Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Investigation of self-learned zeroth-order optimization algorithms

Kilian Schüttler

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Dirk Pflüger
Supervisor:	Peter Domanski, M.Sc.
Commenced:	February 2, 2022
Completed:	August 2, 2022

Abstract

Designing optimization algorithms manually is a laborious process. In Addition, many optimization algorithms rely on hand-crafted heuristics and perform poorly in applications for which they are not specifically designed. Thus, automating the algorithm design process is very appealing. Moreover, learned algorithms minimize the amount of a priori assumptions and do not rely on hyperparameters after training. Several works exist that present methods to learn an optimization algorithm. In this project, we focus on the reinforcement learning perspective. Therefore, any particular optimization algorithm is represented as a policy. Evaluation of the existing methods shows, learned algorithms outperform existing algorithms in terms of convergence speed and final objective value on particular training tasks. However, the inner mechanisms of learned algorithms largely remain a mystery. A first work has discovered that learned first-order algorithms show a set of intuitive mechanisms that are tuned to the training task. We aim to explore the inner workings of learned zeroth-order algorithms and compare our discoveries to previous works. To address this issue, we study properties of learned zeroth-order algorithms to understand the relationship between what is learned and the quantitative and qualitative properties, e.g., curvature or convexity of the objective function. Furthermore, we study the generalization in relation to these properties. Moreover, we explore the feasibility of finetuning a learned zeroth-order optimization algorithm to a related objective function. Finally we provide guidelines for training and application of learned zeroth-order optimization algorithms.

Contents

1 Introduction

Optimization algorithms are of utmost importance to all domains of science and engineering. While algorithms like gradient descent or conjugate gradients are capable of solving a large percentage of optimization problems, they do require the objective function to be differentiable. There are many cases, where computing a gradient is not feasible or even impossible. This is why zeroth-order optimization algorithms like Nelder-Mead and Powell exist. They treat the objective function as a black box and require only evaluations at specific locations. While use cases already exist, for example in post-silicon validation [DPRL21], one can imagine, to apply zeroth-order algorithms instead of computing backpropagation could lead to highly distributed and thus more efficient optimization of deep learning models. [BRL21]
The current approach to designing zeroth-order optimization algorithms, specifically fitted to a task, is a laborious process that requires the designer to devise an algorithm by intuition, theoretical and empirical insight as well as general design paradigms. The designer then needs to study the algorithm's performance and compare it to the performance of existing algorithms. If the devised method falls short, the designer needs to uncover the underlying cause and find ways to overcome them. This process must be iterated until the algorithm outperforms existing methods. Given the complexity of this process, the natural question is: can we automate it?
Unsurprisingly, this question has been asked before and previous works [ADG+16; CHC+16; LJL17; LM16; RXR+19; WMH+17] have already shown, learned optimization algorithms can outperform human-designed algorithms on specific supervised training tasks including linear regression and neural network optimization. This is why more recently the research focus has shifted from showing the feasibility of learned optimizers to understanding their inner workings [MSM+20] which is the focus of our work. Such insight into the mechanisms of learned optimization algorithms is required to improve their performance and applicability. Moreover, we might discover previously unknown mechanisms which improve upon human-designed algorithms.

One of the biggest problems current learn-to-optimize approaches face is the large upfront computational cost associated with learning an optimization algorithm from scratch. The problem of large computational cost is ubiquitous throughout the domain of machine learning as models reach ever greater numbers of parameters to improve upon performance. To address this issue, the use of foundation models [BHA+21] has gained popularity in many areas of machine learning research. A foundation model refers to a machine learning model which has been trained on a large number of tasks to adapt it to a related downstream task. We refer to this process as pretraining and finetuning. This process is widely used in the domain of natural language processing (NLP) where large transformer models [DCLT18] are pretrained unsupervised on large language datasets and then finetuned to perform supervised NLP tasks such as named entity recognition or text classification [SQXH19]. The use of such foundation models can be adapted to the domain of learn-to-optimize by pretraining an optimization algorithm on a large set of objective functions and finetuning it to a specific objective function down the line. We imagine machine

learning libraries such as TensorFlow implementing a pretrained optimization algorithm specifically designed for a set of models which the user then finetunes to their implementation of the model.

Since we learn better optimization algorithms, by observing their execution, we decided on using a reinforcement learning (RL) approach. RL minimizes the necessary amount of a priori assumptions about the objective functions, does not require an elaborate data generation setup like recurrent neural network based approaches [ADG+16], and does not require the use of gradients. Additionally, RL has been shown to be effective at learning such optimization algorithms by a previous work [LM16]. Under this framework, any particular optimization algorithm corresponds to a policy. In the context of RL, a policy represents a parameterized function, which is trained to maximize the sum of rewards. We reward optimization algorithms for quick convergence speeds and quality of solution and penalize the opposites. Learning an optimization algorithm is therefore synonymous with finding an optimal policy, which can be achieved by any RL algorithm. We use off-the-shelf RL algorithms as provided by the TFAgents library such as REINFORCE [Wil92] and PPO [SWD+17]. Both are capable of solving many RL tasks such as robotic control. To distinguish the learned optimization algorithm from the algorithm that performs learning, we will refer to the learned algorithm as 'optimization algorithm' or 'policy' and the learning algorithm that performs the optimization of a policy as the 'agent'.

In this work, we study the behavior and mechanisms of learned zeroth-order algorithms with respect to the quantitative and qualitative properties of the objective function. If we train an optimization algorithm on a specific objective function, which mechanisms does it learn and how well do these generalize to a small change in the objective function or an entirely different objective function? Further, we limited the optimization problems to constrained 2-dimensional objective functions to more easily study the trajectories taken by our learned optimization algorithms. Additionally, we demonstrate and discuss the feasibility of finetuning a learned optimization algorithm to a new objective function to reduce the upfront cost of learn-to-optimize. Moreover, we use the results of our experiments to discover which properties of objective functions are relevant for a learned optimization algorithm to generalize between them. Furthermore, we will discuss the implementation of a testbench capable of performing learn-to-optimize experiments at scale. Finally, we will give a guideline for training a policy using our testbench.

2 Related works

Our work is based on recent works using neural networks (NN) to learn optimization algorithms. In [LM16] Li et al. use a reinforcement learning (RL) approach very similar to ours to learn an optimization algorithm. In comparison to our method, they are using first-order information in their learned optimization algorithm. They have shown that their approach is capable of outperforming human-designed first-order algorithms in a few classic learning scenarios by converging more quickly.
While Li et al. are using a RL approach, most related works depend on the structure of supervised learning and recurrent neural networks (RNN) to model the iterative process of optimization. This requires designing an elaborate data generation setup. Examples include an early work [ADG+16] where Androchowicz et al. use supervised learning with a RNN to learn NN optimizers and show that their approach is superior in convergence speed as well as final objective value compared to typical NN optimizers such as ADAM or RMSProp. In [LJL17] Lv et al. improved this method on robustness by introducing tricks to enhance training data like random scaling. Due to these tricks, their policy converges more slowly than the previous method but can generalize to longer horizons and therefore reach a better final objective value. In [WMH+17] Wichrowska et al. improve on Androchowicz et al.'s work with a novel hierarchical architecture capable of scaling to differently sized optimization tasks. Further, their approach adapts to different problems by performing meta training where an optimizer is trained on many different optimization problems before evaluation. Their work shows that learned optimizers are capable of generalizing to different optimization problems with comparable performance to classical optimizers.
In [CHC+16] Chen et al. also build upon Androchowicz et al.'s work but adapts their method to zeroth-order optimization. Their approach was shown to match the performance of human-designed zeroth-order algorithms with fixed horizon optimization but falls short for longer horizons.
In [RXR+19] Ruan et al. propose a different architecture for supervised learned zeroth-order optimization by splitting the algorithm into a query and update section where both sections work together but are trained consecutively. They show their approach to be vastly superior to human-designed zeroth-order algorithms on their chosen training tasks. However, they only chose to optimize simple NN models and not go into detail about the generalization capabilities of their model.
In [MSM+20] Maheswaranathan et al. uncover the mechanisms learned first-order optimization algorithms use to outperform human-designed algorithms by treating the optimization process as a dynamic system. They show these optimizers use gradient descent with momentum, gradient clipping, and a learning rate schedule. These mechanisms are also used by human-designed gradient descent methods. Additionally, they uncovered some previously unknown mechanisms such as differences in optimization strategy for each layer in a NN. Furthermore, they visualize the trajectories taken by their learned optimization algorithms with common dimensionality reduction techniques such as the singular value decomposition (SVD).
In [BRL21] Bhargava et al. demonstrate an approach different from all previous methods. With a focus on training neural nets without gradients, they also use reinforcement learning similar to our

approach but instead of learning to optimize the entire NN with a single optimizer, they formulate each node of a NN as a synapse that learns to optimize itself. They achieve this by learning a policy that controls the weight adjustment for a single synapse based only on the previous weights of the synapse and the total loss of the NN. The same policy is applied to each synapse allowing for any NN architecture. Although their approach doesn't converge very quickly due to their very small and fixed weight adjustment, they show their method is capable of reaching a comparable final objective value to a standard NN optimizer.

3 Background

In this section, we will introduce important concepts we use in our approach.

3.1 Neural networks

To learn optimization algorithms we decided on a machine learning approach. To be able to learn a complex task like an optimization algorithm, we require a powerful model. Therefore, we decided on using neural networks (NN). NNs are powerful function approximators, which are capable of handling complex data. They minimize the amount of a priori assumptions about the function they are approximating as they extract the relevant features automatically during training. While training a NN requires large amounts of data and computational power, once training is done, inference is fast and easy to compute. Finally, the use of NNs represents the current state-of-the-art in learn-to-optimize with deep reinforcement learning.

3.1.1 Multi layer perceptron

Since NNs are used to approximate functions, lets define an arbitrary function $f : \mathbb{R}^a \to \mathbb{R}^b$. We can illustrate the architecture of a multi layer perceptron (MLP) with an acyclic directed graph as shown in Fig. 3.1. The graph begins with the input layer, with a input nodes, represented by the vector $X \in \mathbb{R}^a$. From here we have multiple hidden layers, represented by the vector $N_i \in \mathbb{R}_i^c$, each with multiple hidden nodes n_{ij}. Finally, there is an output layer $Y \in \mathbb{R}^b$, corresponding to the output of f. For every layer, including the input layer, we draw a directed edge from each node to each node of the next layer. This way we get a dense graph, where every node of every adjacent layer is connected by an edge. These edges form a flow from the input to the output layer. Hence the name feedforward network. Now we assign the input values to the input nodes. For every other

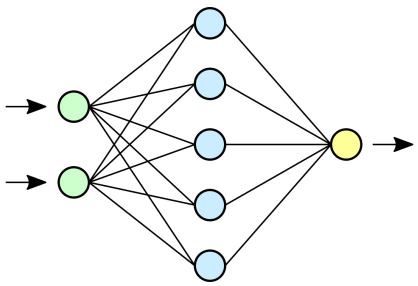

Figure 3.1: Illustration of a multi layer perceptron with two input nodes, a single hidden layer with a breadth of five, and a single output node

node, we assign a value based on its connection to nodes of the previous layer. Along an edge from source node n_{i-1j} to destination node n_{ik} we multiply $n_{(i-1)j}$ with a weight value $w_{ijk} \in \theta$ and add a bias value $b_{ik} \in \theta$, where θ represents the trainable network parameters. These parameters will be updated, such that a performance metric is optimized, for example, the mean squared error as described in Eq 3.4. If we do this for every node the output nodes will eventually be populated with the output values. Every layer N_i of this construct can be expressed recursively by a matrix multiplication and a vector addition including the output of the previous layer Y_{i-1} as in Eq. 3.1

$$N_i = \mathbf{W}_i Y_{i-1} + B_i, \tag{3.1}$$

with $\mathbf{W}_i, B_i \in \theta$

Since this equation is strictly linear, we need to introduce some form of non-linearity in order to be able to approximate non-linear functions. Therefore, we use non-linear activation functions after every layer. In practice many different activation functions are available, but in our case, we exclusively used the Rectified Linear Unit activation function ReLU, due to its computational simplicity and proven effectiveness [Aga18]. Other activation functions include the sigmoid function or the gaussian error linear unit (GELU). The sigmoid function has been used in the past, but it is too costly to compute. The GELU function has gained popularity, due to its advantages in deep learning by addressing the vanishing gradient problem.
The ReLU activation function is defined for an arbitrary scalar input x as:

$$\text{ReLU}(x) = \max(0, x) \tag{3.2}$$

expanded to n layers we get the output function $o(X)$ as defined in Eq. 3.3, the colors of the terms correspond to the coloring of the layers in Fig. 3.1.

$$o(X) = \mathbf{W}_n ReLU(\mathbf{W}_{n-1} ReLU \ldots ReLU(\mathbf{W}_0 X + B_0) \ldots + B_{n-1}) + B_n, \tag{3.3}$$

To optimize a neural net, we define a loss function. The optimal choice of loss function depends on the task. For regression problems, a commonly used loss function is the mean squared error loss (MSE) as defined in Eq. 3.4. Other typical loss functions for regression include the mean absolute error (MAE). For classification tasks, the cross-entropy loss is typically used. We exclusively use the MSE loss function.

$$MSE = \frac{1}{N} \sum_{i}^{N} = 1(f(X_i) - o(X_i))^2, \tag{3.4}$$

With a chosen loss function, we can optimize the MLP, with respect to the trainable parameters θ, by performing gradient descent. We compute the gradients with the backpropagation algorithm. Given an optimal solution is reached, a MLP can approximate an arbitrary continuous function, as such: $|f(X) - o(X)| < \epsilon$ [LLPS93].
In conclusion, while the concept of a neural network can approximate arbitrary functions, it can require massive amounts of data and computational capabilities to reach a good approximation, especially for complex functions.

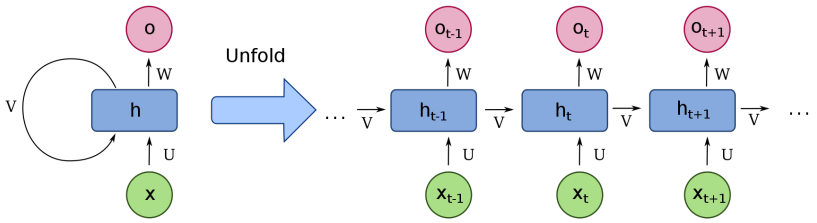

Figure 3.2: A basic recurrent neural network, compressed (left) and unfolded (right)

3.1.2 Recurrent Neural Network

In the case of predicting time series data, where each iteration depends on the previous iterations, a stateless model such as a MLP isn't suitable. It is possible to include the entirety or a part of the time series in the input of the MLP such that it may capture the temporal dependencies. But this increases the complexity of the function to be approximated and thus the data and compute requirements to reach a good solution. So for time series data with long horizons and temporal dependencies, this is unfeasible. Alternatively we could use a recurrent neural net (RNN). RNNs can handle sequential inputs by including a type of memory cell into their architecture. The state of the memory cell, which is called the hidden state H_t, is used in the output function in addition to the current input. The hidden state is updated based on the previous hidden state H_{t-1} and current input X_t. With this memory cell, the network will learn what information from previous iterations to keep and what to forget. The updates in a single iteration are defined as such:

(3.5) $H_t = \mathbf{V}H_{t-1} + \mathbf{U}X_t + B_H,$

with $\mathbf{V}, \mathbf{U}, B_H \in \theta$

(3.6) $o_t(H_t) = \mathbf{W}_t H_{t-1} + B_o$

with $\mathbf{W}, B_o \in \theta$

Fig 3.2 shows the structure of such a basic RNN and illustrates how it unfolds over time while predicting time series data. To train a RNN, we require time series data and a loss function. The loss function follows the same description as in section 3.1.1. We perform training by gradient descent and compute the gradient with the backpropagation through time algorithm. This formulation of a RNN isn't widely used in practice but illustrates the basic principle. This type of RNN struggles to capture long-term dependencies as it has problems with vanishing or exploding gradients over time. Therefore, we used the Long short term memory architecture described in [HS97] as its design addresses the vanishing or exploding gradient problem.

Long short term memory

In a Long short term memory architecture (LSTM) there is the hidden state h_t and the cell state C_t with each of them being passed between iterations. With this architecture the cell state represents the memory cell, which is able to directly decide which input to forget and which to remember.

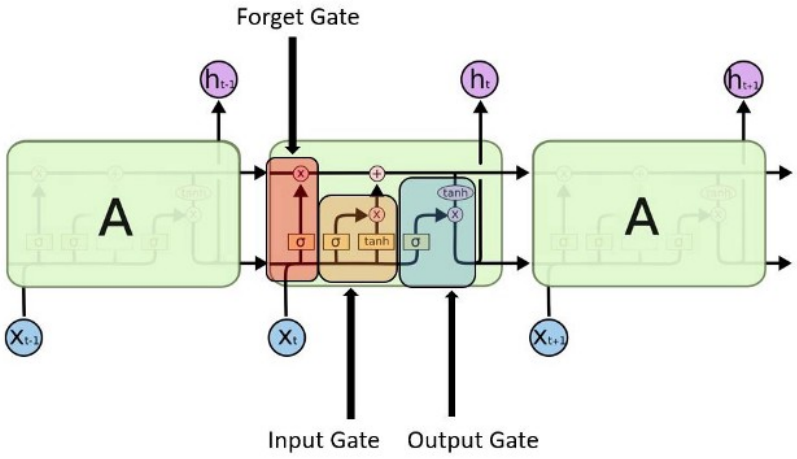

Figure 3.3: Structural overview of the LSTM architecture,that shows the forget gate, input gate and output gate

Their updates are handled by so called gates as illustrated in Fig. 3.3.

Forget gate:
The forget gate decides which values to omit from the previous cell state, by building a mask f_t based on the input an hidden state.

$$f_t = \sigma\left(\mathbf{W}_f \cdot [h_{t-1}, x_t] + B_f\right) \tag{3.7}$$

with $\mathbf{W}_f, B_f \in \theta$

Input gate:
The input gate determines which information from the input and hidden state is used to add to the cell state. $\tilde{C}_t$ represents the proposed updates and i_t represents a mask for these updates.

$$\begin{aligned} i_t &= \sigma\left(\mathbf{W}_i \cdot [h_{t-1}, X_t] + B_i\right) \\ \tilde{C}_t &= \tanh\left(\mathbf{W}_C \cdot [h_{t-1}, X_t] + B_C\right) \end{aligned} \tag{3.8}$$

with $\mathbf{W}_i, \mathbf{W}_C, B_i, B_C \in \theta$

Cell state:
The cell state is updated by performing an element-wise multiplication of the forget mask f_t with the previous cell state C_{t-1}, and the input mask i_t on the proposed cell state update $\tilde{C}_t$.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{3.9}$$

Output gate:
The output gate builds the output function of the cell depending on the input X_t and previous hidden

state h_{t-1}. It also updates the hidden state depending on the output o_t and already updated cell state C_t.

$$\begin{aligned} o_t &= \sigma\left(\mathbf{W}_o\left[h_{t-1}, X_t\right] + B_o\right) \\ h_t &= o_t * \tanh\left(C_t\right) \end{aligned} \tag{3.10}$$

with $\mathbf{W}_o, B_o \in \theta$

3.2 Reinforcement learning

Reinforcement learning refers to several methods of machine learning, where an agent ought to take actions in an environment to maximize the sum of rewards. It is one of the three basic machine learning paradigms alongside supervised learning and unsupervised learning. Reinforcement learning was motivated by trying to model how humans learn. We don't observe large amounts of data but learn by observing how our actions affect our environment and then changing the actions we take. Therefore, reinforcement learning differs from supervised learning in not needing labeled input and output pairs. Instead, the agent generates the data required for training during the training process itself by interacting with the environment and receiving feedback from it in form of a reward. Moreover, minimal a priori knowledge about the data itself is required. We only need to formulate a problem-specific environment.
As reinforcement learning is inherently an iterative process, it is especially useful for modeling time-dependent problems, such as (robotic) control problems or playing games. Examples of this include the DeepMind AlphaStar agent, which in 2019 beat one of the best human professional players in the online real-time-strategy game SStarCraft II". More recently they have adapted their agent to partly solve protein folding and improve YouTube's video compression algorithm. Under the premise of learn-to-optimize, we can list the advantages of reinforcement learning as not needing prior knowledge of the functions we are optimizing, no need for including gradients, and general ease of implementation. The only downside we can identify is the large amount of computation required to train an agent. Further, deep reinforcement learning represents the current state-of-the-art in learning optimization algorithms. Although supervised learning approaches with RNNs are also performing to the same standard. They require a more elaborate setup, as the data generation has to be designed manually.

