-based value functions. In *International Conference on Learning Representations*, 2021a.

Francesco Faccio, Louis Kirsch, and Jürgen Schmidhuber. Parameter-based value functions. In *International Conference on Learning Representations*, 2021b.

Francesco Faccio, Vincent Herrmann, Aditya Ramesh, Louis Kirsch, and Jürgen Schmidhuber. Goal-conditioned generators of deep policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2023.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*, 2023.

Matthias Feurer, Jost Springenberg, and Frank Hutter. Initializing Bayesian hyperparameter optimization via meta-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2015.

Chelsea Finn and Sergey Levine. Meta-Learning and Universality: Deep Representations and Gradient Descent can Approximate any Learning Algorithm. In *International Conference on Learning Representations*, 2018.

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.

Sebastian Flennerhag, Andrei A. Rusu, Razvan Pascanu, Francesco Visin, Hujun Yin, and Raia Hadsell. Meta-learning with warped gradient descent. In *International Conference on Learning Representations*, 2020.

Sebastian Flennerhag, Yannick Schroecker, Tom Zahavy, Hado van Hasselt, David Silver, and Satinder Singh. Bootstrapped meta-learning. *arXiv preprint arXiv:2109.04504*, 2021.

Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *Proceedings of Machine Learning Research*, 80:1587–1596, 2018.

Kunihiko Fukushima. Neural network model for a mechanism of pattern recognition unaffected by shift in position-neocognitron. *IEICE Technical Report, A*, 62(10):658–665, 1979.

Matteo Gagliolo and Jürgen Schmidhuber. Algorithm portfolio selection as a bandit problem with unbounded losses. *Annals of Mathematics and Artificial Intelligence*, 61(2):49–86, 2011.

Shivam Garg, Dimitris Tsipras, Percy Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. *arXiv preprint arXiv:2208.01066*, 2022.

Marta Garnelo, Dan Rosenbaum, Christopher Maddison, Tiago Ramalho, David Saxton, Murray Shanahan, Yee Whye Teh, Danilo Rezende, and

SM Ali Eslami. Conditional neural processes. In *International Conference on Machine Learning*, pages 1704–1713. PMLR, 2018.

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

Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000b.

Ben Goertzel. Artificial General Intelligence: Concept, State of the Art, and Future Prospects. *Journal of Artificial General Intelligence*, 01 2014. doi: 10.2478/jagi-2014-0001.

Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google Vizier: A service for black-box optimization. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495, 2017.

Erin Grant, Chelsea Finn, Sergey Levine, Trevor Darrell, and Thomas Griffiths. Recasting gradient-based meta-learning as hierarchical bayes. In *International Conference on Learning Representations*, 2018.

Karol Gregor. Finding online neural update rules by learning to remember. *arXiv preprint arXiv:2003.03124*, 3 2020.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*, pages 2450–2462, 2018.

David Ha, Andrew M. Dai, and Quoc V. Le. Hypernetworks. In *International Conference on Learning Representations*, 2017.

Jean Harb, Tom Schaul, Doina Precup, and Pierre-Luc Bacon. Policy evaluation networks. *CoRR*, abs/2002.11833, 2020.

Vincent Herrmann, Louis Kirsch, and Jürgen Schmidhuber. Learning one abstract bit at a time through self-invented experiments encoded as neural networks. *arXiv preprint arXiv:2212.14374*, 2022.

Vincent Herrmann, Francesco Faccio, and Jürgen Schmidhuber. Learning useful representations of recurrent neural network weight matrices. In *Forty-first International Conference on Machine Learning*, 2024.

S Hochreiter and J Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997a.

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

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.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Noah Hollmann, Samuel Müller, Katharina Eggensperger, and Frank Hutter. Tabpfn: A transformer that solves small tabular classification problems in a second. *Table Representation Workshop at NeurIPS*, 2022.

Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. *Advances in neural information processing systems*, 29, 2016.

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.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Benchmarking large language models as ai research agents. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

Wenlong Huang, Igor Mordatch, and Deepak Pathak. One Policy to Control Them All: Shared Modular Policies for Agent-Agnostic Control. In *International Conference on Machine Learning*, 2020.