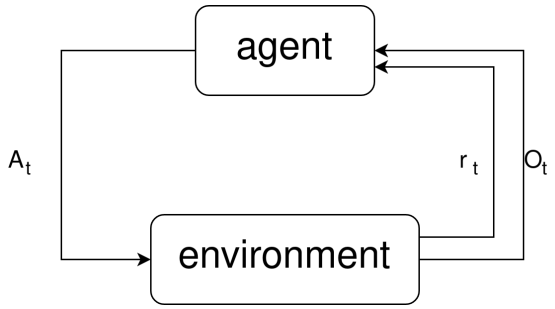

Figure 3.4: Basic reinforcement learning structure

3.2.1 Markov Decision Process

Reinforcement learning is typically formulated as a Markov decision process (MDP). We chose a finite-horizon MDP with continuous state and action spaces as defined by the tuple $(\mathcal{O}, \mathcal{A}, p_0, p, r, \gamma)$, with $\mathcal{O}$ as the set of observations, $\mathcal{A}$ as the set of actions, $p_0 : \mathcal{O} \to \mathbb{R}^+$ is the probability density over initial observations, $p : \mathcal{O} \times \mathcal{A} \times \mathcal{O} \to \mathbb{R}^+$ is the transition probability density, $r : \mathcal{O} \to \mathbb{R}$ is a function that maps an observation to a reward and $\gamma \in (0, 1]$ is the discount factor.
The two main components of reinforcement learning are an environment and an agent as shown in Fig. 3.4. In order to collect a trajectory τ the environment provides the agent with an initial observation O_0, according to the probability density p_0, and reward signal $r(O_0)$. The agent then decides on an action A_0 based on the policy π. The action is passed to the environment, which returns a new observation O_1 according to p and reward signal $r(O_1)$. This loop continues until a stopping condition is met. So we get:

$$\tau = (O_0, A_0, O_1, A_1, \dots, O_T, A_T) \tag{3.11}$$

as a trajectory, which is taken from the probability density:

$$q(\tau) = p_0(O_0) \prod_{t=0}^{T} p(O_{t+1}|O_t, A_t)\pi(A_t|O_t) \tag{3.12}$$

The objective of the agent is to learn an optimal policy $\tilde{\pi} : \mathcal{O} \times \mathcal{A} \to \mathbb{R}^+$, which is a conditional probability density over actions given the current observation, such that the expected discounted sum of rewards is maximized. That is,

$$\tilde{\pi} = \arg\max_{\pi} \mathbb{E}_{\tau}\left[\sum_{i=0}^{n} \gamma^i r(O_i)\right]. \tag{3.13}$$

Therefore, reinforcement learning reduces down to a policy search problem. Many different algorithms exist to solve the policy search problem. They are loosely grouped into model-free and model-based, where model-free methods may be on-policy or off-policy.
On-policy methods try to solve the policy search problem by replacing the policy directly with a trainable parameterized function π_θ, like a NN. We call this the policy network. With this approach, the agent collects several trajectories, computes the cumulative reward, and performs policy gradient ascent. These methods are, generally speaking, computationally efficient compared to off-policy methods. Disadvantages are the low sample efficiency. To compute the gradient, the trajectory has to be sampled with the current policy. Therefore, for every training iteration we need to sample new trajectories. A common extension to on-policy algorithms is the inclusion of a value function, which computes the possible sum of rewards for the current observation. The value function is used as an input to the policy to provide additional information about the quality of the current state. In practice, this value function is replaced by a parameterized function and approximated during training. We call this the value network. Examples of on-policy algorithms include REINFORCE [Wil92] and Proximal policy optimization (PPO) [SWD+17].

REINFORCE

REINFORCE estimates the gradient of the expected reward $\nabla_\theta \mathbf{E}_{\tau\sim q}[R_0]$,
with R_0 being the sum of discounted rewards $\sum_{i=0}^{n} \gamma^i r(O_i)$.

Therefore, the function π_θ must be differentiable with respect to its parameters θ. This is the case if the policy is replaced with a NN.
From the definition of the expected value we get:

$$\begin{aligned}\nabla_\theta \mathbf{E}_{\tau\sim q}[R_0] =& \nabla_\theta \int R_0 q(\tau) d\tau = \int R_0 \nabla_\theta q(\tau) d\tau \\ =& \int R_0 \nabla_\theta \log(q(\tau)) q(\tau) d\tau = \mathbf{E}_{\sim \mathbf{q}}\left[R_0 \nabla_\theta \log(q(\tau))\right]\end{aligned} \tag{3.14}$$

, with

$$\nabla_\theta \log(q(\tau)) = \sum_{t=0}^{T} \nabla_\theta \log(\pi_\theta(O_t, A_t)) \tag{3.15}$$

we get an unbiased estimator $\hat{\mathbf{E}}_\tau \sim q\left[R_0\right]$ for the gradient of the expected reward. By collecting a trajectory τ and then computing

$$\hat{\mathbf{E}}_\tau \sim q\left[R_0\right] = R_0 \sum_{t=0}^{T} \nabla_\theta \log(\pi_\theta(O_t, A_t)), \tag{3.16}$$

we can perform standard gradient ascent to optimize the parameterized policy π_θ.

Proximal policy optimization

PPO is an algorithm similar to REINFORCE as it is also an on-policy algorithm. Its key difference lies in its loss function. This loss function limits the increase in likelihood of an action during a single training iteration. It only limits the likelihood if the actual increase in reward is higher than the expected increase, or if the actual decrease in reward is lower than the expected decrease. This idea is inspired by the observation, that training a policy can be quite unstable. If the algorithm takes a step in the right direction, we do not want to overshoot the optimal solution because the agent might end up in a sub-optimal solution. PPO makes up for this limit in stepsize by doing multiple training sub-iterations on the same data before collecting a new set of trajectories.

Off-policy methods are different in that they do not replace the policy directly with a trainable parameterized function, but use a trainable parameterized function $Q : O \times A \to \mathbb{R}$ which predicts a distribution rating across all possible actions for an observation. This function is closely related to the value function sometimes used in on-policy approaches. More specifically, the value function computes the maximum of the Q function. The policy is replaced by a arg max function to decide which action is the best for the current observation. Off-policy methods converge more slowly and are not guaranteed to converge since they try to estimate a distribution over the entire action and observation space. However, they can use any trajectory to train the Q function and thus be

more sample efficient. Examples of off-policy algorithms include deep Q-networks (DQN) and soft actor-critic (SAC).

Model-based policy search algorithms are entirely different and are engineered to suit a specific environment. A model-based agent has prior knowledge of the environment he acts in. This is in form of a simulation of the environment or a heuristic, which pre-rates the actions before the policy decides on one. For example, if we wanted to train a model-based agent to play chess we could provide this agent with the rules of the game such that the agent can compute possible future moves and search the space of actions.

3.3 Zeroth-order optimization

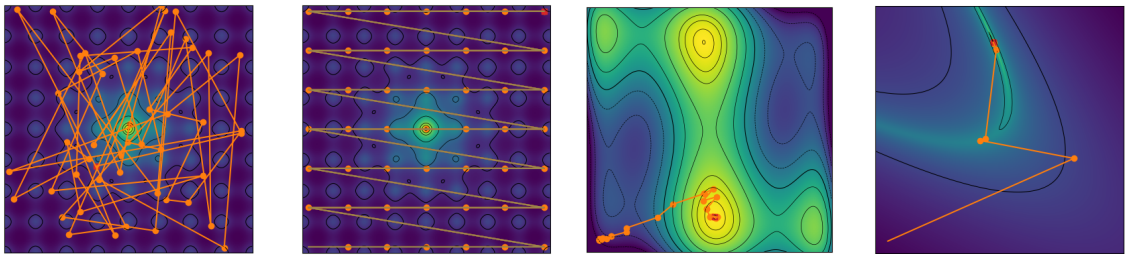

Figure 3.5: Example trajectories of four zeroth-order optimization algorithms, from the left the first image shows random search on Ackley function, second shows grid search on Ackley function, third shows Nelder-Mead on Camel function and the third shows Powell on Rosenbrock function

In the context of optimization, zeroth-order refers to the absence of gradients during optimization. Therefore, the algorithms treat the function as a black box and use evaluations of the function only. The most common type of optimization is gradient descent, which is capable of solving many optimization tasks, especially in the context of machine learning. Due to gradient descent's prevalence in the training of machine learning models, the models are designed with differentiability in mind. If a zeroth-order optimization algorithm capable of competing with gradient descent's performance exists, it would broaden the scope of possible machine learning models considerably. Since the focus of this thesis is exploring zeroth-order optimization and how they compare to human-designed zeroth-order optimization algorithms, we will introduce four different human-designed zeroth-order algorithms which serve as a baseline to compare our learned algorithms against. Fig. 3.5 shows an example trajectory for each of our chosen zeroth-order algorithms.

3.3.1 Random search

Random search is a simple zeroth-order algorithm: For a fixed number of iterations the algorithm evaluates random locations inside the domain of the objective function and at the end of the iterations it returns the best location.

The greatest advantage of random search is its simplicity. Furthermore, it does not require any hyperparameters except the number of iterations. For especially large search spaces, where an exhaustive search or grid search is not feasible, random search might still find good solutions. Although due to its random nature no guarantees for convergence or quality of solution can be given. So finding a good solution can be very time-consuming.

3.3.2 Grid search

Grid search is a simple zeroth-order algorithm: For a fixed number of evaluations, it evaluates the objective function at equidistant locations covering the entire domain and returns the best location. The greatest advantage of grid search is the same as random searches: its simplicity. While it is a good algorithm to almost exhaustively search small domains it suffers from the curse of dimensionality where the time complexity grows exponentially with dimensions given the granularity should remain the same. Another characteristic of grid search is that it does not treat different parameters independently as its evaluation locations are spread evenly regardless of the shape of the domain. For some tasks, this is beneficial while for others it is not.

3.3.3 Nelder-Mead

The Nelder-Mead method is a more complex algorithm. It tries to mimic gradient descent by estimating the gradient with multiple evaluations in close proximity. For an objective function of dimensionality d, it does $d + 1$ guesses which form a simplex around the current location in the domain. A simplex is the simplest possible geometric form, which fills the space it occupies. For one dimension this is a line, for two dimensions a triangle, and for three a tetrahedron. So for a two-dimensional function, the algorithm takes three guesses in an equal triangle around the location. With this simplex, it tries to estimate the gradient at the location and then takes a step in the direction of the estimated gradient.
This method requires many evaluations but is capable of solving any convex problem. It shares many of the benefits, but also shortcomings of the gradient descent method. Nelder-Mead does not include a momentum term and thus has a tendency to get stuck in local minima. It also has many hyperparameters, like learning rate or simplex size which need to be tuned.

3.3.4 Powell

Powell's optimization algorithm is a complex optimization algorithm. It is initialized with a set of n search directions $\mathcal{S}_0 = \{s_1, \dots, s_n\}$. Usually, the unit vectors along each coordinate axis are chosen. The first step is to optimize each initial search direction. Let the minima found along each search direction be $\{x_0 + \alpha_1 s_1, x_0 + \sum_{i=1}^{2} \alpha_i s_i, \dots, x_0 + \sum_{i=1}^{N} \alpha_i s_i\}$, with x_0 being the starting location and α_i the scalar determined by the linear search along search direction s_i. The new position can be expressed as a linear combination of the search vectors $x_1 = x_0 + \sum_{i=1}^{N} \alpha_i s_i$. The new displacement vector $\sum_{i=1}^{N} \alpha_i s_i$, is added to $\mathcal{S}_0$, whereas the search direction, which contributed the most to the displacement vector $i_{max} = \arg\max_{i \leq N} \alpha_i s_i$ is removed from $\mathcal{S}_0$. Therefore, the new set of search directions is $S_1 = \{s_1, \dots, s_{i_{max}-1}, s_{i_{max}+1}, \dots, s_n\}$. The algorithm is repeated until convergence is reached.

The biggest advantage of Powell's method is the quality of its solutions. Especially on polynomial functions of degree two, it is guaranteed to converge to the optimal solution. Disadvantages are the required number of evaluations to perform the linear searches. Furthermore, its complexity makes the algorithm difficult to implement. Finally, the algorithm requires the tuning of hyperparameters like the initial search directions. These greatly influence the required number of iterations. Moreover, choosing a method for the linear searches and the hyperparameters that come with that method.

4 Methodology

4.1 Formulation

In this section, we formulate an optimization task as a reinforcement learning problem. Algorithm 4.1 shows the pseudocode of a general constrained optimization algorithm. Starting from a random location x_0 in the domain Θ of the objective function f, until a stopping condition or a maximum number of steps is reached, the function π : computes a new location $x_i \in \Theta$ with regard to all previous iterations. After the stopping condition or step limit is reached, the latest location x_n is returned as the solution.

Algorithm 4.1 Optimization pseudocode

```
f := objective function
x_0 := random point in Θ
for i in 1, 2, ... do
    if stopping condition is met then
        return x_{i-1}
    end if
    x_i := π(f, {x_0, ..., x_{i-1}})
end for
```

This pseudocode summarizes the general structure of most optimization algorithms. Algorithms differ only in choice and application of π. For example equation 4.1 defines π, such that it only depends on the previous step added to the gradient of f multiplied with a scalar learning rate γ. This choice of π forms the basic gradient descent method.

$$\pi(f, \{x_0, ..., x_{i-1}\}) = x_{i-1} - \gamma \nabla f(x_{i-1}) \tag{4.1}$$

Since we are focusing on zeroth-order optimization, we are trying to learn a choice of π which does not depend on the gradient of the objective function. Eq. 4.2 defines π, such that it does not depend on the gradient of f, but direct evaluations of f only. Further, the choice of π describes the random search algorithm as described in section 3.3.

$$\pi(f, \{x_0, ..., x_{i-1}\}) = \begin{cases} \arg\max_{x_0, ..., x_{i-1}} f(x), & \text{if } i < N_{iterations} \\ \text{random } x \in \Theta, & \text{otherwise} \end{cases} \tag{4.2}$$

With any choice of π possible, it is easy to see how this structure describes any optimization algorithm. Depending on what information we provide to the function π, we can use this structure to describe zeroth-, first- and second-order optimization algorithms. Further, to learn an optimization algorithm, we will formulate the above structure into a reinforcement learning problem by defining

a MDP, as defined in section 3.2. Therefire, we define the set of inputs of the function π as the set of observations $\mathcal{O}$, the set of outputs as the set of actions $\mathcal{A}$. The probability density of initial observations p_0 as an equal distribution across $\mathcal{O}$. The transition probability density p as a deterministic function described by the location updates $x_i := \pi(f, \{x_0, ..., x_{i-1}\})$. The policy to be learned is directly represented by the choice of π as described above. Thus, we denote the policy with π as well. To complete the formulation as a reinforcement learning problem, we need to define the reward function r. The performance of an optimization algorithm is expressed in both speed of convergence, and final objective value. Therefore, we decided to use the objective function directly as the reward function since it satisfies both criteria: slower convergence leads to a lower sum of rewards, so does a lower final objective value. Since we are using a finite-horizon definition of MDP, we may use the discount factor γ to weight the importance of either criterion. $\gamma > 1$ weights later iterations more and consequently incentivizes a better final objective value. $\gamma < 1$ weights earlier iterations more and incentivizes quicker convergence. With this formulation learning an optimization algorithm is synonymous with a policy search problem which can be solved by any of the algorithms described in Sec. 3.2. Additionally, if we parameterize the policy with a NN, the policy can approximate any function. Thus, we are searching the entire space of possible optimization algorithms.
In addition to the formulation in Alg. **??**, we implemented another variant. Instead of using the output of function π as the new location in the domain directly, we add the previous location x_{i-1} to the output as such:

$$x_i := x_{i-1} + \pi(f, \{x_0, ..., x_{i-1}\}) \tag{4.3}$$

We only need to adjust p from the previous definition of an MDP according to equation 4.3, to define a new reinforcement learning problem. To distinguish both variants, we will refer to the first variant as absolute and the second as relative.
Since the function π is computing a distribution over the possible actions and these actions are constrained, the main difference between the two variants is: The absolute variant may move freely throughout the entire domain, whereas the relative variant computes a step vector. Thus the absolute variant can only be used for constrained optimization tasks. Since the step vector is independent of its location in the domain, the relative variant may also be used for unconstrained optimization. Moreover, the absolute variant can evaluate any location in the entire domain at every step, whereas the relative variant is limited in its movement throughout the domain by a step size, as defined by the constraints on $\mathcal{A}$. With our choice of constraints, the relative variant can cover the entire domain with at most two steps.

4.2 Experiment design

In this section, we define the objective functions, operators, and normalization we used for our experiments. Additionally, we describe a standard evaluation, which we use to assign an overall performance score to every policy. Furthermore, we are going to define the terms in-distribution and out-of-distribution. Finally, we are going to introduce the relative performance correlation matrix.

4.2.1 Objective Functions

Since the goal of this work is to study what mechanisms the learned algorithms show dependent on the objective functions they have been trained on we decided to limit this work to 2D objective functions as this provides the ability to visualize the trajectories in a human understandable fashion and analyze the inner workings of the learned algorithms.

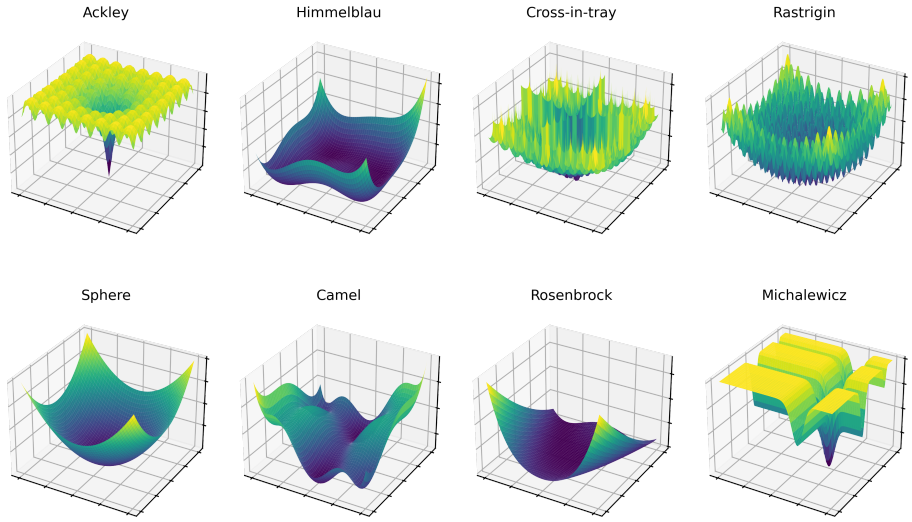

Figure 4.1: The eight used objective functions as 3D visualizations

We decided to use eight different objective functions. We chose each function such that every function shares a property with at least one other function but is also unique in its own way. This enables us to analyze which properties lead to the generalization of optimization performance and which do not. Fig. 4.1 shows a 3D visualization of each objective function as defined in this section.

Ackley

$$f(x,y) = -20\exp\left[-0.2\sqrt{0.5\,(x^2+y^2)}\right] - \exp[0.5(\cos 2\pi x + \cos 2\pi y)] + e + 20 \tag{4.4}$$

The Ackley function is a widely used function for testing optimization algorithms. It has many local minima on a mostly flat surface and one single global minimum, located in a steep valley, in the middle of the domain.