Bernardo A Huberman, Rajan M Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. A modern self-referential weight matrix that learns to modify itself. In *Deep RL Workshop NeurIPS 2021*, 2021a.

Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. *Advances in Neural Information Processing Systems*, 34:7703–7717, 2021b.

Alekseĭ Grigor evich Ivakhnenko and Valentin Grigorévich Lapa. *Cybernetic Predicting Devices*. CCM Information Corporation, 1965.

Matthew Jackson, Chris Lu, Louis Kirsch, Robert Lange, Shimon Whiteson, and Jakob Foerster. Discovering temporally-aware reinforcement learning algorithms. *Advances in Neural Information Processing Systems*, 2023.

Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castañeda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, Nicolas Sonnerat, Tim Green, Louise Deason, Joel Z Leibo, David Silver, Demis Hassabis, Koray Kavukcuoglu, and Thore Graepel. Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science (New York, N.Y.)*, 364(6443):859–865, 5 2019.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.

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

Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. In *International Conference on Learning Representations*, 12 2014.

Louis Kirsch. Differentiable convolutional neural architectures for time series classification. Bachelor's thesis, Hasso Plattner Institute, Potsdam, Germany, 7 2017.

Louis Kirsch. Invited talk: General meta learning. Meta Learning Workshop at Advances in Neural Information Processing Systems, 2020.

Louis Kirsch. Invited talk: General-purpose meta learning. Decision-awareness in Reinforcement Learning Workshop at ICML, 2022.

Louis Kirsch and Jürgen Schmidhuber. Meta learning backpropagation and improving it. *Advances in Neural Information Processing Systems*, 34, 2021. 2020 arXiv preprint arXiv:2012.14905.

Louis Kirsch and Jürgen Schmidhuber. Eliminating meta optimization through self-referential meta learning. *arXiv preprint arXiv:2212.14392 and First Conference on Automated Machine Learning (Workshop)*, 2022a.

Louis Kirsch and Jürgen Schmidhuber. Self-referential meta learning. In *Decision Awareness in Reinforcement Learning Workshop at ICML 2022*, 2022b.

Louis Kirsch and Jürgen Schmiduber. Learning to replace human-engineered reinforcement learning algorithms. Swiss National Supercomputing Centre (CSCS) Project, 2020.

Louis Kirsch and Jürgen Schmiduber. Meta learning general purpose reinforcement learning algorithms. Swiss National Supercomputing Centre (CSCS) Project, 2021.

Louis Kirsch, Julius Kunze, and David Barber. Modular Networks: Learning to Decompose Neural Computation. *Advances in Neural Information Processing Systems*, 2018.

Louis Kirsch, Michael Wand, and Jürgen Schmiduber. Learning learning algorithms. Swiss National Supercomputing Centre (CSCS) Project, 2019.

Louis Kirsch, Ignasi Clavera, Kate Rakelly, Chelsea Finn, Jane Wang, and Jeff Clune. Beyond 'tabula rasa' in reinforcement learning: agents that remember, adapt, and generalize. International Conference on Learning Representations, 2020a.

Louis Kirsch, Sjoerd van Steenkiste, and Juergen Schmidhuber. Improving generalization in meta reinforcement learning using learned objectives. In *International Conference on Learning Representations*, 2020b. 2019 arXiv preprint arXiv:1910.04098.

Louis Kirsch, Sebastian Flennerhag, Hado van Hasselt, Abram Friesen, Junhyuk Oh, and Yutian Chen. Introducing symmetries to black box meta reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 7202–7210, 2022a. 2021 arXiv preprint arXiv:2109.10781.

Louis Kirsch, James Harrison, Jascha Sohl-Dickstein, and Luke Metz. General-purpose in-context learning by meta-learning transformers. *arXiv preprint arXiv:2212.04458 and Workshop on Meta-Learning at NeurIPS*, 2022b.

Louis Kirsch, James Harrison, Daniel Freeman, Jascha Sohl-Dickstein, and Jürgen Schmidhuber. Towards general-purpose in-context learning agents. *Foundation Models for Decision Making Workshop at NeurIPS*, 2023.