Domain: $x, y \in (-32, 32)$
Codomain: $f(x,y) \in (0, 13)$

Himmelblau

$$f(x,y) = \left(x^2 + y - 11\right)^2 + \left(x + y^2 - 7\right)^2 \tag{4.5}$$

The Himmelblau function has four global minima with no local minima. Due to the large codomain, it is usually shown in a logarithmic scale as in Fig. 4.2. When scaled logarithmically, the global minima are located in steep valleys.

Domain: $x, y \in (-6, 6)$
Codomain: $f(x,y) \in (0, 1586)$

Cross-in-tray

$$f(x,y) = -0.0001 \left(\left| \sin(x) \sin(y) \exp\left(\left| 100 - \frac{\sqrt{x^2 + y^2}}{\pi} \right| \right) \right| + 1 \right)^{0.1} \tag{4.6}$$

The Cross-in-tray function has many local minima which are separated by steep and high-reaching walls. It has four global minima located in a cross-like fashion in the middle of the domain, which are also separated by high and steep walls. The minima form a roughly parabolic shape.

Domain: $x, y \in (-15, 15)$
Codomain: $f(x,y) \in (-2.06261, 0)$

Rastrigin

$$f(x,y) = 20 + \left[x^2 - 10\cos(2\pi x)\right] + \left[y^2 - 10\cos(2\pi y)\right] \tag{4.7}$$

The Rastrigin function has many local minima which are separated by sinusoidal hills. It has only a single minimum in the middle of the domain. The minima form a roughly parabolic shape.

Domain: $x, y \in (-5.12, 5.12)$
Codomain: $f(x,y) \in (0, 80)$

Sphere

$$f(x,y) = x^2 + y^2 \tag{4.8}$$

The Sphere function is the most simple function in our selection and has a simple parabolic shape. It is the only strictly convex function.

Domain: $x, y \in (-1, 1)$
Codomain: $f(x,y) \in (0, 1)$

Camel

$$f(x,y) = \left(4 - 2.1x^2 + \frac{x^4}{3}\right)x^2 + xy + \left(-4 + 4y^2\right)y^2 \tag{4.9}$$

The Six-hump-camel function is a simple function. It features 6 local minima of which 2 are global. These local minima are easy to break through, as they are surrounded by gentle hills.

Domain: $x \in (-2, 2), y \in (-1, 1)$
Codomain: $f(x,y) \in (-1.0316, 0)$

Rosenbrock

$$f(x,y) = (1-x)^2 + 100\left(y - x^2\right)^2 \tag{4.10}$$

The Rosenbrock function is a widely used function in testing optimization algorithms and is complex to solve. At first look it shows a canyon surrounded by high walls. Finding this canyon is a relatively easy task. But inside the canyon it features many local minima with a hilly terrain. Finding the global minimum within the canyon proves to be difficult. With a logarithmic scale, as shown in fig. 4.2, the function reveals the hilly terrain inside the canyon to be quite steep and erratic.

Domain: $x, y \in (-2, 2)$
Codomain: $f(x,y) \in (0, 3609)$

Michalewicz

$$f(x,y) = -\sin(x)\sin^{20}\left(\frac{x^2}{\pi}\right) - \sin(y)\sin^{20}\left(\frac{2y^2}{\pi}\right) \tag{4.11}$$

The Michalewicz function shows many flat areas and local minima as flat canyons which makes it difficult to solve. In addition to the flat areas, it also has high and steep walls and only a single global minimum where its valleys meet.

Domain: $x, y \in (0, \pi)$
Codomain: $f(x,y) \in (-1.8013, 0)$

To summarize the properties of the objective functions we provide the table 4.1. The table shows the different properties in the rows and the different objective functions in the columns. The names have been shortened to the first three letters to make the table easier to visualize. An entry of '1' at location i, j means the objective function i has the property j. We can see the Ackley, Rastrigin, Sphere, Rosenbrock, and Michalewicz functions all have only a single global optimum. Camel has two global optima and the other two functions have four each. The Sphere function is the only convex function in our selection. While not convex, Himmelblau, Sphere, Camel, and Rosenbrock are polynomial. While the Ackley, Cross-in-tray, Rastrigin, Camel, Rosenbrock, and Michalewicz functions all show many local minima, only the Cross-in-tray and Rastrigin functions form a parabolic shape. The Cross-in-tray, Rastrigin, and Michalewicz functions have steep hills or walls

separating minima, whereas the Ackley, Himmelblau, Rastrigin, Rosenbrock, and Michalewicz functions have their global minima located in steep valleys. The Michalewicz and Rastrigin function have both. We could argue the Cross-in-tray function also has both. The minima are surrounded by steep walls. However, the valleys in between the walls have gentle curvatures. Hence, we decided not to classify them as steep valleys. The Rosenbrock and Michalewicz functions have canyons that lead to the optimum. Finally, the Ackley and Michalewicz functions share mostly flat areas. Whereas for the Ackley function this mostly flat area has many local minima, whereas Michalewicz has truly flat surfaces.

	Ack	Him	Cro	Ras	Sph	Cam	Ros	Mic
Single optimum	1			1	1		1	1
Multiple optima		1	1			1		
Convex					1			
Polynomial		1			1	1	1	
Many local minima	1		1	1		1	1	1
Minima in parabolic shape			1	1				
Steep hills/walls			1	1				1
Steep valleys	1	1		1			1	1
Canyons							1	1
(Mostly) flat areas	1							1

Table 4.1: A table showing the properties of each objective function, identifiable by their first three letters.

4.2.2 Normalization

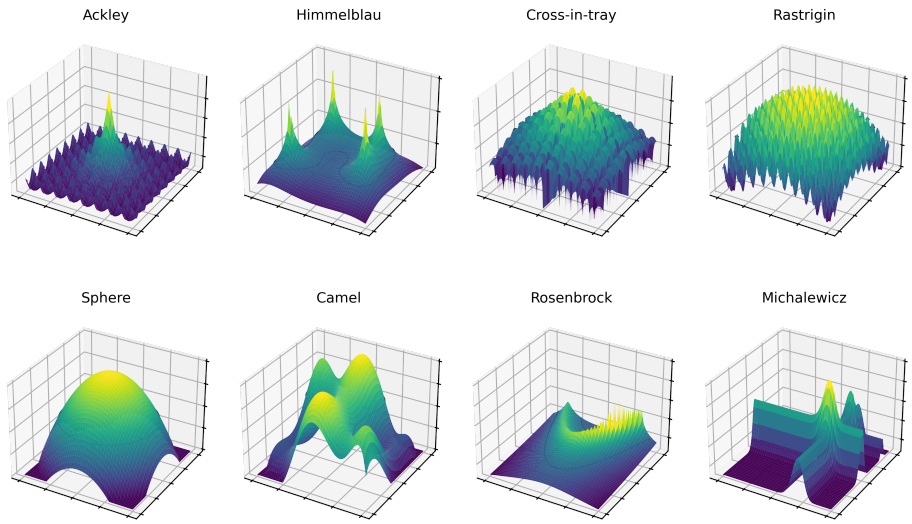

Figure 4.2: The eight objective functions after applying the normalization as 3D visualizations

As shown in section 4.2.1, the chosen objective functions have vastly different domains and codomains. While it is possible for a reinforcement learning agent to learn any of these domains and codomains, our goal is to study how well an agent can learn an optimization algorithm capable of generalizing to different objective functions. We designed our normalization such that the same agent may be trained and evaluated on any set of our normalized objective functions. Since a policy can only be trained and evaluated on the same action set and we use the actions as the updated location directly, the action spaces between environments must be equal. Therefore, the domain of every normalized objective function must be equal. We achieve this by a linear projection of the domain onto the (-1, 1) square.
Furthermore, if we train a policy with multiple objective functions we want the optimization performance for each objective function to affect the loss equally. Since we apply the objective value directly to the reward and the agent tries to maximize the sum of rewards (Eq. 3.13) we need the normalized objective value to measure a comparable optimization performance. Finally, we want our normalization to maintain the characteristics of the objective function. Therefore, we designed our normalization as such:

$$f_n := \frac{\int_\Theta f - f}{\int_\Theta f - \min(f)}, \tag{4.12}$$

with f as the objective function and Θ as its domain.

With Eq 4.12 follows:

$$\int_\Theta f_n = 0 \iff \mathbb{E}[f_n(X)] = 0, \text{ with } X \in \Theta \tag{4.13}$$

Further, $\max(f_n) = 1$. Therefore, the sum of rewards measures a change over the expected value of random evaluations with a maximum value of one per summand and thus represents a comparison of optimization performance between objective functions with vastly different codomains. Moreover, this normalization is strictly linear and consequentially does not affect the characteristics of the objective function.
Since the Rosenbrock and Himmelblau functions have such large codomains that they are typically visualized on a logarithmic scale, we scaled both functions logarithmically prior to our normalization.
Fig. 4.2 shows the resulting normalized objective functions as 3D visualizations and Fig. 4.3 shows them as 2D visualizations with their normalized codomains as a color bar.
While it is impossible to compute this normalization for analytically unknown functions, for most tasks the minimum is known. For example, the optimal loss of a NN is zero. Further, the number of evaluations required to reach a good estimate for the integral is negligible when compared to the number of evaluations done during training.

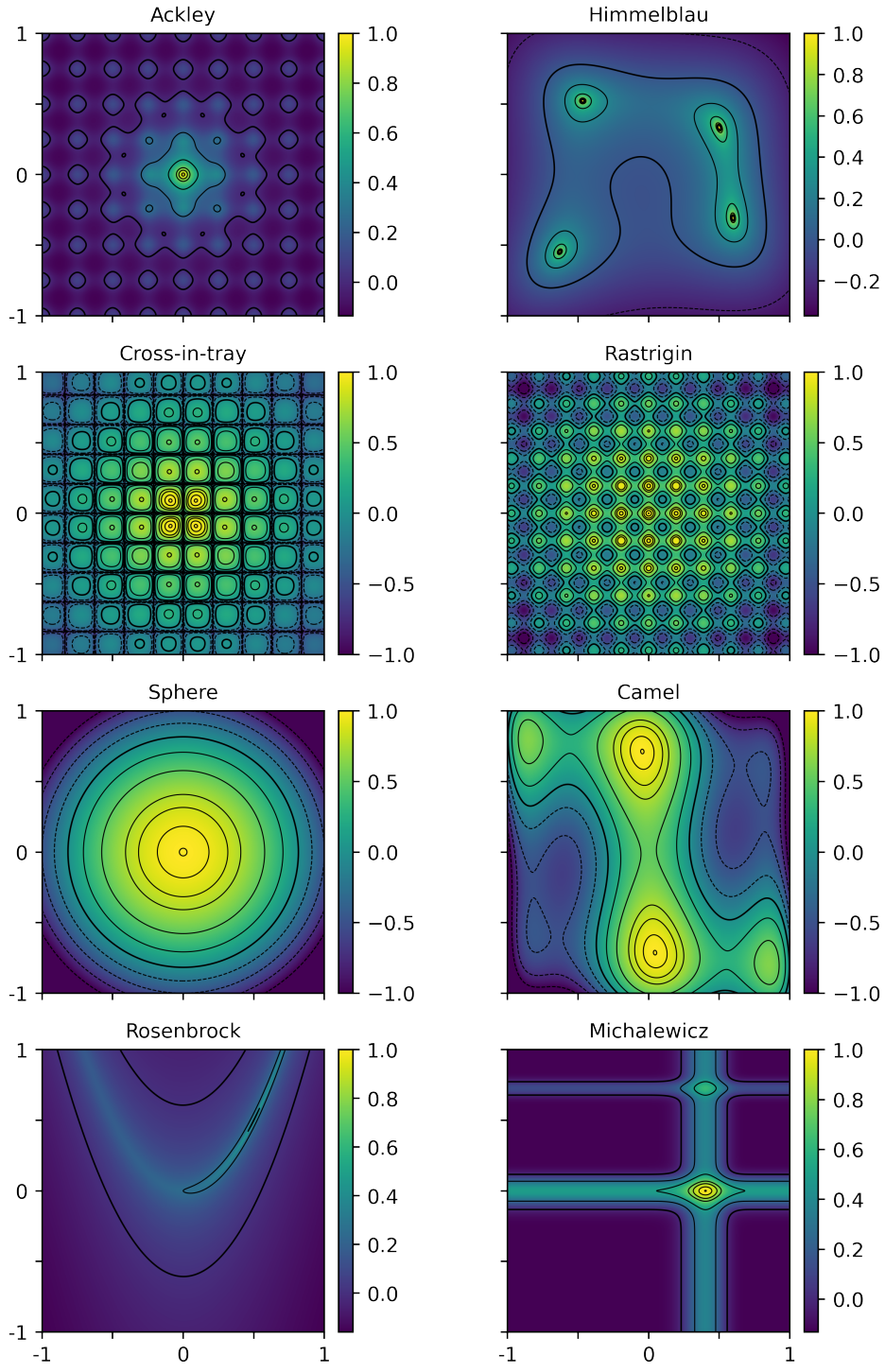

Figure 4.3: The eight objective functions after applying our normalization as 2D visualizations with contours at values -0.95, -0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75, 0.85, 0.95, 0.999 with the contour at 0 being drawn more boldly.

4.2.3 Operators

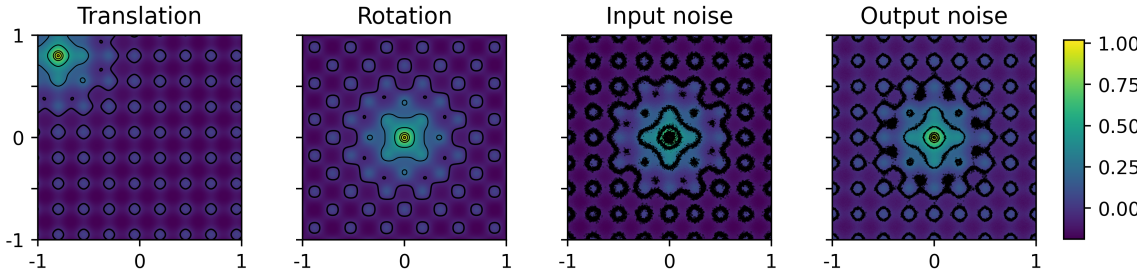

Figure 4.4: The four operators applied to the Ackley function as 2D visualizations

In addition to studying the optimization performance across different objective functions, we want to study how well the learned optimization algorithms perform after applying an operation to the objective functions. Hence, we chose four operations: translation, rotation, input noise, and output noise. Fig. 4.4 illustrates the effects of the different operators. While translation and input noise operators always change the location of the optimum, the output noise operator never does and the rotation operator changes it only if the optimum is not located at $(0, 0)$, which is the case for every function except Ackley, Rastrigin and Sphere. Further, we refer to a function with an operator as such: the translated Ackley function. Interesting to see is, that the input noise operator affects the resulting function in a more profound manner than the output noise operator. Finally, we define a fifth operator which we call the control operator. This operator does not change the objective function.

Translation

$$r_t = f_n(X_t + T), T \in \Theta, \tag{4.14}$$

with Θ the domain of the objective function. We add fixed values to the input of the objective function to translate the objective function and thus the location of the optima.

4.2.4 Rotation

$$r_t = f_n(\mathbf{R}X_t), \mathbf{R} \in \mathbb{R}^{2\times 2} \tag{4.15}$$

We multiply the input of the objective function with a rotation matrix

Input noise

$$r_t = f_n(X_t + \epsilon) \tag{4.16}$$

We add Gaussian noise to the input of the objective function. Compared to translation where the value is fixed, noise changes its value changes for every evaluation.

Output noise

$$r_t = f_n(X_t) + \epsilon \tag{4.17}$$

We add Gaussian noise to the output of the objective function.

Control

$$r_t = f_n(X_t) \tag{4.18}$$

4.2.5 Standard evaluation

In this section, we define a standard evaluation procedure to assign an overall performance to every policy. Thus helping us to conduct a large number of experiments and compare different training setups of objective functions, operators, and agent hyperparameters. The evaluation is designed to identify well-performing algorithms for further evaluation and sort out the ones which are not performing as well. Additionally, we can conduct large-scale comparisons for training configurations.
Such an evaluation shouldn't be computationally intense while still capturing performance across all objective functions and all operators. Since our normalization already achieves comparability of optimization performance across all our objective functions, we use the normalized objective value as the performance.
The evaluation we implemented starts with 16 different starting positions spread in an equal four by four grid across 95% of the domain. For the translation operator, we chose values in an equally spread five by five grid across 90% of the domain. The different grid size reduces the overlap between starting positions and translated optima. Further, we limited the translations to 90% of the domain, due to our optimization algorithms, including the baselines, having trouble finding optima directly on the boundary. For the rotation operator, we use ten different rotation matrices corresponding to a rotation evenly spread throughout $(0°, 180°)$. For the output noise operator we use ten different amounts of gaussian noise with a mean of zero and a standard deviation evenly spread between $(0, 0.1)$. For the input noise operator we also use ten different amounts of gaussian noise, but with the standard deviation evenly spread between $(0, 0.05)$, since the input noise operator affects the resulting objective function more than the output noise operator.

We define the function operator performance for a single operator and objective function as the average performance across all starting positions and values of the operator.
We define the function performance for a single objective function as the average across all function operator performances for that objective function. We define the operator performance as the average across all function operator performances for that operator. Finally, we define the overall performance as the average of each function performance.
Further, we refer to the function performance by the name of the function, e.g. the Ackley performance. We do the same for function operator performances, e.g. the Ackley-translation performance, and operator performances, e.g. translation performance.

With this definition, our performance scores always represent the advantage in optimization performance relative to the expected value of random sampling. Further, the optimal performance is always 1.
For the standard overall performance, we use the performance achieved after 50 iterations.

4.2.6 In-distribution & Out-of-distribution

We define in-distribution to refer to all information that is contained in the training environment, whereas out-of-distribution refers to everything which was not in the training environment.
So in our case in-distribution would refer to every objective function and operator combination that was used during training. Out-of-distribution refers to any combination not used during training. So for example, if we only trained with the Ackley objective function, without any operators in-distribution would only refer to the Ackley-control performance and out-of-distribution to every other possible combination like Ackley-translation.

4.2.7 Relative performance correlation matrix

In this section, we define a relative performance correlation matrix, which we use to assign a relative performance correlation score between two objective functions. We define the columns f to represent the objective functions a policy has been trained on and we define the rows g as the objective functions the policy has been evaluated on. Further, we define a policy trained on a function h as π_h. With the performance score of a single objective function i collected by a policy π_h as $p_i^{\pi_h}$, the correlation score $c_{fg}$is defined as in equation 4.19.

$$c_{fg} = \frac{\max_{\pi_f} p_g^{\pi_f}}{\max_{\pi_g} p_g^{\pi_g}} \tag{4.19}$$

With this definition, c_{fg} represents how well a policy π_g is capable of generalizing to function f. By normalizing by the best performance on g by a policy π_g, we normalize the value and make it easier to interpret. Therefore if the correlation score $c_{fg} = 1$, it does not matter, whether we train a policy on f or g as the best performance on g is equal. We say f generalizes to g. A score lower than 1 means f generalizes less to g. A score higher than 1, means it is better for optimizing g to learn a policy on f than on g. Further $\forall f : c_{ff} = 1$.
Computing the exact maximum possible performance a policy π_f can achieve on a function g is not possible. Therefore we estimate it with the experiments we conducted.

5 Implementation

To run many computationally intense machine learning experiments, one needs an appropriate tool for running them on high-performing hardware. Luckily, one does not need to start from scratch since high-performing libraries such as TensorFlow and PyTorch exist. These libraries have many pre-implemented algorithms and data structures one can adapt to their use case. This may require a significant amount of software engineering based on the task at hand. For our work, we implemented a testbench to easily swap out objective functions, operators, and agents while still maintaining excellent performance. In this section, we introduce the libraries we used and discuss the design choices made for the implementation of the testbench. Further, we will provide explanations and default values for the possible configurations.

5.1 TensorFlow & TFAgents

TensorFlow [1] is an open-source end-to-end machine learning library with a comprehensive and flexible ecosystem of tools, libraries, and community resources designed to enable researchers to push the state-of-the-art in machine learning and also enable developers to build and deploy machine learning applications. While implementing standard but complex machine learning models can be done with a few lines of code, TensorFlow also enables the user to extend existing models and functions, or implement their own, at every level of abstraction. Therefore, TensorFlow is striking a balance between ease of use and performance, suited to any machine learning use case. Its features include many machine learning models, loss functions, metrics, optimizers, and data structures. Further, its features include AutoDifferentiation and AutoGraphing, where the library keeps track of the order of computations and automatically translates it into a computational graph. This tracking works, with a few exceptions, on basic python code. This graph may then be reversed to compute the gradient for the relevant parameters at every step of the computation. While the construction of this graph is computationally expensive, it provides a significant performance boost for the actual computation. Machine learning tasks often require the same computation multiple times, like a forward and backward pass of a neural net or the trajectory sampling in a reinforcement learning task, and thus graph construction is often worth it. Another advantage of TensorFlow's graph computation is its distribute feature, where the user can spread computation to many devices through an API. TensorFlow's only competition is PyTorch [2], which shares many of the features of TensorFlow.
The TFAgents [3] library is a part of the TensorFlow ecosystem which implements modular reinforcement learning components on top of TensorFlow's data structures, such as REINFORCE, PPO, DQN,

[1] https://www.tensorflow.org/
[2] https://pytorch.org/
[3] https://www.tensorflow.org/agents

and SAC agents. It also provides interfaces for implementing custom agents. Moreover, it provides two interfaces for environments, PyEnvironments, and TfEnvironment. PyEnvironments are simple environments that are not inherently batched and are designed for prototyping. TFEnvironments are more complex but also more performant as they are inherently batched and capable of graph execution. TFAgents also provides many standard implementations for environments and an API to easily adapt Gym [4] environments to work in TensorFlow. It also provides standard implementations of replay buffers, like the TfUniformBuffer, which are designed to efficiently store and retrieve large numbers of trajectories. The main competitor to TFAgents is the Stable Baselines 3 [5] library which is a part of PyTorch. It provides many of the same features as TFAgents, but within PyTorch's ecosystem instead of TensorFlow's. We chose TensorFlow and TFAgents over Pytorch and Stable Baselines 3 due to our experience in working with TensorFlow.

5.1.1 Additional libraries

We use multiple libraries including the standard Python libraries, gin-config [6] for lightweight configuration and field injection, NumPy [7] for general data handling and analysis, matplotlib [8] and mayawi [9] for visualizations, scipy [10] for their implementation of zeroth-order optimization algorithms and finally pandas [11] and SQLAlchemy [12] for structuring data and storing/retrieving it with a PostgreSQL [13] database.

[4] https://www.gymlibrary.ml/
[5] https://stable-baselines3.readthedocs.io/
[6] https://github.com/google/gin-config
[7] https://numpy.org/
[8] https://matplotlib.org/
[9] https://github.com/enthought/mayavi
[10] https://scipy.org/
[11] https://pandas.pydata.org/
[12] https://www.sqlalchemy.org/
[13] https://www.postgresql.org/

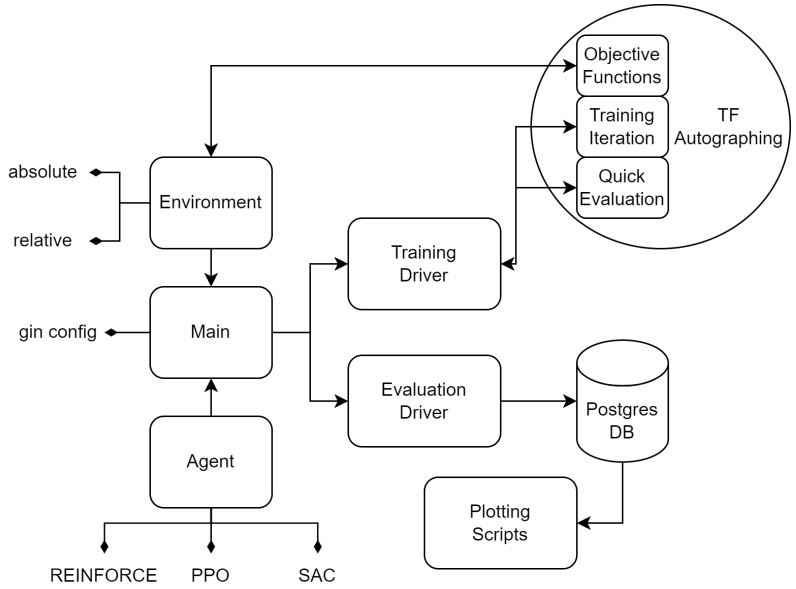

Figure 5.1: Overview of the structure of our learn-to-optimize testbench

5.2 Structure overview

In this section, we give an overview of the structure of our testbench implementation. The goal is to run large numbers of learn-to-optimize experiments. Therefore, we implemented our testbench with configurability in mind such that agents, environments, objective functions, and operators are easily added and removed from the training and evaluation tasks. Furthermore, we kept the structure modular such that extending the configurability is also easily achieved. Fig. 5.1 shows a diagram of the simplified structure. Our structure is heavily influenced by TFAgents' inherent structure of reinforcement learning. Hence, we followed the naming scheme of TFAgents like referring to the observations $\mathcal{O}$ in a MDP as observations instead of states, which is more common in research publications. In our work, state refers to the internal state of the environment like the location in the domain.
To aid our explanation, we will notate all Python objects in `CamelCase`.

There are two main functionalities to our testbench, training and evaluating models. For the training task, the main script handles the instantiation of the `Environment` and `Agent` according to the configuration file which is passed to the main script at runtime. Both objects are then passed to the `TrainingDriver` which handles the training iteration as described by the pseudocode in algorithm 5.1. During a training iteration, the `TrainingDriver` first collects a set of trajectories from the `Environment` by using the `Agent`'s collect policy. The collect policy uses the `Agent`'s policy but includes some randomness in the choice of actions to promote exploration of the environment. The trajectories are inserted into the `ReplayBuffer`. After sufficient trajectories are collected, all trajectories in the `ReplayBuffer` are passed to the `Agent` to perform a training step. If the chosen algorithm for the `Agent` is an on-policy variant, the `ReplayBuffer` is emptied after every training iteration as on-policy algorithms can only use trajectories for training that are generated using the current policy. For more detail we refer to Sec. 3.2. In addition to handling a training iteration, the

`TrainingDriver` provides a quick evaluation by using the `Agent`'s policy on the training `Environment`. This summarizes the progress of training by providing the current in-distribution performance.

Algorithm 5.1 Training iteration

```
f := Objective Function
environment := Environment(f)
O_0, r_0 := environment.reset()
buffer := ReplayBuffer()
for i in (1, ..., N_episode) do
    A_i := agent.collect_policy(O_{i-1}, R_{i-1})
    O_i, r_i := environment.step(A_i)
end for
buffer.add(O, r, A)
agent.train(buffer.gatherAll())
if agent.isOnPolicy then
    buffer.clear()
end if
```

During training, this training iteration is run multiple times in an outer loop. The outer loop is contained in the main script and calls the training iteration and quick evaluation within the `TrainingDriver`. Additionally, the outer loop logs the progress and saves checkpoints of the `Agent` and `ReplayBuffer`. Saving a checkpointing refers to persisting the `Agent`'s trainable parameters or the contents of the `ReplayBuffer` respectively. The checkpoints can be reconstructed at a later point in time to either continue training the `Agent` or perform an evaluation of its policy's performance.

For the evaluation task, the main script only instantiates a policy from a checkpoint of a previously trained `Agent` and passes the policy to the `EvaluationDriver`. The `EvaluationDriver` then performs a standard evaluation on the policy as described in Sec. 4.2.5. We insert all performance metrics of this standard evaluation into an SQL table. Additionally, the `EvaluationDriver` creates visualizations of the trajectories for each objective function and operator combination. Finally, it inserts the average performance over time for each objective function and operator combination into a different SQL table. Since there is an unreasonable number of trajectories we decided not to save them.
Our testbench also provides configurable plotting scripts which visualize the data present in the SQL tables.

5.3 Implementation details

In this section, we explain the exact implementations for the action and observation spaces. Additionally, we discuss the design choices of particular classes in more detail.

5.3.1 Action and observation spaces

We implemented identical action spaces $\mathcal{A}$ for absolute and relative environments. They are implemented continuously within the (-1, 1) square:

$$\mathcal{A} \in \mathbb{R}^2, \text{with} -1 \leq a_1, a_2 \leq 1 \tag{5.1}$$

Thus, the agent can choose to evaluate any location inside the domain of our normalized objective functions at every step in absolute environments. For comparison, in a relative environment the action is added to the previous location and thus the agent can not cover the entire domain with a single step. However, at most two steps are required. Further, the agent can step outside of the boundaries of the domain. When the updated location falls outside of the domain, we decided to clip the location such that the agent always remains inside the domain.

We implemented the observations $\mathcal{O}$ differently for absolute and relative environments. For absolute environments they are defined as such:

$$\begin{aligned} &\mathcal{O} \in \mathbb{R}^{n_{obs} \times 3}, \text{ with} \\ &\mathcal{O}_t = \left\{X_{t-1}, X_{t-2}, \dots, X_{t-n_{obs}}\right\} \cap \left\{r_{t-1}, \dots, r_{t-n_{obs}}\right\}, \end{aligned} \tag{5.2}$$

with X_t being the location in the domain at step t, r_t the reward at step t and n_{obs} being a hyperparameter. With every X_i or r_i with $i < 0$ being defined as $(0, 0)$ and 0 respectively. So the agent sees its past locations in the domain and the reward associated with this location at all times .

For relative environments we define the observations as such:

$$\begin{aligned} &O \in \mathbb{R}^{n_{obs} \times 3}, \text{ with} \\ &O_t = \left\{X_{t-1} - X_{t-2}, \dots, X_{t-n_{obs}} - X_{t-n_{obs}-1}\right\} \cap \left\{r_{t-1} - r_{t-2}, \dots, r_{t-n_{obs}} - r_{t-n_{obs}-1}\right\}, \end{aligned} \tag{5.3}$$

Instead of providing the exact location, the agent only sees the change in the location and reward at step t compared to the location and reward at step $t - 1$. With every X_i or r_i with $i \leq 0$ being defined as $(0, 0)$ and 0 respectively. Hence, the initial location X_0 is hidden from the agent as well. With our implementation, the first part of the observations is equal to the actions the agent has taken in previous iterations. The only exception is if an action is clipped during the location update within a relative environment. Additionally, since the reward is equal to the output of our normalized objective function it is important information for an optimization algorithm and should thus be included in the input of the policy. Our implementation of $\mathcal{O}$ for the relative environment is almost identical to a previous work [LM16]. In comparison, their approach used $n_{obs} = 25$ exclusively. Other learn-to-optimize approaches are also similar. [BRL21]

5.3.2 Agents

For the agents, we used TFAgent's standard implementation of REINFORCE and PPO agents. For either agent, we tried two variants, one which replaces the policy with a MLP only and another which also includes a LSTM. Either network has two outputs, where one output predicts the mean

and the other the standard deviation of the probability distribution over the action space.
The LSTM architecture features two MLPs and an LSTM, the MLPs surround the LSTM between the input and output. We used the same shapes for either MLP.
For the PPO agent, we defined the number of training sub-iterations, as described in Sec. 3.2.1, to be 10.

5.3.3 Environment

We use the `TFEnvironment` interface of TFAgents to implement our environment as an abstract class. We left the construction of observations and update of location as abstract functions to be implemented by a child class. Therefore, different definitions of observation spaces and location updates can quickly be implemented by extending this class. We implemented two variants of this abstract class with the construction of observations and functions as defined by our descriptions of the relative and absolute environment. Furthermore, since we are using an off-the-shelf agent and NN implementation, the execution speed of the environment is the largest factor in decreasing the runtime of the training and evaluation task. Therefore, we implemented the environments to be inherently batched such that we can collect as many trajectories as required in a single parallel run. The operators and starting positions can be shuffled at any point during training allowing for a learning curriculum. Further, multiple objective functions during training are possible by splitting the internal batch of the environment into a minibatch for every objective function. A learning curriculum involving different objective functions is possible by using the checkpointing feature of our `TrainingDriver`. Additionally, this implementation allows the standard evaluation to be run entirely in parallel and consequentially boosts its performance considerably.

5.3.4 Drivers

In this context, a driver refers to the trajectory collection loop including the environment and agent. While TFAgents already provides implementations of drivers, capable of driving batched `TFEnvironmets` with graph execution, their implementations are designed to handle different episode lengths for every parallel environment. As we are using a fixed episode length as our stopping condition, we can implement our own, simpler driver which is easier to debug, without losing performance.

5.4 Configurability

Aside from operator and objective function combinations, this section gives an overview of all possible hyperparameters of this testbench. Additionally, we provide default values.
n_{obs} = 1:
Number of observations as defined in section 5.3
batch size = 512:
Number of parallel environments as described in section 5.3
n_{eps} = 50:
Number of steps taken in a single episode
n_{iter} = $1e^4$:

Number of training iterations as described by algorithm 5.1
$\gamma = 0.9$:
Discount factor as defined in section 3.2
$c_e = 0.01$:
The entropy regularization coefficient c_e weights the entropy regularization term in the loss function of the agents. Higher values incentivize more exploratory behavior of the agent.
$c_v = 1.0$:
Weights the influence of the value net on the loss function.
$\lambda = 1e - 3$:
The learning rate λ defines the step size used during the optimization.
learning schedule, decay rate = 0.9 and decay steps = 1000:
We define the learning schedule as the exponential decay of the learning rate according to the formula:

$$\tilde{\lambda} = \lambda \cdot \text{decay rate}^{\frac{\text{completed iterations}}{\text{decay steps}}} \tag{5.4}$$

$PN_{layers} = (100, 50)$:
This parameter defines the number of hidden layers and hidden nodes the policy networks contain. If we use a LSTM architecture, both MLPs use this definition.
$VN_{layers} = (100,)$:
This parameter defines the number of hidden nodes the value network contains. A parameter of $\emptyset$ means we use no value network.
$RNN_{layers} = (40,)$:
We only use this parameter if the agent uses a LSTM. It defines the breadth and depth of the memory cells. With the default value, we use only a single memory cell of breadth 40.
$trans = 0.0$:
This parameter defines a square $(-trans, trans)$ in the middle of the domain from which we sample a translation vectorduring training. Hence, with a factor of 1.0 the square spans the entire domain.
$rot = 0.0$:
This parameter is multiplied by 2π and defines a partial circle from which we sample rotations during training. So with a factor of 1.0, any amount of rotation is possible.
in_noise = 0.0 and out_noise = 0.0:
This parameter defines the standard deviation of the gaussian noise applied to the input or output during training as a factor of the domain.

5.5 Limitations

While TFAgents' on-policy agents like REINFORCE and PPO work for our environment implementations out of the box, the off-policy agents DQN and SAC did not. DQN isn't implemented for continuous action or observation spaces and is therefore unusable for our formulation. The SAC algorithm isn't compatible with our environments, as, per TFAgents' documentation, it relies on the provided `Actor` and `Learner` APIs which only work for a non-batched `PyEnvironment`. We tried implementing a workaround by writing our own replay buffer and drivers but due to the limited time, insufficient documentation of the internal workings of the SAC implementation, and overall complexity of the problem we were not able to implement a working SAC agent for our environment.

As discussed in section 5.3, we chose to clip the updated locations for a relative environment, such that the agent always remains inside the boundary. There exists a mechanism called action masking where the environment can mask a part of the action space such that the agent is not able to choose it. While this mechanism would be suited for our implementation, the TFAgents library did not implement this feature for their continuous distribution network implementations. Since our experiments show that clipping the actions also leads to the desired behavior we did not implement this feature.

6 Experiments

In this chapter, we are going to discuss the experiments we conducted using our testbench implementation and analyze the learned policies in more detail. We conducted 390 experiments.
Of these experiments, we conducted 304 with the REINFORCE agent. Of those, 257 used only a MLP, 47 also included a LSTM.
86 experiments were conducted using the PPO agent, of which 77 used an MLP only and 9 used a LSTM as well.

6.1 Baseline zeroth-order algorithms

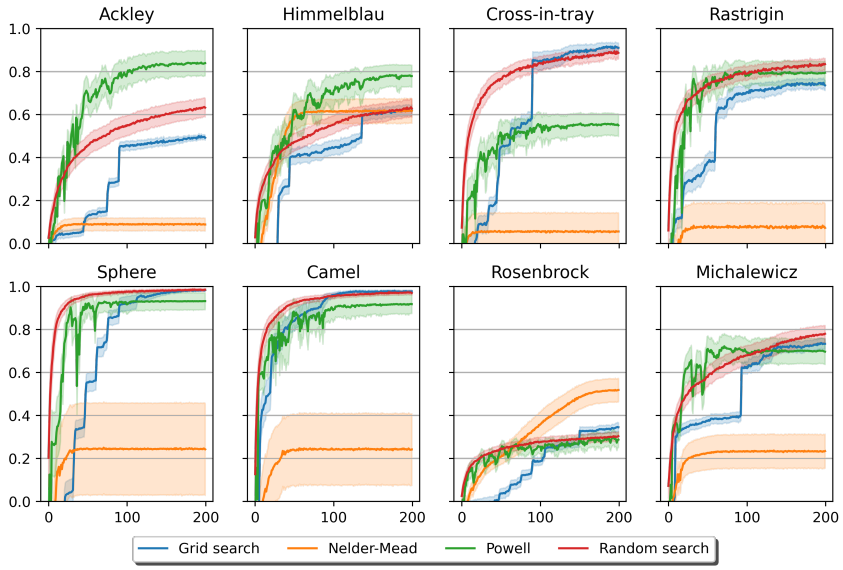

Figure 6.1: The function performance of our baseline methods over 200 iterations

First, we discuss the performance of the baseline methods we introduced in Sec. 3.3. Fig. 6.1 shows their performance over 200 iterations broken down by objective function. For comparability, we define an iteration as an evaluation of the objective function. Further, the line represents the average performance and the shaded area the standard deviation. Due to a large parameter space with good objective values, grid search and random search perform best on the Cross-in-tray, Rastrigin, Sphere, and Camel functions. On Ackley and Himmelblau, where the optima are located on steep hills, Powell's method performs best. Nelder-Mead struggles with every function except for the Rosenbrock function, since it tends to get stuck in local minima.

6.2 Top performances with a single objective function

In this section, we analyze the performances with only a single objective function. For this, the policies are evaluated exclusively on the objective function they were trained on. We chose the policy with the highest function performance for each objective function. We analyze these policies with respect to longer horizons and operator performances. All policies were trained with the REINFORCE agent without a LSTM. Other hyperparameters which differ from the defaults are listed in columns 1-8 of Tab. 6.1.