Werner M Kistler, Wulfram Gerstner, and J Leo van Hemmen. Reduction of the Hodgkin-Huxley equations to a single-variable threshold model. *Neural Computation*, 9(5):1015–1045, 1997.

Akarsh Kumar, Chris Lu, Louis Kirsch, Yujin Tang, Kenneth O Stanley, Phillip Isola, and David Ha. Automating the search for artificial life with foundation models. *arXiv preprint arXiv:2412.17799*, 2024.

Brenden Lake, Ruslan Salakhutdinov, Jason Gross, and Joshua Tenenbaum. One shot learning of simple visual concepts. In *Proceedings of the annual meeting of the cognitive science society*, volume 33, 2011.

Robert Tjarko Lange, Tom Schaul, Yutian Chen, Tom Zahavy, Valentin Dalibard, Chris Lu, Satinder Singh, and Sebastian Flennerhag. Discovering evolution strategies via meta-black-box optimization. In *The Eleventh International Conference on Learning Representations*, 2023.

Christopher G Langton. Artificial life: An overview. 1997.

Michael Laskin, Luyu Wang, Junhyuk Oh, Emilio Parisotto, Stephen Spencer, Richie Steigerwald, DJ Strouse, Steven Hansen, Angelos Filos, Ethan Brooks, et al. In-context reinforcement learning with algorithm distillation. *arXiv preprint arXiv:2210.14215*, 2022.

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

Jonathan N Lee, Annie Xie, Aldo Pacchiano, Yash Chandak, Chelsea Finn, Ofir Nachum, and Emma Brunskill. Supervised pretraining can learn in-context reinforcement learning. *arXiv preprint arXiv:2306.14892*, 2023.

Shane Legg. Machine Super Intelligence. Doctoral Dissertation submitted to the Faculty of Informatics of the University of Lugano, June 2008.

Joel Lehman and Kenneth O Stanley. Novelty search and the problem with objectives. *Genetic programming theory and practice IX*, pages 37–56, 2011.

Gottfried Wilhelm Leibniz. *De arte combinatoria*. Leipzig, 1666. Also see his later works on Characteristica Universalis & Calculus Ratiocinator.

Ke Li and Jitendra Malik. Learning to Optimize. In *International Conference on Learning Representations*, 2017.

Xiaobin Li, Kai Wu, Xiaoyu Zhang, Handing Wang, and Jing Liu. Optformer: Beyond transformer for black-box optimization, 2023.

Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062, 2018.

Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *International Conference on Learning Representations*, 2016.

S Linnainmaa. *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors*. PhD thesis, Univ. Helsinki, 1970.

Hao Liu and Pieter Abbeel. Emergent agentic transformer from chain of hindsight experience. *arXiv preprint arXiv:2305.16554*, 2023.

Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered policy optimisation. *Advances in Neural Information Processing Systems*, 35:16455–16468, 2022.

Chris Lu, Yannick Schroecker, Albert Gu, Emilio Parisotto, Jakob Foerster, Satinder Singh, and Feryal Behbahani. Structured state space models for in-context reinforcement learning. *arXiv preprint arXiv:2303.03982*, 2023.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Luke Metz, Niru Maheswaranathan, Brian Cheung, and Jascha Sohl-Dickstein. Learning Unsupervised Learning Rules. In *International Conference on Learning Representations*, 3 2019a.

Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, pages 4556–4565. PMLR, 2019b.

Luke Metz, Niru Maheswaranathan, C Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein. Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves. *arXiv preprint arXiv:2009.11243*, 2020a.

Luke Metz, Niru Maheswaranathan, Ruoxi Sun, C Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein. Using a thousand optimization tasks to learn hyperparameter search strategies. *arXiv preprint arXiv:2002.11887*, 2020b.

Thomas Miconi, Kenneth Stanley, and Jeff Clune. Differentiable plasticity: training plastic neural networks with backpropagation. In *International Conference on Machine Learning*, pages 3559–3568. PMLR, 2018.

Thomas Miconi, Aditya Rawal, Jeff Clune, and Kenneth O. Stanley. Backpropamine: training self-modifying neural networks with differentiable neuromodulated plasticity. In *International Conference on Learning Representations*, 2019.