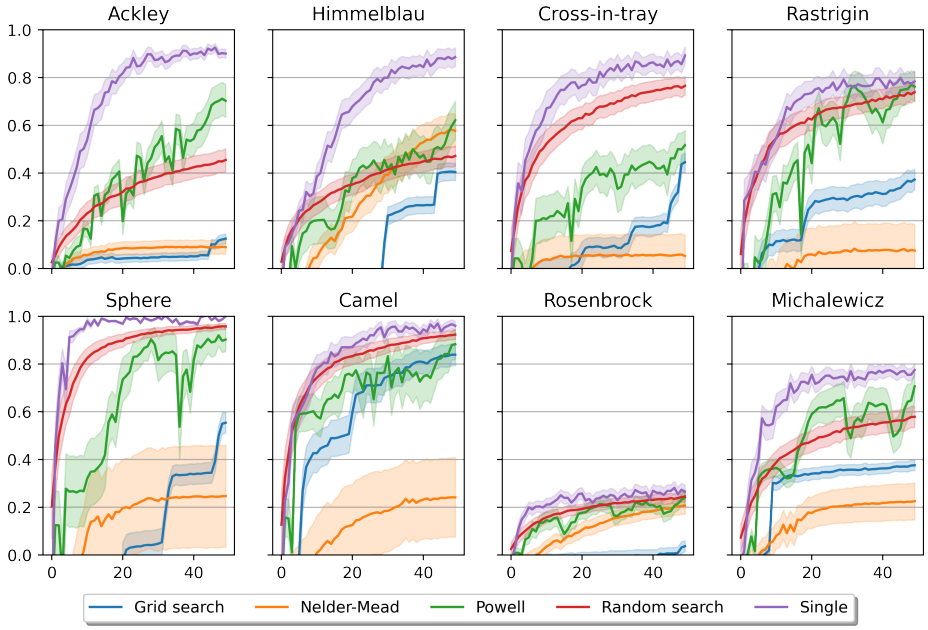

Figure 6.2: The function performance over 50 iterations of the top performing policy on each objective function

Fig. 6.2 shows the performance over 50 iterations for each policy and our baseline methods. Each policy outperforms any of our baseline methods in convergence speed as well as final objective value. This observation is in line with the findings of many other learn-to-optimize approaches. [ADG+16][LM16][CHC+16][RXR+19] Since this advantage is especially large for the Himmelblau and Ackley functions we will analyze the mechanisms of the corresponding policies in Sec. 6.4. In general, the policy can exploit the characteristics of an objective function and thus has an advantage with respect to optimization performance over human-designed algorithms.

6.2.1 Longer horizons

As shown by Andrychowicz et al. [ADG+16; WMH+17] learned optimization algorithms are capable of generalization to longer horizons. We were able to partially confirm these findings. Fig. 6.3 shows some of our top-performing policies can generalize to longer horizons as well. This includes the policies trained on Himmelblau and Sphere. For all other policies, the objective

value decreased with longer horizons. While the policies trained on Ackley or Himmelblau kept their advantage, all other policies were outperformed on final objective value by at least one of our baseline methods.

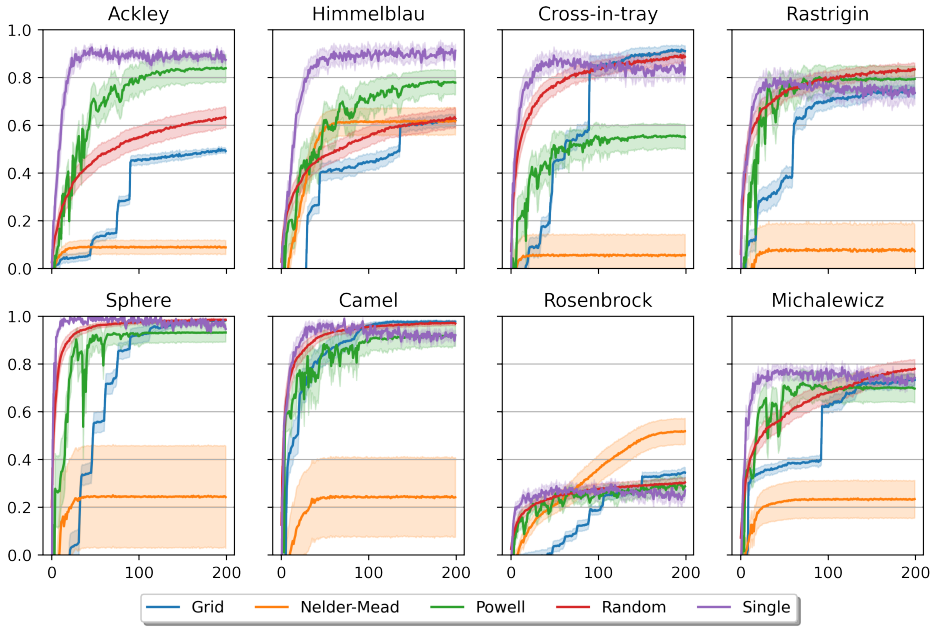

Figure 6.3: The function performance over 200 iterations of the top performing policy on each objective function

	1	2	3	4	5	6	7	8	9	10	11
Fct	Ack	Him	Cro	Ras	Sph	Cam	Ros	Mic	All	All	Cro
Env	Abs	Abs	Abs	Abs	Abs	Abs	Abs	Abs	Abs	Rel	Rel
N_{obs}	10	10	25	10	10	10	10	10	10	25	25
γ	0.9	0.9	0.9	0.5	0.5	0.9	0.9	1.0	0.9		1.0
N_{eps}		100		100		100			100		
trans	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0		1.0
N_{iter}	$1e^5$	$5e^4$	$1e^5$	$5e^4$	$1e^5$	$1e^5$	$1e^5$	$1e^5$	$1e^5$	$1e^5$	$1e^5$
PN_{layers}										deep	
VN_{layers}										deep	

Table 6.1: A table showing the hyperparameters used for training the top performing policies, 'Fct' refers to the objective functions used in training identifiable by their first three letters or 'All' if all were used. 'Env' refers to the type, absolute or relative, of environment used to train the policy. For PN_{layers} 'deep' refers to (100,100,50,50). For VN_{layers} 'deep' refers to (100,50)

6.2.2 Operator performances

Fig. 6.4 shows the performances of the policies over 50 iterations broken down by operators. The control performance is the highest for every function. An exception is the Rastrigin-rotation performance, but the difference is negligible. The Output noise performance sometimes jumps above the control performance but remains lower on average. First, we see the Sphere and Camel performances are invariant to any operator, due to their simplicity and gentle gradients. The Cross-in-tray function is also largely invariant to any operator. For the Ackley, Himmelblau, and Rastrigin function we see a significant drop in input noise performance since their optima are located in steep valleys, such that a minor change in location results in a significant change of objective value.
The Rosenbrock and Michalewicz functions show a notable decrease in translation and rotation performances. Fig. 6.5 shows example trajectories for these functions with the rotation operator. The trajectories show clear overfitting of the policies. For the Michalewicz function, the policy walks in a predetermined straight line and gets stuck in the first local optimum. For the Rosenbrock function, the policy overfits the U shape of the canyon. The policy tries to traverse the canyon in the positive x- and y-direction as shown in the control trajectory. However, due to the rotation operator, the canyon now extends to the negative x- and y-direction, and thus the policy cannot follow the canyon and acts unpredictably.

Tab. 6.2 shows the average operator performance after 50 iterations over every policy's operator performance relative to their control performance. Accordingly, each value measures the average invariance to an operator we can expect a policy to have when evaluated on the same function it has been trained on. As the data shows, our policies are on average more invariant to the translation and noise operators than Nelder-Mead and Powell. Nevertheless, Nelder-Mead is more invariant to the rotation operator. Random search and Grid search are largely invariant to any operator since they act independently of the objective function. The drop in input noise performance is to be expected since the Ackley, Himmelblau, Rastrigin, and Rosenbrock functions are sensitive to it. The drop in translation performance for Grid search is a result of the choice in grid size.

	Translation	Rotation	Input noise	Output noise
Top performer	0.8	0.83	0.83	1
Powell	0.73	0.8	0.75	0.83
Nelder-mead	0.34	0.95	0.12	0.17
Random search	0.95	1.02	0.91	0.98
Grid search	0.8	1.07	0.9	1

Table 6.2: A table showing average operator performance of the top performing policies relative to the control performance

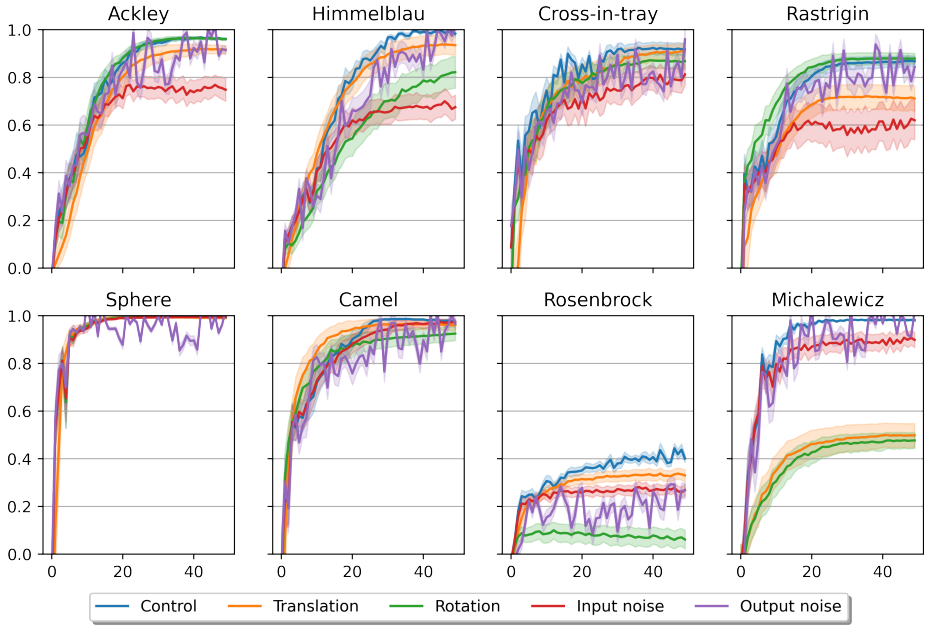

Figure 6.4: The function operator performance over 50 iterations of the top performing policy on each objective function.

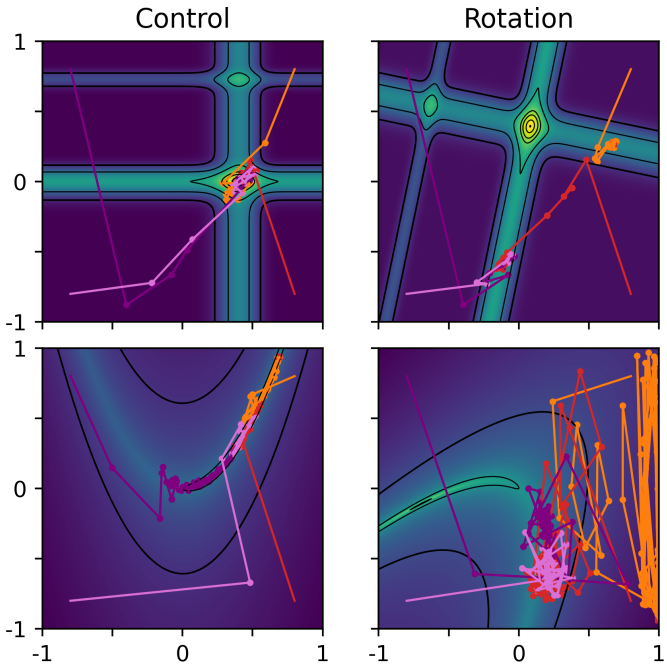

Figure 6.5: Example trajectories of the top performing policy for the Michalewicz and Rosenbrock function with the control and rotation operator

6.3 Top overall performances

In this section, we discuss the performances on all objective functions for four policies that achieved the highest overall performance scores in their categories. The 4 categories are: trained on a single objective function and trained on all objective functions for absolute and relative environments respectively. The hyperparameters which differ from the defaults are listed in Tab. 6.1 as columns 2, 9, 10 and 11. We refer to the policies by the type of environment and functions they have trained on, e.g. the absolute Himmelblau policy or relative All policy.

While the absolute All policy (column 9) achieved the highest overall performance after 50 iterations out of all our experiments, the relative All policy achieved the second highest score. Further, the absolute Himmelblau policy (column 9) almost matched their performance. They scored 0.711, 0.700 and 0.69 respectively. Further, all three outperform any of our baseline algorithms, where the best is Powell's method which achieved an overall performance score of 0.666. The relative Cross-in-tray policy matched Powell's performance score exactly. Therefore, it is possible to learn an optimization algorithm for our selection of objective functions which is capable of equal or better optimization performance than our baselines. Moreover, this is possible by training only on a single objective function. The mechanisms these policies have learned are discussed in Sec. 6.4.

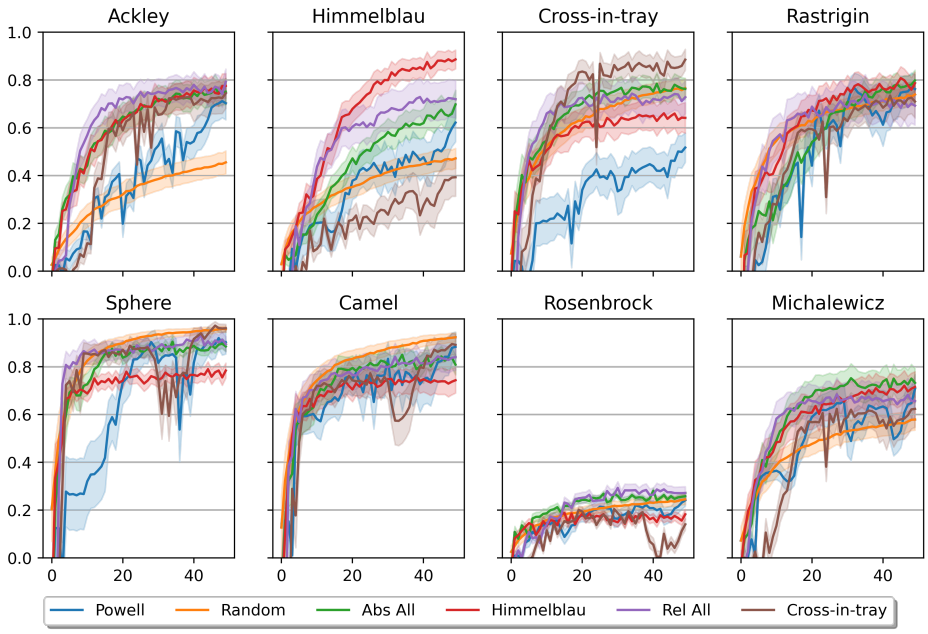

Figure 6.6: The function performance over 50 iterations of the four top overall performing policies.

First, we will look at a comparison of their function performances. Fig. 6.6 shows the function performance over 50 iterations of our top overall performing policies, as well as Powell's method and Random search. We excluded Grid search and Nelder-Mead to improve readability. Additionally, they achieved much lower performances.
Unsurprisingly, the absolute Himmelblau and relative Cross-in-tray policies achieved a significant performance advantage on their respective objective functions. Interestingly, compared to the other

policies, the absolute Himmelblau policy has the largest performance disadvantage on the Cross-in-tray function and vice versa. Compared to the other policies, the Himmelblau policy performs slightly worse on the Camel and Sphere function, whereas the Cross-in-tray policy performs slightly better. On these four functions, the All policies' performances fall in the middle of the other two policies. With these exceptions, the performances of the policies are largely comparable for every other function. Still, all policies are outperformed on the Sphere and Camel function by random search.

6.3.1 Longer Horizons

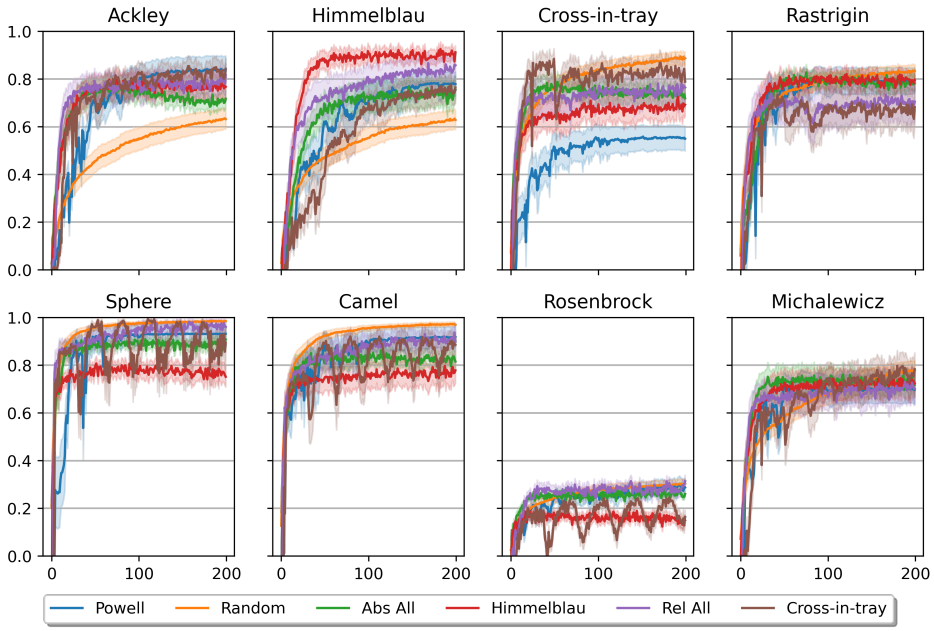

Figure 6.7: The function performance over 200 iterations of the four top overall performing policies.

Fig. 6.7 shows the function performance over 200 iterations of our four top overall performing policies, as well as Powell's method and Random search. Both absolute policies manage to generalize to longer horizons on most functions, the absolute Himmelblau policy does not generalize to longer horizons on the Ackley and Rosenbrock function. The absolute All policy does not generalize to longer horizons on the Ackley and Cross-in-tray functions.
The relative All policy does not generalize to the Rastrigin function but achieves significant performance improvements during the additional iterations on the Himmelblau and Camel functions. The relative Cross-in-tray policy shows a recurring pattern of sudden decreases in performance followed by a gradual increase to higher performance than before the sudden decrease. This behavior is especially pronounced on the Sphere, Camel, and Rosenbrock functions and further causes the policy to outperform any other policy on Ackley and Michalewicz in final objective value after 200 iterations. Interestingly, this behavior does not generalize to Cross-in-tray and Rastrigin. With Cross-in-tray as the training function and Rastrigin as the function closest to it in characteristics, this shows overfitting to n_{eps}.

6.3.2 Operator performance

Fig. 6.8 shows the function operator performance for every top overall performing policy over 50 iterations and Tab. 6.3 shows their operator performance relative to the control performance. The absolute All policy is almost completely invariant to every operator and compares to the invariance showed by the random search and grid search baselines. But, this invariance is due to the advantage of the Himmelblau-translation performance and the disadvantage of the Ackley- and Michalewicz-translation performances averaging out.
The absolute Himmelblau policy is also almost invariant to every operator, except the translation operator. Although, the Himmelblau-, Rastrigin- and Sphere-translation performance is comparable to the control performance.
The relative All policy shows a significantly lower performance for the translation performance as well as the input noise performance. This disadvantage in performance exists across all objective functions. Further, the policy shows a lower performance for the Himmelblau-, Cross-in-tray, Camel- and Michalewicz-rotation performance. These are exactly the functions where rotation changes the location of the optimum. Since this policy did not use the translation operator during training, this shows overfitting to the location of the optima. Furthermore, since this policy is a relative environment, we conclude the policy has to exploit the clipping of actions at the boundaries to infer its location in the domain.
The relative Cross-in-tray policy shows an invariance to the rotation and output noise operators across every objective function. Further, only the Ackley- and Michalewicz-translation performances are significantly lower than the control performance. These functions share the 'flat areas' property.

	Translation	Rotation	Input noise	Output noise
Absolute All	0.95	0.98	0.93	0.93
Absolute Himmelblau	0.79	1.01	0.95	1.05
Relative All	0.58	0.91	0.81	0.93
Relative Cross-in-tray	0.88	0.99	0.86	1.01
Powell	0.73	0.8	0.75	0.83
Nelder-mead	0.34	0.95	0.12	0.17
Random search	0.95	1.02	0.91	0.98
Grid search	0.8	1.07	0.9	1

Table 6.3: The operator performance of the four overall top performing policies relative to their control performance

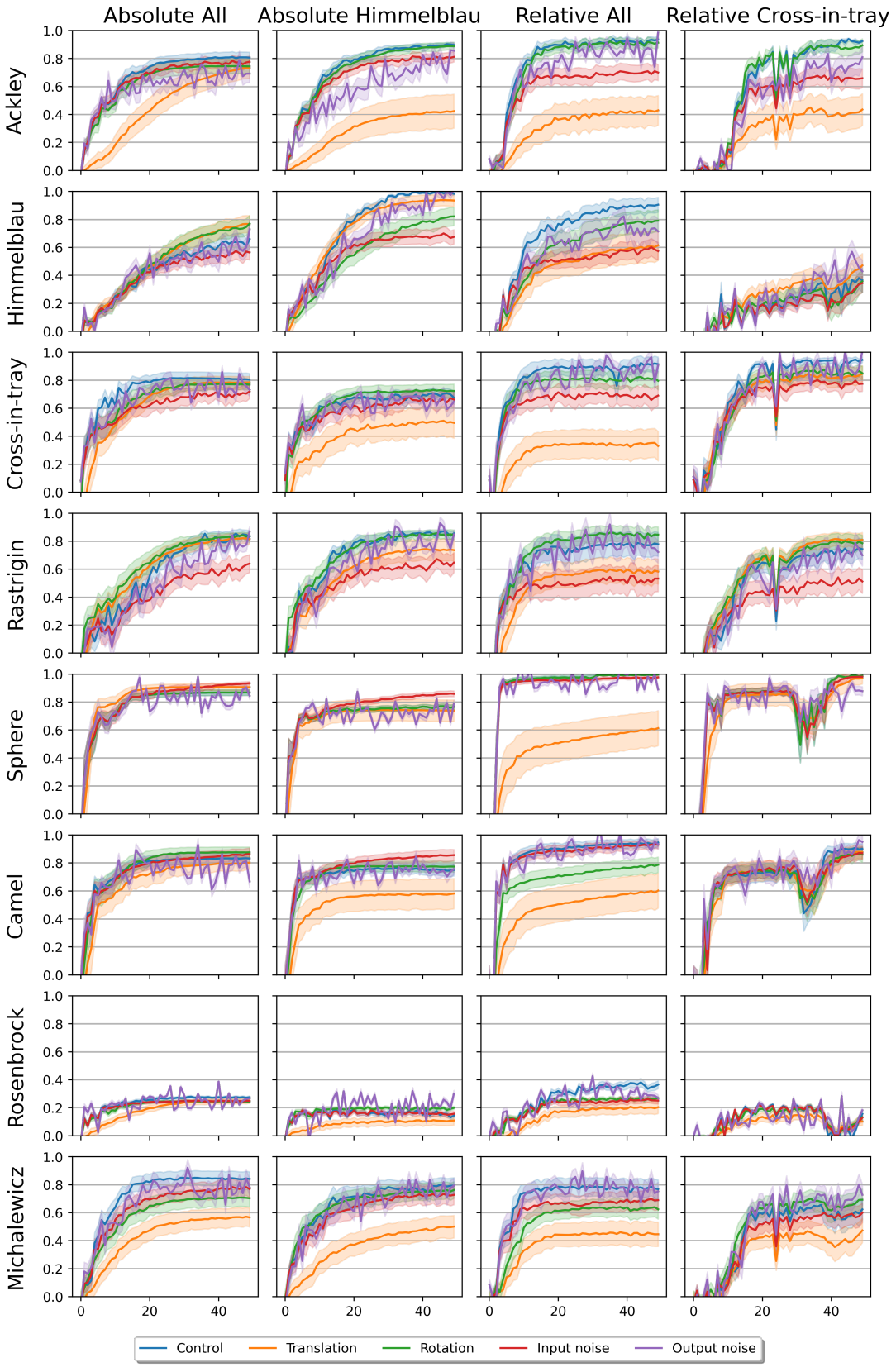

Figure 6.8: The function operator performance over 50 iterations for every top-performing policy

6.4 Learned mechanisms

In this section, we analyze the mechanisms and inner workings of our top overall performing policies by examining the trajectories these policies take. Further, we use these insights to explain the observations of different performances we made in Sec. 6.3. Hence, we use the same four policies we call the absolute All policy, absolute Himmelblau policy, relative All policy, and relative Cross-in-tray policy. Their hyperparameters are listed in Tab. 6.1 as columns 2, 9, 10 and 11 respectively.

6.4.1 Absolute Himmelblau policy

In this section, we discuss the mechanisms of absolute Himmelblau policy. Fig. 6.10 shows the trajectories the policy takes on each of our objective functions with no operators applied. For each objective function four trajectories are shown with initial locations set to $(0.8, 0.8)$, $(0.8, -0.8)$, $(-0.8, 0.8)$, $(-0.8, -0.8)$. The policy shows a two-stage optimization strategy. The first stage is a search with a circular pattern starting from the middle of the domain. With this circular search, the policy estimates the direction of the steepest ascent with a bias to the positive x-direction. Fig. 6.9 shows a trajectory the policy takes on a plane that demonstrates this behavior. The second part of the optimization strategy starts once the policy crosses a reward threshold of roughly 0.5. This threshold is equal to the contour line second from the bold contour line representing a reward of 0. Further, for the Himmelblau function, this threshold is only surpassed at the foot of the steep hills where the optima are located. Since Himmelblau has four of these hills spread throughout its domain, has no local optima, and its gradient smoothly increases up to the global minima, the circular search is efficient at finding the foot of these hills. For all functions other than Rosenbrock and Himmelblau, the circular search is cut short, since the threshold is crossed early in the optimization process and thus not entirely visible. However, we can observe the beginning of it in almost any trajectory. In the second stage of the optimization strategy, the policy estimates the direction of the steepest ascent with small steps that get smaller with increasing rewards. This behavior is learned to ascent the steep hills where the optima of the Himmelblau function are located. With increasing rewards, the step size needs to get smaller so as to not overshoot the top of the hill and miss the optimum. This behavior is clearly shown in the trajectories on the Sphere function.

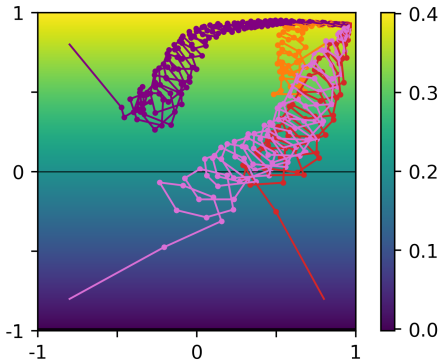

Figure 6.9: A trajectory the absolute Himmelblau policy takes on a plane with a gentle slope over 50 iterations.

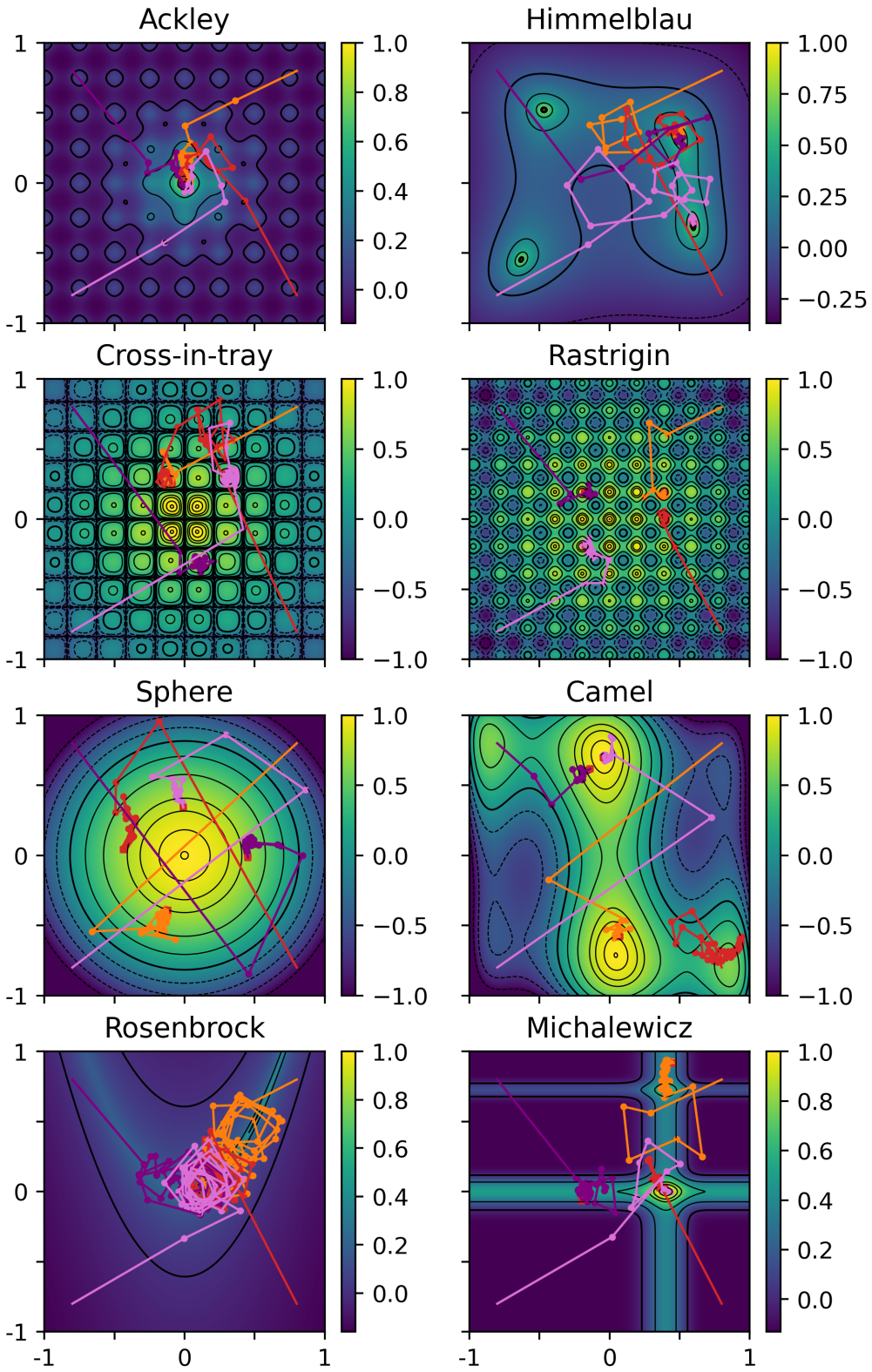

Figure 6.10: Example trajectories for every objective function collected with the absolute Himmelblau policy over 50 iterations.

Transfer

With this insight into the absolute Himmelblau policy, we analyze the performances we discussed in Sec. 6.3 and visualized in Fig. 6.8. First, the positive x-direction bias results in the decrease of the Himmelblau-rotation performance. Furthermore, this strategy generalizes well to optimizing Ackley, since its optima are also located at the top of a steep hill. Moreover, Ackley's only optimum is located in the middle of the domain and thus the search pattern quickly finds it. In comparison to Himmelblau, Ackley is mostly flat with many local optima with a single optimum. Therefore, if a translation operator is applied, the circular search often fails to locate the hill in 50 iterations as shown in Fig. 6.11. The same concept explains the lower Michalewicz-translation performance.
Furthermore, this policy has no mechanism to avoid local minima. This explains the low Cross-in-tray, Rastrigin, Camel, and Michalewicz performances. We can observe this in almost any trajectory taken on these functions. Moreover, the lower translation performance is explained by this, since with the translation operator applied, the local optima towards the middle of the domain has a lower objective value.
For the Sphere and Camel functions, the gentle slope of their hill prevents the agent from ascending to the top. Since the policy takes smaller steps with increasing rewards it never reaches the top of the hill.
For the Rosenbrock function, only the bottom of its canyon, which takes up a small parameter space, has a normalized objective value higher than the threshold. Therefore, the policy remains in the circular search pattern and fails to optimize the Rosenbrock function. This is apparent in every trajectory. For the Michalewicz function the optimum is close to the middle of the domain, the policy does often find the foot of the hill the optimum is located on and can then ascend it, but the policy has a tendency to get stuck in a canyon, as shown by the purple trajectory, which ascends the side of a wall leading to suboptimal performance.
The top performing policy for the Ackley function (Tab. 6.1 column 1) exhibits similar behavior, but with the initial circular search pattern starting with a wide circle. This is due to the Ackley function showing only a single hill and no gentle slope to exploit. Due to our normalization the Ackley function the lowest reward has a value of -0.134. Therefore, if we apply our policy to a different function that includes a much lower reward like the Sphere function, the circular search's initial circle is chosen too wide. Therefore, this policy does not generalize to optimizing many of our chosen objective functions leading to a low overall performance score.

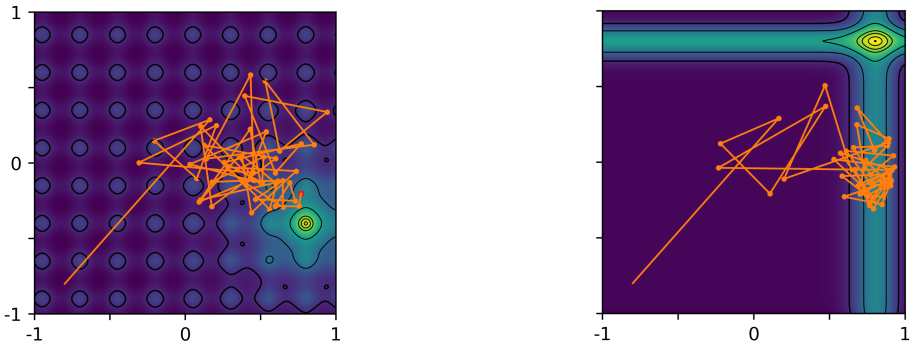

Figure 6.11: A trajectory of the absolute Himmelblau policy on the translated Ackley and Michalewicz function over 200 iterations.

6.4.2 Relative Cross-in-tray policy

In this section, we discuss the mechanisms of the relative Cross-in-tray policy (Tab.6.1 column 11). Fig. 6.13 shows the trajectories the policy takes on each of our objective functions with no operators applied. For each objective function, one trajectory is shown, with the initial location set to $(0.8, 0.8)$.
We were able to identify two main mechanisms this policy uses. The first mechanism is a circular search which estimates a parabolic shape. But instead of searching for a threshold to be crossed, it slowly decreases the size of the circle to converge in a spiraling fashion towards the estimated middle of the parabolic shape. This spiral is apparent in every trajectory. Further, Fig. 6.12 shows a trajectory the policy takes on a translated a Sphere function whose slope has been decreased by a factor of 0.75. This shows the pattern converging in a spiraling fashion to the optimum.
The policy learned a second mechanism to avoid getting stuck in local minima since the Cross-in-tray function has many. This mechanism is best visualized in the trajectories the policy takes on the Ackley and Rosenbrock function. We identify the initial spiral pattern, which converges to a local optimum. For the Ackley function, this is in the positive y-direction of the global optimum and for the Rosenbrock function, the local optimum is located in the canyon towards the middle of the domain. We observe this convergence by an accumulation of evaluations in close proximity. From this convergence, the policy resets its spiral pattern to start over with a wider circle to eventually converge again close to the previous location of convergence. This resetting of the spiral is also shown in the trajectories the policy takes on the Himmelblau, Sphere, and Camel function.

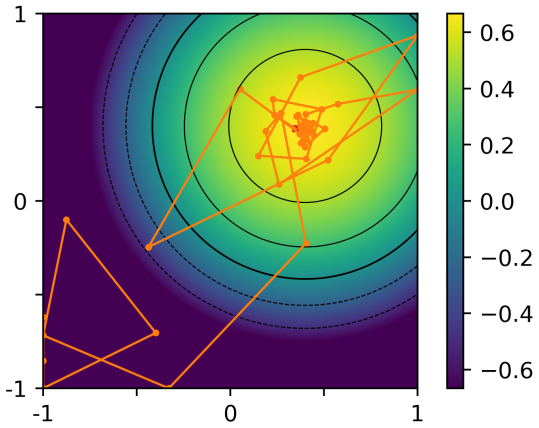

Figure 6.12: A trajectory the relative Cross-in-tray policy takes on the translated sphere function scaled by a factor of 0.75 over 50 iterations.

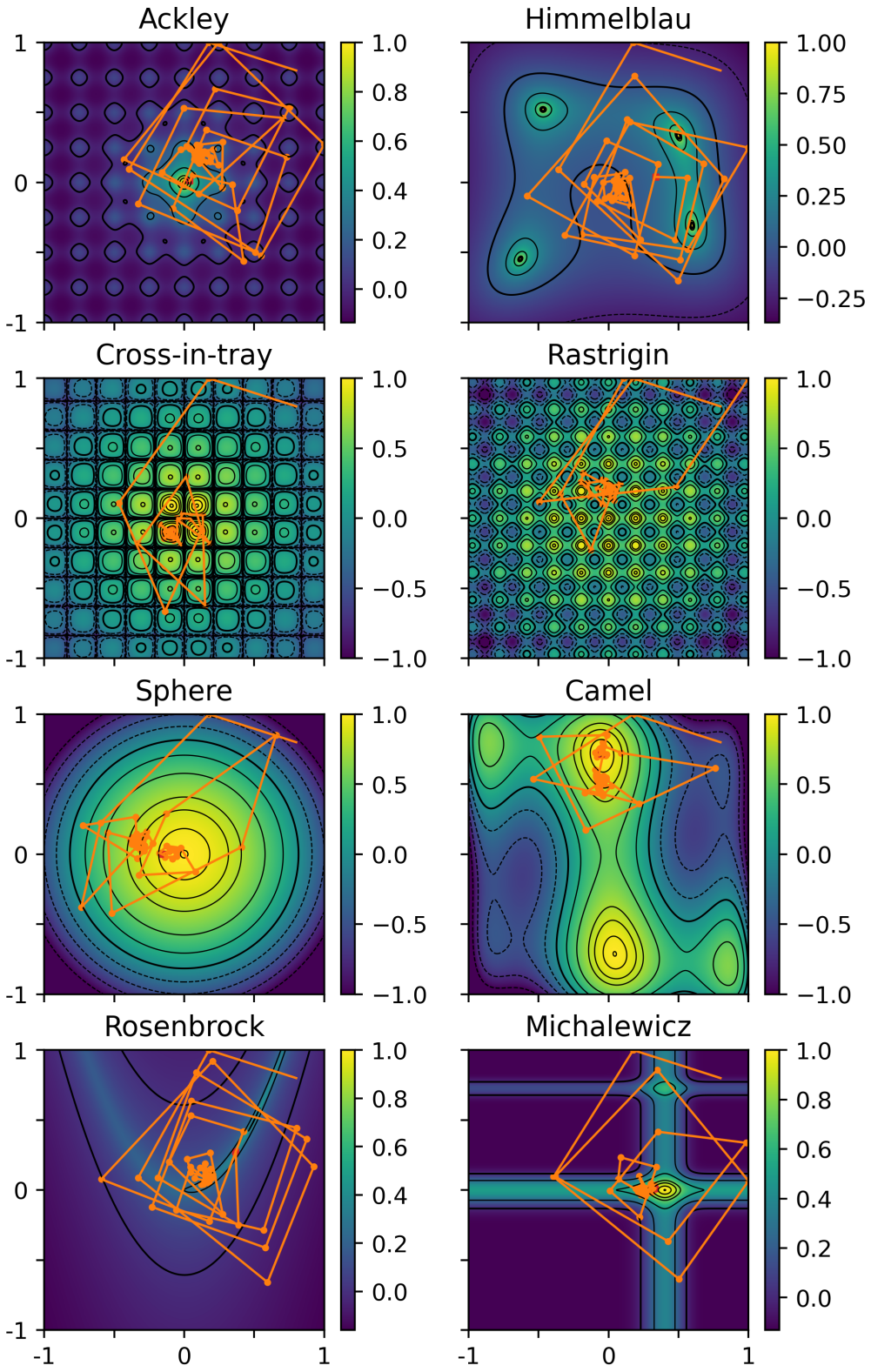

Figure 6.13: Example trajectories for every objective function collected with the relative Cross-in-tray policy over 50 iterations.