Vladimir Mikulik, Grégoire Delétang, Tom McGrath, Tim Genewein, Miljan Martic, Shane Legg, and Pedro Ortega. Meta-trained agents implement bayes-optimal agents. *Advances in neural information processing systems*, 33:18691–18703, 2020.

Geoffrey F Miller, Peter M Todd, and Shailesh U Hegde. Designing neural networks using genetic algorithms. In *ICGA*, volume 89, pages 379–384, 1989.

Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. In *International Conference on Learning Representations*, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. Growing neural cellular automata. *Distill*, 5(2):e23, 2020.

Meredith Ringel Morris, Jascha Sohl-Dickstein, Noah Fiedel, Tris Warkentin, Allan Dafoe, Aleksandra Faust, Clement Farabet, and Shane Legg. Levels of agi: Operationalizing progress on the path to agi. *arXiv preprint arXiv:2311.02462*, 2023.

Michael C Mozer and Sreerupa Das. A connectionist symbol manipulator that discovers the structure of context-free languages. In *Advances in neural information processing systems*, pages 863–870, 1993.

Deepak Mukunthu, Parashar Shah, and Wee Hyong Tok. *Practical Automated Machine Learning on Azure: Using Azure Machine Learning to Quickly Build AI Solutions*. O'Reilly Media, 2019.

Samuel Müller, Noah Hollmann, Sebastian Pineda Arango, Josif Grabocka, and Frank Hutter. Transformers can do bayesian inference. In *International Conference on Learning Representations*, 2022.

Elias Najarro and Sebastian Risi. Meta-learning through hebbian plasticity in random networks. *Advances in Neural Information Processing Systems*, 33: 20719–20731, 2020.

Aviv Navon, Aviv Shamsian, Idan Achituve, Ethan Fetaya, Gal Chechik, and Haggai Maron. Equivariant architectures for learning in deep weight spaces. In *International Conference on Machine Learning*, pages 25790–25816. PMLR, 2023.

Tung Nguyen and Aditya Grover. Transformer neural processes: Uncertainty-aware meta learning via sequence modeling. In *International Conference on Machine Learning*, pages 16569–16594. PMLR, 2022.

Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman Openai. Gotta Learn Fast: A New Benchmark for Generalization in RL. *arXiv preprint arXiv:1804.03720*, 2018.

Scott Niekum, Lee Spector, and Andrew Barto. Evolution of reward functions for reinforcement learning. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 177–178, 2011.

Junhyuk Oh, Matteo Hessel, Wojciech M Czarnecki, Zhongwen Xu, Hado P van Hasselt, Satinder Singh, and David Silver. Discovering reinforcement learning algorithms. *Advances in Neural Information Processing Systems*, 33: 1060–1070, 2020.

Pedro A Ortega, Jane X Wang, Mark Rowland, Tim Genewein, Zeb Kurth-Nelson, Razvan Pascanu, Nicolas Heess, Joel Veness, Alex Pritzel, Pablo Sprechmann, et al. Meta-learning of sequential strategies. *arXiv preprint arXiv:1905.03030*, 2019.

Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. *arXiv preprint arXiv:2203.01302*, 2022a.

Jack Parker-Holder, Raghu Rajan, Xingyou Song, André Biedenkapp, Yingjie Miao, Theresa Eimer, Baohe Zhang, Vu Nguyen, Roberto Calandra,

Aleksandra Faust, et al. Automated reinforcement learning (autorl): A survey and open problems. *Journal of Artificial Intelligence Research*, 74:517–568, 2022b.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning*, pages 2778–2787. PMLR, 2017.

Deepak Pathak, Chris Lu, Trevor Darrell, Phillip Isola, and Alexei A. Efros. Learning to Control Self-Assembling Morphologies: A Study of Generalization via Modularity. In *Advances in Neural Information Processing Systems*, 2019.

Joachim Winther Pedersen and Sebastian Risi. Evolving and merging hebbian learning rules: increasing generalization by decreasing the number of rules. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 892–900, 2021.

Joachim Winther Pedersen and Sebastian Risi. Minimal neural network models for permutation invariant agents. *arXiv preprint arXiv:2205.07868*, 2022.

Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, et al. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *arXiv preprint arXiv:1802.09464*, 2018.

Alethea Power, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. Grokking: Generalization beyond overfitting on small algorithmic datasets. *arXiv preprint arXiv:2201.02177*, 2022.

Sébastien Racanière, Théophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. *Advances in neural information processing systems*, 30, 2017.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763. PMLR, 2021.

Aniruddh Raghu, Maithra Raghu, Samy Bengio, and Oriol Vinyals. Rapid learning or feature reuse? towards understanding the effectiveness of maml. In *International Conference on Learning Representations*, 2020.

Aditya Ramesh, Louis Kirsch, and Jürgen Schmiduber. Improving curiosity-driven exploration for reinforcement learning. Swiss National Supercomputing Centre (CSCS) Project, 2022a.

Aditya Ramesh, Louis Kirsch, Sjoerd van Steenkiste, and Jürgen Schmidhuber. Exploring through random curiosity with general value functions. *Advances in Neural Information Processing Systems*, 2022b.

Ettore Randazzo, Eyvind Niklasson, and Alexander Mordvintsev. MPLP: Learning a Message Passing Learning Protocol. *arXiv preprint arXiv:2007.00970*, 2020.

Sharath Chandra Raparthy, Eric Hambro, Robert Kirk, Mikael Henaff, and Roberta Raileanu. Generalization to new sequential decision making tasks with in-context learning. In *Forty-first International Conference on Machine Learning*, 2024.

Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. In *International Conference on Learning Representations*, 2017.

Esteban Real, Chen Liang, David So, and Quoc Le. Automl-zero: evolving machine learning algorithms from scratch. In *International Conference on Machine Learning*, pages 8007–8019. PMLR, 2020.

Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gómez Colmenarejo, Alexander Novikov, Gabriel Barth-maron, Mai Giménez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A generalist agent. *Transactions on Machine Learning Research*, 2022. ISSN 2835-8856.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In *International Conference on Machine Learning*, pages 1278–1286, 2014. ISBN 9781634393973. doi: 10.1051/0004-6361/201527329.

Sebastian Risi. The future of artificial intelligence is self-organizing and self-assembling. *sebastianrisi.com*, 2021.

Marek Rosa, Olga Afanasjeva, Simon Andersson, Joseph Davidson, Nicholas Guttenberg, Petr Hlubuček, Martin Poliak, Jaroslav Vítku, and Jan Feyereisl. BADGER: Learning to (Learn [Learning Algorithms] through Multi-Agent Communication). *arXiv preprint arXiv:1912.01513*, 2019a.

Marek Rosa, Olga Afanasjeva, Simon Andersson, Joseph Davidson, Nicholas Guttenberg, Petr Hlubuček, Martin Poliak, Jaroslav Vítku, and Jan Feyereisl. Badger: Learning to (learn [learning algorithms] through multi-agent communication). *arXiv preprint arXiv:1912.01513*, 2019b.

Clemens Rosenbaum, Tim Klinger, and Matthew Riemer. Routing Networks: Adaptive Selection of Non-Linear Functions for Multi-Task Learning. In *International Conference on Learning Representations*, 2018.

Andrei A. Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy Distillation. In *International Conference on Learning Representations*, 2016.

Andrei A. Rusu, Dushyant Rao, Jakub Sygnowski, Oriol Vinyals, Razvan Pascanu, Simon Osindero, and Raia Hadsell. Meta-Learning with Latent Embedding Optimization. In *International Conference on Learning Representations*, 7 2019.

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.

Mark Sandler, Max Vladymyrov, Andrey Zhmoginov, Nolan Miller, Tom Madams, Andrew Jackson, and Blaise Agüera Y Arcas. Meta-learning bidirectional update rules. In *International Conference on Machine Learning*, pages 9288–9300. PMLR, 2021.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pages 1842–1850. PMLR, 2016.

Imanol Schlag and Jürgen Schmidhuber. Gated fast weights for on-the-fly neural program generation. In *NIPS Metalearning Workshop*, 2017.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pages 9355–9366. PMLR, 2021a.

Imanol Schlag, Tsendsuren Munkhdalai, and Jürgen Schmidhuber. Learning associative inference using fast weight memory. In *International Conference on Learning Representations*, 2021b.

Samuel Schmidgall, Jascha Achterberg, Thomas Miconi, Louis Kirsch, Rojin Ziaei, S Hajiseyedrazi, and Jason Eshraghian. Brain-inspired learning in artificial neural networks: a review. *arXiv preprint arXiv:2305.11252*, 2023.

Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

Jürgen Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proc. of the international conference on simulation of adaptive behavior: From animals to animats*, pages 222–227, 1991a.

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

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992b.

Jürgen Schmidhuber. Steps towards self-referential neural learning: A thought experiment. 1992c.

Jürgen Schmidhuber. Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets. In *International Conference on Artificial Neural Networks*, pages 460–463. Springer, 1993a.

Jürgen Schmidhuber. A 'self-referential'weight matrix. In *International conference on artificial neural networks*, pages 446–450. Springer, 1993b.

Jürgen Schmidhuber. On learning how to learn learning strategies. Technical Report FKI-198-94, Fakultät für Informatik, Technische Universität München, 1994a. Revised 1995.

Jürgen Schmidhuber. What's interesting? Technical Report IDSIA-35-97, IDSIA, 1997. ftp://ftp.idsia.ch/pub/juergen/interest.ps.gz; extended abstract in Proc. Snowbird'98, Utah, 1998; see also Schmidhuber [2003].

Jürgen Schmidhuber. Exploring the predictable. In *Advances in evolutionary computing: theory and applications*, pages 579–612. Springer, 2003.

Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial general intelligence*, pages 199–226. Springer, 2007.

Jürgen Schmidhuber. Ultimate cognition *à la* Gödel. *Cognitive Computation*, 1 (2):177–193, 2009.

Jürgen Schmidhuber. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in psychology*, 4:313, 2013.

Jürgen Schmidhuber. Jürgen schmidhuber's homepage. The Swiss AI Lab IDSIA, 2014. URL `https://people.idsia.ch/~juergen/`. Accessed: 2025-11-03.

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

Jürgen Schmidhuber. Reinforcement learning upside down: Don't predict rewards–just map them to actions. *arXiv preprint arXiv:1912.02875*, 2019.

Jürgen Schmidhuber, Jieyu Zhao, and Marco Wiering. Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. *Machine Learning*, 28(1):105–130, 1997.

Jürgen Schmidhuber. Making the world differentiable: on using self-supervision fully recurrent neural networks for dynamic reinforcement learning and planning in non-stationary environments. *International Business*, pages 62–81, 1990. ISSN 02624079.

Jürgen Schmidhuber. Learning to Generate Sub-Goals for Action Sequences. In T Kohonen, K Mäkisara, O Simula, and J Kangas, editors, *Artificial Neural Networks*, pages 967–972. Elsevier Science Publishers B.V., North-Holland, 1991b.

Jürgen Schmidhuber. A Possibility for Implementing Curiosity and Boredom in Model-Building Neural Controllers. In J A Meyer and S W Wilson, editors, *Proc. of the International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, pages 222–227. MIT Press/Bradford Books, 1991c.

Jürgen Schmidhuber. Neural sequence chunkers. Technical Report FKI-148-91, Institut für Informatik, Technische Universität München, April 1991d.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to recurrent nets. *Neural Computation*, 4(1):131–139, 1992d.

Jürgen Schmidhuber. On decreasing the ratio between learning complexity and number of time-varying variables in fully recurrent nets. In *Proceedings of the International Conference on Artificial Neural Networks, Amsterdam*, pages 460–463. Springer, 1993c.

Jürgen Schmidhuber. Discovering Problem Solutions with Low Kolmogorov Complexity and High Generalization Capability. Technical Report FKI-194-94, Fakultät für Informatik, Technische Universität München, 1994b.

Jürgen Schmidhuber and J Zhao. Direct policy search and uncertain policy evaluation. In *AAAI Spring Symposium on Search under Uncertain and Incomplete Information, Stanford Univ.*, pages 119–124. American Association for Artificial Intelligence, Menlo Park, Calif., 1999.

Nicol N Schraudolph. Local gain adaptation in stochastic gradient descent. In *1999 Ninth international conference on artificial neural networks ICANN 99.(Conf. Publ. No. 470)*, volume 2, pages 569–574. IET, 1999.

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

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

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

Dale Schuurmans, Hanjun Dai, and Francesco Zanini. Autoregressive large language models are computationally universal. *arXiv preprint arXiv:2410.03170*, 2024.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *International Conference on Learning Representations*, 2017.

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

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *31st International Conference on Machine Learning, ICML 2014*, volume 1, pages 605–619, 1 2014. ISBN 9781634393973.

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

Sims, Karl. *Evolving Virtual Creatures*. ACM SIGGRAPH, 1994.

Satinder Singh, Satinder Singh, A.G. Barto, A.G. Barto, Nuttapong Chentanez, and Nuttapong Chentanez. Intrinsically motivated reinforcement learning. *18th Annual Conference on Neural Information Processing Systems (NIPS)*, 17:1281–1288, 2004. ISSN 1943-0604. doi: 10.1109/TAMD.2010.2051031.

Alessandro Sperduti. Encoding labeled graphs by labeling raam. *Advances in Neural Information Processing Systems*, pages 1125–1125, 1994.

Aleksandar Stanić, Dylan Ashley, Oleg Serikov, Louis Kirsch, Francesco Faccio, Jürgen Schmidhuber, Thomas Hofmann, and Imanol Schlag. The languini kitchen: Enabling language modelling research at different scales of compute. *arXiv preprint arXiv:2309.11197*, 2023.

Kenneth O Stanley. Why open-endedness matters. *Artificial life*, 25(3): 232–235, 2019.

Felipe Petroski Such, Aditya Rawal, Joel Lehman, Kenneth Stanley, and Jeffrey Clune. Generative teaching networks: Accelerating neural architecture

search by learning to generate synthetic training data. In *International Conference on Machine Learning*, pages 9206–9216. PMLR, 2020.

Shyam Sudhakaran, Djordje Grbic, Siyan Li, Adam Katona, Elias Najarro, Claire Glanois, and Sebastian Risi. Growing 3d artefacts and functional machines with neural cellular automata. In *ALIFE 2021: The 2021 Conference on Artificial Life*. MIT Press, 2021.

G Z Sun. Neural networks with external memeory stack that learn context-free grammars from examples. In *Proceedings of the Conference on Information Science and Systems, 1991*, pages 649–653. Princeton University, 1991.

Flood Sung, Li Zhang, Tao Xiang, Timothy Hospedales, and Yongxin Yang. Learning to learn: Meta-critic networks for sample efficient learning. *arXiv preprint arXiv:1706.09529*, 2017.

Richard S Sutton. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *AAAI*, pages 171–176. San Jose, CA, 1992.

Yujin Tang and David Ha. The sensory neuron as a transformer: Permutation-invariant neural networks for reinforcement learning. *Advances in Neural Information Processing Systems*, 34:22574–22587, 2021.

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.

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

Eleni Triantafillou, Tyler Zhu, Vincent Dumoulin, Pascal Lamblin, Utku Evci, Kelvin Xu, Ross Goroshin, Carles Gelada, Kevin Swersky, Pierre-Antoine Manzagol, and Hugo Larochelle. Meta-dataset: A dataset of datasets for learning to learn from few examples. In *International Conference on Learning Representations*, 2020.

Hung-Yu Tseng, Hsin-Ying Lee, Jia-Bin Huang, and Ming-Hsuan Yang. Cross-domain few-shot classification via learned feature-wise transformation. In *International Conference on Learning Representations*, 2020.

Maria Tsimpoukelli, Jacob L Menick, Serkan Cabi, SM Eslami, Oriol Vinyals, and Felix Hill. Multimodal few-shot learning with frozen language models. *Advances in Neural Information Processing Systems*, 34:200–212, 2021.

Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, Nicolas Heess, and Yuval Tassa. dm_control: Software and tasks for continuous control. *Software Impacts*, 6: 100022, 2020.

A.M. Turing. Computing Machinery and Intelligence. *Mind*, LIX:433–460, October 1950.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Johannes Von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. In *International Conference on Machine Learning*, pages 35151–35174. PMLR, 2023.

Neha Wadia, Daniel Duckworth, Samuel S Schoenholz, Ethan Dyer, and Jascha Sohl-Dickstein. Whitening and second order optimization both make information in the dataset unusable during training, and can reduce or prevent generalization. In *International Conference on Machine Learning*, pages 10617–10629. PMLR, 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. *arXiv preprint arXiv:1611.05763*, 2016.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

M Wiering and J Schmidhuber. HQ-Learning: Discovering Markovian Subgoals for Non-Markovian Reinforcement Learning. Technical Report IDSIA-95-96, IDSIA, 1996a.

Marco A Wiering and Jürgen Schmidhuber. Solving pomdps with levin search and eira. 1996b.

Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. Natural evolution strategies. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 3381–3387. IEEE, 2008.

R J Williams. On the Use of Backpropagation in Associative Reinforcement Learning. In *IEEE International Conference on Neural Networks, San Diego*, volume 2, pages 263–270, 1988.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

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.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR*, abs/1708.0, 2017.

Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-gradient reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

Zhongwen Xu, Hado P van Hasselt, Matteo Hessel, Junhyuk Oh, Satinder Singh, and David Silver. Meta-gradient reinforcement learning with an objective discovered online. *Advances in Neural Information Processing Systems*, 33:15254–15264, 2020.

Jaesik Yoon, Taesup Kim, Ousmane Dia, Sungwoong Kim, Yoshua Bengio, and Sungjin Ahn. Bayesian model-agnostic meta-learning. *Advances in neural information processing systems*, 31, 2018.

Kenny John Young, Aditya Ramesh, Louis Kirsch, and Jürgen Schmidhuber. The benefits of model-based generalization in reinforcement learning. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 40254–40276. PMLR, 23–29 Jul 2023.

Tianhe Yu, Chelsea Finn, Annie Xie, Sudeep Dasari, Tianhao Zhang, Pieter Abbeel, and Sergey Levine. One-shot imitation from observing humans via domain-adaptive meta-learning. *International Conference on Learning Representations, Workshop Track*, 2018.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. *Advances in neural information processing systems*, 30, 2017.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation. *arXiv preprint arXiv:2310.02304*, 2023.

Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. Omni: Open-endedness via models of human notions of interestingness. *arXiv preprint arXiv:2306.01711*, 2023a.

Yiyuan Zhang, Kaixiong Gong, Kaipeng Zhang, Hongsheng Li, Yu Qiao, Wanli Ouyang, and Xiangyu Yue. Meta-transformer: A unified framework for multimodal learning. *arXiv preprint arXiv:2307.10802*, 2023b.

Zeyu Zheng, Junhyuk Oh, and Satinder Singh. On learning intrinsic rewards for policy gradient methods. In *Advances in Neural Information Processing Systems*, pages 4644–4654, 2018.

Andrey Zhmoginov, Mark Sandler, and Maksym Vladymyrov. Hypertransformer: Model generation for supervised and semi-supervised few-shot learning. In *International Conference on Machine Learning*, pages 27075–27098. PMLR, 2022.

Allan Zhou, Tom Knowles, and Chelsea Finn. Meta-learning symmetries by reparameterization. In *International Conference on Learning Representations*, 2021.

Allan Zhou, Kaien Yang, Kaylee Burns, Adriano Cardace, Yiding Jiang, Samuel Sokota, J Zico Kolter, and Chelsea Finn. Permutation equivariant neural functionals. *Advances in neural information processing systems*, 36, 2024.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

Mingchen Zhuge, Haozhe Liu, Francesco Faccio, Dylan R Ashley, Róbert Csordás, Anand Gopalakrishnan, Abdullah Hamdi, Hasan Abed Al Kader Hammoud, Vincent Herrmann, Kazuki Irie, et al. Mindstorms in natural language-based societies of mind. *arXiv preprint arXiv:2305.17066*, 2023.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024.

Luisa Zintgraf, Zita Marinho, Iurii Kemaev, Louis Kirsch, Junhyuk Oh, and Tom Schaul. Rl2x: Reinforcement learning to explore. In *The Multi-disciplinary Conference on Reinforcement Learning and Decision Making*, 2022.