Transfer

With this insight, we analyze the performances we discussed in sec 6.3 and visualized in Fig. 6.8. First, the sudden dips in performance we observed in Fig. 6.7 are caused by the resetting mechanism the policy learned. Since the policy shows an improvement in performance after these dips on almost all objective functions, this mechanism generalizes to optimizing multiple objective functions. Further, we see the circular search estimating a parabola is overfitted to the slope of the parabolic shape the optima of the Cross-in-tray function form. This parabolic shape is gentler than our normalized Sphere function. As the trajectory the policy took on the Sphere function shows, the policy does not find the optimum with its first convergence. Only after resetting the optimum is reached. Moreover, Fig. 6.12 shows, a gentler slope on a sphere function leads to faster convergence to the optimum.
For the Ackley and Michalewicz functions, the policy shows a drop in translation performance. This is due to the mostly flat area these functions show. If no translation is applied, this flat area is spread evenly around the optima and therefore the function can often find the optimum with a single reset as the trajectory taken on Ackley shows. But if the optimum is translated to the edge of the domain, this flat area now covers most of the domain. The policy now requires too many iterations to cross this flat area. Fig. 6.14 shows trajectories taken on the translated Ackley and Michalewicz functions over 200 iterations. They demonstrate the behavior of periodic resetting to cross the flat area.
The low performance over 50 iterations on Himmelblau and Rosenbrock is explained by the spiral initially converging towards the middle of the domain. For the Himmelblau function, this is due to the optima being roughly equidistant to the middle of the domain and thus the circular search estimates an optimum in the middle of them and converges to this location. For the Rosenbrock function, we see the circular search evaluations do not land on the canyon but only on the gentle slope around it which leads to the middle of the domain and thus the spiral converges to the middle of the domain. As Fig. 6.7 and Fig. 6.14 on the right shows, the policy is able to better optimize Himmelblau and Rosenbrock when given more steps. With each reset, the spiral converges closer to the optima.
Further, this spiraling search pattern is exhibited by many of our trained policies. Especially by policies, which are trained on relative environments.

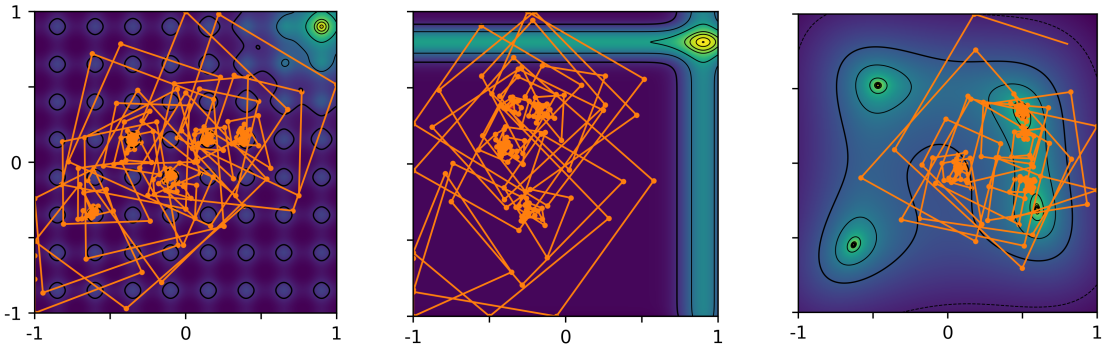

Figure 6.14: A trajectory of the relative Cross-in-tray policy on the translated Ackley (left) and Michalewicz (middle) function and the Himmelblau (right) function over 200 iterations.

6.4.3 All policies

In this section, we discuss the mechanisms of both policies trained on all objective functions. Although they use different environments, the exhibited behaviors are similar. The trajectories these policies take are shown in Fig. 6.16 for the relative All policy and Fig. 6.17 for the absolute All policy. For either policy, we were not able to identify a clear optimization strategy. The policies perform a few steps with a mostly fixed pattern followed by a jump to a location close to the optima. They then perform a random search in near proximity. This behavior exemplifies overfitting to the objective functions and not learning a general optimization algorithm. The policy remembers the patterns the objective functions show during the first steps and exploits them to infer the approximate location of the optima. This overfitting explains the excellent performance of the policies on each objective function. Further, the relative All policy did not use the translation operator during training and thus also overfitted to the location of optima without translation. This explains its drop in translation performance.
If we evaluate the policies on a plane with a gentle slope as Fig. 6.15 shows, we can gain some insights into their behavior. For the relative All policy we observe a behavior similar to the absolute Himmelblau policy which starts with a circular search estimating the direction of the steepest ascent that turns into a series of increasingly smaller steps. For the absolute All policy, we observe a series of small steps initially in the direction of the steepest ascent which then takes a turn to continue orthogonal to the direction of the steepest ascent.

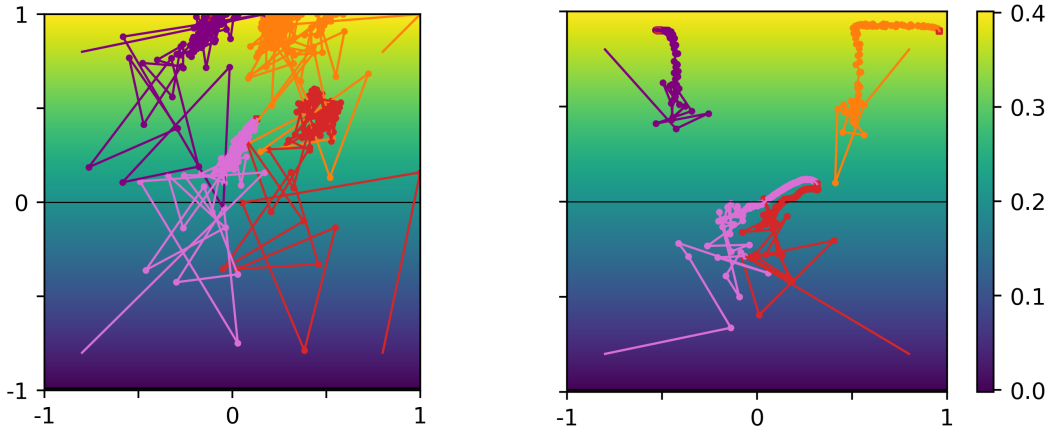

Figure 6.15: Example trajectories the relative All policy (left) and absolute All policy (right) take on a plane with a gentle slope over 200 iterations.

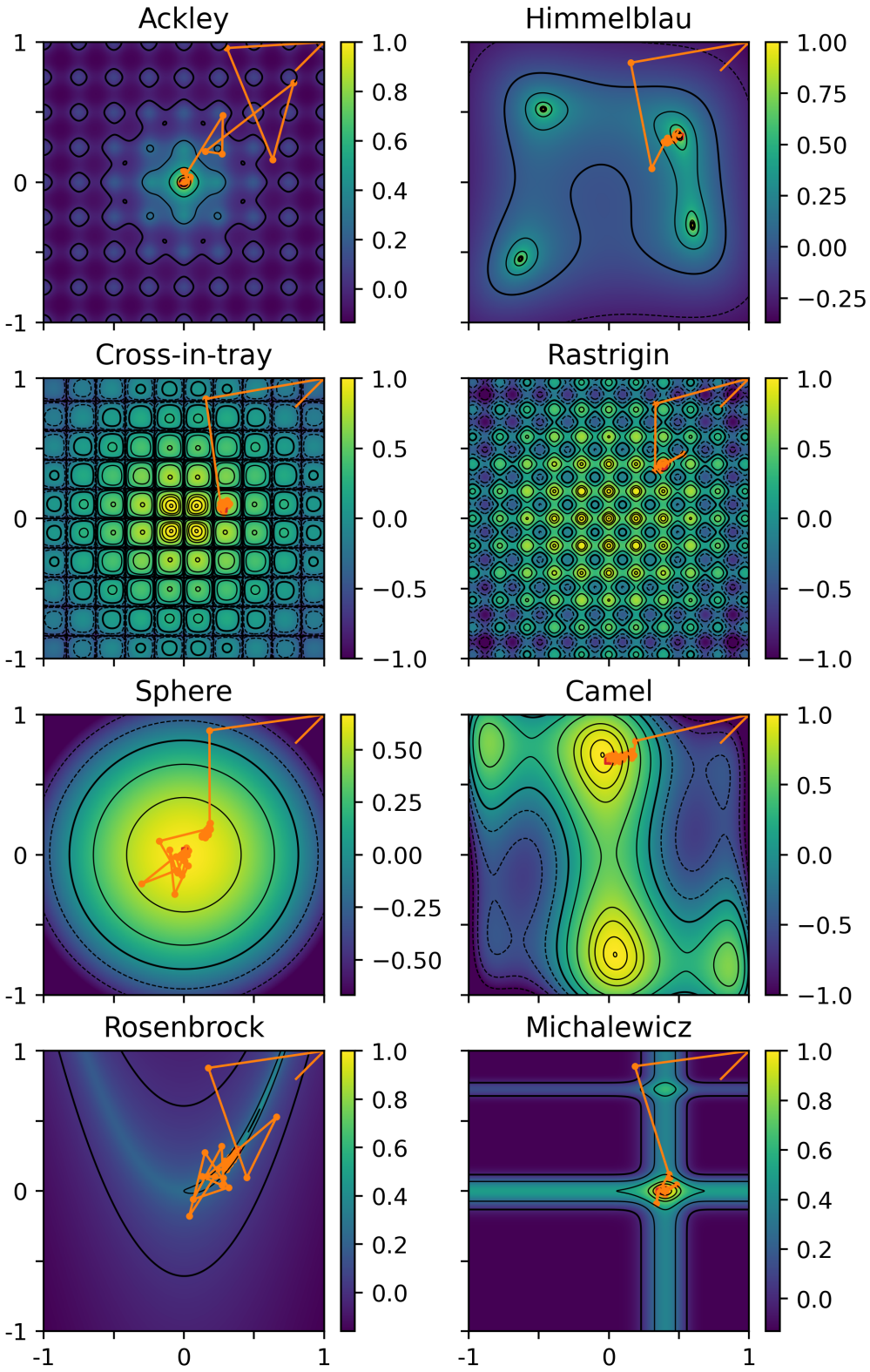

Figure 6.16: Example trajectories for every objective function collected with the relative All policy.

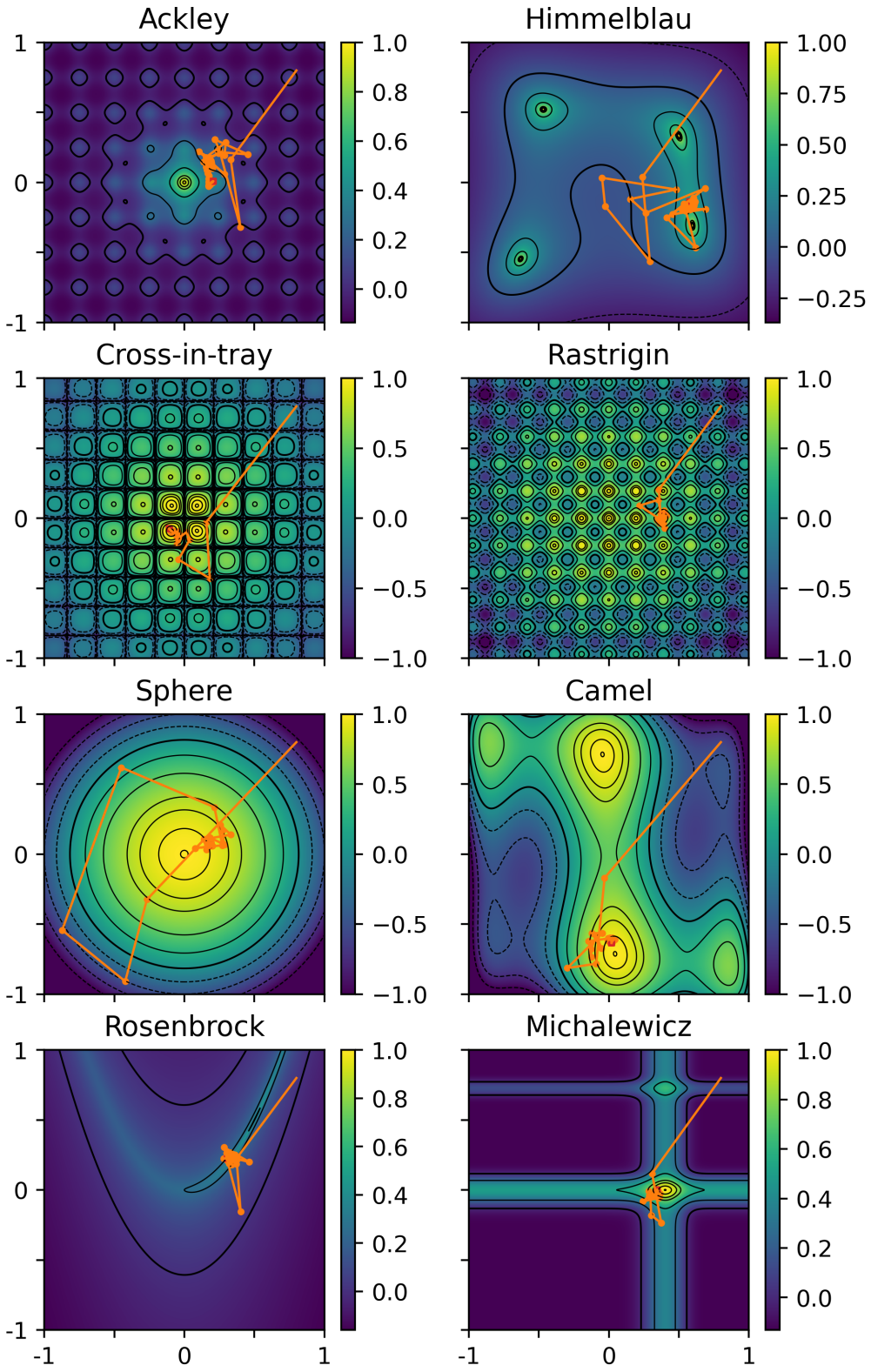

Figure 6.17: Example trajectories for every objective function collected with the absolute All policy.

6.5 Finetuning performance

One of the biggest problems learn-to-optimize is facing is the immense upfront computational cost to learn a policy capable of optimizing an objective function. Finetuning refers to the process of continued training of a model which has already learned to solve a particular task. A pretrained model is trained for a few additional steps to adapt it to a related task. A similar concept could be used for learn-to-optimize, where a policy is pretrained to perform optimization on some objective functions and is then finetuned to a specific objective function related to the initial training functions. This reduces the amount of upfront training required to learn a well-performing optimizer.

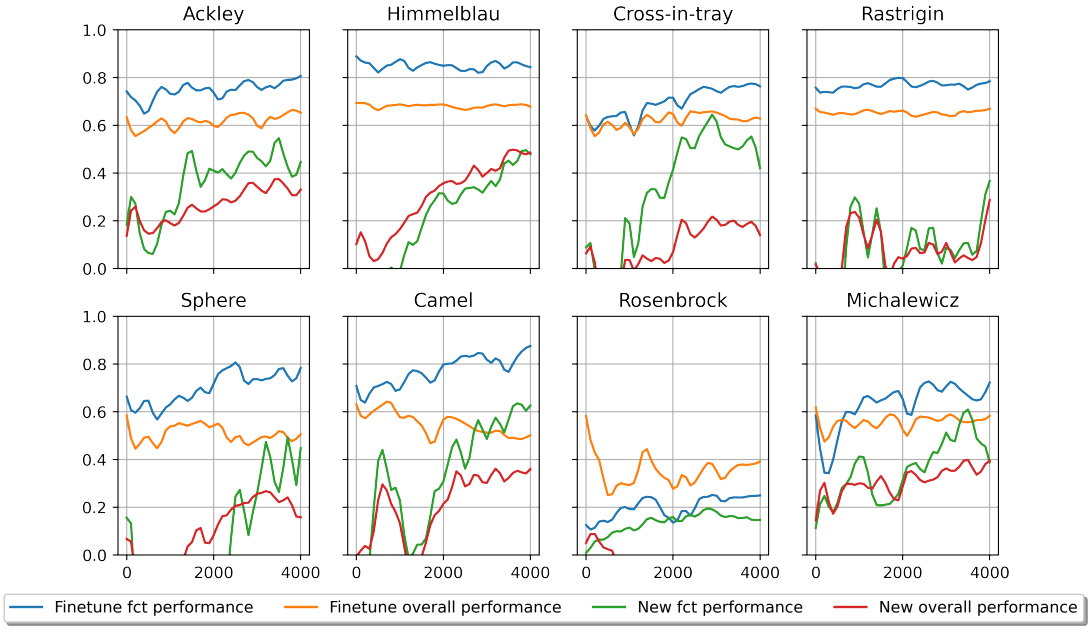

Figure 6.18: Change in overall and training function performance over $4e^3$ training iterations of finetuning our absolute Himmelblau policy compared to training a new policy from scratch

In this section, we discuss the performance changes we observed while finetuning our absolute Himmelblau policy (Tab. 6.1 column 2) to each of our objective functions. We used the same agent we used to train our absolute Himmelblau policy for $5e^4$ training iterations. Further, we lowered the learning rate from $1e^{-3}$ to $1e^{-4}$ and disabled the learning schedule. Finally, we trained the policy with an additional $4e^3$ training iterations on each objective function. In addition to the policy we finetuned, we also trained a policy with the REINFORCE agent without LSTM and the default hyperparameters from scratch. Fig. 6.18 shows the change in overall and function performance over the additional training iterations for both policies. This experiment shows, that we were able to improve performance on every new objective function we finetuned our policy on. While in the case of the Camel and Cross-in-tray functions the policy we trained from scratch was able to match the initial function performance of the pretrained policy, none of the policies we trained from scratch were able to achieve a better function performance after $4e^3$ training steps. Moreover, we see large

differences in the performance changes. For example, the Camel performance increased by over 0.1, whereas the performance for Rastrigin did not change significantly. We conclude, that not all functions are suited to finetuning from the same policy. Furthermore, a policy trained from scratch is prone to quick changes in performance, good or bad, until it has been trained for at least $1e^4$ steps. The policy we train from scratch on the Cross-in-tray function demonstrates such a sudden decrease in performance after $3e^3$ iterations. Additionally, while the optimization performances for the training functions improved by finetuning, the overall optimization performance decreased during finetuning. This is to be expected since we are training a policy capable of general optimization to perform better on a specific function. Therefore, overfitting to the new training function is expected and also desired. Finally, the continued training on the Himmelblau function did not change the performance. In conclusion, finetuning promises to be a useful tool to reduce upfront computational costs for learn-to-optimize tasks, but a more in-depth analysis of the classes of objective functions suited to finetuning is required.

6.6 Generalization between objective functions

In order to find sets in our objective functions for which a learned optimization algorithm generalizes well, we propose to analyze the ability of a learned optimizer to generalize from one function to another based on the function's properties. In this section, we compute a relative performance correlation matrix as defined in Eq 4.19. We use the results of our experiments to estimate the relative performance correlation scores and discuss which function generalizes to which and compare it to the qualitative properties of our objective functions as described in Tab. 4.1. Since our experiments include highly overfitted policies to their specific objective function or policies that did not learn to optimize at all, we chose to exclude all experiments with an in-distribution performance lower than 0.2 or a translation performance lower than 0.1. This reduces the number of experiments to 225.

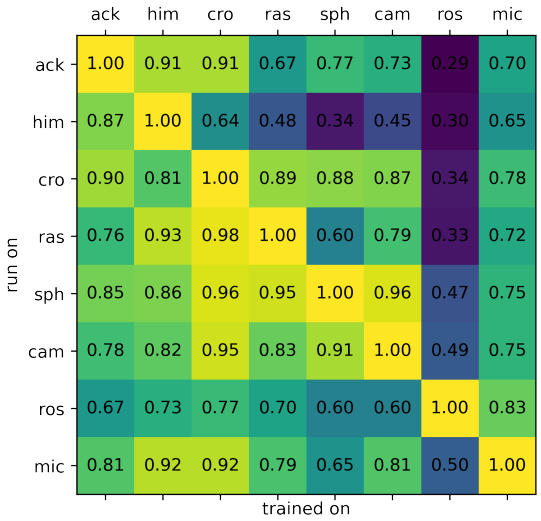

Figure 6.19: The estimated relative performance correlation matrix between our objective functions

Fig. 6.19 shows the estimated relative performance correlation matrix. All values along the diagonal are 1.0 which follows by definition. Further, Rosenbrock does not generalize well to any other function of our lineup. In contrast, Cross-in-tray generalizes well to every other function, except Himmelblau. Also, almost no function generalizes well to Himmelblau, except for Ackley. This is interesting since Himmelblau generalizes well to every other function.
For further investigation we used the properties described in Tab. 4.1. For each property, we computed the average relative performance correlation score between all functions that share a certain property. For example for the property 'Minima in parabolic shape', we compute the average of the relative performance correlation scores of Cross-in-tray to Rastrigin and Rastrigin to Cross-in-tray. Repeated for every property we get a score that predicts the generalization capabilities between functions that share this property. We call this the property generalization score. We exclude the convex property since Sphere is the only function with this property. The resulting scores are listed in Tab. 6.4. Based on these scores we can deduce, objective functions which have local minima that form a parabolic shape form a set where training on one function should lead to policies that generalize well to any other function in the class. The same is true to a lesser extent for functions showing steep hills or walls, or multiple optima.

	score
Single optimum	0.67
Multiple optima	0.76
Polynomial	0.63
Many local minima	0.73
Minima in parabolic shape	0.93
Steep hills/walls	0.85
Steep valleys	0.68
Canyons	0.67
(Mostly) flat areas	0.75

Table 6.4: A table showing the property generalization scores as computed with our experiments

Discussion

The properties we chose do not represent an exhaustive set of properties our functions have. Further, we assigned the properties in a binary fashion, where a function either has it or not. Therefore, we fail to capture the entirety of the functions by reducing them to these properties. Moreover, the number of objective functions we used for our analysis is small, especially for statistical analysis. For example, the score for "Minima forming a parabolic shape"consists of only two objective functions that generalize well to one another. Is this only because of this specific property or did we fail to capture the true reason?
Similarly, the experiments we used to estimate the relative performance correlation matrix do not represent an exhaustive search of all possible policies and may therefore not be accurate.

6.7 Training guidelines

In this section, we give a training guideline by providing insights into various performance improvements we have gained from running a large number of experiments.

6.7.1 Observations

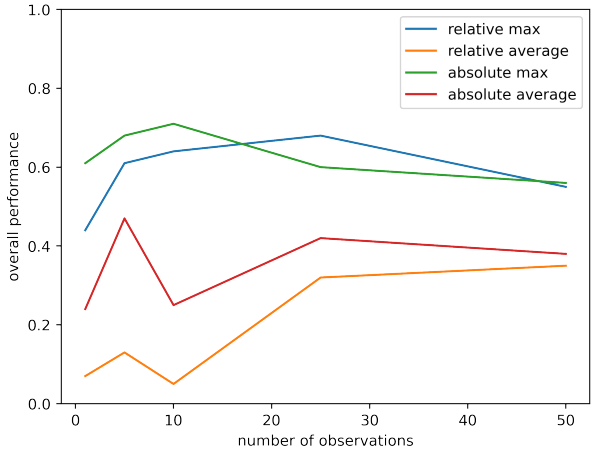

Figure 6.20: The maximum and average overall performance by n_{obs} for relative and absolute environments

Fig 6.20 shows the average and maximum overall performance of all experiments with n_{obs} equal to 1, 5, 10, 25, and 50, split by relative and absolute environments. It shows, that for absolute environments the best overall performance was achieved with $n_{obs} = 10$. The drop in average performance at $n_{obs} = 10$ is due to our initial hyperparameter search being conducted with this hyperparameter. The spike in average performance at $n_{obs} = 5$ can be explained by the small number (10) of experiments we conducted.
For relative environments the maximum overall performance was achieved with $n_{obs} = 25$. Further, the average performance increases with n_{obs}.

Discussion

We set the trailing observations to 0 for iterations smaller than n_{obs}. Therefore, for a trajectory with $n_{eps} = 50$ and $n_{obs} = 50$, every observation except the last has at least one trailing zero. In contrast, if $n_{obs} = 10$ only the first 9 observations show trailing zeros. Therefore, with $n_{obs} = 10$, the trajectory contains 41 observations with the same structure, whereas, with $n_{obs} = 50$, the trajectory never contains an observation with the same structure twice. Thus, overfitting to n_{eps} should be expected. Our experiments confirm, no policy with $n_{obs} = n_{eps}$ is generalizes to more than n_{eps} iterations. For $n_{obs} < n_{eps}$, our experiments show generalization to longer horizons is possible.

6.7.2 Agent choice

Tab. 6.5 shows the average, maximum, and minimum overall performance achieved by different agent configurations, broken down by relative and absolute environments. The data shows, that for absolute environments the REINFORCE without LSTM agents achieved the highest maximum overall performance by a wide margin. Additionally, only the PPO with LSTM agent achieved a higher average score. However, this is due to the small sample size of this agent configuration.

	Env	Net	Avg	Max	Min	N
REINFORCE	Abs	MLP	0.3	0.71	-0.45	159
		LSTM	0.25	0.63	-0.17	37
PPO		MLP	0.14	0.59	-0.35	48
		LSTM	0.364	0.534	0.12	9
REINFORCE	Rel	MLP	0.35	0.7	-0.25	98
		LSTM	-0.1	0.43	-0.45	10
PPO		MLP	-0.16	0.22	-0.45	29
		LSTM				0

Table 6.5: A table showing the average, maximum and minimum overall performance scores achieved by different agent choices

6.7.3 Value Network & environment type

In this section, we discuss the function performance changes by including a value network in the parameterization of our policy. We also discuss the function performance changes for absolute and relative environments.

Tab. 6.6 shows the maximum scores for overall performance and function performance with or

	Absolute	Absolute	Relative	Relative
VN_{layers}	Any	$\emptyset$	Any	$\emptyset$
Overall	0.711	0.67	0.7	0.455
Ackley	0.9	0.71	0.8	0.63
Himmelblau	0.89	0.67	0.89	0.72
Cross-in-tray	0.89	0.74	0.88	0.52
Rastrigin	0.83	0.78	0.85	0.32
Sphere	1	0.99	0.97	0.84
Camel	0.96	0.91	0.95	0.65
Rosenbrock	0.27	0.25	0.3	0.13
Michalewicz	0.78	0.67	0.74	0.4

Table 6.6: A table showing the maximum performance scores with or without a value network broken down by objective functions and environment

without a value network and relative or absolute environments. This data shows, for either type of environment or objective function, the inclusion of a value network did improve performance. The increase in performance is small for the Camel function and minimal for the Sphere function. This is likely since these functions are easy to solve by any optimization algorithm and thus the policy does not need the additional information a value network provides. Furthermore, the data shows the increase in performance to be much larger for relative environments. This suggests learning relative environments is more reliant on the information about the quality of its state. Finally, the data shows no significant difference in performance achieved by either relative or absolute environments. The only exception is the Ackley function, where absolute environments achieved a higher performance score.

6.8 Runtime improvements

Since runtime is a large factor when conducting computationally intense machine learning experiments, we tried different approaches to reach our current implementation. In this section, we discuss some of the choices we made and which changes to runtime they provided.

6.8.1 Agent choice

In this section, we are going to look at the runtime differences between REINFORCE and PPO agents. All experiments were conducted on a system running Linux and an Intel Haswell 72-Core processor.

While REINFORCE models normally converge after $1e^4$-$1e^5$ training iterations, PPO models consistently require $1e^6$ steps to reach convergence, although in this number the training sub-iterations as described in Sec. 3.2.1 of the PPO algorithm are already included. Therefore PPO only requires the sampling of new trajectories every 10 iterations. An average iteration of PPO is therefore much faster.
With our given hardware and an agent with the standard policy net and value net size, $5e^4$ iterations of REINFORCE are required to reach convergence on average and they require roughly 3.5 hours. In comparison, $1e^6$ iterations of PPO are required to reach convergence and require on average 11 hours.

If we include a LSTM into the policy of the agent with the default size, the required iterations for REINFORCE to converge increase to $6e^4$, in addition, a single iteration takes longer as the LSTM needs to be trained as well. Therefore training time on average increases to 8 hours. For PPO including a LSTM causes the iterations to increase as well, but we still only trained for $1e^6$, since the training time exploded to 29 hours. The increased cost of training is due to the increased cost of a LSTM affecting every sub-iteration of a PPO agent.

6.8.2 Implementation

In this section, we discuss various runtime improvements achieved by our implementation. All experiments were performed on a Windows 10 system containing a Ryzen 7 3700x 8 core processor with 32 GB of RAM and an NVIDIA GeForce 2070.

As discussed in section 5.3, since we use off-the-shelf agents, the largest factor in decreasing runtime is the implementation of the environment. We compared the runtime of our inherently batched `TFEnvironemnt` to the runtime of a trivial implementation of a non-batched `PyEnvironment`. For a batch size of 20 with a simple REINFORCE agent and a single objective function without operators, we observed a runtime decrease of a factor of 45. For larger batch sizes this performance boost increased considerably. With n as the batch size and $20 < n < 2048$, we observed empirically the runtime of the `PyEnvironment` to be in $\mathcal{O}(n)$, whereas the runtime of our implementation is in $\mathcal{O}(\log n)$.

Given 100 training iterations, REINFORCE agent with the default hyperparameters and a single objective function:

We observed an increase in runtime by a factor of 4 between graph execution, using TensorFlow's AutoGraphing feature, and eager execution. This shows, that using graph computation can lead to significant performance boosts.

We observed a runtime increase by a factor of more than 100 when trying to move computation to a GPU. Since the outer loop is not converted to a graph, and thus the data must be written back and forth between RAM and VRAM of the GPU leading to significant overhead. Since we use small neural nets, the performance boost a GPU provides during the training of the agent is not significant. If we increase the size of the neural net to (1024, 1024, 1024, 1024), we observe a small performance boost by including the GPU.

7 Future Work

In this section, we will discuss a few ideas we had while working on this thesis, which build upon our investigation. First, we will discuss a few approaches, which our testbench could include with minimal changes. An obvious improvement to our testbench is to increase the number of objective functions and operators used during training and evaluation. Combined with a larger number of experiments, this helps to more accurately estimate the relative performance correlation matrix and thus enable a more representative evaluation of sets of functions as described in Sec. 6.6. Some operators to add include random scaling, where the objective functions are scaled by a certain amount in the direction of a parameter. Also, a noise map, such that the evaluation of a location of the domain receives the same noise vector every time, instead of sampling a new random noise vector. Finally, an output translation, where we add a bias term to the reward.
Furthermore, the inclusion of higher-dimensional objective functions or even the training of a NN to confirm our findings. But the optimization of NNs requires a move to an unconstrained optimization task, which we also consider for future work.
Different agents are of interest as well, especially the use of an off-policy agent, which could provide an interesting comparison, as off-policy agents are known to learn policies, whose generalization to different environments is better than on-policy variants.
Since in this work we were limited to a visual and quantitative interpretation of the inner workings of learned optimization algorithms, a dynamic system approach as in Maheswaranathan et al. [MSM+20] could lead to a deeper understanding of the mechanisms we uncovered.
A well-known problem for learn-to-optimize is its scalability. Once trained, a policy, formulated like ours, can only attempt to solve 2D functions. We propose two architectures, which enable policies to scale to any dimensional functions. The first approach we propose is a model-based reinforcement learning approach, which uses the framework of Powell's method to reduce any optimization task to a linear search task. This linear search task remains the same, independent of the number of parameters, as only the parameter α_i for search direction s_i needs to be predicted. If we use our formulation of reinforcement learning as an optimization task, where the inputs for the policy are the reward signal, previous values of α_i, and the rewards associated with them. Further, the output of the policy is α_i, this framework could scale to any dimension.
The other approach builds on Bhargava et al.'s [BRL21] synapse-inspired model, where every node of a NN learns to optimize itself, by formulating a reinforcement learning problem, where the reward is the objective value and the policy's input is its weight and the reward signal for the last 3 iterations and the output is a very small and fixed weight adjustment. This formulation is capable of scaling to any size of optimization task, as every parameter is optimized independently by the same policy. Their approach converges slowly due to the very small and fixed step size and the limited information provided to each synapse. We propose changing the input to include more previous iterations. As our work shows, more observations up to a certain number lead to better results. Also, we propose a variable direct weight adjustment predicted by a distribution like in our formulation. Further, we propose to include the weights of surrounding synapses in the input of the policy. As

the term from learning psychology 'fire together, wire together' suggests, neurons in the human brain, which become active at the same time, tend to associate. [Heb49] To model this phenomenon, we have to provide a synapse with information about the activity of other synapses.

8 Conclusion

In this work, we presented a method for learning general optimization algorithms for constrained 2D optimization. We formulated it as a reinforcement learning problem, in which any optimization algorithm can be represented as a policy. We used the REINFORCE and PPO agents to learn multiple such optimization algorithms and evaluated them on eight different objective functions. We also evaluated them with respect to four operators: translation, rotation, input noise, and output noise. For this evaluation, we introduced our implementation of a testbench, where many different policies can be trained and evaluated. With this testbench, we identified eight policies trained on a single objective function, that are capable of outperforming any of our human-designed zeroth-order baseline algorithms on their respective objective function. We discussed the performance of these policies with respect to our operators and longer horizons. Further, we identified four trained policies, that are capable of optimizing all our eight objective functions with our operators applied as good or better than our human-designed zeroth-order baseline algorithms. Moreover, we visualized the trajectories taken by these policies and evaluated them to uncover the mechanisms they employ and discovered intuitive optimization strategies, such as estimation of steepest ascent, a converging spiral, and resetting to avoid local minima. We used the insight gained from these mechanisms to further analyze the optimization performances of these policies. Additionally, we demonstrated the feasibility of finetuning a pretrained policy to improve its performance on a related objective function. We proposed a method for discovering sets of related objective functions, where an optimization algorithm capable of generalization between them can be learned. Furthermore, we proposed a method to assign a generalization score to the properties of objective functions. We used our experiments to estimate them both. Finally, we provided a training guideline by discussing various performance improvements and discussed various runtime improvements for choices of agents and our testbench implementation.

Bibliography

[ADG+16] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, B. Shillingford, N. de Freitas. *Learning to learn by gradient descent by gradient descent*. 2016. DOI: 10.48550/ARXIV.1606.04474. URL: https://arxiv.org/abs/1606.04474 (cit. on pp. 7–9, 42).

[Aga18] A. F. Agarap. "Deep learning using rectified linear units (relu)". In: *arXiv preprint arXiv:1803.08375* (2018) (cit. on p. 12).

[BHA+21] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, E. Brynjolfsson, S. Buch, D. Card, R. Castellon, N. S. Chatterji, A. S. Chen, K. Creel, J. Q. Davis, D. Demszky, C. Donahue, M. Doumbouya, E. Durmus, S. Ermon, J. Etchemendy, K. Ethayarajh, L. Fei-Fei, C. Finn, T. Gale, L. Gillespie, K. Goel, N. D. Goodman, S. Grossman, N. Guha, T. Hashimoto, P. Henderson, J. Hewitt, D. E. Ho, J. Hong, K. Hsu, J. Huang, T. Icard, S. Jain, D. Jurafsky, P. Kalluri, S. Karamcheti, G. Keeling, F. Khani, O. Khattab, P. W. Koh, M. S. Krass, R. Krishna, R. Kuditipudi, et al. "On the Opportunities and Risks of Foundation Models". In: *CoRR* abs/2108.07258 (2021). arXiv: 2108.07258. URL: https://arxiv.org/abs/2108.07258 (cit. on p. 7).

[BRL21] A. Bhargava, M. R. Rezaei, M. Lankarany. *Gradient-Free Neural Network Training via Synaptic-Level Reinforcement Learning*. 2021. DOI: 10.48550/ARXIV.2105.14383. URL: https://arxiv.org/abs/2105.14383 (cit. on pp. 7, 9, 37, 67).

[CHC+16] Y. Chen, M. W. Hoffman, S. G. Colmenarejo, M. Denil, T. P. Lillicrap, M. Botvinick, N. de Freitas. *Learning to Learn without Gradient Descent by Gradient Descent*. 2016. DOI: 10.48550/ARXIV.1611.03824. URL: https://arxiv.org/abs/1611.03824 (cit. on pp. 7, 9, 42).

[DCLT18] J. Devlin, M. Chang, K. Lee, K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805 (cit. on p. 7).

[DPRL21] P. Domanski, D. Pflüger, J. Rivoir, R. Latty. "Self-Learning Tuning for Post-Silicon Validation". In: *CoRR* abs/2111.08995 (2021). arXiv: 2111.08995. URL: https://arxiv.org/abs/2111.08995 (cit. on p. 7).

[Heb49] D. O. Hebb. *The organization of behavior: A neuropsychological theory*. New York: Wiley, June 1949. ISBN: 0-8058-4300-0 (cit. on p. 68).

[HS97] S. Hochreiter, J. Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735 (cit. on p. 13).

[LJL17] K. Lv, S. Jiang, J. Li. *Learning Gradient Descent: Better Generalization and Longer Horizons*. 2017. DOI: 10.48550/ARXIV.1703.03633. URL: https://arxiv.org/abs/1703.03633 (cit. on pp. 7, 9).

[LLPS93] M. Leshno, V. Y. Lin, A. Pinkus, S. Schocken. “Multilayer feedforward networks with a nonpolynomial activation function can approximate any function”. In: *Neural Networks* 6.6 (1993), pp. 861–867. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(05)80131-5. URL: https://www.sciencedirect.com/science/article/pii/S0893608005801315 (cit. on p. 12).

[LM16] K. Li, J. Malik. *Learning to Optimize*. 2016. arXiv: 1606.01885 [cs.LG] (cit. on pp. 7–9, 37, 42).

[MSM+20] N. Maheswaranathan, D. Sussillo, L. Metz, R. Sun, J. Sohl-Dickstein. *Reverse engineering learned optimizers reveals known and novel mechanisms*. 2020. DOI: 10.48550/ARXIV.2011.02159. URL: https://arxiv.org/abs/2011.02159 (cit. on pp. 7, 9, 67).

[RXR+19] Y. Ruan, Y. Xiong, S. Reddi, S. Kumar, C.-J. Hsieh. *Learning to Learn by Zeroth-Order Oracle*. 2019. DOI: 10.48550/ARXIV.1910.09464. URL: https://arxiv.org/abs/1910.09464 (cit. on pp. 7, 9, 42).

[SQXH19] C. Sun, X. Qiu, Y. Xu, X. Huang. “How to Fine-Tune BERT for Text Classification?” In: *CoRR* abs/1905.05583 (2019). arXiv: 1905.05583. URL: http://arxiv.org/abs/1905.05583 (cit. on p. 7).

[SWD+17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov. “Proximal Policy Optimization Algorithms.” In: *CoRR* abs/1707.06347 (2017). URL: http://dblp.uni-trier.de/db/journals/corr/corr1707.html#SchulmanWDRK17 (cit. on pp. 8, 16).

[Wil92] R. J. Williams. “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. In: *Mach. Learn.* 8.3–4 (May 1992), pp. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696 (cit. on pp. 8, 16).

[WMH+17] O. Wichrowska, N. Maheswaranathan, M. W. Hoffman, S. G. Colmenarejo, M. Denil, N. de Freitas, J. Sohl-Dickstein. *Learned Optimizers that Scale and Generalize*. 2017. DOI: 10.48550/ARXIV.1703.04813. URL: https://arxiv.org/abs/1703.04813 (cit. on pp. 7, 9, 42).

All links were last followed on July 26th, 2022.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature