Reinforcement Learning with General Evaluators and Generators of Policies

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

presented by
Francesco Faccio

under the supervision of
Prof. Jürgen Schmidhuber

January 2024

Dissertation Committee

Prof. Cesare Alippi	Università della Svizzera italiana, Switzerland
Prof. Rolf Krause	Università della Svizzera italiana, Switzerland
Dr. Alex Graves	NNAISENSE, Lugano, Switzerland
Prof. Marcello Restelli	Politecnico di Milano, Milan, Italy

Dissertation accepted on 18 January 2024

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

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

Francesco Faccio
Lugano, 18 January 2024

Abstract

Reinforcement Learning (RL) is a subfield of Artificial Intelligence that studies how machines can make decisions by learning from their interactions with an environment. The key aspect of RL is evaluating and improving policies, which dictate the behavior of artificial agents by mapping sensory input to actions. Typically, RL algorithms evaluate these policies using a value function, generally specific to one policy. However, when value functions are updated to track the learned policy, they can forget potentially useful information about previous policies. To address the problem of generalization across many policies, we introduce Parameter-Based Value Functions (PBVFs), a class of value functions that take policy parameters as inputs. A PBVF is a single model capable of evaluating the performance of any policy, given a state, a state-action pair, or a distribution over the RL agent's initial states, and it can generalize across different policies. We derive off-policy actor-critic algorithms based on PBVFs. To input the policy into the value function, we employ a technique called policy fingerprinting. This method compresses the policy parameters, rendering PBVFs invariant to changes in the policy architecture. This policy embedding extracts crucial abstract knowledge about the environment, distilled into a limited number of states sufficient to define the behavior of various policies. A policy can improve solely by modifying actions in such states, following the gradient of the value function's predictions. Extensive experiments demonstrate that our method outperforms evolutionary algorithms, demonstrating a more efficient direct search in the policy space. Furthermore, it achieves performance comparable to that of competitive continuous control algorithms. We apply this technique to learn useful representations of Recurrent Neural Network weight matrices, showing its effectiveness in several supervised learning tasks. Lastly, we empirically demonstrate how this approach can be integrated with HyperNetworks to train a single goal-conditioned neural network (NN) capable of generating deep NN policies that achieve any desired return observed during training. The majority of this thesis is based on previous papers published by the author [Faccio et al., 2021, 2022, 2023; Herrmann et al., 2023].

Acknowledgements

I would like to express my gratitude to my PhD advisor, Jürgen Schmidhuber, for guiding me through my PhD journey and making it an enjoyable experience between Switzerland and Saudi Arabia. I am deeply thankful to my mother, Teresa, and my father, Bruno, for their unwavering support and love over these years. Papà, you would be proud of this moment if you were still here! My appreciation also goes to my brother, Federico, for our long remote conversations, often while having dinner very far from each other, and to my grandparents, nonno Erminio and nonna Liliana.

I extend my thanks to my current and past colleagues in Switzerland – Aleksandar, Robert, Louis, Paulo, Imanol, Sjoerd, Xingdong, Felipe, Raoul, Anand, Aditya, Vincent, Dylan, Krsto, Mikhail, Michael, Kazuki, Mirek, Qinhan, and to my colleagues at KAUST – Yuhui, Haozhe, Mingchen, Wenyi, Piotr, Oleg. I am particularly grateful to those who shared in several endless hikes, board game nights, and holidays with me.

My heartfelt thanks go to Noor, for all her support throughout these years, and to Bernard, Liliana, Tagreed, and Sarah for welcoming me to KAUST. Finally, I want to mention some of the other people with whom I spent the most and best time: Lady, Silvio, Qian, Anthony, Yulian, Vanessa, Romina, Julius, Camila, and Kana.

Contents

Contents vii

1 Introduction 1
1.1 Contributions . . . 2

2 Background 5
2.1 Machine Learning . . . 5
2.1.1 Supervised Learning . . . 5
2.1.2 Unsupervised Learning . . . 6
2.1.3 Reinforcement Learning . . . 7
2.2 Deep Learning in Neural Networks . . . 7
2.2.1 Neural Network Architectures . . . 8
2.2.2 Learning in Neural Networks . . . 10
2.3 Reinforcement Learning . . . 11
2.3.1 General Remarks . . . 11
2.3.2 Markov Decision Processes . . . 12
2.3.3 Off-policy RL . . . 13

3 Parameter-Based Value Functions 15
3.1 Method . . . 15
3.1.1 Parameter-based Start-State-Value Function $V(\theta)$ . . . 16
3.1.2 Parameter-based State-Value Function $V(s,\theta)$ . . . 17
3.1.3 Parameter-based Action-Value Function $Q(s,a,\theta)$ . . . 18
3.2 Experiments and Results . . . 20
3.2.1 Visualizing PBVFs using LQRs . . . 21
3.2.2 Main results . . . 22
3.2.3 Zero-shot learning . . . 25
3.2.4 Offline learning with fragmented behaviors . . . 28
3.3 Related Work . . . 29

3.4 Discussion 31

4 General Policy Evaluation and Improvement by Learning to Identify Few But Crucial States **33**

4.1 Method 33
4.1.1 Static Policy fingerprinting 33
4.1.2 Recurrent Policy Fingerprinting 35
4.2 Experiments and Results 36
4.2.1 Motivating experiments on MNIST 36
4.2.2 Main experiments on MuJoCo 40
4.2.3 Zero-shot learning of new policy architectures 43
4.2.4 Fingerprint Analysis 43
4.3 Related Work 50
4.4 Discussion 51

5 Learning Useful Representations of Recurrent Neural Network Weight Matrices **53**

5.1 Method 54
5.1.1 RNN Encoders 55
5.2 Experiments and Results 57
5.2.1 Dataset 57
5.2.2 Emulator 57
5.2.3 Results 57
5.3 Discussion 61

6 Goal-Conditioned Generators of Deep Policies **63**

6.1 Method 63
6.1.1 Background 64
6.1.2 Fast Weights Programmers 64
6.1.3 Gogepo 65
6.1.4 HyperNetworks 68
6.2 Experiments and Results 70
6.2.1 Results on Continuous Control RL Environments 70
6.2.2 Analyzing the Generator's Learning Process 72
6.2.3 Limitation: obtaining suitable policies from the start 74
6.3 Related Work 75
6.3.1 Hindsight and Upside Down RL. 75
6.3.2 On the convergence of Upside Down RL 76
6.3.3 Fast Weight Programmers and HyperNetworks 76

6.4 Discussion 77

7 Outlook **79**

Publications during the PhD program **81**

A Parameter-Based Value Functions **83**
A.1 Proofs and derivations 83
A.2 Implementation details 86
A.3 Experimental details 91
A.3.1 LQR 91
A.3.2 Main Experiments 92
A.3.3 Sensitivity analysis 93
A.3.4 Table of best hyperparameters 93

B General Policy Evaluation and Improvement by Learning to Identify Few But Crucial States **109**
B.1 Implementation details 109
B.1.1 MNIST Implementation 109
B.1.2 RL Implementation 110
B.1.3 GPU usage / Computation requirements 112
B.2 Experimental details 112
B.2.1 Main experiments on MuJoCo 112
B.2.2 Ablation on weighted sampling 113
B.2.3 Comparison to vanilla PSSVF 114

C Learning Useful Representations of Recurrent Neural Network Weight Matrices **115**
C.1 Experimental details 115
C.1.1 Hyperparameters 115

D Goal-Conditioned Generators of Deep Policies **117**
D.1 Implementation details 117
D.1.1 Hyperparameters 117
D.1.2 Generator implementation details 120
D.1.3 GPU usage / compute 121
D.2 Experimental details 121
D.2.1 Main experiments on MuJoCo 121
D.2.2 Command strategies 121
D.3 Environment details 122

Bibliography **123**

Chapter 1

Introduction

Artificial Intelligence (AI) is a field of computer science focused on creating machines capable of tasks that typically require human intelligence. This includes activities like learning, problem-solving, and language understanding. AI is divided into two main categories: Narrow AI, designed for specific tasks, and General AI, which mimics human intelligence more broadly. AI applications are extensive and growing, involving technologies like machine learning and natural language processing, and have the potential to significantly impact many aspects of daily life and various industries.

A prominent branch of AI is Reinforcement Learning (RL) [Sutton and Barto, 2018], which centers on sequential decision-making. This process involves an agent interacting with an environment in a series of steps, where it chooses actions based on received observations and rewards. The primary objective of the agent is to maximize the total rewards accumulated during an episode, which begins from an initial state and is directed towards achieving specific goals as indicated by the rewards. These rewards vary depending on the context. For example, in locomotion tasks, rewards may be linked to the agent's directional velocity, while in gaming, they might correlate with the player's score. The agent's policy is a function that dictates which actions should be taken based on any given observation. Initially, these policies may not correspond to desired behaviours. Therefore, RL algorithms are typically adopted to alternate between collecting new data with the current policy, and using such data to improve the policy.

A crucial aspect of policy improvement relies on accurate evaluation of policies. Value functions are designed to estimate the expected sum of rewards of a given policy when starting from a particular state or after choosing a specific action within that state. This evaluation is key to understanding and improving the effectiveness of the policy in achieving the desired outcomes in various RL scenarios. Many RL

breakthroughs were achieved through improved estimates of such values. These advancements span a diverse range of applications, from board games such as backgammon [Tesauro, 1995], extending to video games [Mnih et al., 2015], and even including the complex task of controlling nuclear fusion reactors [Degrave et al., 2022]. However, learning value functions of arbitrary policies without observing their behavior in the environment is not trivial. Such off-policy learning must correct the mismatch between the distribution of updates induced by the behavioral policy and the one we want to learn. Common techniques include Importance Sampling (IS) [Hesterberg, 1988] and deterministic policy gradient methods (DPG) [Silver et al., 2014], which adopt the actor-critic architecture [Sutton, 1984; Konda and Tsitsiklis, 2001; Peters and Schaal, 2008].

Unfortunately, these approaches have limitations. While estimators of value functions using IS are generally unbiased, their variance can increase exponentially when the policy we want to learn diverges significantly from the behavioral policy [Cortes et al., 2010; Metelli et al., 2018; Wang et al., 2016]. Moreover, traditional off-policy actor-critic methods introduce off-policy objectives whose gradients are difficult to follow since they involve the gradient of the action-value function with respect to the policy parameters [Degris et al., 2012; Silver et al., 2014]. This term is usually ignored, resulting in biased gradients for the off-policy objective. Furthermore, off-policy actor-critic algorithms learn value functions of a single target policy. When value functions are updated to track the learned policy, the information about old policies is lost. While traditional value functions are designed to generalize across new states or state-action pairs, achieving generalization over new policies remains an open problem. This thesis introduces a set of novel techniques to tackle this challenge.

1.1 Contributions

We address the problem of generalization across many policies in the off-policy setting by introducing a class of *parameter-based value functions* (PBVFs) defined for any policy. PBVFs are value functions that take as input the policy parameters, along with a given state, state-action pair, or distribution over the RL agent's initial states. These functions can be learned using Monte Carlo (MC) [Metropolis and Ulam, 1949] or Temporal Difference (TD) [Sutton, 1988] methods. The PBVF that considers state-action pairs and policy parameters leads to a novel stochastic and deterministic off-policy policy gradient theorem. Unlike previous approaches, it can directly compute the gradient of the action-value function with respect to the

policy parameters. Based on these results, we develop off-policy actor-critic methods and compare our algorithms to two strong baselines, ARS and DDPG [Mania et al., 2018; Lillicrap et al., 2015], outperforming them in some environments. PBVFs are formally introduced in Chapter 3.

A crucial problem in the application of such value functions is choosing a suitable representation of the policy. Flattening the policy parameters leads to vanilla PBVFs that are difficult to scale to larger policies. In Chapter 4, we present an approach that connects PBVFs and a policy embedding method called "fingerprint mechanism" by Harb et al. [2020]. Using policy fingerprinting allows us to scale PBVFs to handle larger neural network (NN) policies and also achieve invariance with respect to the policy architecture. Policy fingerprinting was introduced to learn maps from policy parameters to expected return offline and prior to this work was never applied to the online RL setting. In this approach, which we term *static* policy fingerprinting, our PBVF is designed to learn a set of crucial abstract states alongside an evaluator function. For policy evaluation, these abstract states are passed as input to the policy, and the evaluator function then maps the resulting actions of the policy in these states to the expected return. A policy improves solely by changing actions in probing states, following the gradient of the value function's predictions. We show in continuous control problems that our approach can identify a small number of critical "probing states" that are highly informative of the policies performance. Our learned value function generalizes across many NN-based policies. It combines the behavior of many bad policies to learn a better policy, and is able to zero-shot learn policies with a different architecture. We compare our approach with strong baselines in continuous control tasks: our method is competitive with DDPG [Lillicrap et al., 2015] and evolutionary approaches.

While static policy fingerprinting is promising, it presents a crucial drawback: it uses the same set of learned states to evaluate many different policies. In practice, the action of each policy might need to be measured in different states for accurate evaluation. In Section 4.1.2, we introduce a technique called *recurrent* policy fingerprinting, which sequentially generates a set of states used to probe each specific policy. While this technique has not been evaluated yet on RL tasks, we show in Chapter 5 how it can be used to learn useful representations of Recurrent Neural Networks weight matrices.

Finally, in Chapter 6, we focus on the context of Goal-conditioned RL, which aims to learn optimal policies when given goals encoded as special command inputs. We study goal-conditioned NNs that learn to generate deep NN policies in the form of context-specific weight matrices. This approach is similar to Fast Weight Programmers (FWP) [Schmidhuber, 1992b, 1993] and other methods from the 1990s. In our research, we combine PBVFs with policy fingerprinting and a form of weight-

sharing HyperNetworks [Ha et al., 2016] to train an FWP to generate the parameters of the desired deep policy in response to a “desired return” command. This facilitates end-to-end optimization of the return-conditioned generator, producing deep NN policies by matching the commands (desired returns) to the evaluated returns. Our analysis reveals that a single learned policy generator can produce policies that achieve any desired return observed during training.

This thesis concludes with Chapter 7, which outlines promising directions for future research. In the next chapter, Chapter 2, we provide a succinct introduction to Machine Learning and, in particular, Reinforcement Learning. This background is essential for understanding the main methodologies proposed in the thesis.

Chapter 2

Background

This Chapter provides an introduction to Deep Learning and Reinforcement Learning. For a more advanced introduction, the reader is invited to consult Sutton and Barto [2018] and Bishop [2006].

2.1 Machine Learning

Machine Learning is a specialized subfield of Artificial Intelligence (AI) that emphasizes the development of algorithms capable of identifying patterns in data. In this context, the *learned machine* typically refers to a function that operates based on a set of adjustable parameters. Learning in this field involves using data to modify these parameters, enabling the function to exhibit desired behaviors. How each component of a learned machine influences its performance has been identified as the *fundamental credit assignment problem* [Minsky, 1963]. The primary aim of Machine Learning models is to achieve generalization, which means ensuring the model performs effectively on new, unseen data.

In this overview, we will explore the most prevalent paradigms in Machine Learning: Supervised Learning, Unsupervised Learning, and Reinforcement Learning.

2.1.1 Supervised Learning

In certain Machine Learning problems, the primary objective is to construct a model capable of associating input patterns, denoted as $x \in \mathbb{R}^{n_x}$, with corresponding targets, denoted as $y \in \mathbb{R}^{n_y}$. Here the model can assume the form of a parameterized function $f_w : \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_y}$, so that $y = f_w(x)$. The vector $w \in \mathbb{R}^{n_w}$ represents the model's parameters. When the target values are provided in the dataset, meaning that the data comprises pairs of (x, y), this type of problem is identified as supervised learning.

Regression If the target data are known and assume continuous values, we have a Regression problem. An example of such a problem is the prediction of temperature, precipitation levels, or humidity based on weather data and atmospheric conditions. Given some data $D = \{(x_i, y_i) | i = 1, 2, \dots\}$, we can train a model f_w by finding the parameters w^* minimizing the square of the difference between the predicted target and the true target in the dataset, i.e.:

$$w^* = \arg\min_w L(w, D) = \arg\min_w \mathop{\mathbb{E}}_{(x,y)\in D} [(f_w(x) - y)^T (f_w(x) - y)] \tag{2.1}$$

Classification Classification is a type of supervised learning problem where the target is represented as a discrete set of categories. This is typically achieved using one-hot encoding, where each category is represented by a vector containing a single '1' corresponding to the class label, and '0's in all other positions. For example, in a problem with three classes, the target vectors would be [1, 0, 0], [0, 1, 0], and [0, 0, 1] for the first, second, and third classes, respectively.

Given a dataset $D = \{(x_i, y_i) | i = 1, 2, \dots\}$, where $y_i \in \{0, 1\}^C$ and C is the number of classes, the goal is to train a model f_w that accurately predicts the class label. The function $f_w : \mathbb{R}^{n_x} \to \{0, 1\}^C$ maps input features to one-hot encoded class labels. The objective is to find the model parameters w^* that minimize the discrepancy between predicted and actual target vectors, often using a loss function like cross-entropy for multi-class classification.

The optimization problem can be formalized as:

$$w^* = \arg\min_w L(w, D) = \arg\min_w \mathop{\mathbb{E}}_{(x,y)\in D} [-\sum_{c=1}^{C} y_c \log f_w(x)_c] \tag{2.2}$$

where y_c is the cth element of the one-hot encoded vector y, and $f_w(x)_c$ is the predicted probability of class c.

2.1.2 Unsupervised Learning

In Unsupervised Learning, the target value y is typically not present in the dataset. The objective of these problems is to identify groups within the data that share similar patterns or to develop models that estimate the data's distribution. These models can then be employed to generate new data that resembles the existing dataset. Additionally, Unsupervised Learning can be utilized to understand objects, their composition and relationships [Greff et al., 2020], and how object representation can aid various downstream tasks [van Steenkiste et al., 2019]. In specific applications like data compression, the target values are effectively the input data itself. Here,

models can be trained to compress and reconstruct the input data in an unsupervised manner.

2.1.3 Reinforcement Learning

Reinforcement Learning (RL) is a specialized area within Machine Learning that explores how an artificial agent interacts with its environment. In RL, an agent begins in an initial state and, through iterative processes, decides which action to take at each subsequent state. With each action, the environment shifts to a new state, and the agent receives a reward that reflects the effectiveness of its action. In particularly complex environments, the impact of an action on the reward can extend far into the future. RL agents must learn through experimentation to identify actions that yield the highest rewards.

2.2 Deep Learning in Neural Networks

In this thesis, we will frequently discuss Neural Networks (NNs). NNs comprise multiple interconnected processors known as neurons. Input neurons gather environmental information through observations, generating real-valued activations. These values are processed by subsequent neurons via weighted connections, often revealing complex nonlinear patterns. Output neurons utilize these patterns to initiate actions that may influence the environment. The process of learning, or credit assignment, involves adjusting the NN's weights to achieve desired behavior. This is typically defined by minimizing a loss function or maximizing accumulated rewards.

The concept of shallow NNs dates back to the 19th century when Gauss used simple linear models with astronomical data to predict the dwarf planet Ceres's location, although the method was published only later in 1809 [Gauss, 1809; Stigler, 1981]. The development of deeper, multi-layer NNs began in the 1950s with the perception [Rosenblatt, 1958]. However, it wasn't until 1965 that Ivakhnenko and Lapa [1965] introduced the first practical Deep Learning algorithm for NNs. Initially, NNs were trained layer by layer, but in 1970, Linnainmaa [1970] introduced an efficient method for backpropagating errors in deep NNs, which later gained popularity for learning representations in hidden layers [Rumelhart et al., 1986]. Even more than two decades later, training very deep NNs (with over 10 layers) remained challenging due to the vanishing or exploding gradient problem, identified by Hochreiter [1991]. In deep NNs, backpropagated errors either grow or diminish exponentially across layers. This issue affects both feedforward NNs, where neurons form an acyclic graph, and recurrent NNs (RNNs), where the graph is cyclic.

The introduction of gated mechanisms like the Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997] for RNNs and the LSTM-based Highway Networks [Srivastava et al., 2015] for feedforward NNs helped mitigate this fundamental deep learning problem, leading to the development of very deep NNs. Advances in LSTM [Gers et al., 2000; Graves et al., 2006] and feedforward NNs, particularly in language modeling and computer vision, have produced models capable of superhuman performance, winning several international contests [Graves and Schmidhuber, 2009; Ciresan et al., 2012, 2011; Krizhevsky et al., 2012; He et al., 2015].

For an in-depth exploration of NN history, readers are encouraged to consult Schmidhuber [2022, 2015a]. The next chapter will formally introduce the NNs utilized in this thesis.

2.2.1 Neural Network Architectures

Feedforward Neural Networks The simplest and most commonly used type of neural network (NN) is composed of multiple fully connected layers. This includes the multi-layer perceptron (MLP), which has L fully connected hidden layers, denoted as $l \in 1,\ldots,L$, and an output layer. This structure forms a linear graph. The MLP models a function $f_w : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$, parameterized by $w \in \mathbb{R}^{n_w}$. Each hidden layer l transforms an input vector x^l into an output vector y^l. The input to each layer is the output from the previous layer, meaning $x^l = y^{l-1}$. The first layer takes external data $x \in \mathbb{R}^{n_x}$ as its input, while the final output layer produces the output $\hat{y}$. Each layer l linearly transforms the input into a vector z^l, which has the same dimensionality as the next layer, and then applies a nonlinear activation function σ element-wise. The transformation performed by the jth neuron in layer l is as follows:

$$z^l_j = \sum_{i=0}^{n_{l-1}-1} W^l_{i,j} \cdot y^{l-1}_i + b^l_j \tag{2.3}$$

$$y^l_j = \sigma(z^l_j). \tag{2.4}$$

Here, $W \in \mathbb{R}^{n_{l-1} \times n_l}$ and $b \in \mathbb{R}^{n_l}$ represent the weight and bias parameters from layer $l-1$ to layer l. Here, y^0 is the input layer, and y^{L+1} is the output of the NN. Common activation functions used in these networks are the Hyperbolic Tangent ($tanh(z)$) and the Rectified Linear Unit (ReLU) [Fukushima, 1969], defined as $\max(0,z)$.

Convolutional Neural Networks Convolutional Neural Networks (CNNs) [Fukushima, 1979; Waibel, 1987; Zhang et al., 1988] are specialized for processing data with a grid-like structure, such as images. They use convolutional layers, where small, learnable filters slide across the input image to create feature maps that capture

spatial features. Unlike fully connected layers, these filters focus on small regions (receptive fields) and are repeated across the entire input, enabling the network to detect features like edges or textures regardless of their position in the image. CNNs also typically include pooling layers, like max pooling, to reduce the size of the feature maps and control overfitting, making them highly efficient for image recognition tasks.

Recurrent Neural Networks Many Machine Learning problems involve finding patterns in sequential data. In the most challenging case, a learned model should capture long time-lags between relevant events. To address this, contemporary Recurrent Neural Networks (RNNs) [Elman, 1990] incorporate a cyclic computational graph. This design ensures that the network's output depends not only on the current input data but also on its previous outputs. In this process, each data sequence element is fed into the RNN at a distinct time step, denoted as t. In its simplest form, the RNN consists of a single recurrent fully connected layer and transforms at time step t the input data as follows:

$$z_j[t] = \sum_{i=0}^{n_x-1} W_{i,j} \cdot x_i[t] + \sum_{i=0}^{n_l-1} R_{i,j} \cdot y_j[t-1] + b_j \tag{2.5}$$

$$y_j[t] = \sigma(z_j). \tag{2.6}$$

Here, $W \in \mathbb{R}^{n_x \times n_l}$ and $b \in \mathbb{R}^{n_l}$ represent the weight and bias parameters from the input layer to the hidden layer. $R \in \mathbb{R}^{n_l \times n_l}$ represents the weight of the connections between the hidden layer at time step $t-1$ and the hidden layer at time step t.

Output layer The choice of activation functions in the final layer of a neural network (NN) is influenced by the specific task it is designed to perform. For classification tasks, where the goal is to output a probability distribution across various categories, the softmax activation function is commonly employed. This function modifies each element of the output vector y_k using the formula $y_j = \frac{e^{y_j}}{\sum_{i=0}^{n_y-1} e^{y_i}}$, effectively converting raw scores into probabilities. In contrast, for regression tasks where the output value is unbounded, the identity activation function is appropriate, as it leaves the output unchanged. Meanwhile, in situations where the NN's output needs to fall within a specific range, the tanh activation function is utilized. This function scales the values to lie between $[-1,1]$, and these values can subsequently be adjusted to fit any desired range.

2.2.2 Learning in Neural Networks

To train an NN, we assume that we can compute a loss function $L(w,D)$ which is differentiable over the NN parameters w. The loss depends on the NN and on the data we are using, and can assume different forms based on the task.

Backpropagation As outlined in Section 2.2, backpropagation is a method that efficiently calculates the derivatives of a neural network's (NN) loss function relative to its parameters. This process facilitates credit assignment by iteratively employing the chain rule.

In this context, we focus on the backpropagation algorithm as it applies to a Multilayer Perceptron (MLP). The objective here is to determine $\frac{\partial L(w,D)}{\partial W^l_{i,j}}$ and $\frac{\partial L(w,D)}{\partial b^l_j}$. Let $\delta^l_i := \frac{\partial L(w,D)}{\partial z^l_i}$ be the error for neuron i in layer l. We can rewrite the derivatives for each weight as: $\frac{\partial L(w,D)}{\partial W^l_{i,j}} = \delta^l_j \cdot y^{l-1}_i$, and $\frac{\partial L(w,D)}{\partial b^l_j} = \delta^l_j$. The calculation of δ^l_j in backpropagation begins at the output layer and systematically applies the chain rule through the following steps:

$$\delta^l_i = \frac{\partial L(w,D)}{\partial z^l_i} \tag{2.7}$$

$$= \frac{\partial y^l_i}{\partial z^l_i}\frac{\partial L(w,D)}{\partial y^l_i} \tag{2.8}$$

$$= \sigma'(z^l_i)\sum_{j=0}^{n_{l+1}-1}\frac{\partial L(w,D)}{\partial z^{l+1}_j}\frac{\partial z^{l+1}_j}{\partial y^l_i} \tag{2.9}$$

$$= \sigma'(z^l_i)\sum_{j=0}^{n_{l+1}-1}\delta^{l+1}_j \cdot W^l_{i,j}. \tag{2.10}$$

Given that each neuron is computed only once, the complexity mirrors that of the forward pass.

When applying backpropagation to Recurrent Neural Networks (RNNs), the key strategy is to unfold the network's computational graph forward in time and then directly implement the backpropagation algorithm. This approach leads to a variant known as backpropagation through time (BPTT) [Werbos, 1990; Williams and Zipser, 1994].

Gradient Descent After computing the gradients of the loss function with respect to the neural network (NN) parameters, any optimization technique can be applied to improve the NN's performance. The most widely used method is gradient descent.

This technique iteratively updates the NN parameters in the gradient's direction, as expressed in the following equation:

$$w_{new} = w_{old} - \alpha \frac{\partial L(w_{old}, D)}{\partial w_{old}}. \tag{2.11}$$

Here, $\alpha > 0$ represents the learning rate. Typically, the loss is not calculated over the entire dataset but on smaller subsets, known as batches. This approach leads to a method named Stochastic Gradient Descent (SGD) [Robbins and Monro, 1951; Kiefer and Wolfowitz, 1952; Amari, 1967]. The most prevalent variations of SGD incorporate momentum techniques to quicken convergence. For instance, the Adam [Kingma and Ba, 2015] algorithm adjusts learning rates and momentum throughout the optimization process, aiming for more effective and quicker convergence.

2.3 Reinforcement Learning

2.3.1 General Remarks

In recent years, Reinforcement Learning has achieved remarkable success in simulated and real world applications. For instance, it has been used to learn to achieve superhuman performance in videogames, including Atari games [Mnih et al., 2015], Go [Silver et al., 2016], Dota2 [Berner et al., 2019], Starcraft [Vinyals et al., 2019], and Gran Turismo [Wurman et al., 2022]. Video games are particularly well-suited for studying RL algorithms [Schaul et al., 2011]. Remarkably, it has also been applied in diverse areas such as controlling a nuclear fusion reactor [Degrave et al., 2022], piloting a balloon to the stratosphere [Bellemare et al., 2020], solving the Rubik's Cube with a robotic hand [OpenAI et al., 2020], discovering new algorithms for matrix multiplication [Fawzi et al., 2022], and aligning large language models [Ouyang et al., 2022]. The RL framework is highly general, allowing a wide range of learning problems to be formulated within its paradigm. The core objective in RL is for the agent to determine which actions in specific states yield the highest returns. However, this process involves a critical tradeoff between exploration, the costly task of discovering new actions and their consequences, and exploitation, where the agent uses its existing knowledge to make beneficial decisions. Exploration is essential but often resource-intensive, as each interaction with the environment incurs a cost. There are several ways to efficiently explore the environment, including curiosity-based algorithms [Schmidhuber, 1990], which reward the agent by visiting surprising states, and reward-free methods [Jin et al., 2020]. To exploit the environment, researchers build models that can either predict the environment's reaction to

the artificial agent, or evaluate the agent's performance. Model-based methods, for instance, utilize a model of the environment, either pre-existing or learned, to predict future states resulting from an agent's actions. These methods enable planning by using the model as a surrogate for the actual environment. Examples include World models [Schmidhuber, 1990; Ha and Schmidhuber, 2018], Dyna [Sutton, 1990], and methods based on Monte Carlo Tree Search [Silver et al., 2016]. In scenarios where an environmental model is unavailable or challenging to construct, model-free algorithms come into play. Model-free, value-based methods focus on learning value functions, which estimate the expected return of using a specific policy in a given state or following a particular action. Q-Learning [Watkins and Dayan, 1992], one of the most well-known RL algorithms, employs a value function to identify actions that maximize expected returns from any state. Alternatively, policy-based methods such as policy gradient [Sutton et al., 1999] directly learn a parameterized policy that dictates the agent's behavior. These methods define an objective function that is differentiable with respect to policy parameters and typically learn the policy by applying gradient ascent to this objective. Policy gradient techniques learn by increasing the probability of achieving higher returns and may concurrently learn a value function to accelerate policy improvement. This is the case for actor-critic methods [Sutton, 1984; Konda and Tsitsiklis, 2001; Peters and Schaal, 2008], where both policy and value function are iteratively updated. Another distinction can be made between on-policy and off-policy [Precup et al., 2001] algorithms. On-policy learning frameworks use data generated by the same policy that is being learned. Conversely, off-policy learning involves leveraging data gathered from one or multiple different behavioral policies to learn about a policy that has not been directly tested in the environment. The next section will present a formal definition of RL within the framework of Markov Decision Processes.

2.3.2 Markov Decision Processes

We consider a Markov Decision Process (MDP) [Stratonovich, 1960; Puterman, 2014] $\mathscr{M} = (\mathscr{S}, \mathscr{A}, P, R, \gamma, \mu_0)$. The state space $\mathscr{S} \subset \mathbb{R}^{n_S}$ and the action space $\mathscr{A} \subset \mathbb{R}^{n_A}$ are assumed to be compact sub-spaces. At each step, an agent observes a state $s \in \mathscr{S}$, chooses action $a \in \mathscr{A}$, transitions into state s' with probability $P(s'|s,a)$ and receives a reward $R(s,a) \in \mathbb{R}$. The agent starts from an initial state, chosen with probability $\mu_0(s)$. It is represented by a parametrized stochastic policy $\pi_\theta : \mathscr{S} \to \Delta(\mathscr{A})$, which provides the probability of performing action a in state s, where $\theta \in \Theta \subset \mathbb{R}^{n_\Theta}$ are the policy parameters. The policy is deterministic if for each state s there exists an action a such that $\pi_\theta(a|s) = 1$. The return R_t is defined as the cumulative discounted reward from time step t: $R_t = \sum_{k=0}^{T-t-1} \gamma^k R(s_{t+k+1}, a_{t+k+1})$, where T denotes the time

horizon and $\gamma \in (0,1]$ a discount factor. The performance of the agent is measured by the cumulative discounted expected reward (expected return), defined as $J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[R_0]$. Given a policy π_θ, the state-value function $V^{\pi_\theta}(s) = \mathbb{E}_{\pi_\theta}[R_t|s_t = s]$ is defined as the expected return for being in a state s and following policy π_θ. By integrating over the state space $\mathscr{S}$, we can express the maximization of the expected cumulative reward in terms of the state-value function $J(\pi_\theta) = \int_{\mathscr{S}} \mu_0(s) V^{\pi_\theta}(s)\,\mathrm{d}s$. The action-value function $Q^{\pi_\theta}(s,a)$, which is defined as the expected return for performing action a in state s, and following the policy π_θ, is $Q^{\pi_\theta}(s,a) = \mathbb{E}_{\pi_\theta}[R_t|s_t = s, a_t = a]$, and it is related to the state-value function by $V^{\pi_\theta}(s) = \int_{\mathscr{A}} \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\,\mathrm{d}a$. We define as $d^{\pi_\theta}(s')$ the discounted weighting of states encountered starting at $s_0 \sim \mu_0(s)$ and following the policy π_θ: $d^{\pi_\theta}(s') = \int_{\mathscr{S}} \sum_{t=1}^{\infty} \gamma^{t-1} \mu_0(s) P(s \to s', t, \pi_\theta)\,\mathrm{d}s$, where $P(s \to s', t, \pi_\theta)$ is the probability of transitioning to s' after t time steps, starting from s and following policy π_θ. Sutton et al. [1999] showed that, for stochastic policies, the gradient of $J(\pi_\theta)$ does not involve the derivative of $d^{\pi_\theta}(s)$ and can be expressed in a simple form:

$$\nabla_\theta J(\pi_\theta) = \int_{\mathscr{S}} d^{\pi_\theta}(s) \int_{\mathscr{A}} \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\,\mathrm{d}a\,\mathrm{d}s. \tag{2.12}$$

Similarly, for deterministic policies, Silver et al. [2014] obtained the following:

$$\nabla_\theta J(\pi_\theta) = \int_{\mathscr{S}} d^{\pi_\theta}(s) \nabla_\theta \pi_\theta(s) \nabla_a Q^{\pi_\theta}(s,a)|_{a=\pi_\theta(s)}\,\mathrm{d}s. \tag{2.13}$$

2.3.3 Off-policy RL

In off-policy policy optimization, we seek to find the parameters of the policy maximizing a performance index $J_b(\pi_\theta)$ using data collected from a behavioral policy π_b. Here the objective function $J_b(\pi_\theta)$ is typically modified to be the value function of the target policy, integrated over $d_\infty^{\pi_b}(s) = \lim_{t\to\infty} P(s_t = s|s_0, \pi_b)$, the limiting distribution of states under π_b (assuming it exists) [Degris et al., 2012; Imani et al., 2018; Wang et al., 2016]. Throughout this chapter, we assume that the support of $d_\infty^{\pi_b}$ includes the support of μ_0 so that the optimal solution for J_b is also optimal for J. Formally, we want to find:

$$J_b(\pi_{\theta^*}) = \max_\theta \int_{\mathscr{S}} d_\infty^{\pi_b}(s) V^{\pi_\theta}(s)\,\mathrm{d}s = \max_\theta \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \int_{\mathscr{A}} \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\,\mathrm{d}a\,\mathrm{d}s. \tag{2.14}$$

Unfortunately, in the off-policy setting, the states are obtained from $d_\infty^{\pi_b}$ and not from $d_\infty^{\pi_\theta}$, hence the gradients suffer from a distribution shift [Liu et al., 2019; Nachum

et al., 2019]. Moreover, since we have no access to $d_\infty^{\pi_\theta}$, a term in the policy gradient theorem corresponding to the gradient of the action value function with respect to the policy parameters needs to be estimated. This term is usually ignored in traditional off-policy policy gradient theorems[1]. In particular, when the policy is stochastic, Degris et al. [2012] showed that:

$$\nabla_\theta J_b(\pi_\theta) = \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \int_{\mathscr{A}} \pi_b(a|s) \frac{\pi_\theta(a|s)}{\pi_b(a|s)} \left(Q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q^{\pi_\theta}(s,a)\right) \mathrm{d}a\, \mathrm{d}s \tag{2.15}$$

$$\approx \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \int_{\mathscr{A}} \pi_b(a|s) \frac{\pi_\theta(a|s)}{\pi_b(a|s)} \left(Q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\right) \mathrm{d}a\, \mathrm{d}s. \tag{2.16}$$

Analogously, Silver et al. [2014] provided the following approximation for deterministic policies [2]:

$$\nabla_\theta J_b(\pi_\theta) = \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \left(\nabla_\theta \pi_\theta(s) \nabla_a Q^{\pi_\theta}(s,a)|_{a=\pi_\theta(s)} + \nabla_\theta Q^{\pi_\theta}(s,a)|_{a=\pi_\theta(s)}\right) \mathrm{d}s \tag{2.17}$$

$$\approx \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \left(\nabla_\theta \pi_\theta(s) \nabla_a Q^{\pi_\theta}(s,a)|_{a=\pi_\theta(s)}\right) \mathrm{d}s. \tag{2.18}$$

Although the term $\nabla_\theta Q^{\pi_\theta}(s,a)$ is dropped, there might be advantages in using the approximate gradient of J_b in order to find the maximum of the original RL objective J. Indeed, if we were on-policy, the approximated off-policy policy gradients [Degris et al., 2012; Silver et al., 2014] would revert to the on-policy policy gradients, while an exact gradient for J_b would necessarily introduce a bias. However, when we are off-policy, it is not clear whether this would be better than using the exact gradient of J_b in order to maximize J. In this work, we assume that J_b can be considered a good objective for off-policy RL and we derive an exact gradient for it.

[1] With tabular policies, dropping this term still results in a convergent algorithm [Degris et al., 2012].

[2] In the original formulation of Silver et al. [2014] $d_\infty^{\pi_b}(s)$ is replaced by $d^{\pi_b}(s)$.

Chapter 3

Parameter-Based Value Functions

3.1 Method

In this section, we introduce our parameter-based value functions, the PSSVF $V(\theta)$, PSVF $V(s,\theta)$, and PAVF $Q(s,a,\theta)$ and their corresponding learning algorithms. All the proofs of the theorems stated in this section can be found in Appendix A.1. First, we augment the state and action-value functions, allowing them to receive as an input also the weights of a parametric policy. The parameter-based state-value function (PSVF) $V(s,\theta) = \mathbb{E}[R_t|s_t = s,\theta]$ is defined as the expected return for being in state s and following policy parameterized by θ. Similarly, the parameter-based action-value function (PAVF) $Q(s,a,\theta) = \mathbb{E}[R_t|s_t = s, a_t = a,\theta]$ is defined as the expected return for being in state s, taking action a and following policy parameterized by θ. Using PBVFs, the RL objective becomes: $J(\pi_\theta) = \int_{\mathscr{S}} \mu_0(s) V^\pi(s,\theta)\,\mathrm{d}s$. Maximizing this objective leads to on-policy policy gradient theorems that are analogous to the traditional ones [Sutton et al., 1999; Silver et al., 2014]:

Theorem 3.1.1. *Let π_θ be stochastic. For any Markov Decision Process, the following holds:*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim d^{\pi_\theta}(s), a\sim\pi_\theta(.|s)}\left[(Q(s,a,\theta)\nabla_\theta \log \pi_\theta(a|s))\right]. \tag{3.1}$$

Theorem 3.1.2. *Let π_θ be deterministic. Under standard regularity assumptions [Silver et al., 2014], for any Markov Decision Process, the following holds:*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim d^{\pi_\theta}(s)}\left[\nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s)\right]. \tag{3.2}$$

Parameter-based value functions allow us also to learn a function of the policy parameters that directly approximates $J(\pi_\theta)$. In particular, the parameter-based start-state-value function (PSSVF) is defined as:

$$V(\theta) := \mathbb{E}_{s\sim\mu_0(s)}[V(s,\theta)] = \int_{\mathscr{S}} \mu_0(s)V(s,\theta)\,\mathrm{d}s = J(\pi_\theta). \tag{3.3}$$

Off-policy RL. In the off-policy setting, we assume that we collected data using a behavioral policy π_b. The objective to be maximized becomes:

$$J_b(\pi_{\theta^*}) = \max_\theta \int_{\mathscr{S}} d_\infty^{\pi_b}(s)V(s,\theta)\,\mathrm{d}s = \max_\theta \int_{\mathscr{S}}\int_{\mathscr{A}} d_\infty^{\pi_b}(s)\pi_\theta(a|s)Q(s,a,\theta)\,\mathrm{d}a\,\mathrm{d}s. \tag{3.4}$$

By taking the gradient of the performance J_b with respect to the policy parameters θ we obtain novel policy gradient theorems. Since θ is continuous, we need to use function approximators $V_\mathbf{w}(\theta) \approx V(\theta)$, $V_\mathbf{w}(s,\theta) \approx V(s,\theta)$ and $Q_\mathbf{w}(s,a,\theta) \approx Q(s,a,\theta)$ parametrized by $\mathbf{w} \in W \subset \mathbb{R}^{n_W}$. Compatible function approximations can be derived to ensure that the approximated value function is following the true gradient. Like in previous approaches [Sutton et al., 1999; Silver et al., 2014], this would result in linearity conditions. However, here we consider nonlinear function approximation and we leave the convergence analysis of linear PBVFs as future work. In episodic settings, we do not have access to $d_\infty^{\pi_b}$, so in the algorithm derivations and in the experiments we approximate it by sampling trajectories generated by the behavioral policy. In all cases, the policy improvement step can be very expensive, due to the computation of the $\arg\max$ over a continuous space Θ. Actor-critic methods can be derived to solve this optimization problem, where the critic (PBVFs) can be learned using Temporal Difference (TD) or Monte Carlo (MC) methods, while the actor is updated following the gradient with respect to the critic. All our algorithms make use of a replay buffer.

3.1.1 Parameter-based Start-State-Value Function $V(\theta)$

We first derive the PSSVF $V(\theta)$. Given the original performance index J, and taking the gradient with respect to θ, we obtain:

$$\nabla_\theta J(\pi_\theta) = \int_{\mathscr{S}} \mu_0(s)\nabla_\theta V(s,\theta)\,\mathrm{d}s = \mathbb{E}_{s\sim\mu_0(s)}[\nabla_\theta V(s,\theta)] = \nabla_\theta V(\theta). \tag{3.5}$$

In Algorithm 1, the critic $V_\mathbf{w}(\theta)$ with learnable parameters $\mathbf{w}$ is learned using MC to estimate the value of any policy θ. The actor is then updated following the direction

of improvement suggested by the critic. Since the main application of PSSVF is in episodic tasks, we optimize for the undiscounted objective.

Algorithm 1 Actor-critic with Monte Carlo prediction for $V(\theta)$

Input: Differentiable critic $V_{\mathbf{w}} : \Theta \to \mathbb{R}$ with parameters $\mathbf{w}$; deterministic or stochastic actor π_θ with parameters θ; empty replay buffer D
Output : Learned $V_{\mathbf{w}} \approx V(\theta) \forall \theta$, learned $\pi_\theta \approx \pi_{\theta^*}$

Initialize critic and actor weights $\mathbf{w}, \theta$
repeat:
  Generate an episode $s_0, a_0, r_1, s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T$ with policy π_θ
  Compute return $r = \sum_{k=1}^{T} r_k$
  Store (θ, r) in the replay buffer D
  for many steps **do**:
    Sample a batch $B = \{(r, \theta)\}$ from D
    Update critic by stochastic gradient descent: $\nabla_{\mathbf{w}} \mathbb{E}_{(r,\theta)\in B}[r - V_{\mathbf{w}}(\theta)]^2$
  end for
  for many steps **do**:
    Update actor by gradient ascent: $\nabla_\theta V_{\mathbf{w}}(\theta)$
  end for
until convergence

3.1.2 Parameter-based State-Value Function $V(s, \theta)$

Learning the value function using MC approaches can be difficult due to the high variance of the estimate. Furthermore, episode-based algorithms like Algorithm 1 are unable to credit good actions in bad episodes. Gradient methods based on TD updates provide a biased estimate of $V(s, \theta)$ with much lower variance and can credit actions at each time step. Taking the gradient of $J_b(\pi_\theta)$ in the PSVF formulation[1], we obtain:

$$\nabla_\theta J_b(\pi_\theta) = \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \nabla_\theta V(s, \theta) \, \mathrm{d}s = \mathbb{E}_{s \sim d_\infty^{\pi_b}(s)}[\nabla_\theta V(s, \theta)]. \tag{3.6}$$

Algorithm 2 uses the actor-critic architecture, where the critic is learned via TD[2].

[1] Compared to standard methods based on the state-value function, we can directly optimize the policy following the performance gradient of the PSVF, obtaining a policy improvement step in a model-free way.

[2] Note that the differentiability of the policy π_θ is never required in PSSVF and PSVF.

Algorithm 2 Actor-critic with TD prediction for $V(s,\theta)$

Input: Differentiable critic $V_{\mathbf{w}} : \mathscr{S} \times \Theta \to \mathbb{R}$ with parameters $\mathbf{w}$; deterministic or stochastic actor π_θ with parameters θ; empty replay buffer D
Output : Learned $V_{\mathbf{w}} \approx V(s,\theta)$, learned $\pi_\theta \approx \pi_{\theta^*}$
Initialize critic and actor weights $\mathbf{w}, \theta$
repeat:
  Observe state s, take action $a = \pi_\theta(s)$, observe reward r and next state s'
  Store (s,θ,r,s') in the replay buffer D
  if it's time to update **then**:
    for many steps **do**:
      Sample a batch $B_1 = \{(s,\theta,r,s')\}$ from D
      Update critic by stochastic gradient descent:
      $\nabla_{\mathbf{w}} \mathbb{E}_{(s,\theta,r,s') \in B_1}[V_{\mathbf{w}}(s,\theta) - (r + \gamma V_{\mathbf{w}}(s',\theta))]^2$
    end for
    for many steps **do**:
      Sample a batch $B_2 = \{(s)\}$ from D
      Update actor by stochastic gradient ascent: $\nabla_\theta \mathbb{E}_{s \in B_2}[V_{\mathbf{w}}(s,\theta)]$
    end for
  end if
until convergence

3.1.3 Parameter-based Action-Value Function $Q(s,a,\theta)$

The introduction of the PAVF $Q(s,a,\theta)$ allows us to derive new policy gradients theorems when using a stochastic or deterministic policy.

Stochastic policy gradients. We want to use data collected from some stochastic behavioral policy π_b in order to learn the action-value of a target policy π_θ. Traditional off-policy actor-critic algorithms only approximate the gradient of J_b, since they do not estimate the gradient of the action-value function with respect to the policy parameters $\nabla_\theta Q^{\pi_\theta}(s,a)$ [Degris et al., 2012; Silver et al., 2014]. With PBVFs, we can directly compute this contribution to the gradient. This yields an exact policy gradient theorem for J_b:

Theorem 3.1.3. *For any Markov Decision Process, the following holds:*

$$\nabla_\theta J_b(\pi_\theta) = \mathbb{E}_{s \sim d_\infty^{\pi_b}(s), a \sim \pi_b(.|s)} \left[\frac{\pi_\theta(a|s)}{\pi_b(a|s)} \left(Q(s,a,\theta) \nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q(s,a,\theta) \right) \right]. \tag{3.7}$$

Algorithm 3 uses an actor-critic architecture and can be seen as an extension of Off-PAC [Degris et al., 2012] to PAVF.

Algorithm 3 Stochastic actor-critic with TD prediction for $Q(s, a, \theta)$

Input: Differentiable critic $Q_{\mathbf{w}} : \mathscr{S} \times \mathscr{A} \times \Theta \rightarrow \mathbb{R}$ with parameters $\mathbf{w}$; stochastic differentiable actor π_θ with parameters θ; empty replay buffer D

Output : Learned $Q_{\mathbf{w}} \approx Q(s, a, \theta)$, learned $\pi_\theta \approx \pi_{\theta^*}$

Initialize critic and actor weights $\mathbf{w}, \theta$

repeat:

  Observe state s, take action $a = \pi_\theta(s)$, observe reward r and next state s'

  Store (s, a, θ, r, s') in the replay buffer D

  if it's time to update **then**:

    for many steps **do**:

      Sample a batch $B_1 = \{(s, a, \tilde{\theta}, r, s')\}$ from D

      Update critic by stochastic gradient descent:

      $\nabla_{\mathbf{w}} \mathbb{E}_{(s,a,\tilde{\theta},r,s') \in B_1} [Q_{\mathbf{w}}(s, a, \tilde{\theta}) - (r + \gamma Q_{\mathbf{w}}(s', a' \sim \pi_{\tilde{\theta}}(s'), \tilde{\theta}))]^2$

    end for

    for many steps **do**:

      Sample a batch $B_2 = \{(s, a, \tilde{\theta})\}$ from D

      Update actor by stochastic gradient ascent:

      $\mathbb{E}_{(s,a,\tilde{\theta}) \in B_2} \left[\frac{\pi_\theta(a|s)}{\pi_{\tilde{\theta}}(a|s)} \left(Q(s, a, \theta) \nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q(s, a, \theta) \right) \right]$

    end for

  end if

until convergence

Deterministic policy gradients. Estimating $Q(s, a, \theta)$ is in general a difficult problem due to the stochasticity of the policy. Deterministic policies of the form $\pi : \mathscr{S} \rightarrow \mathscr{A}$ can help improving the efficiency in learning value functions, since the expectation over the action space is no longer required. Using PBVFs, we can write the performance of a policy π_θ as:

$$J_b(\pi_\theta) = \int_{\mathscr{S}} d_\infty^{\pi_b}(s) V(s, \theta) \, \mathrm{d}s = \int_{\mathscr{S}} d_\infty^{\pi_b}(s) Q(s, \pi_\theta(s), \theta) \, \mathrm{d}s. \tag{3.8}$$

Taking the gradient with respect to θ we obtain a deterministic policy gradient theorem:

Theorem 3.1.4. *Under standard regularity assumptions [Silver et al., 2014], for any Markov Decision Process, the following holds:*

$$\nabla_\theta J_b(\pi_\theta) = \mathbb{E}_{s \sim d_\infty^{\pi_b}(s)} \left[\nabla_a Q(s, a, \theta)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s) + \nabla_\theta Q(s, a, \theta)|_{a=\pi_\theta(s)} \right]. \tag{3.9}$$

Algorithm 4 uses an actor-critic architecture and can be seen as an extension of DPG [Silver et al., 2014] to PAVF.

Algorithm 4 Deterministic actor-critic with TD prediction for $Q(s,a,\theta)$

Input: Differentiable critic $Q_{\mathbf{w}} : \mathscr{S} \times \mathscr{A} \times \Theta \to \mathbb{R}$ with parameters $\mathbf{w}$; differentiable deterministic actor π_θ with parameters θ; empty replay buffer D
Output : Learned $Q_{\mathbf{w}} \approx Q(s,a,\theta)$, learned $\pi_\theta \approx \pi_{\theta^*}$
Initialize critic and actor weights $\mathbf{w}, \theta$
repeat:
  Observe state s, take action $a = \pi_\theta(s)$, observe reward r and next state s'
  Store (s,a,θ,r,s') in the replay buffer D
  if it's time to update **then**:
    for many steps **do**:
      Sample a batch $B_1 = \{(s,a,\tilde{\theta},r,s')\}$ from D
      Update critic by stochastic gradient descent:
      $\nabla_{\mathbf{w}} \mathbb{E}_{(s,a,\tilde{\theta},r,s') \in B_1}[Q_{\mathbf{w}}(s,a,\tilde{\theta}) - (r + \gamma Q_{\mathbf{w}}(s', \pi_{\tilde{\theta}}(s'), \tilde{\theta}))]^2$
    end for
    for many steps **do**:
      Sample a batch $B_2 = \{(s)\}$ from D
      Update actor by stochastic gradient ascent:
      $\mathbb{E}_{s \in B_2}[\nabla_\theta \pi_\theta(s) \nabla_a Q_{\mathbf{w}}(s,a,\theta)|_{a=\pi_\theta(s)} + \nabla_\theta Q_{\mathbf{w}}(s,a,\theta)|_{a=\pi_\theta(s)}]$
    end for
  end if
until convergence

3.2 Experiments and Results

Applying algorithms 1, 2 and 4 directly can lead to convergence to local optima, due to the lack of exploration. In practice, like in standard deterministic actor-critic algorithms, we use a noisy version of the current learned policy in order to act in the environment and collect data to encourage exploration. More precisely, at each episode we use $\pi_{\tilde{\theta}}$ with $\tilde{\theta} = \theta + \varepsilon, \varepsilon \sim \mathscr{N}(0, \sigma^2 I)$ instead of π_θ and then store $\tilde{\theta}$ in the replay buffer. In our experiments, we report both for our methods as well as the baselines the performance of the policy without parameter noise.

3.2.1 Visualizing PBVFs using LQRs

We start with an illustrative example that allows us to visualize how PBVFs are learning to estimate the expected return over the parameter space. For this purpose, we use an instance of the 1D Linear Quadratic Regulator (LQR) problem and a linear deterministic policy with bias. Here, the agent observes a 1-D state, corresponding to its position and chooses a 1-D action. The transitions are $s' = s + a$ and there is a quadratic negative term for the reward: $R(s,a) = -s^2 - a^2$. The agent starts in state $s_0 = 1$ and acts in the environment for 50 time steps. The state space is bounded in [-2,2]. To maximize the sum of rewards, the agent must reach and remain in the origin. The agent is expected to perform small steps towards the origin when it uses the optimal policy. For this task, we use a deterministic policy without tanh nonlinearity and we do not use observation normalization. Appendix A.3.1 contains environment details and hyperparameters used.

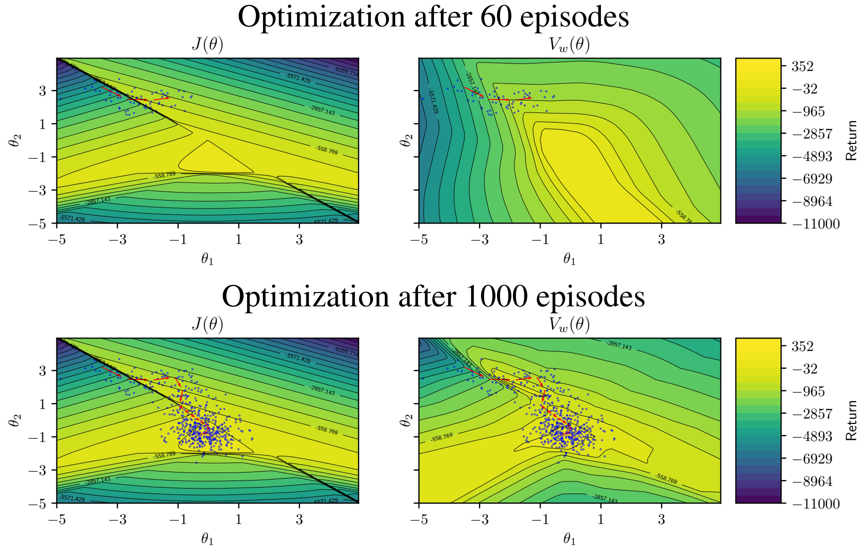


Figure 3.1: True episodic return $J(\theta)$ and PSSVF estimation $V(\theta)$ as a function of the policy parameters at two different stages in training. The red arrows represent an optimization trajectory in parameter space. The blue dots represent the perturbed policies used to train $V(\theta)$. θ_1 is the bias and θ_2 is the weight of the policy.

In figure 3.1, we plot the episodic $J(\theta)$, the cumulative undiscounted reward that an agent would obtain by acting in the environment using policy π_θ for a single episode, and the expected return predicted by the PSSVF $V(\theta)$ for two different times during learning. At the beginning of the learning process, the PSSVF is able

to provide just a local estimation of the performance of the agent, since only few data have been observed. However, after 1000 episodes, it is able to provide a more accurate global estimate over the parameter space.

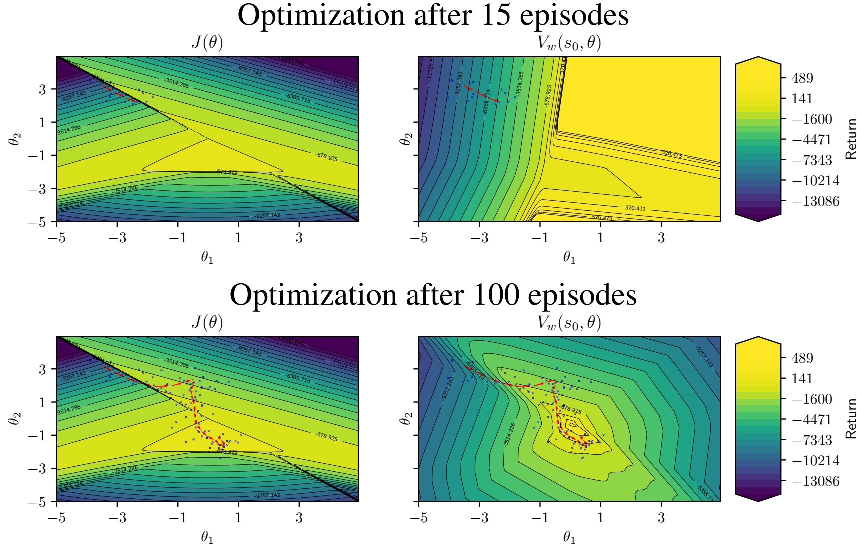


Figure 3.2: True cumulative discounted reward $J(\theta)$ and PSVF estimation $V_w(s_0, \theta)$ as a function of the policy parameters at two different stages in training. The red arrows represent an optimization trajectory in parameter space. The blue dots represent the perturbed policies used to train $V_w(s_0, \theta)$.

In Figures 3.2 and 3.3 we report $J(\theta)$, the cumulative discounted reward that an agent would obtain by acting in the environment for infinite time steps using policy π_θ and the expected return predicted by the PSVF and PAVF for two different times during learning. Like in the PSSVF experiment, the critic is able to improve its predictions over the parameter space. In the plots $V(s, \theta)$ and $Q(s, \pi_\theta(s), \theta)$ are evaluated only in s_0. The results show that PBVFs are able to effectively bootstrap the values of future states. Each red arrow in Figures 3.2 and 3.3 represents 50 update steps of the policy.

3.2.2 Main results

Given the similarities between our PAVF and DPG, Deep Deterministic Policy Gradients (DDPG) [Lillicrap et al., 2015] is a natural choice for the baseline. Additionally, the PSSVF $V(\theta)$ resembles evolutionary methods as the critic can be interpreted as a global fitness function. Therefore, we include in the comparison Augmented

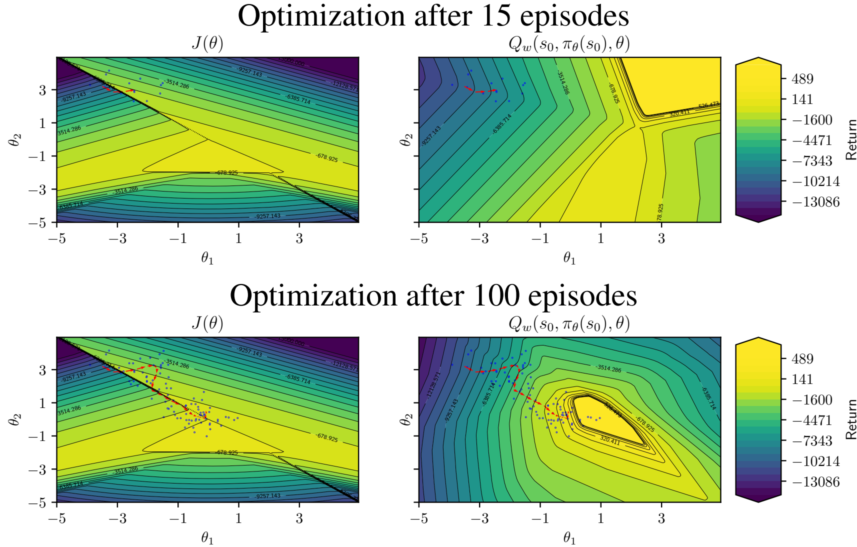


Figure 3.3: True cumulative discounted reward $J(\theta)$ and PAVF estimation $Q_w(s_0, \pi_\theta(s_0), \theta)$ as a function of the policy parameters at two different stages in training. The red arrows represent an optimization trajectory in parameter space. The blue dots represent the perturbed policies used to train $Q_w(s_0, \pi_\theta(s_0), \theta)$.

Random Search (ARS) [Mania et al., 2018], which is known for its state-of-the-art performance using only linear policies in continuous control tasks. For the policy, we use a 2-layer MLP (64,64) with tanh activations and a shallow policy composed of a linear layer followed by a tanh nonlinearity. Appendix A.3.2 details the methodology for identifying the optimal hyperparameters for the main experiments, whereas Appendix A.3.4 presents the best hyperparameters that were identified. Extended details about the implementation, as well as ablation studies on several critical implementation and optimization choices, can be found in Appendix A.2. We compare our methods with the baselines by measuring the expected return achieved over a fixed number of interactions in various continuous control environments.

Deterministic policies Figure 3.4 shows results for deterministic policies with both architectures. In all the tasks the PSSVF is able to achieve at least the same performance as ARS, often outperforming it. In the Inverted Pendulum environment, PSVF and PAVF with deep policy are very slow to converge, but they excel in the Swimmer task and MountainCarContinuous. In Reacher, all PBVFs fail to learn the task, while DDPG converges quickly to the optimal policy. We conjecture that for this task it is difficult to perform a search in parameter space. On the other hand, in

MountainCarContinuous, the reward is more sparse and DDPG only rarely observes positive reward when exploring in action space. We analyze the sensitivity of the algorithms on the choice of hyperparameters in Appendix A.3.3.

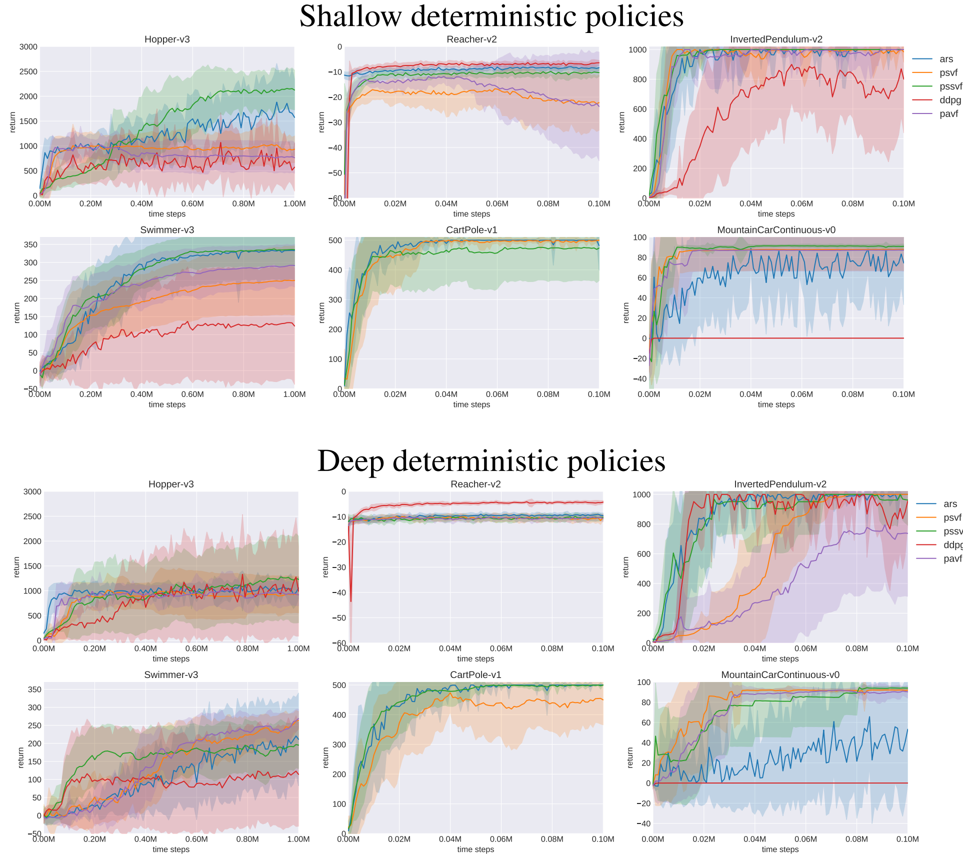


Figure 3.4: Average return of shallow and deep deterministic policies as a function of the number of time steps used for learning (across 20 runs, one standard deviation), for different environments and algorithms. We use the best hyperparameters found when maximizing the average return.

Stochastic policies We include some results for stochastic policies when using PSSVF and PSVF. Figure 3.5 shows a comparison with the baselines when using shallow and deep policies respectively. We observe results sometimes comparable, but often inferior with respect to deterministic policies. In particular, when using shallow policies, PBVFs are able to outperform the baselines in the MountainCar environment, while obtaining comparable performance in CartPole and InvertedPendulum. Like in previous experiments, PBVFs fail to learn a good policy in Reacher. When using

deep policies, the results are slightly different: PBVFs outperform ARS and DDPG in Swimmer, but fail to learn InvertedPendulum. Although the use of stochastic policies can help smoothing the objective function and allows the agent exploring in action space, we believe that the lower variance of the value function estimate provided by deterministic policies can facilitate learning PBVFs.

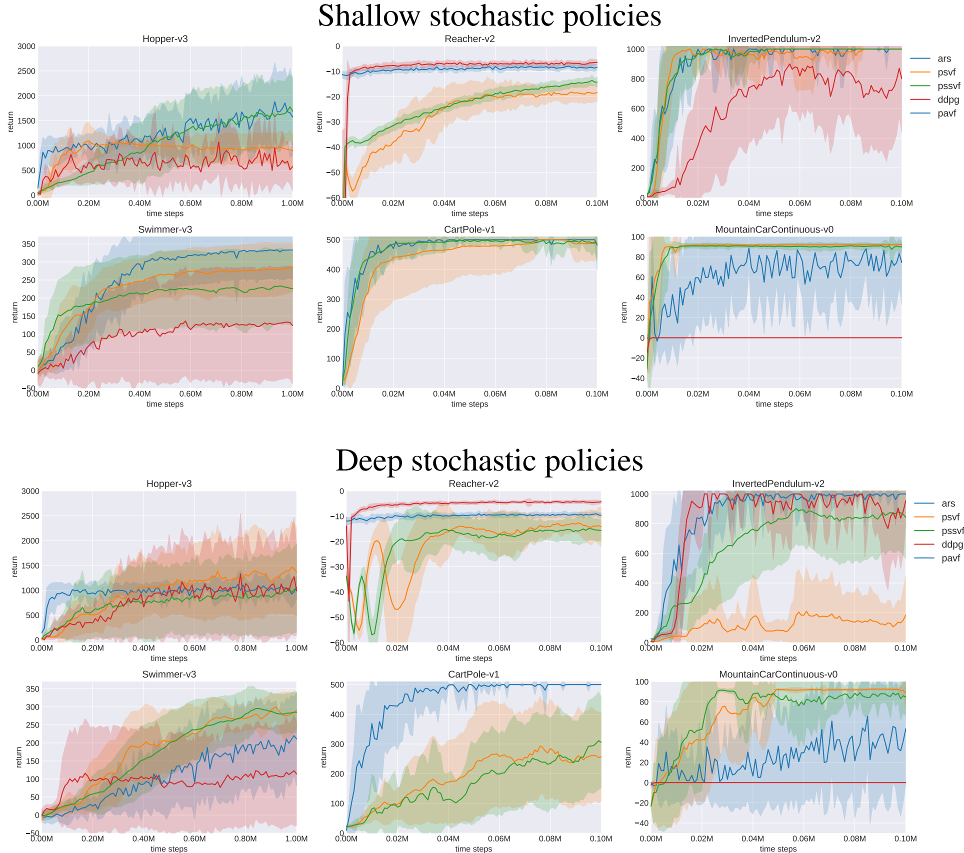


Figure 3.5: Average return of shallow and deep stochastic policies as a function of the number of time steps used for learning (across 20 runs, one standard deviation), for different environments and algorithms. We use the best hyperparameters found when maximizing the average return.

3.2.3 Zero-shot learning

Shallow policies on Swimmer In order to test whether PBVFs are generalizing across the policy space, we perform the following experiment with shallow deterministic policies: while learning using Algorithm 1, we stop training and randomly initialize 5 policies. Then, without interacting with the environment, we train these

policies offline, in a zero-shot manner, following only the direction of improvement suggested by $\nabla_\theta V_w(\theta)$, whose weights w remain frozen. We observe that shallow policies can be effectively zero-shot trained. Results for PSSVFs in Swimmer-v3 are displayed in Figure 3.6. In particular, we compare the performance of the policy learned, the best perturbed policy for exploration seen during training and five policies zero-shot learned at three different stages in training. We note that after the PSSVF has been trained for 100,000 time steps interactions with the environment (first snapshot), these policies are already able to outperform both the current policy and any policy seen while training the PSSVF. They achieve an average return of 297, while the best observed return is 225. We evaluate the performance of the policies zero-shot learned evaluating them with 5 test trajectories every 5 gradient steps. For this task, we use the same hyperparameters as in Figure 3.4.

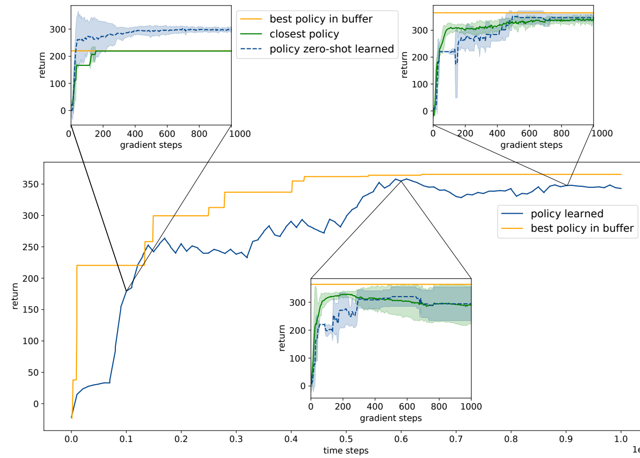


Figure 3.6: Policies zero-shot learned during training. The plot in the center represents the return of the agent learning while interacting with the environment using Algorithm 1. We compare the best noisy policy $\pi_{\tilde{\theta}}$ used for exploration to the policy π_θ learned through the critic. The learning curves in the small plots represent the return obtained by policies zero-shot trained following the fixed critic $V_{\mathbf{w}}(\theta)$ after different time steps of training. The return of the closest policy (L2 distance) in the replay buffer with respect to the policy zero-shot learned is depicted in green.

Extended results with deep policies We report in Figures 3.7 and 3.8 a comparison of zero-shot performance between PSSVF, PSVF and PAVF in three different environments using deterministic shallow and deep policies (2-layers MLP(64,64)). In this task we use the same hyperparameters found in tables A.4, A.6 and A.8. In

Figure 3.6, we use a tuned learning rate of 0.02 for policies zero-shot trained. In the additional experiments in Figures 3.7 and 3.8, we use a learning rate of 0.05 across all policies, environments and algorithms when learning zero-shot.

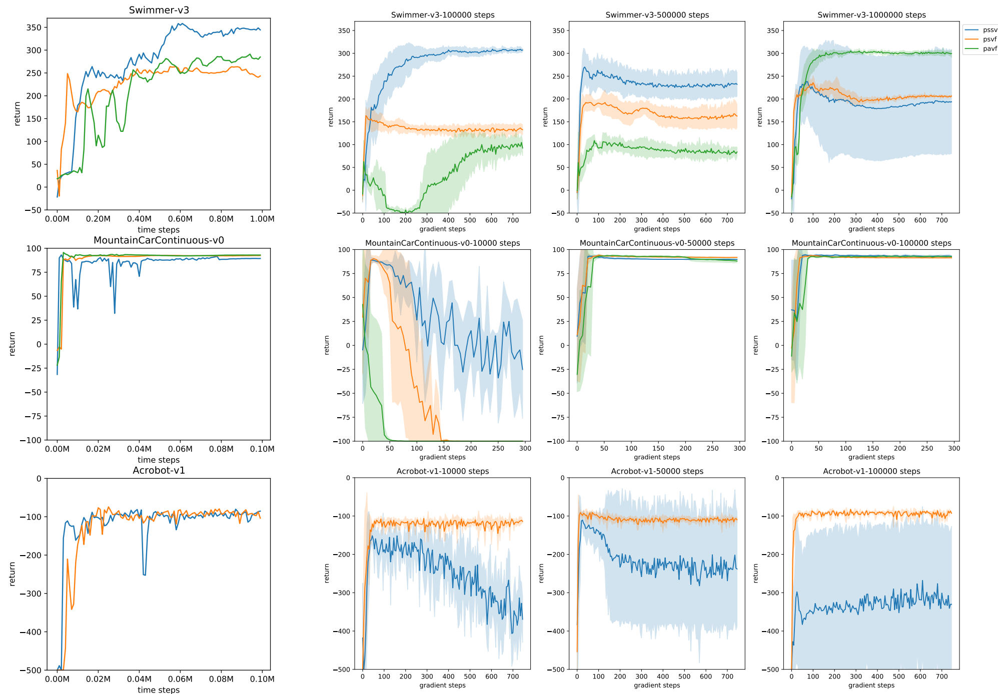


Figure 3.7: Shallow policies zero-shot learned during training. The plots in the left column represent the return of agents learning while interacting with the environment using different algorithms. The learning curves in the other plots represent the return obtained by policies zero-shot trained following the fixed critics after different time steps of training. Zero-shot learning curves are averaged over 5 seeds.

We observe that, using shallow policies, PBVFs can effectively zero-shot learn policies with performance comparable to the policy learned in the environment without additional tuning for the learning rate. We note the regular presence of a spike in performance followed by a decline due to the policy going to regions of the parameter space never observed. This suggests that there is a trade-off between exploiting the generalization of the critic and remaining in the part of the parameter space where the critic is accurate. Measuring the width of these spikes can be useful for determining the number of offline gradient steps to perform in the general algorithm. When using deep policies the results become much worse and zero-shot learned policies can recover the performance of the main policy being learned only in simple environments and at beginning of training (e.g., MountainCarContinuous). We observe that, when the critic is trained (last column), the replay buffer contains

policies that are very distant to policies randomly initialized. This might explain why the zero-shot performance is better sometimes at the beginning of training (e.g., second column in Figure 3.8). However, since PBVFs in practice perform mostly local off-policy evaluation around the learned policy, this problem is less prone to arise in our main experiments.

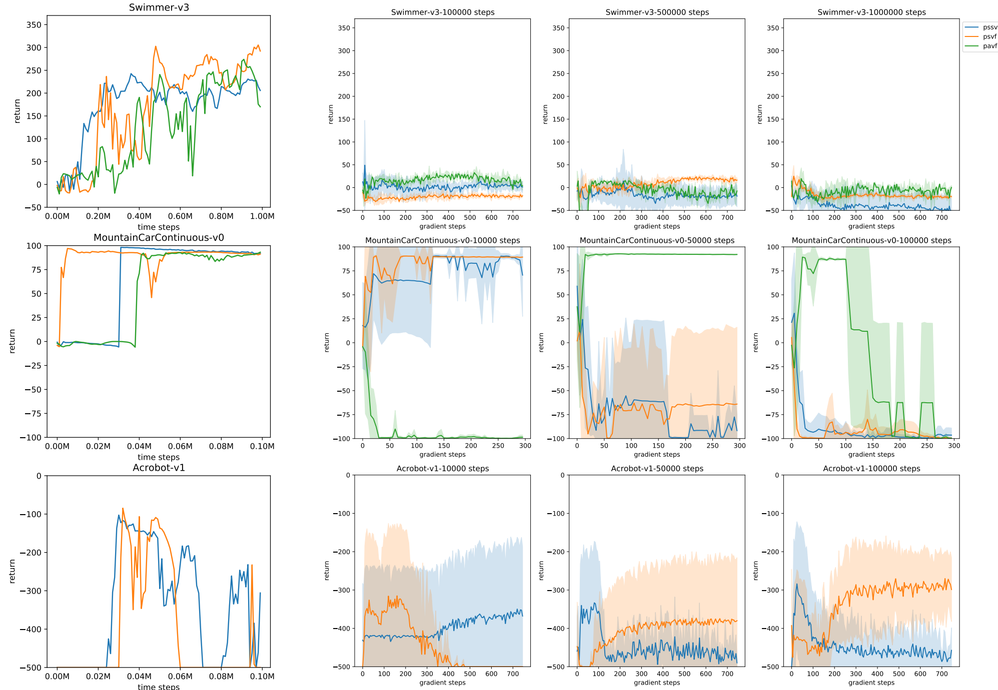


Figure 3.8: Deep policies zero-shot learned during training. The plots in the left column represent the return of agents learning while interacting with the environment using different algorithms. The learning curves in the other plots represent the return obtained by policies zero-shot trained following the fixed critics after different time steps of training. Zero-shot learning curves are averaged over 5 seeds.

3.2.4 Offline learning with fragmented behaviors

In our last experiment, we investigate how PSVFs are able to learn in a completely offline setting. The goal is to learn a good policy in Swimmer-v3 given a fixed dataset containing 100,000 transitions, without additional environment interactions. Furthermore, the policy generating the data is perturbed every 200 time steps, for a total of 5 policies per episode. Observing only incomplete trajectories for each policy parameter makes TD bootstrapping harder: in order to learn a good policy, the PSVF needs to generalize across both the state and the parameter space.

Given the fixed dataset, we first train the PSVF, minimizing the TD error. Then, at different stages during learning, we zero-shot train 5 new shallow deterministic policies, following only the gradient of the PSVF. Figure 3.9 describes this process. In this task, data are generated by perturbing a randomly initialized deterministic policy every 200 time steps and using it to act in the environment. We use $\sigma = 0.5$ for the perturbations. After the dataset is collected, the PSVF is trained using a learning rate of $1e-3$ with a batch size of 128. We use a learning rate of 0.02 for learning the policy. All other hyperparameters are set to default values. We note that at the beginning of training, when the PSVF $V(s,\theta)$ has a larger TD error, the policies learned have poor performance. However, after 7000 gradient updates, they are able to achieve a reward of 237, before eventually degrading to 167. They outperform the best policy in the dataset used to train the PSVF, whose return is only of 58.

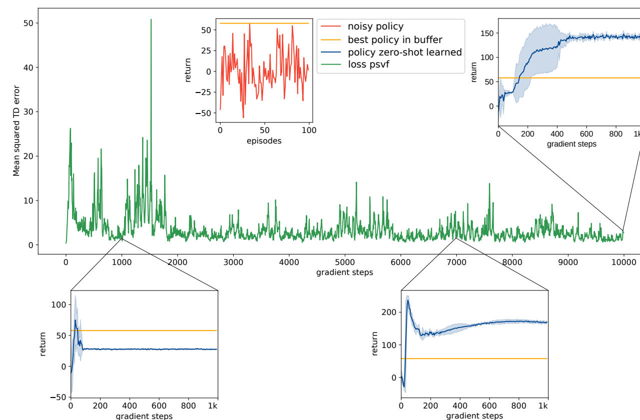


Figure 3.9: Offline learning of PSVF. We plot the mean squared TD error of a PSVF trained using data coming from a set of noisy policies. In the small plots, we compare the return obtained by policies zero-shot trained following the fixed critic $V_{\mathbf{w}}(s,\theta)$ after different time steps of value function training and the return of the best noisy policy used to train V.

3.3 Related Work

Parameter-based Search. There is a long history of RL algorithms performing direct search in parameter space or policy space. The most common approaches include evolution strategies, e.g., [Rechenberg, 1971; Sehnke et al., 2010, 2008; Wierstra et al., 2014; Salimans et al., 2017]. They iteratively simulate a population of policies and use the result to estimate a direction of improvement in parameter space.

Evolution strategies, however, don't reuse data: the information contained in the population is lost as soon as an update is performed, making them sample-inefficient. Several attempts have been made to reuse past data, often involving importance sampling (IS) [Zhao et al., 2013], but these methods suffer from high variance of the fitness estimator [Metelli et al., 2018]. Our method directly estimates a fitness for each policy observed in the history and makes efficient reuse of past data without involving IS. Direct search can be facilitated by compressed network search [Koutnik et al., 2010] and algorithms that distill the knowledge of an NN into another NN [Schmidhuber, 1992a]. Estimating a global objective function is common in control theory, where usually a gaussian process is maintained over the policy parameters. This allows to perform direct policy optimization during the parameter search. Such approaches are often used in the Bayesian optimization framework [Snoek et al., 2015, 2012], where a tractable posterior over the parameter space is used to drive policy improvements. Despite the soundness of these approaches, they usually employ very small control policies and scale badly with the dimension of the policy parameters.

Off-Policy Learning. Gradient Temporal Difference [Sutton et al., 2009a,b; Maei et al., 2009, 2010; Maei, 2011] and Emphatic Temporal Difference methods [Sutton et al., 2016] were developed to address convergence under on-policy and off-policy [Precup et al., 2001] learning with function approximation. The first attempt to obtain a stable off-policy actor-critic algorithm under linear function approximation was called Off-PAC [Degris et al., 2012], where the critic is updated using GTD(λ) [Maei, 2011] to estimate the state-value function. This algorithm converges when using tabular policies. However, in general, the actor does not follow the true gradient direction for J_b, the off-policy objective we defined in Chapter 2.3. A paper on DPG [Silver et al., 2014] extended the Off-PAC policy gradient theorem [Degris et al., 2012] to deterministic policies. This was coupled with a deep neural network to solve continuous control tasks through Deep Deterministic Policy Gradients [Lillicrap et al., 2015]. Imani et al. [2018] used emphatic weights to derive an exact off-policy policy gradient theorem for J_b. Differently from Off-PAC, they do not ignore the gradient of the action-value function with respect to the policy, which is incorporated in the emphatic weighting: a vector that needs to be estimated. Our off-policy policy gradients provide an alternative approach that does not need emphatic weights. The widely used off-policy objective function J_b suffers the distribution shift problem. Liu et al. [2019] provided an off-policy policy gradient theorem which is unbiased for the true RL objective $J(\pi_\theta)$. This theorem introduces a term $d_\infty^{\pi_\theta}/d_\infty^{\pi_b}$, which corrects the mismatch between the limiting distributions of states under the target and behavioral policies. Despite their sound off-policy

formulation, estimating the state weighting ratio remains challenging. All our algorithms are based on the off-policy actor-critic architecture. The two algorithms based on $Q(s,a,\theta)$ can be viewed as analogous to Off-PAC and DPG where the critic is defined for all policies and the actor is updated following the true gradient with respect to the critic.

3.4 Discussion

We introduced PBVFs, a novel class of value functions that receive the parameters of a policy as input and can be used for off-policy learning. We empirically demonstrated that PBVFs are competitive with ARS and DDPG [Mania et al., 2018; Lillicrap et al., 2015], generalizing across policies and enabling zero-shot training in an offline setting. While our PSSVF simply maps policy parameters to their expected return, the PSVF and PAVF algorithms can assign credit for each state (or state-action pair), effectively generalizing over both state and parameter spaces.

Despite their positive results with both shallow and deep policies, PBVFs suffer from the curse of dimensionality when the number of policy parameters is high. To scale up to deeper policies and tackle more challenging environments, it is necessary to design suitable policy representations to input into the value function. A crucial aspect of these representations is that they must be differentiable with respect to the policy parameters, enabling policy improvement through simple backpropagation through the PBVF to find a better policy. Such embeddings can save not only memory and computational time but also facilitate the search in the parameter space. We will investigate a class of such embeddings in the next chapter.

Chapter 4

General Policy Evaluation and Improvement by Learning to Identify Few But Crucial States

4.1 Method

4.1.1 Static Policy fingerprinting

While the algorithms described in the previous sections are straightforward and easy to implement, feeding the policy parameters as inputs to the value function remains a challenge. Recently Harb et al. [2020] showed that a form of policy embedding can be suitable for this task. Their *policy fingerprinting* creates a lower-dimensional policy representation. It learns a set of K 'probing states' $\{\tilde{s}_k\}_{k=1}^{K}$ and an evaluation function U—like the PSSVF. To evaluate a policy π_θ, they first compute the 'probing actions' $\tilde{a}_k$ that the policy produces in the probing states. Then the concatenated vector of these actions is given as input to $U : \mathbb{R}^{K \times n_A} \to \mathbb{R}$. While the learned probing states remain fixed when evaluating multiple policies, the probing actions in such states depend on the policy we are evaluating. The parameters of the value function V are the probing states AND the weights of the MLP U_ϕ parametrized by $\phi \in \Phi \subset \mathbb{R}^{n_\Phi}$ that maps the 'probing actions' to the return. When the policy π_θ is deterministic, the probing actions for such policy are the deterministic actions $\{\tilde{a}_k = \pi_\theta(\tilde{s}_k)\}$ produced in the probing states [1].

This mechanism has an intuitive interpretation: to evaluate the behavior of

[1] If the policy is stochastic, the probing actions are the parameters of the output distribution of the policy in such states (the vector of probability distribution if the action space is discrete)

an agent, the PSSVF with policy fingerprinting learns a set of situations (or states), observes how the agent acts in those situations, and then maps the agent's actions to a score. Arguably, this is also how a teacher would evaluate multiple different students by simultaneously learning which questions to ask the students and how to score the student's answers. Therefore the parameters of the value function (probing states and evaluator function) can be learned by minimizing MSE loss $\mathscr{L}_V$ between the prediction of the value function and the observed return. Setting $w = \{\phi, \tilde{s}_1, \dots \tilde{s}_K\}$, we retrieve the common notation of $V_w(\theta)$ for the PSSVF with fingerprint mechanism. Given a batch B of data $(\pi_\theta, r) \in B$, the value function optimization problem is:

$$\min_w \mathscr{L}_V := \min_w \mathop{\mathbb{E}}_{(\pi_\theta, r)\in B}[(V_w(\theta) - r)^2] = \min_{\phi, \tilde{s}_1, \dots \tilde{s}_K} \mathop{\mathbb{E}}_{(\pi_\theta, r)\in B}[(U_\phi([\pi_\theta(\tilde{s}_1), \dots, \pi_\theta(\tilde{s}_K)]) - r)^2]. \tag{4.1}$$

If the prediction of the value function is accurate, policy improvement can be achieved by changing the way a policy acts in the learned probing states in order to maximize the prediction of the value function, like in the original PSSVF. This process connects to the same interpretation as before: a student (the policy) observes which questions the teacher asks and how the teacher evaluates the student's answers, and subsequently tries to improve in such a way to maximize the score predicted by the teacher. This iterative method is depicted in Figure 4.1.

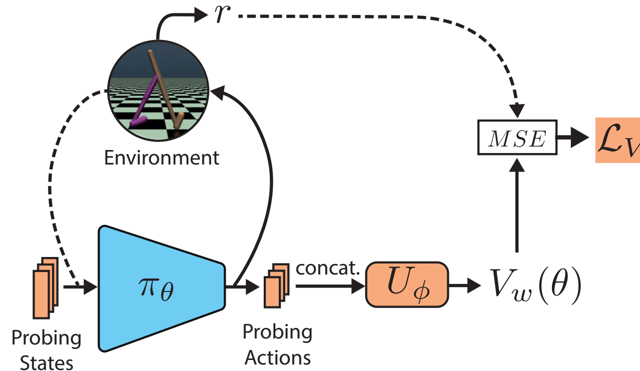


Figure 4.1: General policy evaluation aims to evaluate any given policy's return based on the policy's actions (referred to as probing actions) in the learned probing states. The policy can be improved through maximising the prediction of the learned value function via gradient ascent.

While Harb et al. applied this technique to the offline setting, we can easily incorporate it into the online setting when the value function and the policy are

learned iteratively. Note that Algorithm1 applies directly to this setting. The only distinction is that the probing states are part of the learned value function. Throughout this chapter, with the exception of the MNIST experiments, we consider deterministic policies.

Static Policy fingerprinting presents some significant limitations. First, the set of probing states that is learned is fixed; therefore, the same set of states is used to evaluate different policies. In practice, good policies might observe states whose distribution is very different from that of bad policies. Second, if the number of probing states is large, the vector of concatenated probing actions will have high dimensionality. This will cause the number of parameters in U_ϕ to grow linearly with the number of probing states. To overcome this issue, we introduce a new method that recurrently generates probing states.

4.1.2 Recurrent Policy Fingerprinting

We change the architecture of our evaluator function U_ϕ to be a recurrent neural network (RNN). To evaluate a policy π_θ, our RNN iteratively generates a probing state $\tilde{s}$, which is given as input to π_θ. The resulting probing action $\tilde{a}$ is fed back into the RNN, which again generates a new probing state, and so on. The RNN U_ϕ is also trained to predict the expected return $\hat{r}$. Formally, during the forward pass of the RNN, we have $\tilde{s}, \hat{r}, h = U_\phi(\tilde{a}, h)$, where h represents the hidden state of the RNN. Here, $\tilde{s} = f_{\phi_1}(h)$ and $\hat{r} = g_{\phi_2}(h)$, where f and g are MLPs with parameters $\phi_1, \phi_2 \subset \phi$. The probing action is simply determined by running a forward pass of the policy, i.e., $\tilde{a} = \pi_\theta(\tilde{s})$. The initial probing state is generated using the initial hidden state h_0, as $\tilde{s_0} = f_{\phi_1}(h_0)$. After unrolling the RNN of K steps, which involves generating K probing states based on the previous probing states and probing actions, the RNN outputs a prediction of the expected return $\hat{r}$. One can consider either the final return prediction or a weighted average of all return predictions (one for each step of the RNN) in the loss function to facilitate credit assignment.

We can retrieve the same notation as before if we call V_w the unrolled network U_ϕ. In this case, w contains the parameters ϕ and the initial hidden state h_0, and $\hat{r} = V_w(\theta)$ is the return predicted after unrolling U_ϕ for K steps. The policy can then be improved following Algorithm1.

This mechanism can be intuitively interpreted with a connection to the example of standard fingerprinting. It simulates how a teacher would learn to evaluate multiple students in an oral exam. The teacher iteratively asks new questions based on the previous student's answers, aiming to improve their ability to predict the

student's final score. In this analogy, the probing states generated by the RNN can be seen as the questions posed by the teacher. On the other hand, the student (policy) observes this process and seeks to modify their answers (actions) in order to maximize the score predicted by the teacher.

Recurrent policy fingerprinting is experimentally analyzed in Chapter 5, focusing on learning representations of Recurrent Neural Network Weight Matrices. Future work will concentrate on applying recurrent policy fingerprinting in RL tasks. The next section will exclusively focus on experiments with PSSVF using static fingerprinting.

4.2 Experiments and Results

This section presents an empirical study of parameter-based value functions (PBVFs) with static fingerprinting. We begin with a demonstration that fingerprinting can learn interesting states in MNIST purely through the designated evaluation task of mapping randomly initialized Convolutional Neural Networks (CNNs) to their expected loss. We also show that such a procedure could be used to construct a value function for offline improvement in MNIST. Next, we proceed to our main experiments on continuous control tasks in MuJoCo [Todorov et al., 2012]. Here we show that our approach is competitive with strong baselines like DDPG [Lillicrap et al., 2015] and ARS [Mania et al., 2018], while it lacks sample efficiency when compared to SAC [Haarnoja et al., 2018b]. A strength of our approach is invariance to policy architecture. To illustrate this, we provide results on zero-shot learning of new policy architectures. Thereafter, we present a detailed analysis of the learned probing states in various MuJoCo environments. We conclude our study with the surprising observation that very few probing states are required to clone near-optimal behaviour in certain MuJoCo environments.

4.2.1 Motivating experiments on MNIST

We begin our experimental section with an intuitive demonstration of how PBVFs with fingerprinting work, using the MNIST digit classification problem. The policy is a CNN, mapping images to a probability distribution over digit classes. The environment simulation consists of running a forward pass of the CNN on a batch of data and receiving the reward, which in this case is the negative cross-entropy between the output of the CNN and the labels of the data. The value function learns to map CNN parameters to the reward (the negative loss) obtained during the simulation.

Then the CNN learns to improve itself only by following the prediction of the value function, without access to the supervised learning loss. These MNIST experiments can be considered as a contextual bandit problem, where the initial state (or context) is given by the batch of training data sampled and there are no transition dynamics. We start with a randomly initialized CNN and value function and iteratively update them following the PSSVF Algorithm 1. Using only 10 probing states, we obtain a test set accuracy of 82.5%. When increasing the number of probing states to 50, the accuracy increases to 87%.

We use the hyperparameters described in Appendix B.1.1. Figure 4.2 shows the performance of PSSVF using CNNs on MNIST with 10 and 50 probing states as a function of the number of interactions with the dataset. Each interaction consists of perturbing the current policy with random noise, computing the loss of the perturbed policy on a batch of data, storing the perturbed policy and its loss, and updating.

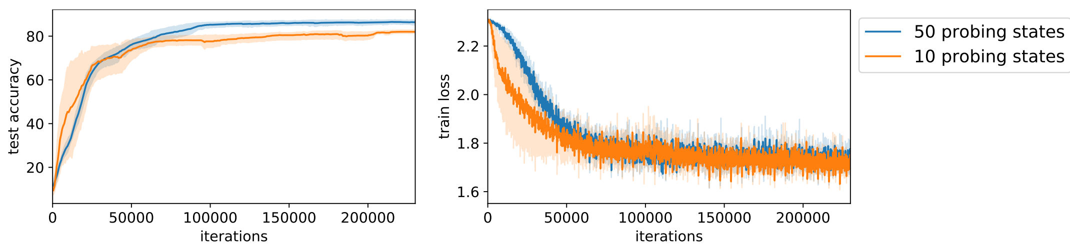


Figure 4.2: On the left: test accuracy of PSSVF as a function of the interactions with the dataset. On the right: loss of the perturbed CNN on the training set. Average over 5 independent runs and 95% bootstrapped confidence interval.

Visualization of probing states Figure 4.3 shows some of the probing states learned by our model, starting from random noise. During learning, we observe the appearance of various digits (sometimes the same digit in different shapes). Since probing states are states in which the action of the policy is informative about its global behavior, it is intuitive that digits should appear. We emphasize that both the CNNs and the value function are starting from random initializations. The convolutional filters and the probing states are learned using Algorithm 1, without access to the supervised loss. For more complex datasets like CIFAR10 our method found it difficult to learn meaningful probing states. This is possibly due to the high variance in the training data given a specific class and highlights a limitation of our method.

Figure 4.3: Samples of probing states learned while training Algorithm 1 on MNIST.

We plot the evolution of some of the probing states, starting from random noise, until the PSSVF is learned. We consider one run of the previous experiment with 10 probing states and show how they change during learning. This is depicted in Figure 4.4 where randomly initialized probing states slowly become similar to digits.

Offline policy improvement Using this setting, we perform another experiment to evaluate the ability of the PSSVF to generalize when using bad data. We collect one offline dataset $\{\pi_{\theta_i}, l_i\}_{i=1}^{N}$ of N randomly initialized CNN policies and their losses. We constrain the maximum accuracy of these CNNs in the training set to be 12%. Here every iteration encompasses the following steps. We perturb a randomly initialized CNN with gaussian noise with standard deviation 0.1. Then we compute the loss on a batch of 1024 training data. If the accuracy on such batch is below 12%, we store the CNN and its loss, otherwise we discard the data. At every iteration we also train a PSSVF with 200 probing states, using the data collected (whose accuracy is at most 12%). We repeat this for 90000 iterations. Then, we randomly initialize a new CNN and train it by taking gradient steps through the fixed PSSVF, without further seeing training data. In Figure 4.5 we plot the performance of the zero-shot learned CNN. Surprisingly, it achieves a test accuracy of 65%, although only CNNs with at most 12% accuracy are used in training. From the same figure we also observe that the prediction of the PSSVF is quite accurate up to 80 gradient steps, after which the performance degrades. We use a learning rate of $1e-3$ for the CNN.

Visualization of learned probing states When training the PSSVF using CNNs whose accuracy is at most 12%, we also observe the formation of "numbers"

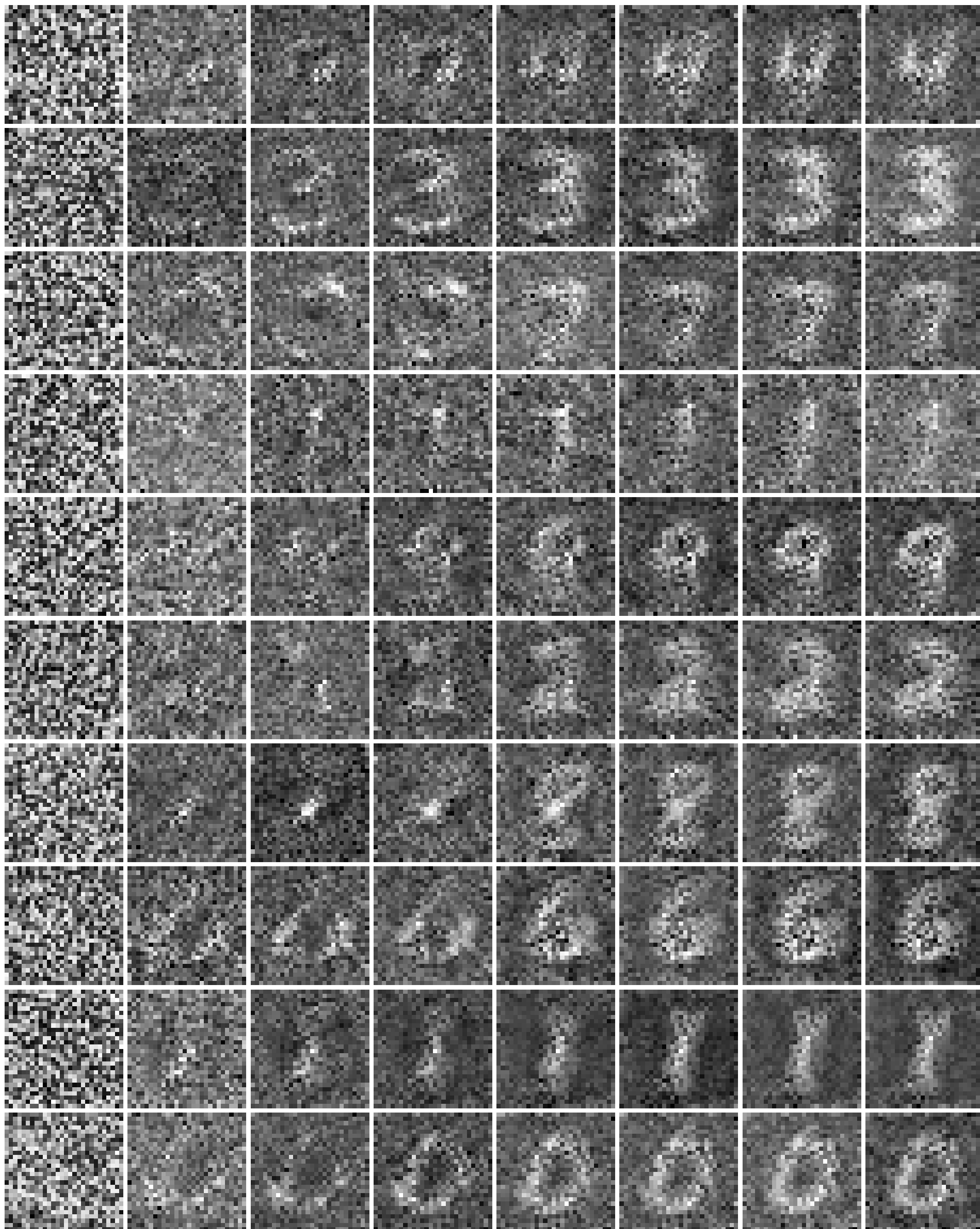


Figure 4.4: From left to right, the 10 probing states learned by the PSSVF using Algorithm 1. Each column represents 12500 interactions.

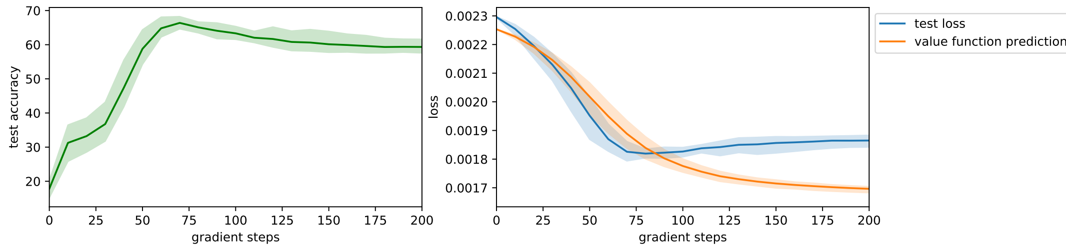


Figure 4.5: On the left: test accuracy of a random initialized CNN zero-shot learned using a learned PSSVF. On the right, the prediction of the performance of the CNN given by the PSSVF and the true performance on the test set. Average over 5 independent runs and 95% bootstrapped c.i.

as probing states, although they are not as evident as in the online setting. We provide some examples in Figure 4.6.

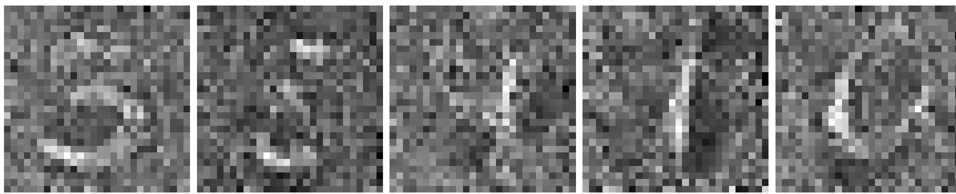


Figure 4.6: Samples of probing states learned by the PSSVF using CNNs with at most 12% training set accuracy.

4.2.2 Main experiments on MuJoCo

Here we present our main evaluation on selected continuous control problems from MuJoCo [Todorov et al., 2012]. Like in the previous chapter, since our algorithm performs direct search in parameter space, we choose *Augmented Random Search (ARS)* [Mania et al., 2018] as baseline for comparison. Moreover, since our algorithm employs deterministic policies, off-policy data, and an actor-critic architecture, a natural competitor is the *Deep Deterministic Policy Gradient (DDPG)* algorithm [Lillicrap et al., 2015], a strong baseline for continuous control. We also compare our method with the state-of-the-art *Soft Actor-Critic (SAC)* [Haarnoja et al., 2018b].

Implementation details For the policy architecture, we use an MLP with 2 hidden layers and 256 neurons for each layer. We use 200 probing states and later

provide an analysis of them. In some MuJoCo environments like Hopper and Walker, a bad agent can fail and the episode ends after very few time steps. This results in an excessive number of bad policies in the replay buffer, which can bias learning. Indeed, by the time a good policy is observed, it becomes difficult to use it for training when uniformly sampling experience from the replay buffer. We find that by prioritizing more recent data we are able to achieve a more uniform distribution over the buffer and increase the sample efficiency. We provide an ablation in Appendix B.2.2, showing the contribution of this component and of policy fingerprinting. Like in the previous chapter, we use observation normalization and remove the survival bonus for the reward. The survival bonus, which provides reward 1 at each time step for remaining alive in Hopper, Walker and Ant, induces a challenging local optimum in parameter space where the agent would learn to keep still.

For DDPG and SAC, we use the default hyperparameters, yielding results on par with the best reported results for the method. For ARS, we tune for each environment step size, number of population and noise. For our method, we use a fixed set of hyperparameters, with the only exception of Ant. In Ant, we observe that setting the parameter noise for perturbations to 0.05 results in very rare positive returns for ARS and PSSVF (after subtracting the survival bonus). Therefore we use less noise in this environment. We discuss implementation details and hyperparameters in Appendices B.1.2 and B.2.1.

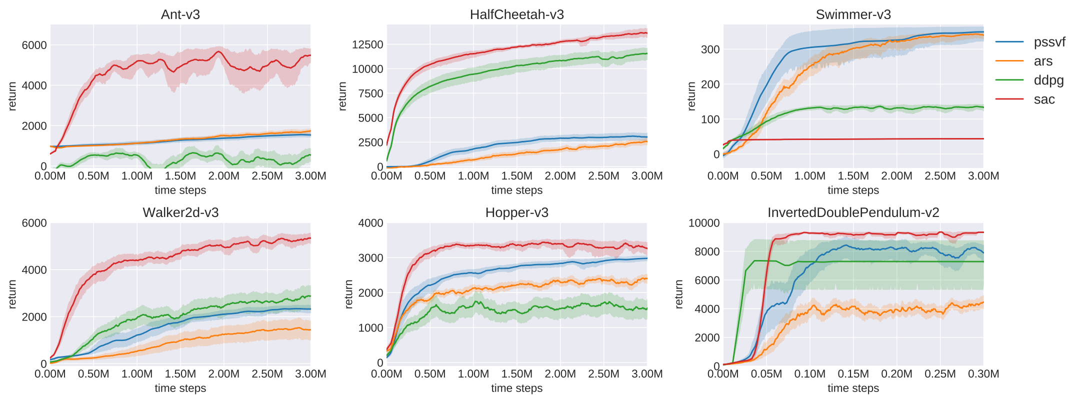


Figure 4.7: Return as a function of the environment interactions. The solid curve represents the mean (across 20 runs), and the shaded region represents a 95% bootstrapped confidence interval.

Results Figure 4.7 shows learning curves in terms of expected return (mean and 95% confidence interval) achieved by our algorithm and the baselines across time in the environments. Our algorithm is very competitive with DDPG and ARS. It outperforms DDPG in all environments with the exception of HalfCheetah and Walker, and displays faster initial learning than ARS. In the Swimmer environment, DDPG and SAC fails to learn an optimal policy due to the problem of discounting[2]. On the other hand, in HalfCheetah, parameter-based methods take a long time to improve, and the ability of DDPG to give credit to sub-episodes is crucial here to learn quickly. Furthermore, the variance of our method's performance is less than DDPG's and comparable to ARS's. Like evolutionary approaches, our method uses only the return as learning data, while ignoring what happens in each state-action pairs. This is a limitation of our method and it is evident how PSSVF and ARS are less sample efficient in comparison to SAC in many environments.

Learning with $V(s_0,\theta)$ In preliminary experiments we tried to learn also a function $V(s_0,\theta)$, incorporating the information on the initial state. In practice, we can store in the buffer tuples (s_0,θ,r) consisting of initial state, policy parameters and episodic return. When training the PSSVF (now similar to the PSVF), we concatenate the initial state to the probing actions and map the vector of probing actions and initial state to the return. Then policy improvement is achieved by finding the policy parameters that maximize the value function's prediction taking an expectation over the initial states sampled from the buffer. The results were very similar to those we presented in this section, so we decided to use the more straightforward approach that ignores the initial state and directly maps policy parameters to the expected return.

Comparison to vanilla PSSVF A direct comparison to the standard Parameter-Based Value function is unfeasible for large NNs. This is because in the vanilla PSSVF, flattened policy parameters are directly fed to the value function. In our policy configuration, the flattened vector of policy parameters contains about 70K elements, which is significantly more than $200 \times n_A$ elements used to represent policies with fingerprinting. Nevertheless, we provide a direct comparison between the two approaches using a smaller policy architecture which consists of an MLP with 2 hidden layers and 64 neurons per layer. The complete results are provided in Appendix B.2.3. Our results in this setting show that the fingerprint mechanism could be useful even for smaller policies.

[2]This is a common problem for Temporal Difference methods: the policy optimizing expected return in Swimmer with $\gamma = 0.99$ is sub-optimal when considering the expected return with $\gamma = 1$. See the ablation in Figure A.4

4.2.3 Zero-shot learning of new policy architectures

Here we show that our method can generalize across policy architectures. We train a PSSVF using NN policies as in the main experiments. Then we randomly initialize a linear policy and start taking gradient ascent steps through the fixed value function, finding the parameters of the policy that maximizes the value function's prediction. For this task, we use the same hyperparameters as in the main experiments (see Appendix B.1.2). We use a learning rate of $1e-4$ to zero-shot learn the linear policy. In Figure 4.8 we observe that a near-optimal linear policy can be zero-shot-learned through the value function even if it was trained using policies with different architecture. It achieves an expected return of 345, while the return of best NN used for training was 360.

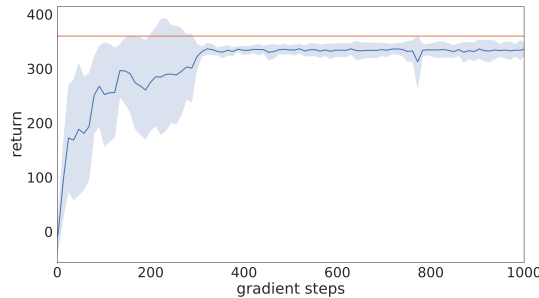


Figure 4.8: Performance of a linear policy (in blue) zero-shot learned (averaged over 5 runs, 95% bootstrapped CI). The orange line shows the best performance of the deep NN when training the PSSVF.

4.2.4 Fingerprint Analysis

Ablation on number of probing states Our experiments show that learning probing states helps evaluating the performance of many policies, but how many of such probing states are necessary for learning? We run our main experiments again, with fewer probing states, and discover that in many environments, a very small number of states is enough to achieve good performance. In particular, we find that the PSSVF with 5 probing states achieves 314 and 2790 final return in Swimmer and Hopper respectively, while Walker needs at least 50 probing states to obtain a return above 2000. In general, 200 probing states represent a good trade-off between learning stability and computational complexity. We compare the performances of PSSVF versions with varying numbers of probing states. We use the same hyperparameters as in the main experiments (see Appendix B.1.2), apart for the number of probing states. Figure 4.9 shows that in Hopper and Swimmer 10 probing

states are sufficient to learn a good policy, while Walker needs a larger number of probing states to provide stability in learning.

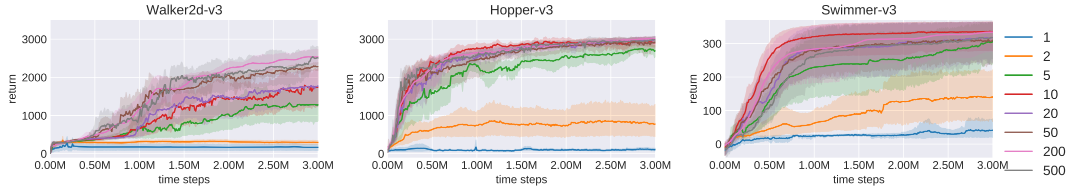


Figure 4.9: Average return of PSSVF with different number of probing states as a function of the number of time steps in the environment. The solid line is the average over 10 independent runs; the shading indicates 95% bootstrapped confidence intervals.

Learning Swimmer with 3 states The most surprising result is that a randomly initialized policy can learn near-optimal behaviors in Swimmer and Hopper by knowing how to act only in 3 (5) such crucial learned states (out of infinitely many in the continuous state space). To verify this, we first train a PSSVF with 5 probing states following Algorithm 1 for $2M$ time steps. Then we manually select 3 of the 5 learned probing states in Swimmer, and compute the actions of an optimal policy in such states. Then we train a new, randomly initialized policy, to just fit these 3 data points minimizing MSE loss. After many gradient steps, the policy obtains a return of 355, compared to the return of 364 of the optimal policy that was used to compute such actions. Figure 4.10 includes a detailed analysis of this experiment. The probing actions are the vectors $[-0.97,-0.86],[-0.18,-0.99],[0.86,0.68]$. In the plot we notice that when the agent's state is close to the first probing state (bottom plot, depicted in blue), then both components of the actions are close to -1, like the probing action in such state. When the agent's state is close to the second state (bottom plot, depicted in orange), the first component of the action moves from -1 to 0 (and then to +1) in a smooth way, while the second component jumps directly to +1. This behavior is consistent with the second probing action, since the second component is more negative than the first. Notably, although the distance between the agent's state and the third probing state (bottom plot, depicted in green) is never close to zero, such a probing state is crucial: it induces the agent to take positive actions whenever the other probing states are far away. We observe similar behavior for other environments, although they need more of such states to encode the behavior of an optimal policy.

In order to select the 3 transitions we try all combinations of 3 probing states out

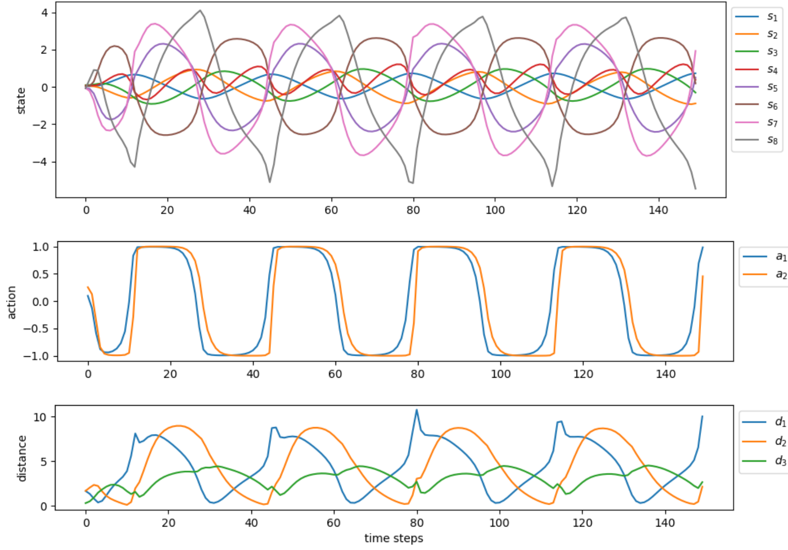


Figure 4.10: Behavior of the policy learned from 3 probing state-probing action pairs in Swimmer. From top to bottom: each component of the state vector across time steps in an environment simulation; each component of the action vector; L2 distance of the current state to each of the 3 probing states used for learning.

of 5 that we used to train our PSSVF. When cloning using all 5 probing states, the performance is very similar to the optimal policy. When choosing 4 out of 5 probing states, we notice that the performance highly depends on which probing state is removed, suggesting that some of the learned probing states are more important than others. When trying 3 out of 5 probing states this effect is more evident, and many combinations of 3 probing states lead to poor cloning performance. We can see in Figure 4.11 that as the MSE loss goes to zero when fitting the 3 transitions, the return of the policy increases until it almost matches the optimal value. We use a batch size of 3 and a learning rate of $2e-5$ to fit the new policy. The other hyperparameters are the same as in the main experiments (see Appendix B.1.2).

Learning Hopper with 5 states We repeat the same experiment of cloning near-optimal behaviour from a few states in the Hopper environment. Using the action of a good policy (whose return is 2450) in 5 probing states, we are able to fit a new policy and obtain a final return of 2200. We use a batch size of 5 and a learning rate of $1e-4$ for the randomly initialized policy. All other hyperparameters are like in the Swimmer experiment with 3 transitions. Figure 4.12 shows the learning curve, while Figure 4.13 relates the behavior of the policy learned using the 5 transitions to the distance of the current agent's state to the probing states. The 5 probing actions

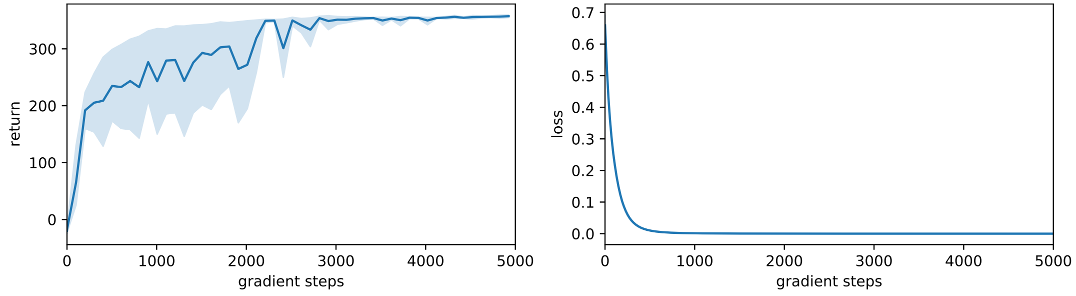


Figure 4.11: On the left: return of the policy learned using 3 transitions in Swimmer. On the right, MSE for fitting the 3 transitions. Average over 5 independent runs and 95% bootstrapped confidence interval.

$\{\tilde{a}_k\}_{k=1}^{5}$ are:

$$
\begin{aligned}
\tilde{a}_1 &= [0.4859, 0.6492, -0.7818], \\
\tilde{a}_2 &= [0.9251, 0.9100, 0.2322], \\
\tilde{a}_3 &= [0.0405, 0.0475, 0.9091], \\
\tilde{a}_4 &= [0.2925, -0.4677, -0.1329], \\
\tilde{a}_5 &= [0.7578, 0.4327, -0.1521].
\end{aligned}
$$

We observe a similar behavior of the Swimmer experiments (Figure 4.10), where the action chosen by the agent is similar to the probing action of a probing state whenever the agent's state is close to the probing state. Although the dynamics in Hopper are more complex than in Swimmer, 5 probing states are enough to make the agent perform non-trivial actions in the environment.

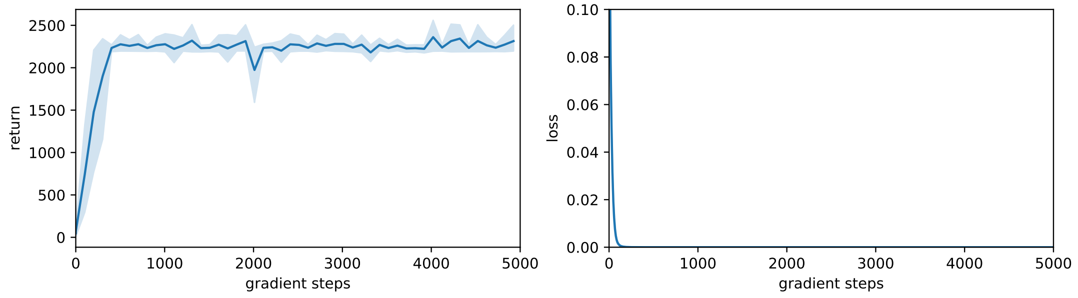


Figure 4.12: On the left: return of the policy learned using 5 transitions in Hopper. On the right, MSE for fitting the 5 transitions. Average over 5 independent runs and 95% bootstrapped confidence interval.

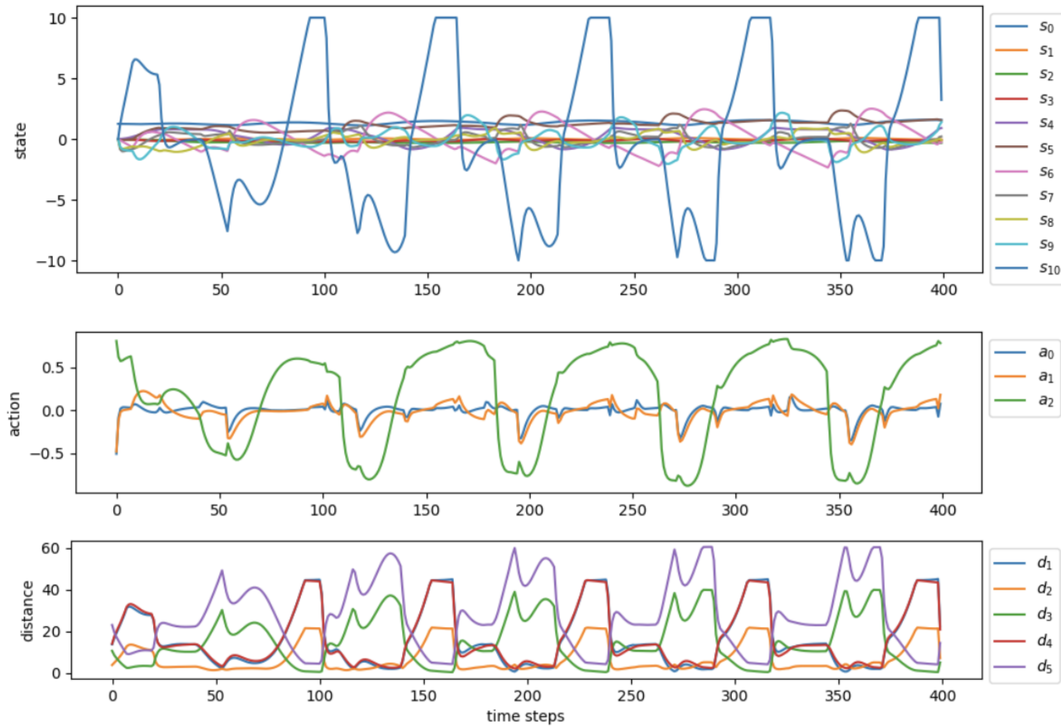


Figure 4.13: Behavior of the policy learned from 5 probing state-probing action pairs in Hopper. From top to bottom: each component of the state vector across time steps in an environmental simulation; each component of the action vector; L2 distance of the current state to each of the 5 probing states used for learning.

Visualization of RL probing states It is possible to visualize the probing states learned by the PSSVF. To understand the behaviour in probing states, we initialize the MuJoCo environment to the learned probing state (when possible) and let it evolve for a few time steps while performing no action. In environments like Hopper and Walker, probing states might not correspond to a real state in the environment (e.g., some components of the probing state are outside a specific range). We notice that this is usually not the case and that the learned probing states generally correspond to valid environmental states. Moreover, we observe that probing states tend to get closer to certain critical situations over learning. These are states where certain actions have a significant effect on the future. In the Ant environment, we notice that all components of the probing state vector from index 28 to 111 learn a value of around $1e-8$. Interestingly, the process of fingerprinting discovers this 'bug' in MuJoCo 2.0.2.2 that sets all contact forces in Ant to zero. Since these components of the state vector remain constant during the environmental interactions, and are therefore not relevant for learning, the PSSVF learns to set them to zero as well.

Figure 4.14 shows the evolution of the Swimmer environment from the selected probing states when no action is taken. The 3 probing states reported are those used for the experiment of Figures 4.11 and 4.10.

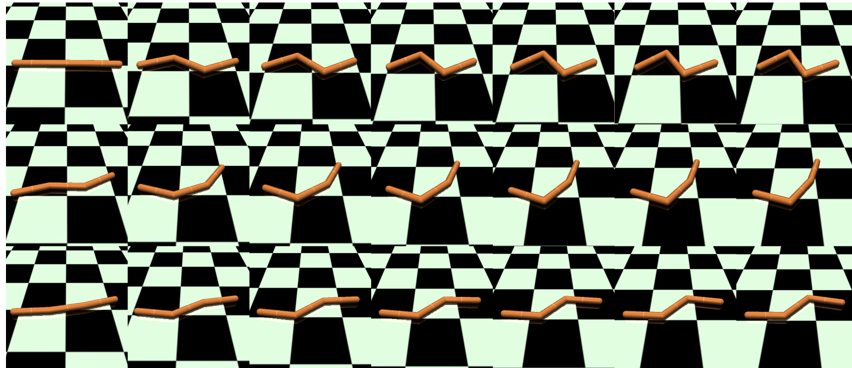


Figure 4.14: From top to bottom: the three learned probing states in Swimmer. From left to right: evolution of the environment over time steps. The agent is initialized in the probing state and performs no action.

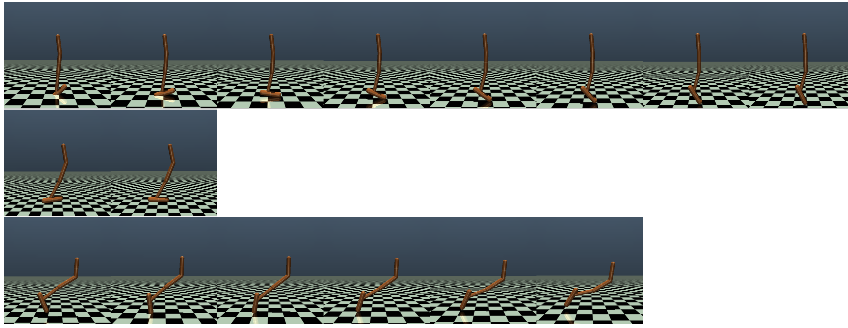


Figure 4.15: From top to bottom: the 5 learned probing states on Hopper. From left to right: various time steps in the environment. The agent is initialized in the probing state and performs no action.

Figure 4.15 shows 3 out of the 5 learned probing states on Hopper in the experiment of Figures 4.12 and 4.13. The other 2 probing states do not correspond to valid states in Hopper and are therefore not visualized. Since the probing state does not depend on a specific policy, no action is taken in the probing state, and the environment is allowed to evolve naturally from that state. The duration of interaction differs in each row of the figure as termination occurs at different points from the probing states.

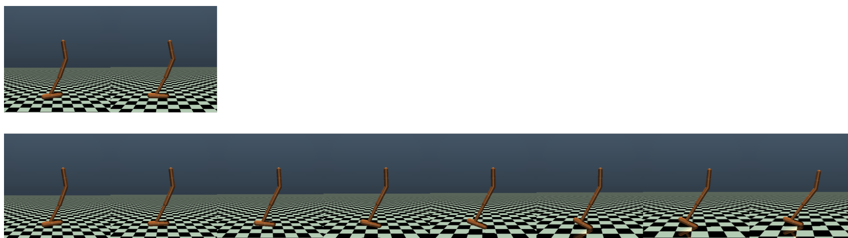


Figure 4.16: Evolution of the environment from a probing state when (Top) no actions taken, (Bottom) the first action in the probing state is taken using a good policy. Then no action is performed.

Figure 4.16 supports our hypothesis that some probing states might capture critical scenarios. In the considered probing state from Hopper we see that taking no action results in immediate failure as indicated by the shorter span of interaction in the top panel of Figure 4.16. In contrast, acting for a single time-step with a successful policy in that situation helps the agent survive and prolongs the interaction (bottom panel of Figure 4.16).

Additional probing states for all environments can be seen in animated form on the website `https://policyevaluator.github.io/`.

4.3 Related Work

Policy Evaluation Networks. Our method is based on a recent class of algorithms that were developed to address global estimation and improvement of policies. For Policy Evaluation Networks (PVNs) [Harb et al., 2020], an actor-critic algorithm for offline learning through policy fingerprinting was proposed. PVNs focus on the offline RL setting. In PVNs, first a dataset of randomly initialized policies with their returns is collected. Then, once their $V(\theta)$ with policy fingerprinting is trained, they perform policy improvement through gradient ascent steps on V. Here, however, we empirically demonstrated that PBVFs with policy fingerprinting mechanisms can be efficient in the online scenario. A minor difference between our approach and PVNs is that PVNs predict a discretized distribution of the return, whereas our approach simply predicts the expected return. Our PSSVF with policy fingerprinting can be seen like an online version of PVN without some of the tricks used. Fingerprinting itself is similar to a technique for "learning to think" [Schmidhuber, 2015b] where one NN learns to send queries (sequences of activation vectors) into another NN and learns to use the answers (sequences of activation vectors) to improve its performance.

Policy-extended value functions. Recent work [Tang et al., 2020] learned Parameter-Based State-Value Functions which, coupled with PPO, improved performance. The authors did not use the value function to directly backpropagate gradients through the policy parameters, but only exploited the general policy evaluation properties of the method. They also proposed two dimensionality reduction techniques. The first, called *Surface Policy Representation*, consists of learning a state-action embedding that encodes possible information from a policy π_θ. This requires feeding state-action pairs to a common MLP whose output is received as input to the value function. The MLP is trained such that it allows for both low prediction error in the value function and low reconstruction error of the action, given a state and the embedding. This method is not differentiable in the policy parameters, therefore it cannot be used for gradient-based policy improvement. The second method, called *Origin Policy Representation* (OPR), consists of using an MLP that performs layer-wise extraction of features from policy parameters. OPR uses MLPs to take as input direcly the weight matrix of each layer. This approach is almost identical to directly feeding the policy parameters to the value function (they concatenate the state to the last layer of

such MLP), and suffers from the curse of dimensionality. Also, OPR was not used to directly improve the policy parameters, but only to provide better policy evaluation.

Policy Representation. Alternative strategies to represent policies have been studied in previous work. One such strategy aims to learn a representation function mapping trajectories to a policy embedding through an auto-encoding objective [Grover et al., 2018; Raileanu et al., 2020]. In particular, Grover et al. [2018] use this idea to model the agent's behavior in a multi-agent setting. A recent approach [Raileanu et al., 2020] performs gradient ascent steps finding a policy embedding that maximizes the value function's predicted return. While this maximization through the value function is similar to our setting, it relies on a representation function (or policy decoder). Our method does not use a decoder and instead directly backpropagates the gradients into the policy parameters for policy improvement. Closer to our fingerprinting setup, Pacchiano et al. [2020] utilize pairs of states and actions (from the corresponding policy) as a policy representation. However, unlike in our approach, the probing states are not learned, but sampled from a chosen probing state distribution. Closely related to our fingerprint embedding is also the concept of Dataset Distillation [Wang et al., 2018]. However, in our RL setting, learning to distill crucial states from an environment is harder due to the non-differentiability of the environment. Recent work [Kanervisto et al., 2020] suggests representing policies based on visited states via Gaussian Mixture Models applied to an offline dataset of data from multiple policies. The authors mention that their current version of policy supervectors is intended for analysing policies and is not yet suitable for online optimization. Value functions conditioned on other quantities include vector-valued adaptive critics [Schmidhuber, 1991a], General Value Functions [Sutton et al., 2011], and Universal Value Function Approximators [Schaul et al., 2015]. Unlike our approach these methods typically generalize over achieving different goals, and are not used to generalize across policies.

4.4 Discussion

Our approach connects Parameter-Based Value Functions (PBVFs) and the fingerprinting mechanism of Policy Evaluation Networks. It can efficiently evaluate large Neural Networks, is suitable for off-policy data reuse, and competitive with existing baselines for online RL tasks. Zero-shot learning experiments on MNIST and continuous control problems demonstrated our method's generalization capabilities. Our value function is invariant to policy architecture changes, and can extract essential

knowledge about a complex environment by learning a small number of situations that are important to evaluate the success of a policy. A randomly initialized policy can learn optimal behaviors in Swimmer (Hopper) by knowing how to act only in 3 (5) such crucial learned states (out of infinitely many in the continuous state space). This suggests that some of the most commonly used RL benchmarks require to learn only a few crucial state-action pairs. Our set of learned probing states is instead used to evaluate any policy, while in practice different policies may need different probing states for efficient evaluation.

Preliminary experiments using the PSVF in Algorithm 2 with static policy fingerprinting for learning $V(s,\theta)$ showed results that were slightly inferior to those of the PSSVF $V(\theta)$ in all tested environments. Further analysis is necessary to develop training techniques and hyperparameter tuning to achieve more competitive results. We leave this as a topic for future work.

A natural direction for improving PBVFs and scaling them to more complex tasks is to generate probing states using recurrent policy fingerprinting. In the next chapter, we will apply this method to learn representations of Recurrent Neural Networks weights.

Chapter 5

Learning Useful Representations of Recurrent Neural Network Weight Matrices

For decades, researchers have developed techniques for learning representations in deep neural networks (NNs). This expertise has significantly advanced the field by enabling models to convert data into useful formats for solving problems. In particular, Recurrent NNs (RNNs) have been widely adopted due to their computational universality [Siegelmann and Sontag, 1991]. Low-dimensional representations of the programs of RNNs (their weight matrices) are of great interest as they can speed up the search for solutions to given problems. For instance, compressed RNN representations have been used to evolve RNN parameters [Koutnik et al., 2010] for controlling a car from video input [Koutník et al., 2013]. However, such representations have employed Fourier-type transforms, e.g., the coefficient of the Discrete Cosine Transform (DCT) [Srivastava et al., 2012], and did not use the capabilities of NNs to learn such representations. Recent work, including the methods presented in the previous chapters, has seen a rise of representation learning techniques for NN weights using powerful neural networks as encoders [Unterthiner et al., 2020; Schürholt et al., 2021; Dupont et al., 2022; Faccio et al., 2022]. However, there is a lack of methods for learning representations of RNNs. This chapter introduces novel techniques for learning them, using powerful NNs which may be RNNs themselves. Just like representation learning in other fields, such as Computer Vision, facilitates solving specific tasks, such techniques can facilitate learning, searching, and planning with RNNs.

5.1 Method

We consider a Recurrent Neural Network (RNN), $f_\theta : \mathbb{R}^X \times \mathbb{R}^H \to \mathbb{R}^Y \times \mathbb{R}^H; (x, h_o) \mapsto (y, h_n)$, parametrized by $\theta \in \Theta \subset \mathbb{R}^{n_\Theta}$, which maps an input x and hidden state h_o to an output y and a new hidden state h_n. The RNN interacts with a potentially stochastic environment, $\mathscr{E}$, that maps an RNN's output y to a new input x. The environment may have its own hidden state η. By sequentially interacting with the environment, the RNN produces a rollout defined by:

$$\begin{cases} x_t, \eta_t = \mathscr{E}(y_{t-1}, \eta_{t-1}) \\ y_t, h_t = f_\theta(x_t, h_{t-1}), \end{cases}$$

with fixed initial states y_0, η_0 and h_0. For instance, f_θ might be an autoregressive generative model, with $\mathscr{E}$ acting as a stochastic environment that receives a probability distribution over some language tokens, y_t—the output of f—, and produces a representation (e.g., a one-hot vector) of the new input token x_{t+1}. When the environment is stochastic, numerous rollouts can be generated for any $\theta \in \Theta$. A rollout sequence of a function f_θ in environment $\mathscr{E}$ has the form $S_\theta = (x_1, y_1, x_2, y_2, \dots)$.

Encoder and Emulator Our primary objective is to propose, analyze, and train several methods for representing RNN weights. We define the Encoder $E_\phi : \Theta \to \mathbb{R}^M; \theta \mapsto z$, parametrized by $\phi \in \Phi \subset \mathbb{R}^{n_\Phi}$ as a function mapping the RNN parameters θ to a lower-dimensional representation z. To train the encoder E_ϕ, we consider an Emulator $A_\xi : \mathbb{R}^X \times \mathbb{R}^B \times \mathbb{R}^Z \to \mathbb{R}^Y \times \mathbb{R}^B; (x, b_o, z) \mapsto (\tilde{y}, b_n)$, parametrized by $\xi \in \Xi \subset \mathbb{R}^{n_\Xi}$. The Emulator is an RNN with hidden state b that learns to imitate different RNNs f_θ based on their function encoding $z = E(\theta)$.

Dataset and Training We consider a dataset $\mathscr{D} = \{(\theta_i, S_{\theta_i}) | i = 1, 2, \dots\}$ composed of tuples, each containing the parameters of a different RNN and a corresponding rollout sequence. We assume that all RNNs have the same initial state h_0 but have been trained on different tasks. Our self-supervised learning approach to training function representations is inspired by the work of Raileanu et al. [2020]). The Encoder E_ϕ and the Emulator A_ξ are jointly trained by minimizing a loss function $\mathscr{L}$. This loss function measures the behavioral similarity between an RNN f_θ and the Emulator A_ξ, which is conditioned on the function representation $z = E_\phi(\theta)$ of θ as produced by the Encoder E_ϕ. Put simply, the Emulator utilizes the representations of a set of diverse RNNs f_θ to imitate their behavior:

$$\min_{\phi, \xi} \mathbb{E}_{(\theta, S) \sim \mathscr{D}} \sum_{(x_i, y_i) \in S} \mathscr{L}\big(A_\xi(x_i, b_{i-1}, E_\phi(\theta)), y_i\big). \tag{5.1}$$

In the case of continuous outputs y, the mean-squared error provides a suitable loss function. Conversely, for categorical outputs, we employ the inverse Kullback-Leibler divergence.

5.1.1 RNN Encoders

In this section, we explore various mechanistic and functionalist methods for constructing RNN encoders. These approaches will be compared in our experimental section.

Flattened Weights (Mechanistic) Flattening the weights into a single vector presents the most straightforward method for encoding an RNN. While this technique has shown efficacy on a modest scale [Faccio et al., 2021; Herrmann et al., 2022], it faces challenges when applied to larger parameter vectors, especially in handling weight-space symmetries such as neuron permutations. For the flattened weights Encoder, all parameters θ of the RNN to be encoded are flattened into a vector. This weight vector is given as input to a multi-layer perceptron (MLP) with ReLU nonlinearities, which outputs the RNN encoding z.

Neural Functional (Mechanistic) Fast Weight Programmers [Schmidhuber, 1992b, 1993; Schlag et al., 2021; Irie et al., 2021b] are neural networks that can process the gradients or weights of another neural network. A recent variant thereof, called Neural Functionals[Navon et al., 2023], has been used to learn representations of neural network weights that are invariant to the permutation of hidden neurons. The architecture comprises layers that display equivariance to neuron permutations, followed by a final pooling operation that ensures the invariance property. Neural Functionals have been theoretically proven to be able to extract all information from the weights of a neural network [Navon et al., 2023]. However, their implementation to date has been confined to feedforward networks, such as MLPs and CNNs. For the neural functional (NF) Encoder, we adapt the equivariant NF-layer [Zhou et al., 2023] for LSTMs. To preserve both equivariance to neuron permutation and functional universality, the appropriate row- and column-wise feature extractors have to be added for input-to-hidden and hidden-to-hidden weights, considering rollouts across time and depth of the network.

Non-Interactive RNN Probing (Functionalist) In the context of Reinforcement Learning and Markov Decision Processes, policy fingerprinting has emerged as an effective way to evaluate feedforward neural network policies [Faccio et al.,

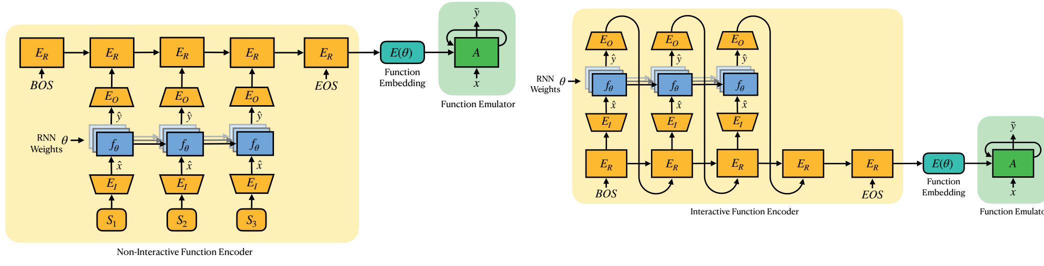


Figure 5.1: Left: Non-Interactive Encoder. Right: Interactive Encoder.

2022; Harb et al., 2020; Faccio et al., 2023]. In static policy fingerprinting, a set of learnable probing inputs is given to the network using the techniques introduced in Section 4.1.1. Based on the set of corresponding policy outputs, a function (policy) representation is produced. This approach can be adapted in a straightforward way for RNNs by learning whole probing sequences instead of probing inputs (see Figure 5.1, left). In the context of this chapter, we refer to this approach as non-interactive RNN probing. RNN probing Encoders have three main components: the core LSTM E_R, an input projection MLP E_I, and an output projection MLP E_O.

For the non-interactive Encoder, a learnable latent probing sequence $(S_1, S_2, \dots, S_l)$ with a fixed length l is given to E_I. $E_I(S_i)$ is interpreted as either one a several parallel probing inputs $\hat{x}_i$ and given to f_θ. The resulting probing outputs $\hat{y}_i := f_\theta(\hat{x}_i)$ are given to E_O (in the case of multiple parallel probing outputs, the values $\hat{y}_i$ are concatenated). The sequence of probing output projections $(E_O(\hat{y}_1), \dots, E_O(\hat{y}_l))$ is given as input to E_R, preceded by a begin-of-sequence (BOS) and followed by an end-of-sequence (EOS) token. E_R's output after the EOS token is transformed with a learned linear projection into the RNN representation z.

Interactive RNN Probing (Functionalist) The probing sequences for non-interactive RNN probing are static, i.e., at test time, the probing sequences do not depend on the specific RNN being evaluated. The alternative is to make the probing sequences dynamically dependent on the given RNN by using the recurrent policy fingerprinting technique described in Section 4.1.2. Each item in the probing sequences should depend on the outputs of the given RNN to the previous items (Figure 5.1, right). The interactive probing encoder differs from the non-interactive one in one crucial aspect: Instead of having a learned but static latent probing sequence, the probing inputs at each step are based on the output of E_R from the current step, which in turn depends on the probing outputs of the previous step. This means that the interactive probing Encoder can dynamically adapt the probing sequences to the particular RNN f_θ that is being encoded. This idea has been described previously to

extract arbitrary information from a recurrent world model [Schmidhuber, 2015b].

5.2 Experiments and Results

The experiments in this section were performed by Vincent Herrmann in a common paper [Herrmann et al., 2023] and are reported here for completeness.

5.2.1 Dataset

We consider the family of context-sensitive languages:

$$L_{m_1,\ldots,m_k} := \{a_1^{n+m_1} a_2^{n+m_2} \ldots a_k^{n+m_k} | n \in \mathbb{N}\}, \tag{5.2}$$

with $m_1, \ldots, m_k \in \mathbb{N}$ and $a_1, \ldots, a_k$ being the letters/tokens of the language. The parameters m_i define the relative number of times different tokens may appear. As an example, one member of the language $L_{3,1,2}$ is the string $a_1a_1a_1a_1a_2a_2a_3a_3a_3$.

The set of languages used is $\{L_{r,r+o_1,r+o_2,r+o_3} | o_1, o_2, o_3 \in \{-3, \ldots, 2\}$ and $r = -\min\{o_1, o_2, o_3\}\}$, with L defined in Equation 5.2. This set contains $6^3 = 216$ uniquely identifiable languages. The training data for each LSTM are strings from a particular language of length ≤ 40, with an additional begin-of-sequence and end-of-sequence token.

The LSTMs trained for the dataset have two layers with a hidden size of 32, resulting in a total of 13766 parameters. In total, 1000 such networks are trained, each on one of the 216 possible languages. For each LSTM, 10 snapshots (at steps 0, 100, 200, 500, 1000, 2000, 5000, 10000 and 20000) are saved during training. A snapshot consists of the LSTM's current weights and 100 sequences, also of length 40, generated by it.

5.2.2 Emulator

The Emulator A_ξ is an LSTM network. The conditioning on the function representation z is done by adding a learned linear projection of z to the embedding of the begin-of-sequence token.

5.2.3 Results

We empirically analyze various approaches to learning representations of RNNs, with a specific focus on LSTM [Hochreiter and Schmidhuber, 1997] weights. Each LSTM in our dataset serves as an autoregressive generative model of a specific formal

language. Each LSTM is trained on strings from a particular language using the standard language modelling objective. We split the dataset into training, validation, and out-of-distribution (OOD) test parts. The OOD split includes only tasks in which the relative frequencies of each token appearance are small (i.e., all tokens appear approximately the same number of times). The validation set is utilized for early stopping during training. All shown results are derived from the test set. The experiments employ the four types of function encoders described in Section 5.1.1. The encoders' hyperparameters are selected to ensure a comparable number of parameters among them. The training details remain consistent across all runs (further details are available in Appendix C.1.1). All encoders are trained end-to-end together with an LSTM emulator to minimize the loss defined in Equation 5.1, utilizing the reverse Kullback-Leibler divergence as the loss function $\mathscr{L}$. The objective is to ensure that the LSTM weight encodings z serve as generally useful representations. We verify this by training models for two downstream tasks using the fixed representations provided by the encoder E. The first task involves classifying the language on which an LSTM f_θ was trained, given its encoding $E(\theta)$. This classification is inherently challenging, considering the dataset contains a total of 216 different languages, and some networks are nearly untrained. The second task aims to predict the performance of f_θ, defined as the percentage of strings generated by f_θ belonging to the language that f_θ was trained on. We present the results for these tasks in Figure 5.2.

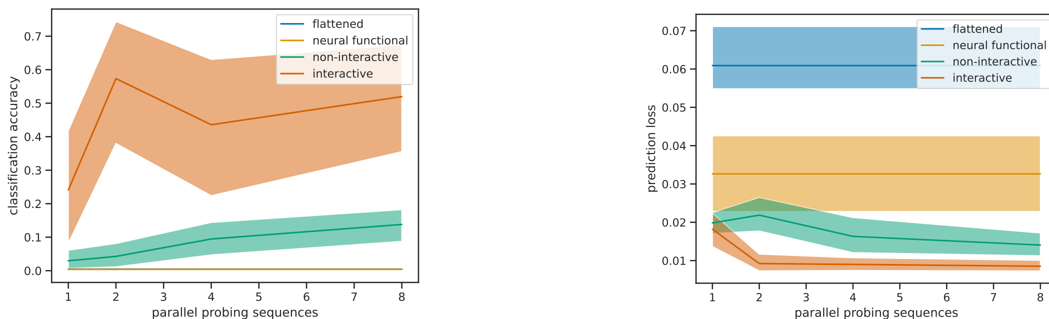
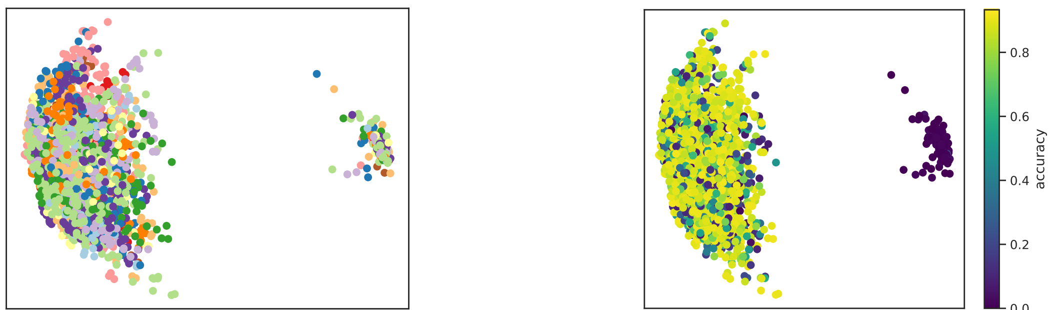


Figure 5.2: Left: Accuracy of a language classifier, trained using the generated function encodings. Right: Loss of a performance predictor, also trained on the generated function encodings, depicted on the test set. Plots are presented as a function of the number of parallel probing sequences (only relevant for interactive and non-interactive probing encoders). Both graphs display the mean and bootstrapped 95% confidence intervals, aggregated across 15 seeds.

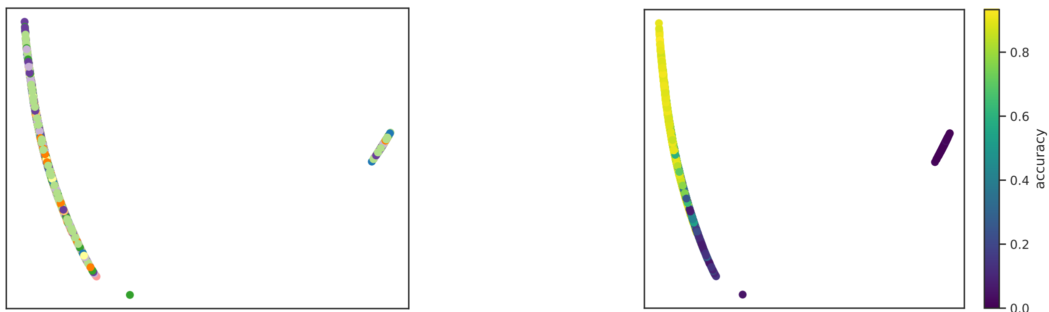
From the results, it is evident that the interactive probing encoder yields the most useful representations for both tasks. Having multiple probing sequences in parallel

benefits both interactive and non-interactive encoders. The representations derived from the flattened weights and the neural functional encoder appear to contain no useful information for the language classifier. In predicting accuracy, representations from neural functionals outperform those based on flattened weights but fall short when compared to functionalist representations. A visualization of the learned embedding spaces can be found in Figure 5.6.

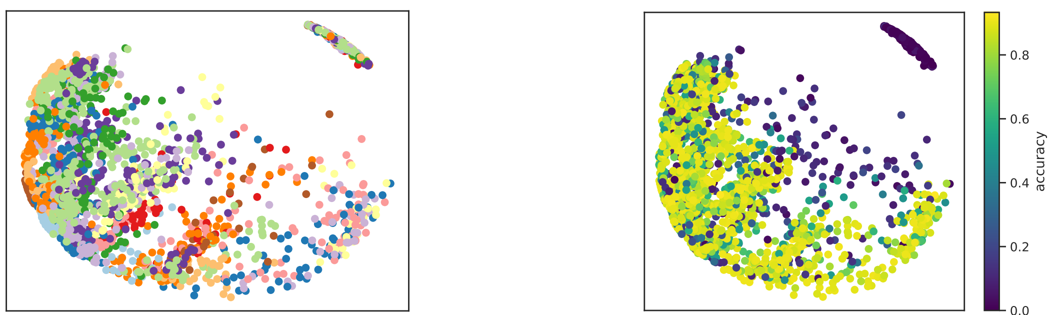
Flattened



Neural Functional



Non-interactive



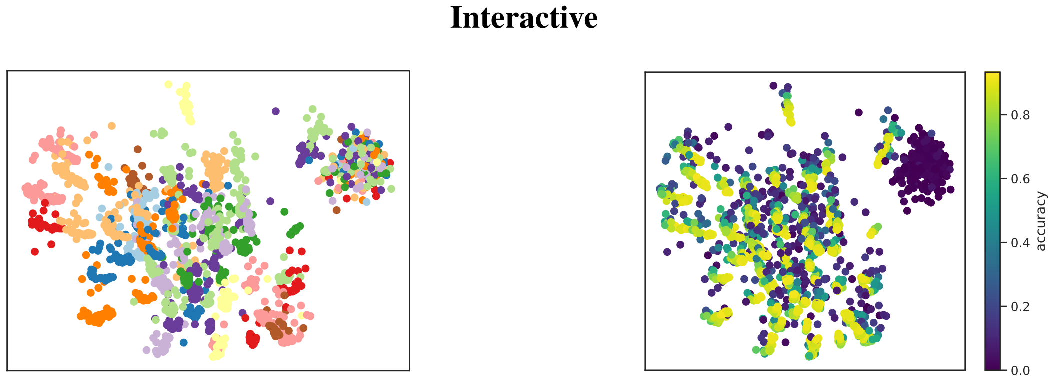


Figure 5.6: PCA of the function encodings generated by different encoders. Left column: Colored by language, Right column: Colored by perfomance (accuracy).

Figure 5.7 shows the test losses of the different Emulators (as defined in Equation 5.1). The relative performance of the different encoders types is similar as for the downstream tasks shown Figure 5.2.

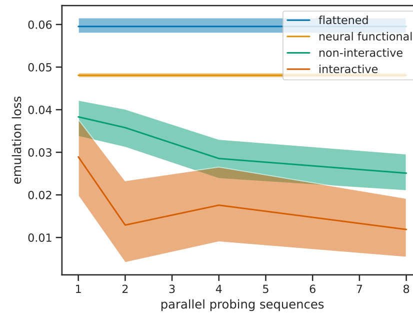


Figure 5.7: Loss of the emulator on the test. Plotted as a function of the number of parallel probing sequences. Mean and bootstrapped 95% confidence intervals across 15 seeds.

We investigate the results for different lengths of the probing sequence for both interactive and non-interactive probing encoders, as illustrated in Figure 5.8. The plot reveals a trade-off: a small number of sequential probing steps corresponds to a simpler emulator function to learn, while a large number of probing steps correlates with an emulator capable of learning useful representations, albeit resulting in a more challenging network to train. The optimal number of probing steps appears to be around 22.

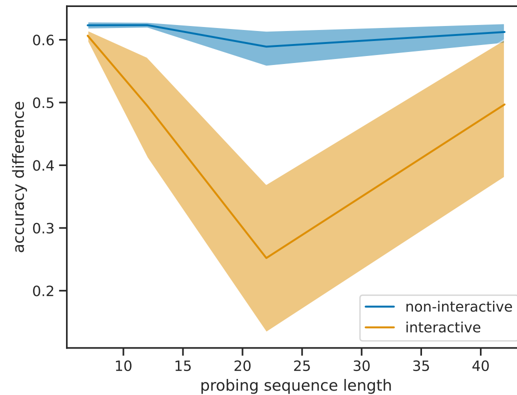


Figure 5.8: Difference in generation accuracy of the emulated function compared to the original one. Validation set. Plotted as a function of the length of the probing sequence. Mean and bootstrapped 95% confidence intervals across 15 seeds.

In Figure 5.9 we plot probing sequences generated by an encoder using recurrent fingerprinting for different instances of a particular language. For the sequence lengths 12, 22 and 42, the encoder produces an insightful probing sequence, i.e., probing sequences that belong to the corresponding language.

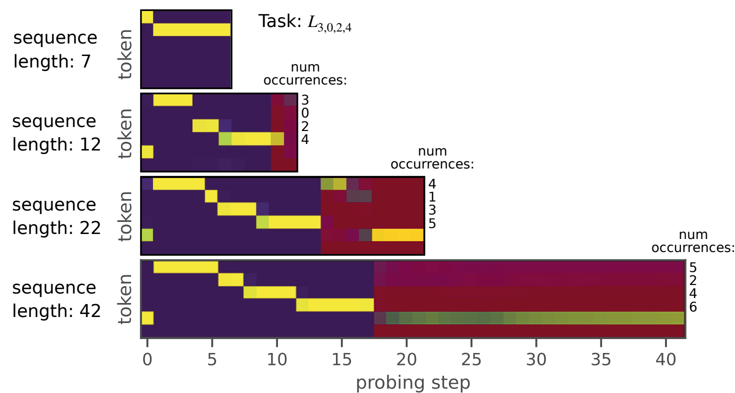


Figure 5.9: Probing sequences generated for language $L_{3,0,2,4}$ by the best performing interactive function encoder with different sequence lengths.

5.3 Discussion

We identified two classes of methods for learning RNN weight representations. Firstly, we adapted the Mechanistic Neural Functional approach to RNNs and, secondly, presented two novel Functionalist methods, theoretically demonstrating when

their representations can be utilized to identify RNNs. Functionalist methods outperformed Mechanistic ones, learning more useful RNN weight representations for two downstream tasks. Future work will explore the combination of both approaches and evaluate their performance on more challenging problems.

Recurrent Policy Fingerprinting offers several advantages. Firstly, for each policy, it learns to generate distinct sets of probing states, specifically those that are most crucial for policy evaluation. Secondly, the number of weights in the evaluator function remains constant even if the number of probing states increases. Note that if the number of probing states generated is the same as the number of time steps for the RNN in the environment, our value function could, in principle, learn to always generate the next state as a probing state, effectively learning a simulator of the environment. Future work will focus on applying the techniques developed in this Chapter to problems involving natural language, and to Reinforcement Learning problems.

Chapter 6

Goal-Conditioned Generators of Deep Policies

6.1 Method

Goal-conditioned RL agents can learn to solve many different tasks, where the present task is encoded by special command inputs [Schmidhuber and Huber, 1991; Schaul et al., 2015]. Many RL methods learn value functions [Sutton and Barto, 2018] or estimate stochastic policy gradients (with possibly high variance) [Williams, 1992; Sutton et al., 1999]. Upside-down RL (UDRL) [Srivastava et al., 2019; Schmidhuber, 2019] and related methods [Ghosh et al., 2019], however, use supervised learning to train goal-conditioned RL agents. UDRL agents receive command inputs of the form "act in the environment and achieve a desired return within so much time" [Schmidhuber, 2019]. Typically, hindsight learning [Andrychowicz et al., 2017; Rauber et al., 2018] is used to transform the RL problem into the problem of predicting actions, given reward commands. Consider a command-based agent interacting with an environment, given a random command c, and achieving return r. Its behavior would have been optimal if the command had been r. Hence the agent's parameters can be learned by maximizing the likelihood of the agent's behavior, given command r. Unfortunately, in the episodic setting, many behaviors may satisfy the same command. Hence the function to be learned may be highly multimodal, and a simple Gaussian maximum likelihood approach may fail to capture the variability in the data. In this chapter, we present a method that proposes an alternative solution to this challenge.

6.1.1 Background

Here, we use a slightly different formulation of the Markov Decision Process that we formally report for completeness. We define a trajectory $\tau \in \mathscr{T}$ as the sequence of state-action pairs that an agent encounters during an episode in the MDP $\tau = (s_{\tau,0}, a_{\tau,0}, s_{\tau,1}, a_{\tau,1}, \dots, s_{\tau,T}, a_{\tau,T})$, where T denotes the time-step at the end of the episode ($T \leq H$). The return of a trajectory $R(\tau)$ is defined as the cumulative discounted sum of rewards over the trajectory $R(\tau) = \sum_{t=0}^{T} \gamma^t R(s_{\tau,t}, a_{\tau,t})$, where $\gamma \in (0,1]$ is the discount factor.

The RL problem consists in finding the policy π_{θ^*} that maximizes the expected return obtained from the environment, i.e., $\pi_{\theta^*} = \arg\max_{\pi_\theta} J(\theta)$:

$$J(\theta) = \int_{\mathscr{T}} p(\tau|\theta) R(\tau) \, \mathrm{d}\tau, \tag{6.1}$$

where $p(\tau|\theta) = \mu_0(s_0) \prod_{t=0}^{T} \pi_\theta(a_t|s_t) P(s_{t+1}|s_t, a_t)$ is the distribution over trajectories induced by π_θ in the MDP. When the policy is stochastic and differentiable, by taking the gradient of $J(\theta)$ with respect to the policy parameters we obtain an algorithm called REINFORCE [Williams, 1992]: $\nabla_\theta J(\theta) = \int_{\mathscr{T}} p(\tau|\theta) \nabla_\theta p(\tau|\theta) R(\tau) \, \mathrm{d}\tau$.

In parameter-based methods [Sehnke et al., 2010, 2008; Salimans et al., 2017; Mania et al., 2018], at the beginning of each episode, the weights of a policy are sampled from a distribution $\nu_\rho(\theta)$, called the hyperpolicy, which is parametrized by $\rho \in \mathrm{P} \subset \mathbb{R}^{n_\mathrm{P}}$. Typically, the stochasticity of the hyperpolicy is sufficient for exploration, and deterministic policies are used. The RL problem translates into finding the hyperpolicy parameters ρ maximizing expected return, i.e., $\nu_{\rho^*} = \arg\max_{\nu_\rho} J(\rho)$:

$$J(\rho) = \int_{\Theta} \nu_\rho(\theta) \int_{\mathscr{T}} p(\tau|\theta) R(\tau) \, \mathrm{d}\tau \, \mathrm{d}\theta. \tag{6.2}$$

This objective is maximized by taking the gradient of $J(\rho)$ with respect to the hyperpolicy parameters: $\nabla_\rho J(\rho) = \int_\Theta \int_{\mathscr{T}} \nu_\rho(\theta) \nabla_\rho \log \nu_\rho(\theta) p(\tau|\theta) R(\tau) \, \mathrm{d}\tau \, \mathrm{d}\theta$. This gradient can be either approximated through samples [Sehnke et al., 2010, 2008; Salimans et al., 2017] or estimated using finite difference methods [Mania et al., 2018]. This only requires differentiability and stochasticity of the hyperpolicy.

6.1.2 Fast Weights Programmers

Fast Weight Programmers (FWPs) [Schmidhuber, 1992b, 1993] are NNs that generate changes of weights of another NN conditioned on some contextual input. In our UDRL-like case, the context is the desired return to be obtained by a generated policy.

The outputs of the FWP are the policy parameters $\theta \in \Theta$. Formally, our FWP is a function $G_\rho : \mathbb{R}^{n_c} \longrightarrow \Theta$, where $c \in \mathbb{R}^{n_c}$ is the context-input and $\rho \in \mathrm{P}$ are the FWP parameters. Here, we consider a probabilistic FWP of the form $g_\rho(\theta|c) = G_\rho(c) + \varepsilon$, with $\varepsilon \sim \mathscr{N}(0, \sigma^2 I)$ and $\sigma \in \mathbb{R}^{n_\Theta}$ is fixed, with n_Θ being the dimensionality of the policy parameter space. In this setting, the FWP conditioned on context c induces a probability distribution over the parameter space, similar to the one induced by the hyperpolicy in the previous Section. Using the FWP to generate the weights of a policy, we can rewrite the RL objective, making it context-dependent:

$$J(\rho, c) = \int_\Theta g_\rho(\theta|c) \int_{\mathscr{T}} p(\tau|\theta) R(\tau) \,\mathrm{d}\tau \,\mathrm{d}\theta. \tag{6.3}$$

Compared to Eq. 6.2, $J(\rho, c)$ induces a set of optimization problems that now are context-specific [1]. Here, $J(\rho, c)$ is the expected return for generating a policy with a generator parametrized by ρ, when observing context c. Instead of optimizing Eq. 6.2 using policy gradient methods, we are interested in learning a good policy through pure supervised learning by following a sequence of context-commands of the form "generate a policy that achieves a desired expected return." Under such commands, for any c, the objective $J(\rho, c)$ can be optimized with respect to ρ to equal c. FWPs offer a suitable framework for this setting, since the generator network can learn to create weights of the policy network so that it achieves what the given context requires. In the subsequent sections, we will consider unidimensional context commands $c \in \mathbb{R}$.

6.1.3 Gogepo

Here we develop *GoGePo*, our algorithm to generate policies that achieve any desired return. In the supervised learning scenario, it is straightforward to learn the parameters of the FWP that minimize the error $\mathscr{L}_G(\rho) = \mathbb{E}_{c \in D, \theta \sim g_\rho(\cdot|c)}[(J(\theta) - c)^2]$, where the context c comes from some set of possible commands D. This is because in supervised learning $J(\theta)$, the expected return, is a differentiable function of the policy parameters, unlike in general RL. Therefore, to make the objective differentiable, we learn a PSSVF $V_\mathbf{w} : \Theta \longrightarrow \mathbb{R}$ parametrized by $\mathbf{w}$ that estimates $J(\theta)$ [Faccio et al., 2021]. This evaluator function is a map from the policy parameters to the expected return. Once V is learned, the objective $\mathscr{L}_G(\rho)$ can be optimized end-to-end, like in the supervised learning scenario, to directly learn the generator's parameters.

[1] Note the generality of Eq. 6.3. In supervised learning, common FWP applications include the case where g is deterministic, θ are the weights of an NN (possibly recurrent), $p(\tau|\theta)$ is the output of the NN given a batch of input data, $R(\tau)$ is the negative supervised loss.

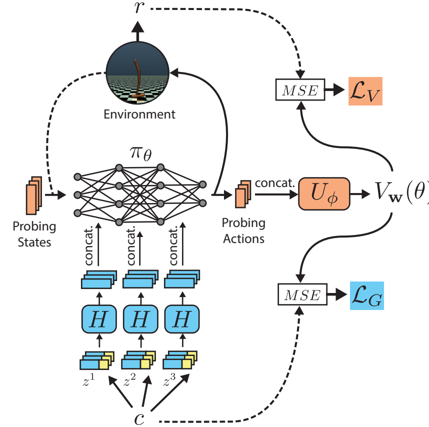


Figure 6.1: GoGePo generates policies using a Fast Weight Programmer (hypernetwork) conditioned on a desired return and evaluates the resulting policy using a parameter-based value function based on fingerprinting. This enables training using supervised learning.

Concretely, we minimize $\mathscr{L}_G(\rho) = \mathbb{E}_{c \in D}[(V_\mathbf{w}(G_\rho(c)) - c)^2]$ to learn the parameters ρ.

Our method is described in Algorithm 5 and consists of three steps. **First**, in each iteration, a command c is chosen following some strategy. Ideally, to ensure that the generated policies improve over time, the generator should be instructed to produce larger and larger returns. We discuss command strategies in the next paragraph. The generator observes c and produces policy π_θ which is run in the environment. The return and the policy (r, θ) are then stored in a replay buffer. **Second**, the evaluator function is trained to predict the return of the policies observed during training. This is achieved by minimizing MSE loss $\mathscr{L}_V(w) = \mathbb{E}_{(r,\theta) \in B}[(r - V_\mathbf{w}(\theta))^2]$. **Third**, we use the learned evaluator to directly minimize $\mathscr{L}_G(\rho) = \mathbb{E}_{r \in B}[(r - V_\mathbf{w}(G_\rho(r)))^2]$.

By relying on an evaluator function V that maps policy parameters to expected return, we do not need to model multimodal behavior. The generator is trained to produce policies such that the scalar return command matches the evaluator's scalar return prediction. In practice, the generator finds for each return command c a point θ in the domain of the evaluator function such that $V(\theta) = c$. As a very simple example, assume $J(\theta) = \theta^2$, $\theta \in [-1, 1]$. If methods like UDRL observe $\theta_1 = 1/2$ with $r_1 = 1/4$ and $\theta_2 = -1/2$ with $r_2 = 1/4$, a Gaussian maximum likelihood approach will learn to map $r = 1/4$ to $\theta = 0$, but for $\theta = 0$ the return is 0. Our evaluator function learns to approximate $J(\theta)$, so our generator will learn to find a single θ such that $V(\theta) = 1/4$. The value of θ produced by the generator depends

Algorithm 5 Gogepo with return commands

Input: Differentiable generator $G_\rho : \mathbb{R} \to \Theta$ with parameters ρ; differentiable evaluator $V_\mathbf{w} : \Theta \to \mathbb{R}$ with parameters $\mathbf{w}$; empty replay buffer D

Output : Learned $V_\mathbf{w} \approx V(\theta) \forall \theta$, learned $G_\rho s.t. V(G_\rho(r)) \approx r \forall r$

1: Initialize generator and critic weights $\rho, \mathbf{w}$, set initial return command $c = 0$
2: **repeat**
3: Sample policy parameters $\theta \sim g_\rho(\theta, c)$
4: Generate an episode $s_0, a_0, r_1, s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T$ with policy π_θ
5: Compute return $r = \sum_{k=1}^{T} r_k$
6: Store (r, θ) in the replay buffer D
7: **for** many steps **do**
8: Sample a batch $B = \{(r, \theta)\}$ from D
9: Update evaluator by stochastic gradient descent: $\nabla_\mathbf{w} \mathbb{E}_{(r,\theta) \in B}[(r - V_\mathbf{w}(\theta))^2]$
10: **end for**
11: **for** many steps **do**
12: Sample a batch $B = \{r\}$ from D
13: Update generator by stochastic gradient descent: $\nabla_\rho \mathbb{E}_{r \in B}[(r - V_\mathbf{w}(G_\rho(r)))^2]$
14: **end for**
15: Set next return command c using some strategy
16: **until** convergence

on the optimization process and on the shape of V. In other words, the multimodality issue is turned into having multiple optimal points in the optimization of the generator.

Choosing the Command The strategy of choosing the command c before interacting with the environment is important. Intuitively, asking the generator to produce low return policies will not necessarily help finding better policies. On the other hand, asking for too much will produce policies that are out of distribution, given the training data, and the generator cannot be trusted to produce such values. Hence it is reasonable to ask the generator to produce a return close to the highest one observed so far. More on command strategies can be found in Section 6.2.1.

Scaling to Deep Policies Both generating and evaluating the weights of a deep feedforward MLP-based policy is difficult for large policies. The sheer number of policy weights, as well as their lack of easily recognizable structure, requires special solutions for the generator and evaluator. To scale FWPs to deep policies, we rely on the relaxed weight-sharing of hypernetworks [Ha et al., 2016] for the generator, and on parameter-based value functions [Faccio et al., 2021] using a static fingerprinting mechanism [Harb et al., 2020; Faccio et al., 2022] for the evaluator. We discuss hypernetworks in the next section.

6.1.4 HyperNetworks

The idea behind certain feed-forward FWPs called hypernetworks [Ha et al., 2016] is to split the parameters of the generated network θ into smaller slices sl_l. A shared NN H with parameters $\xi \in \Xi \subset \mathbb{R}^{n_\Xi}$ receives as input a learned embedding z_l and outputs the slice sl_l for each l, i.e., $sl_l = H_\xi(z_l)$. Following von Oswald et al. [2020], further context information can be given to H in form of an additional conditioning input c, which can be either scalar or vector-valued: $sl_l = H_\xi(z_l, c)$. Then the weights are combined by concatenating all generated slices:

$$\theta = \begin{bmatrix} sl_1 & sl_2 & sl_3 & \dots \end{bmatrix}. \tag{6.4}$$

The splitting of θ into slices and the choice of H depend on the specific architecture of the generated policy. Here we are interested in generating MLP policies whose parameters θ consist of weight matrices K^j with $j \in \{1, 2, \dots, n_K\}$, where n_K is the policy's number of layers. We use an MLP H_ξ to generate each slice of each weight matrix: the hypernetwork generator G_ρ splits each weight matrix into slices $sl^j_{mn} \in \mathbb{R}^{f \times f}$, where j is the policy layer, and m, n are indexes of the slice in weight matrix of layer l. For each of these slices, a small embedding vector $z^j_{mn} \in \mathbb{R}^d$ is

learned. Our network H_ξ is an MLP, followed by a reshaping operation that turns a vector of size f^2 into an $f \times f$ matrix:

$$sl^j_{mn} = H_\xi(z^j_{mn}, c). \tag{6.5}$$

The slices are then concatenated over two dimensions to obtain the full weight matrices:

$$K^j = \begin{bmatrix} sl^j_{11} & sl^j_{12} & \cdots \\ sl^j_{21} & sl^j_{22} & \\ \vdots & & \ddots \end{bmatrix}. \tag{6.6}$$

The full hypernetwork generator G_ρ consists of the shared network H_ξ, as well as all embeddings z^j_{mn}. Its learnable parameters are $\rho = \{\xi, z^j_{mn} \forall m, n, j\}$.

Generator G_ρ is supposed to dynamically generate policy parameters, conditioned on the total return these policies should achieve. The conditioning input c is simply this scalar return command. It is appended to each learned slice embedding z^j_{mn}. The resulting vectors are the inputs to the network H. Figure 6.2 shows a diagram of this process.

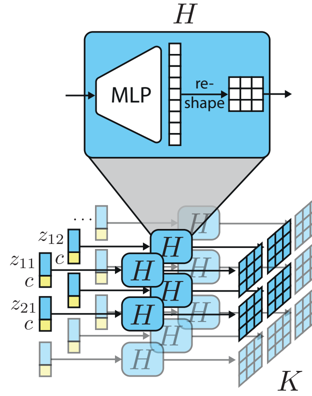


Figure 6.2: Generating a weight matrix K by concatenating slices that are generated from learned embeddings z and conditioning c using a shared network H.

For the the slicing to work, the widths and heights of the weight matrices have to be multiples of f. For the hidden layers of an MLP, this is easily achieved since we can freely choose the numbers of neurons. For the input and output layers, however, we are constrained by the dimensions of environmental observations and actions. To accommodate any number of input and output neurons, we use dedicated networks H_i and H_o for the input and output layers. The generated slices have the shape $f \times n_i$

for the input layer (n_i is the number of input neurons) and $n_o \times f$ for the output layer (n_o is the number of output neurons).

Figure 6.1 shows a diagram of our method with a hypernetwork generator and a fingerprinting value function.

6.2 Experiments and Results

We empirically evaluate GoGePo as follows: first, we show competitive performance on common continuous control problems. Then we use the the learned fingerprinting mechanism to visualize the policies created by the generator over the course of training, and investigate its learning behavior.

6.2.1 Results on Continuous Control RL Environments

We evaluate our method on continuous control tasks from the MuJoCo [Todorov et al., 2012] suite. Augmented Random Search (ARS) [Mania et al., 2018], a competitive parameter-based method, serves as a strong baseline. We also compare our method to other popular algorithms for continual control tasks: Deep Deterministic Policy Gradients (DDPG) [Silver et al., 2014], Soft Actor Critic (SAC) [Haarnoja et al., 2018a] and Twin Delayed Deep Deterministic Policy Gradients (TD3) [Fujimoto et al., 2018]. In addition, we include as a baseline UDRL [Srivastava et al., 2019] and confirm that UDRL is not sample efficient for continuous control in environments with episodic resets [Schmidhuber, 2019], in line with previous experimental results.

In the experiments, all policies are MLPs with two hidden layers, each having 256 neurons. Our method uses the same set of hyperparameters in all environments. For ARS and UDRL, we tune a set of hyperparameters separately for each environment (step size, population size, and noise for ARS; nonlinearity, learning rate and the "last few" parameter for UDRL). For DDPG, SAC and TD3, we use the established sets of default hyperparameters. Details can be found in Appendix D.1.

We find that while always asking to generate a policy with return equal to the best return ever seen, there is a slight advantage when asking for more than that. In particular, we empirically demonstrate that a simple strategy such as "produce a policy whose return is 20 above the one of the best policy seen so far" can be very effective. We present an ablation showing that this strategy is slightly better than the strategy "produce a policy whose return equal to the one of the best policy seen so far" in Appendix D.2.2. This suggests that our method's success is not only due to random exploration in parameter space but also to generalization over commands: *it*

learns to understand and exploit the nature of performance improvements in a given environment.

For our method and ARS, we use observation normalization (see [Mania et al., 2018; Faccio et al., 2021]). Furthermore, following ARS, the survival bonus of +1 for every timestep is removed for the Hopper-v3 environment, since for parameter-based methods it leads to the local optimum of staying alive without any movement. In tasks without fixed episode length, quickly failing bad policies from the early stages of training tend to dominate the replay buffer. To counteract this, we introduce a recency bias when sampling training batches from the buffer, assigning higher

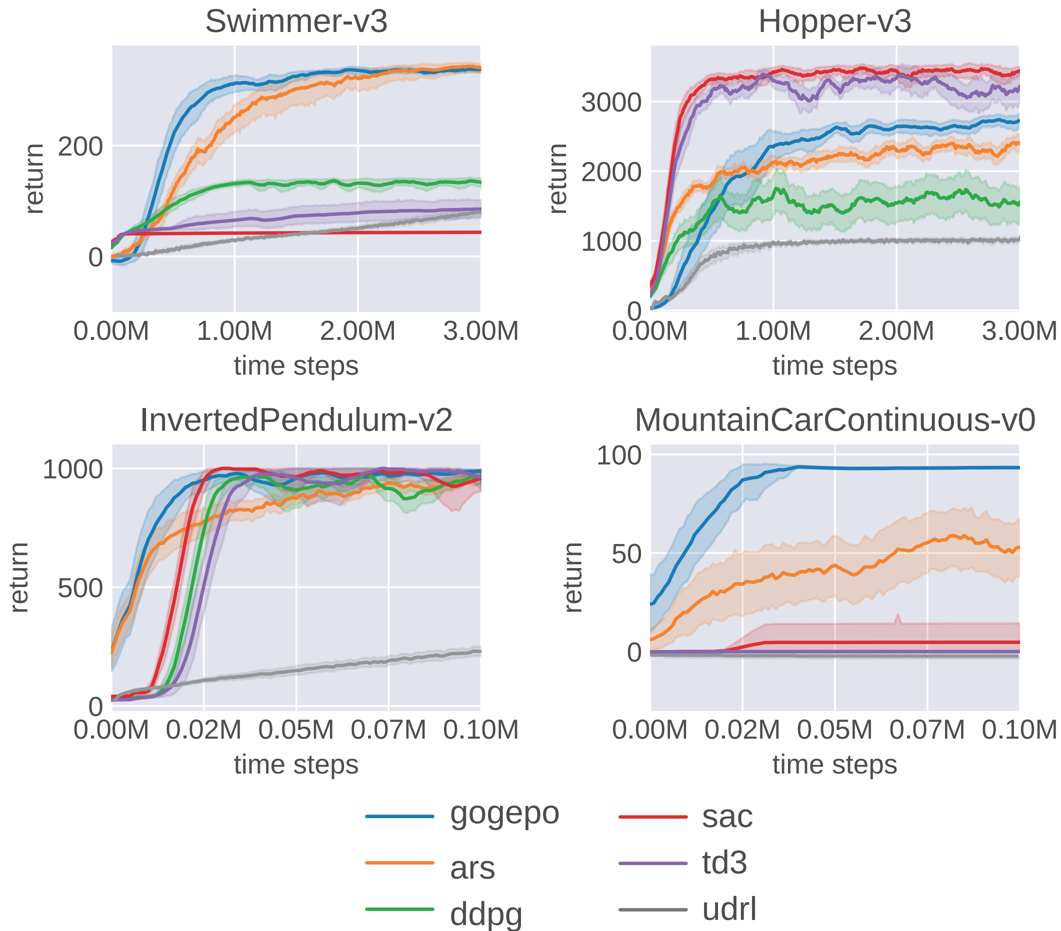


Figure 6.3: Performance of policies created with GoGePo (our method), ARS, DDPG, SAC, TD3 and UDRL over the course of training. Curves show the mean return and 95% bootstrapped confidence intervals from 20 runs as a function of total environment interactions.

probability to newer policies. It is treated as an additional hyperparameter. In Figure B.1 in Appendix B.2.2 we provide an ablation showing the importance of this component. Figure 6.3 shows our main experimental result (see also Table D.1 in Appendix D.2.1).

Our Algorithm 5 performs very competitively in the tested environments with the exception of Hopper, where TD3 and SAC achieve higher expected return. In Swimmer and Hopper environments, our method learns faster than ARS, while eventually reaching the same asymptotic performance. In MountainCarContinuous, DDPG, SAC and TD3 are unable to explore the action space, and parameter-based methods quickly learn the optimal policy. Our method always outperforms UDRL.

6.2.2 Analyzing the Generator's Learning Process

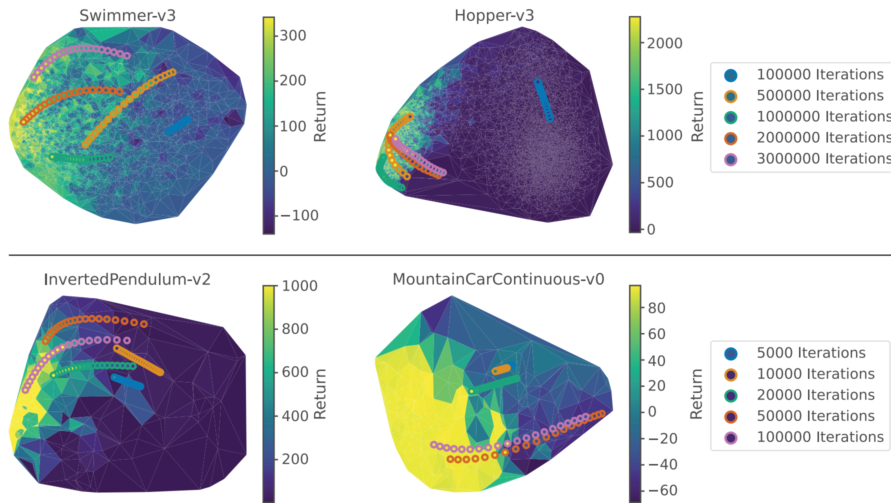


Figure 6.4: Policies generated by the generator during different stages of training. The background shows all policies executed during training (i.e., in the replay buffer), colored according to their returns. The 2D coordinates of the policies are determined by the PCA of their probing actions (obtained by the final critic $V_{\mathbf{w}}$). The chains of points show the policies created by the generator when given return commands ranging from the minimum (darker end of the chain) to the maximum (last point at the brighter end) possible return in the environment. Each chain represents a different stage of training, from almost untrained to fully trained. After training, the generator is able to produce policies across the whole performance spectrum.

The probing actions created by the fingerprinting mechanism of the value function $V_{\mathbf{w}}$ can be seen as a compact meaningful policy embedding useful to visualize policies

for a specific environment. In Figure 6.4 we apply PCA to probing actions to show all policies in the buffer after training, as well as policies created by the generator at different stages of training when given the same range of return commands. Policies are colored in line with achieved return. The generator's objective can be seen as finding a trajectory through policy space, defined by the return commands, connecting the lowest with the highest return. In Figure 6.4, this corresponds to a trajectory going from a dark to a bright area. Indeed, we observe that the generator starts out being confined to the dark region (producing only bad policies) and over the course of training finds a trajectory leading from the darkest (low return) to the brightest (high return) regions.

To create Figure 6.4, we perform Principal Component Analysis (PCA) on the probing actions of all policies in the buffer after training. The first two principal components indicate a policy's position in our visualization. Using Delaunay triangulation, we assign an area to every policy and color it according to its achieved return. We then take the generator at different stages of training (of the same run). Each of these generators is given a set of 20 commands, evenly spaced across the range of possible returns ($[-100, 365]$ for Swimmer, $[-100, 3000]$ for Hopper, $[0, 1000]$ for InvertedPendulum and $[-100, 100]$ for MountainCarContinuous). The resulting policies are plotted using probing actions on the probing states of the fully trained value function $V_{\mathbf{w}}$ (and the same PCA).

Figure 6.5 shows the the returns achieved by policies that are created by a fully trained generator when given a range of return commands. This highlights a feature of the policy generator: while most RL algorithms generate only the best-performing policy, our generator is in principle able to produce by command policies across the whole performance spectrum. For the environments Swimmer and Hopper (Figures 6.5a and 6.5b), this works in a relatively reliable fashion. In Hopper the return used does not include survival bonus. A return of 2000 without survival bonus corresponds roughly to a return of 3000 with survival bonus.

It is worth noting, however, that in some environments it is hard or even impossible to achieve every given intermediate return. This might be the case, for example, if the optimal policy is much simpler than a slightly sub-optimal one, or if a large reward is given once a goal state is reached. We can observe this effect for the environments InvertedPendulum and MountainCar—see Figures 6.5c and 6.5d. There the generator struggles to produce the desired identity of return command and achieved return—instead we get something closer to a step function. However, this does not prevent our method from quickly finding optimal policies in these environments.

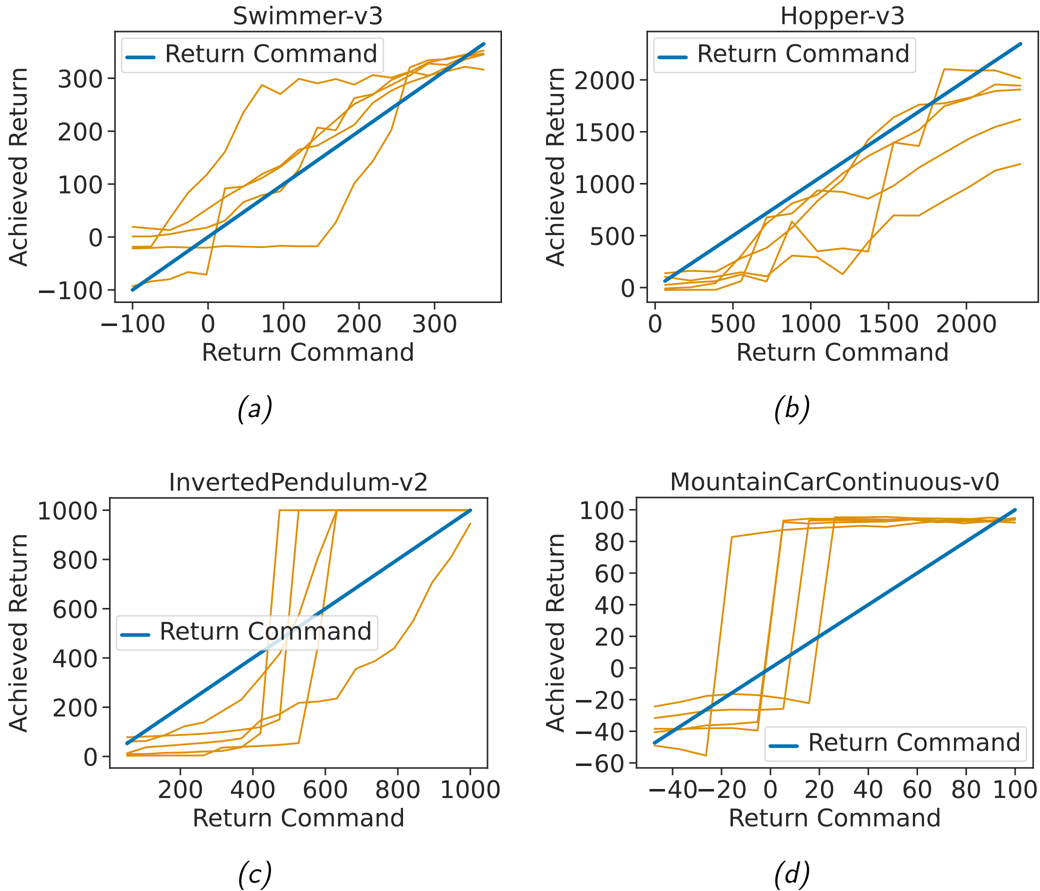


Figure 6.5: Achieved returns (mean of 10 episodes) of policies created by fully trained generators as a function of the given return command. A perfect generator would produce policies that lie on the diagonal identity line (if the environment permits such returns). For each environment, results of five independent runs are shown.

6.2.3 Limitation: obtaining suitable policies from the start

Randomly initialized policy generators produce weights far from those of typical initialization schemes. In particular, the standard PyTorch [Paszke et al., 2019] initialization is uniform in $[-1/\sqrt{n}, 1/\sqrt{n}]$, where n is the number of neurons in the previous layer, resulting in a distribution uniform in $[-0.0625, 0.0625]$ in the second and last layers. Our network tends to generate much larger weights, roughly uniform in every NN layer. We therefore scale our output such that it is close to the default initialization. Concretely, we multiply for each layer the output of the generator by $2/\sqrt{n}$, where n is the number of neurons in the previous layer. Here we provide

an ablation showing that this choice is crucial. Figure 6.6 shows the importance of scaling the output of the generator in Swimmer and Hopper. We compare this with and without weighted sampling from the replay buffer. We observe that in Swimmer, output scaling is very important, while in Hopper, most of the performance gain is due to weighted sampling. This choice of output scaling is rather heuristic and does not match the standard PyTorch initialization for all environments. It might happen that a randomly initialized generator produces policies that are difficult to perturb. This exploration issue seems to cause some difficulties for InvertedDoublePendulum, highlighting a possible limitation of our method.

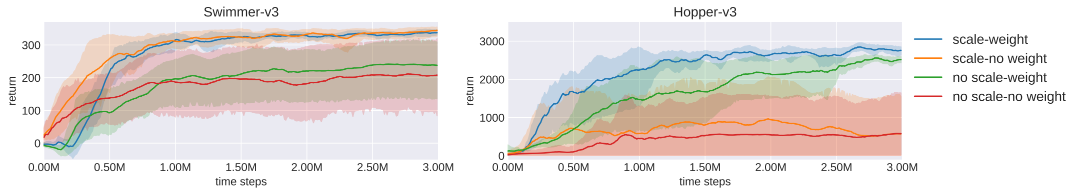


Figure 6.6: Comparison between our algorithm with/without weighted sampling from the replay buffer and output scaling. “No weight” denotes uniform sampling from the replay buffer. Average over 5 independent runs and 95% bootstrapped confidence intervals.

6.3 Related Work

6.3.1 Hindsight and Upside Down RL.

Upside Down RL (UDRL) transforms the RL problem into a supervised learning problem by conditioning the policy on commands such as “achieve a desired return” [Schmidhuber, 2019; Srivastava et al., 2019]. UDRL methods are related to hindsight RL where the commands correspond to desired goal states in the environment [Schmidhuber, 1991b; Kaelbling, 1993; Andrychowicz et al., 2017; Rauber et al., 2018]. In UDRL, just as in our method Gogepo, the required dataset of states, actions, and rewards is collected online during iterative improvements of the policy [Srivastava et al., 2019]. The conceptually highly related Decision Transformer (DT) [Janner et al., 2021]) is designed for offline RL and thus requires a dataset of experiences from policies trained using other methods. A recent DT variant called “Online DT” [Chen et al., 2021] alternates between an offline pretraining phase, using data already collected, and an online finetuning phase, where hindsight learning is used. Online DTs and UDRL suffer from the same multi-modality issues when fitting

data using unimodal distributions and a maximum likelihood approach. Our method learns online and does not rely on offline pretraining. It solves the multimodality issue of UDRL and outperforms it in terms of sample efficiency. Instead of optimizing the policy to achieve a desired reward in action space, our method Gogepo evaluates the generated policies in command space. This is done by generating, conditioning on a command, a policy that is then evaluated using a parameter-based value function and trained to match the command to the evaluated return. This side-steps the issue with multi-modality in certain types of UDRL for episodic environments, where a command may be achieved through many different behaviors, and fitting the policy to varying actions may lead to sub-optimal policies. Alternatively, the multimodality problem could be solved using a return-conditioned policy that directly outputs a multimodal action distribution, or is conditioned on random latent variables. To the best of our knowledge, this has not been tried for return-conditioned RL.

6.3.2 On the convergence of Upside Down RL

In a separate paper [Štrupl et al., 2022a], we proved the divergence of UDRL and Goal-Conditioned Supervised Learning (GCSL) in stochastic environments with episodic tasks. The proofs, primarily developed by Mirek Strupl, were not included in this thesis. Specifically, we showed that in a simple MDP, following UDRL's update, under certain simplifications, the sequence of learned policies fails to converge to the optimal policy. This optimal policy, capable of satisfying any command, resembles a standard policy in an augmented MDP, where UDRL's command and horizon are part of the agent's state space.

Our study of UDRL's recursive update also led to proving the convergence of the algorithm Reward-Weighted Regression (RWR) [Peters and Schaal, 2007] to a global optimum [Štrupl et al., 2022b]. RWR shares a similar recursive update mechanism with UDRL. We have not investigated the convergence properties of GoGePo. Our approach differs from that of Štrupl et al. [2022a] as it does not involve hindsight learning, making the counterexamples in his paper not applicable to our work.

6.3.3 Fast Weight Programmers and HyperNetworks

The idea of using a neural network (NN) to generate weight changes for another NN dates back to Fast Weight Programmers (FWPs) [Schmidhuber, 1992b, 1993], later scaled up to deeper neural networks under the name of hypernetworks [Ha et al., 2016]. While in traditional NNs the weight matrix remains fixed after training, FWPs make these weights context-dependent. More generally, FWPs can be used as neural functions that involve multiplicative interactions and parameter sharing [Kirsch and

Schmidhuber, 2021]. When updated in recurrent fashion, FWPs can be used as memory mechanisms. Linear transformers are a type of FWP where information is stored through outer products of keys and values [Schlag et al., 2021; Schmidhuber, 1992b]. FWPs are used in the context of memory-based meta learning [Schmidhuber, 1993; Miconi et al., 2018; Gregor, 2020; Kirsch and Schmidhuber, 2021; Irie et al., 2021a; Kirsch et al., 2022], predicting parameters for varying architectures [Knyazev et al., 2021], and reinforcement learning [Gomez and Schmidhuber, 2005; Najarro and Risi, 2020; Kirsch et al., 2022]. In contrast to all of these approaches, ours uses FWPs to conditionally generate policies given a command.

6.4 Discussion

Our GoGePo is an RL framework for generating policies yielding given desired returns. Hypernetworks in conjunction with fingerprinting-based value functions can be used to train a Fast Weight Programmer through supervised learning to directly generate parameters of a policy that achieves a given return. By iteratively asking for higher returns than those observed so far, our algorithm trains the generator to produce highly performant policies zero-shot. Empirically, GoGePo is competitive with ARS and DDPG and outperforms UDRL on continuous control tasks. It also circumvents the multi-modality issue found in many approaches of the UDRL family. Further, our approach can be used to generate policies with any desired return.

It remains uncertain whether learning a policy conditioned on a goal is better than learning a policy generator conditioned on the same goal. This dilemma shares its foundation with the question of whether it is more effective to learn a Recurrent Neural Network based on some contextual input versus learning a Fast Weight Programmer capable of generating the weights for a Neural Network given similar context. Our framework does not compare these two approaches directly; instead, it suggests employing the PSSVF to tackle the multi-modality challenges associated with UDRL. Exploring similar strategies, potentially utilizing the PSVF $V(s,\theta)$ to improve UDRL, represents an avenue for future research. There are two current limitations of our approach that we want to highlight: First, NNs created by an untrained generator might have weights that are far from typical initialization schemes. Exploration starting with such policies might be hard. Second, our method is based on the episodic return signal. Extending it by considering also the state of the agent might help to increase sample efficiency. Future work will also consider context commands other than those asking for particular returns, as well as generators based on latent variable models (e.g., conditional variational autoencoders) allowing for capturing diverse sets of policies, to improve exploration of complex RL environments.

Chapter 7

Outlook

This thesis introduced Parameter-Based Value Functions (PBVFs), which are value functions capable of incorporating policy parameters alongside states or state-action pairs as inputs. PBVFs excel in generalizing across a broad range of policies, rapidly zero-shot learning new policies, and demonstrating strong performance in continuous control tasks. We explored two methods for reducing policy dimensionality, namely static and recurrent policy fingerprinting. Static fingerprinting has been evaluated in Reinforcement Learning (RL) and goal-conditioned problems, aiming to develop deep policies that achieve specific returns in various environments. Our preliminary experiments with recurrent fingerprinting in supervised learning tasks have demonstrated its potential in learning effective representations for Recurrent Neural Network (RNN) weight matrices. The following paragraphs will present prospective research directions.

Recurrent fingerprinting for Reinforcement Learning Recurrent policy fingerprinting, having demonstrated promising results in supervised learning, is now well-positioned for significant applications in RL. The method's effectiveness stems from the ability of the PBVF to consistently query the policy using either the subsequent state or the most probable upcoming state encountered in the environment. Additionally, by unrolling the PBVF for fewer steps, it can input only the most crucial states to the policy. Such an approach is particularly effective in environments where understanding a policy's behavior across various states is key to accurately predicting its return. This is especially true in scenarios where different policies might experience vastly distinct states. However, the challenge in applying recurrent fingerprinting arises when we solely map policy parameters to expected returns. The PBVF tends to develop an implicit model of the environment based only on the loss associated with the expected return prediction error, which might be a weak learning

signal. To overcome this, PBVFs that incorporate states or state-action pairs, such as our PSVF $V(s,\theta)$ and our PAVF $Q(s,a,\theta)$, are anticipated to offer a richer learning signal, since they can give credit to every action taken. Future work will focus on learning our PBVFs with recurrent fingerprinting on complex RL tasks.

Recurrent Fingerprinting and Language In Chapter 5, we empirically demonstrated the use of recurrent fingerprinting to determine the language an LSTM was trained in. Initially, our experiments focused on formal languages, but they have the potential to be extended to tasks involving natural languages. In these tasks, probing states and actions would encompass complete sentences. Recurrent fingerprinting enables us to train neural networks to pose natural language questions, thereby extracting essential information from other neural networks, such as Large Language Models [Radford et al., 2019; Brown et al., 2020]. This approach is closely linked to the concept of "Learning To Think" [Schmidhuber, 2015b], which we have recently adapted for multiagent settings [Zhuge et al., 2023]. In "Learning to Think," the policy can learn to ask questions to a world model to solve a task efficiently. Here, however, it is the value function that is learning to ask questions to the policy to determine its behavior. Potential applications of this method include cloning language models, identifying harmful content, and detecting misalignments [Bai et al., 2022; Ouyang et al., 2022]. A significant advantage of using recurrent fingerprinting with natural language is its transparency and explainability, making the information extraction process directly understandable to humans.

Avoiding Backpropagation Through Time with PBVFs The traditional way of training an RNN by backpropagation through time (BPTT) or similar methods [Werbos, 1988; Williams and Zipser, 1994; Robinson and Fallside, 1987] is based on the runtime view of what the RNN does while it operates on a sequence of inputs. In the case of long time lags between relevant events, this leads to well-known issues (e.g., vanishing gradients [Hochreiter, 1991]) at the heart of all of deep learning [Schmidhuber, 2015a]. An alternative is to use our PBVFs, which allow us to view the RNN as a static object and optimize it directly using Algorithm 1, without having to resort to BPTT. In future work, we will train RNNs by directly minimizing the loss predicted by a PBVF.

Publications during the PhD program

Francesco Faccio, Louis Kirsch, and Jürgen Schmidhuber. Parameter-based value functions. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=tV6oBfuyLTQ.

Francesco Faccio, Aditya Ramesh, Vincent Herrmann, Jean Harb, and Jürgen Schmidhuber. General policy evaluation and improvement by learning to identify few but crucial states. *The 15th European Workshop on Reinforcement Learning*, 2022. URL https://arxiv.org/abs/2207.01566.

Francesco Faccio, Vincent Herrmann, Aditya Ramesh, Louis Kirsch, and Jürgen Schmidhuber. Goal-conditioned generators of deep policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 7503–7511, 2023. URL https://arxiv.org/abs/2207.01570.

Vincent Herrmann, Francesco Faccio, and Jürgen Schmidhuber. Learning useful representations of recurrent neural network weight matrices. In *NeurIPS 2023 Workshop on Symmetry and Geometry in Neural Representations*, 2023. URL https://openreview.net/forum?id=yqGoKziEvY.

Kazuki Irie, Francesco Faccio, and Jürgen Schmidhuber. Neural differential equations for learning to program neural nets through continuous learning rules. *Advances in Neural Information Processing Systems*, 35:38614–38628, 2022. URL https://arxiv.org/abs/2206.01649.

Haozhe Liu, Mingchen Zhuge, Bing Li, Yuhui Wang, Francesco Faccio, Bernard Ghanem, and Jürgen Schmidhuber. Learning to identify critical states for reinforcement learning from videos. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1955–1965, 2023. URL https://arxiv.org/abs/2308.07795.

Piotr Piękos, Aditya Ramesh, Francesco Faccio, and Jürgen Schmidhuber. Efficient value propagation with the compositional optimality equation. In *NeurIPS 2023 Workshop on Goal-Conditioned Reinforcement Learning*, 2023. URL https://openreview.net/forum?id=UyNjQ3UKO2.

Noor Sajid, Francesco Faccio, Lancelot Da Costa, Thomas Parr, Jürgen Schmidhuber, and Karl Friston. Bayesian brains and the rényi divergence. *Neural Computation*, 34(4):829–855, 2022. URL https://arxiv.org/abs/2107.05438.

Aleksandar Stanić, Dylan Ashley, Oleg Serikov, Louis Kirsch, Francesco Faccio, Jürgen Schmidhuber, Thomas Hofmann, and Imanol Schlag. The languini kitchen: Enabling language modelling research at different scales of compute. 2023. URL https://arxiv.org/abs/2309.11197.

Miroslav Štrupl, Francesco Faccio, Dylan R Ashley, Jürgen Schmidhuber, and Rupesh Kumar Srivastava. Upside-down reinforcement learning can diverge in stochastic environments with episodic resets. *The 15th European Workshop on Reinforcement Learning*, 2022a. URL https://arxiv.org/abs/2205.06595.

Miroslav Štrupl, Francesco Faccio, Dylan R Ashley, Rupesh Kumar Srivastava, and Jürgen Schmidhuber. Reward-weighted regression converges to a global optimum. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8361–8369, 2022b. URL https://arxiv.org/abs/2107.09088.

Wenyi Wang, Louis Kirsch, Francesco Faccio, Mingchen Zhuge, and Jürgen Schmidhuber. Continually adapting optimizers improve meta-generalization. In *NeurIPS 2023 Workshop on Optimization for Machine Learning*, 2023a. URL https://openreview.net/forum?id=Aw8GuIevIa.

Yuhui Wang, Haozhe Liu, Miroslav Strupl, Francesco Faccio, Qingyuan Wu, Xiaoyang Tan, and Jürgen Schmidhuber. Highway reinforcement learning. *Open Review*, 2023b. URL https://openreview.net/pdf?id=NFcRC4aYSWf.

Mingchen Zhuge, Haozhe Liu, Francesco Faccio, Dylan R Ashley, Róbert Csordás, Anand Gopalakrishnan, Abdullah Hamdi, Hasan Abed Al Kader Hammoud, Vincent Herrmann, Kazuki Irie, et al. Mindstorms in natural language-based societies of mind. In *NeurIPS 2023 Workshop on Robustness of Few-shot and Zero-shot Learning in Foundation Models*, 2023. URL https://arxiv.org/abs/2305.17066.

Appendix A

Parameter-Based Value Functions

A.1 Proofs and derivations

Theorem 3.1.1. *Let π_θ be stochastic. For any Markov Decision Process, the following holds:*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim d^{\pi_\theta}(s), a\sim\pi_\theta(.|s)}\left[(Q(s,a,\theta)\nabla_\theta \log \pi_\theta(a|s))\right]. \tag{3.1}$$

Proof. The proof follows the standard approach by Sutton et al. [1999] and we report it for completeness. We start by deriving an expression for $\nabla_\theta V(s,\theta)$:

$$\begin{aligned}
\nabla_\theta V(s,\theta) &= \nabla_\theta \int_{\mathscr{A}} \pi_\theta(a|s) Q(s,a,\theta)\,\mathrm{d}a = \int_{\mathscr{A}} \nabla_\theta \pi_\theta(a|s) Q(s,a,\theta) + \pi_\theta(a|s)\nabla_\theta Q(s,a,\theta)\,\mathrm{d}a \\
&= \int_{\mathscr{A}} \nabla_\theta \pi_\theta(a|s) Q(s,a,\theta) + \pi_\theta(a|s)\nabla_\theta \left(R(s,a) + \gamma \int_{\mathscr{S}} P(s'|s,a)V(s',\theta)\,\mathrm{d}s'\right)\mathrm{d}a \\
&= \int_{\mathscr{A}} \nabla_\theta \pi_\theta(a|s) Q(s,a,\theta) + \pi_\theta(a|s)\gamma \int_{\mathscr{S}} P(s'|s,a)\nabla_\theta V(s',\theta)\,\mathrm{d}s'\,\mathrm{d}a \\
&= \int_{\mathscr{A}} \nabla_\theta \pi_\theta(a|s) Q(s,a,\theta) + \pi_\theta(a|s)\gamma \int_{\mathscr{S}} P(s'|s,a) \times \\
&\quad \times \int_{\mathscr{A}} \nabla_\theta \pi_\theta(a'|s') Q(s',a',\theta) + \pi_\theta(a'|s')\gamma \int_{\mathscr{S}} P(s''|s',a')\nabla_\theta V(s'',\theta)\,\mathrm{d}s''\,\mathrm{d}a'\,\mathrm{d}s'\,\mathrm{d}a \\
&= \int_{\mathscr{S}} \sum_{t=0}^{\infty} \gamma^t P(s\to s',t,\pi_\theta) \int_{\mathscr{A}} \nabla_\theta \pi_\theta(a|s') Q(s',a,\theta)\,\mathrm{d}a\,\mathrm{d}s'.
\end{aligned}$$

Taking the expectation with respect to $s_0 \sim \mu_0(s)$ we have:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \int_{\mathscr{S}} \mu_0(s) V(s,\theta)\,\mathrm{d}s = \int_{\mathscr{S}} \mu_0(s) \nabla_\theta V(s,\theta)\,\mathrm{d}s \\
&= \int_{\mathscr{S}} \mu_0(s) \int_{\mathscr{S}} \sum_{t=0}^{\infty} \gamma^t P(s \to s', t, \pi_\theta) \int_{\mathscr{A}} \nabla_\theta \pi_\theta(a|s) Q(s,a,\theta)\,\mathrm{d}s'\,\mathrm{d}a\,\mathrm{d}s \\
&= \int_{\mathscr{S}} d^{\pi_\theta}(s) \int_{\mathscr{A}} \nabla_\theta \pi_\theta(a|s) Q(s,a,\theta)\,\mathrm{d}a\,\mathrm{d}s \\
&= \mathbb{E}_{s \sim d^{\pi_\theta}(s), a \sim \pi_\theta(.|s)} \left[(Q(s,a,\theta) \nabla_\theta \log \pi_\theta(a|s)) \right].
\end{aligned}$$

□

Theorem 3.1.2. *Let π_θ be deterministic. Under standard regularity assumptions [Silver et al., 2014], for any Markov Decision Process, the following holds:*

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s \sim d^{\pi_\theta}(s)} \left[\nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s) \right]. \tag{3.2}$$

Proof. The proof follows the standard approach by Silver et al. [2014] and we report it for completeness. We start by deriving an expression for $\nabla_\theta V(s,\theta)$:

$$\begin{aligned}
\nabla_\theta V(s,\theta) &= \nabla_\theta Q(s,\pi_\theta(s),\theta) = \nabla_\theta \left(R(s,\pi_\theta(s)) + \gamma \int_{\mathscr{S}} P(s'|s,\pi_\theta(s)) V(s',\theta)\,\mathrm{d}s' \right) \\
&= \nabla_\theta \pi_\theta(s) \nabla_a R(s,a)|_{a=\pi_\theta(s)} + \\
&+ \gamma \int_{\mathscr{S}} P(s'|s,\pi_\theta(s)) \nabla_\theta V(s',\theta) + \nabla_\theta \pi_\theta(s) \nabla_a P(s'|s,a)|_{a=\pi_\theta(s)}\,\mathrm{d}s' \\
&= \nabla_\theta \pi_\theta(s) \nabla_a \left(R(s,a) + \gamma \int_{\mathscr{S}} P(s'|s,a) V(s',\theta)\,\mathrm{d}s' \right) |_{a=\pi_\theta(s)} + \\
&+ \gamma \int_{\mathscr{S}} P(s'|s,\pi_\theta(s)) \nabla_\theta V(s',\theta)\,\mathrm{d}s' \\
&= \nabla_\theta \pi_\theta(s) \nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)} + \gamma \int_{\mathscr{S}} P(s'|s,\pi_\theta(s)) \nabla_\theta V(s',\theta)\,\mathrm{d}s' \\
&= \nabla_\theta \pi_\theta(s) \nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)} + \\
&+ \gamma \int_{\mathscr{S}} P(s'|s,\pi_\theta(s)) \nabla_\theta \pi_\theta(s') \nabla_a Q(s',a,\theta)|_{a=\pi_\theta(s')}\,\mathrm{d}s' + \\
&+ \gamma \int_{\mathscr{S}} P(s'|s,\pi_\theta(s)) \gamma \int_{\mathscr{S}} P(s''|s',\pi_\theta(s')) \nabla_\theta V(s'',\theta)\,\mathrm{d}s''\,\mathrm{d}s' \\
&= \int_{\mathscr{S}} \sum_{t=0}^{\infty} \gamma^t P(s \to s', t, \pi_\theta) \nabla_\theta \pi_\theta(s') \nabla_a Q(s',a,\theta)|_{a=\pi_\theta(s')}\,\mathrm{d}s'
\end{aligned}$$

Taking the expectation with respect to $s_0 \sim \mu_0(s)$ we have:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \int_{\mathscr{S}} \mu_0(s) V(s,\theta)\,\mathrm{d}s = \int_{\mathscr{S}} \mu_0(s) \nabla_\theta V(s,\theta)\,\mathrm{d}s \\
&= \int_{\mathscr{S}} \mu_0(s) \int_{\mathscr{S}} \sum_{t=0}^{\infty} \gamma^t P(s \to s', t, \pi_\theta) \nabla_\theta \pi_\theta(s') \nabla_a Q(s', a, \theta)|_{a=\pi_\theta(s')}\,\mathrm{d}s'\,\mathrm{d}s \\
&= \int_{\mathscr{S}} d^{\pi_\theta}(s) \nabla_\theta \pi_\theta(s) \nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)}\,\mathrm{d}s \\
&= \mathbb{E}_{s\sim d^{\pi_\theta}(s)}\left[\nabla_\theta \pi_\theta(s) \nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)}\right]
\end{aligned}$$

□

Theorem 3.1.3. *For any Markov Decision Process, the following holds:*

$$\nabla_\theta J_b(\pi_\theta) = \mathbb{E}_{s\sim d_\infty^{\pi_b}(s), a\sim \pi_b(.|s)}\left[\frac{\pi_\theta(a|s)}{\pi_b(a|s)}\left(Q(s,a,\theta)\nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q(s,a,\theta)\right)\right]. \tag{3.7}$$

Proof.

$$\begin{aligned}
\nabla_\theta J_b(\pi_\theta) &= \nabla_\theta \int_{\mathscr{S}} d_\infty^{\pi_b}(s) V(s,\theta)\,\mathrm{d}s && \text{(A.1)} \\
&= \nabla_\theta \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \int_{\mathscr{A}} \pi_\theta(a|s) Q(s,a,\theta)\,\mathrm{d}a\,\mathrm{d}s && \text{(A.2)} \\
&= \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \int_{\mathscr{A}} [Q(s,a,\theta)\nabla_\theta \pi_\theta(a|s) + \pi_\theta(a|s)\nabla_\theta Q(s,a,\theta)]\,\mathrm{d}a\,\mathrm{d}s && \text{(A.3)} \\
&= \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \int_{\mathscr{A}} \frac{\pi_b(a|s)}{\pi_b(a|s)} \pi_\theta(a|s)[Q(s,a,\theta)\nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q(s,a,\theta)]\,\mathrm{d}a\,\mathrm{d}s && \text{(A.4)} \\
&= \mathbb{E}_{s\sim d_\infty^{\pi_b}(s), a\sim \pi_b(.|s)}\left[\frac{\pi_\theta(a|s)}{\pi_b(a|s)}\left(Q(s,a,\theta)\nabla_\theta \log \pi_\theta(a|s) + \nabla_\theta Q(s,a,\theta)\right)\right] && \text{(A.5)}
\end{aligned}$$

□

Theorem 3.1.4. *Under standard regularity assumptions [Silver et al., 2014], for any Markov Decision Process, the following holds:*

$$\nabla_\theta J_b(\pi_\theta) = \mathbb{E}_{s\sim d_\infty^{\pi_b}(s)}\left[\nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s) + \nabla_\theta Q(s,a,\theta)|_{a=\pi_\theta(s)}\right]. \tag{3.9}$$

Proof.

$$\nabla_\theta J_b(\pi_\theta) = \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \nabla_\theta Q(s, \pi_\theta(s), \theta)\,\mathrm{d}s \tag{A.6}$$

$$= \int_{\mathscr{S}} d_\infty^{\pi_b}(s) \left[\nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s) + \nabla_\theta Q(s,a,\theta)|_{a=\pi_\theta(s)} \right] \mathrm{d}s \tag{A.7}$$

$$= \mathbb{E}_{s \sim d_\infty^{\pi_b}(s)} \left[\nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)} \nabla_\theta \pi_\theta(s) + \nabla_\theta Q(s,a,\theta)|_{a=\pi_\theta(s)} \right] \tag{A.8}$$

□

A.2 Implementation details

In this appendix, we report the implementation details for PSSVF, PSVF, PAVF and the baselines. We specify for each hyperparameter, which algorithms and tasks are sharing them.

Shared hyperparameters:

- Deterministic policy architecture (continuous control tasks): We use three different deterministic policies: a linear mapping between states and actions; a single-layer MLP with 32 neurons and tanh activation; a 2-layers MLP (64,64) with tanh activations. All policies contain a bias term and are followed by a tanh nonlinearity in order to bound the action.

- Deterministic policy architecture (discrete control tasks): We use three different deterministic policies: a linear mapping between states and a probability distribution over actions; a single-layer MLP with 32 neurons and tanh activation; a 2-layers MLP (64,64) with tanh activations. The deterministic action a is obtained choosing $a = \arg\max \pi_\theta(a|s)$. All policies contain a bias term.

- Stochastic policy architecture (continuous control tasks): We use three different stochastic policies: a linear mapping; a single-layer MLP with 32 neurons and tanh activation; a 2-layers MLP (64,64) with tanh activations all mapping from states to the mean of a Normal distribution. The variance is state-independent and parametrized as $e^{2\Omega}$ with diagonal Ω. All policies contain a bias term. Actions sampled are given as input to a tanh nonlinearity in order to bound them in the action space.

- Stochastic policy architecture (discrete control tasks): We use three different deterministic policies: a linear mapping between states and a probability distribution over actions; a single-layer MLP with 32 neurons and tanh activation; a 2-layers MLP (64,64) with tanh activations. All policies contain a bias term.
- Policy initialization: all weights and biases are initialized using the default Pytorch initialization for PBVFs and DDPG and are set to zero for ARS.
- Critic architecture: 2-layers MLP (512,512) with bias and ReLU activation functions for PSVF, PAVF; 2-layers MLP (256,256) with bias and ReLU activation functions for DDPG.
- Critic initialization: all weights and biases are initialized using the default Pytorch initialization for PBVFs and DDPG.
- Batch size: 128 for DDPG, PSVF, PAVF; 16 for PSSVF.
- Actor's frequency of updates: every episode for PSSVF; every batch of episodes for ARS; every 50 time steps for DDPG, PSVF, PAVF.
- Critic's frequency of updates: every episode for PSSVF; every 50 time steps for DDPG, PSVF, PAVF.
- Replay buffer: the size is 100k; data are sampled uniformly.
- Optimizer: Adam for PBVFs and DDPG.

Tuned hyperparameters:

- Number of directions and elite directions for ARS ([directions, elite directions]): tuned with values in $[[1,1],[4,1],[4,4],[16,1],[16,4],[16,16]]$.
- Policy's learning rate: tuned with values in $[1e-2, 1e-3, 1e-4]$.
- Critic's learning rate: tuned with values in $[1e-2, 1e-3, 1e-4]$.
- Noise for exploration: the perturbation for the action (DDPG) or the parameter is sampled from $\mathcal{N}(0, \sigma I)$ with σ tuned with values in $[1, 1e-1]$ for PSSVF, PSVF, PAVF; $[1e-1, 1e-2]$ for DDPG; $[1, 1e-1, 1e-2, 1e-3]$ for ARS. For stochastic PSSVF and PSVF we include also the value $\sigma = 0$, although it almost never results optimal.

Environment hyperparameters:

- Environment interactions: 1M time steps for Swimmer-v3 and Hopper-v3; 100k time steps for all other environments.
- Discount factor for TD algorithms: 0.999 for Swimmer; 0.99 for all other environments.
- Survival reward in Hopper: True for DDPG, PSVF, PAVF; False for ARS, PSSVF.

Algorithm-specific hyperparameters:

- Critic's number of updates: 50 for DDPG, 5 for PSVF and PAVF; 10 for PSSVF.
- Actor's number of updates: 50 for DDPG, 1 for PSVF and PAVF; 10 for PSSVF.
- Observation normalization: False for DDPG; True for all other algorithms.
- Starting steps in DDPG (random actions and no training): first 1%.
- Polyak parameter in DDPG: 0.995.

PAVF $\nabla_\theta Q(s,a,\theta)$ ablation We investigate the effect of the term $\nabla_\theta Q(s,a,\theta)$ in the off-policy policy gradient theorem for deterministic PAVF. We follow the same methodology as in our main experiments to find the optimal hyperparameters when updating using the now biased gradient:

$$\nabla_\theta J_b(\pi_\theta) \approx \mathbb{E}_{s\sim d_\infty^{\pi_b}(s)}\left[\nabla_a Q(s,a,\theta)|_{a=\pi_\theta(s)}\nabla_\theta \pi_\theta(s)\right], \tag{A.9}$$

which corresponds to the gradient that DDPG is following. Figure A.1 reports the results for Hopper and Swimmer using shallow and deep policies. We observe a significant drop in performance in Swimmer when removing part of the gradient. In Hopper the loss of performance is less significant, possibly because both algorithms tend to converge to the same sub-optimal behavior.

ARS For ARS, we use the official implementation provided by the authors and we modified it in order to use nonlinear policies. More precisely, we adopt the implementation of ARSv2-t [Mania et al., 2018], which uses observation normalization, elite directions and an adaptive learning rate based on the standard deviation of the return collected. To avoid divisions by zero, which may happen if all data sampled have the same return, we perform the standardization only in case the standard deviation is not

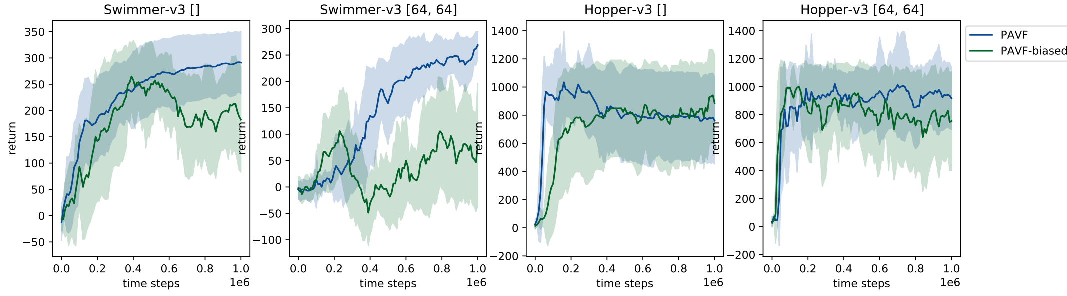


Figure A.1: Performance of PAVF and biased PAVF (PAVF without the gradient of the action-value function with respect to the policy parameters) using deterministic policies. We use the hyperparameters maximizing average return and report the best hyperparameters found for the biased version in Table A.1. Learning curves are averaged over 20 seeds.

Learning rate policy	Policy:	[]	[64,64]
	Metric:	avg	avg
Swimmer-v3		1e-3	1e-4
Hopper-v3		1e-4	1e-4
Learning rate critic			
Swimmer-v3		1e-4	1e-4
Hopper-v3		1e-3	1e-3
Noise for exploration			
Swimmer-v3		1.0	1.0
Hopper-v3		0.1	0.1

Table A.1: Table of best hyperparameters for biased PAVFs

zero. In the original implementation of ARS [Mania et al., 2018], the survival bonus for the reward in the Hopper environment is removed to avoid local minima. Since we want our PSSVF to be close to their setting, we also apply this modification. We do not remove the survival bonus from all TD algorithms, and we have not investigatet how this could affect their performance. We provide a comparison of the performance of PSSVF with and without the bonus in figure A.2 using deterministic policies.

DDPG For DDPG, we use the Spinning Up implementation provided by OpenAI [Achiam, 2018], which includes target networks for the actor and the critic and no learning for a fixed set of time steps, called starting steps. We do not include target networks and starting steps in our PBVFs, although they could potentially help

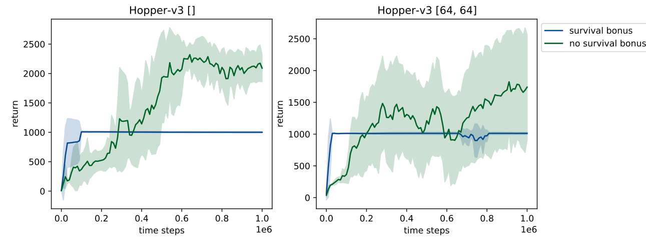


Figure A.2: Performance of PSSVF with and without the survival bonus for the reward in Hopper-v3 when using the hyperparameters maximizing the average return. Learning curves are averaged over 5 seeds.

stabilize training. The implementation of DDPG that we use [Achiam, 2018] does not use observation normalization. In preliminary experiments, we observed that it failed to increase or decrease performance significantly; hence, we decided not to use it. Another difference between our TD algorithms and DDPG consists in the number of updates of the actor and the critic. Since DDPG's critic needs to keep track of the current policy, the critic and the actor are updated in a nested form, with the actor's update depending on the critic's and vice versa. Our PSVF and PAVF do not need to track the policy learned, hence, when it is time to update, we need only to train once the critic for many gradient steps and then train the actor for many gradient steps. This requires less compute. On the other hand, when using nonlinear policies, our PBVFs suffer the curse of dimensionality. For this reason, we profited from using a bigger critic. In preliminary experiments, we observed that DDPG's performance did not change significantly through a bigger critic. We show differences in performance for our methods when removing observation normalization and when using a smaller critic (MLP(256,256)) in figure A.3. We observe that the performance decreases if observation normalization is removed. However, only for shallow policies in Swimmer and deep policies in Hopper, there seems to be a significant benefit. Future work will assess when bigger critics help.

Discounting in Swimmer For TD algorithms, we choose a fixed discount factor $\gamma = 0.99$ for all environments but Swimmer-v3. This environment is known to be challenging for TD-based algorithms because discounting causes the agents to become too short-sighted. We observe that, with the standard discounting, DDPG, PSVF and PAVF were not able to learn the task. However, making the algorithms more far-sighted greatly improved their performance. In Figure A.4, we report the return obtained by DDPG, PSVF, and PAVF for different values of the discount factor in Swimmer when using deterministic policies.

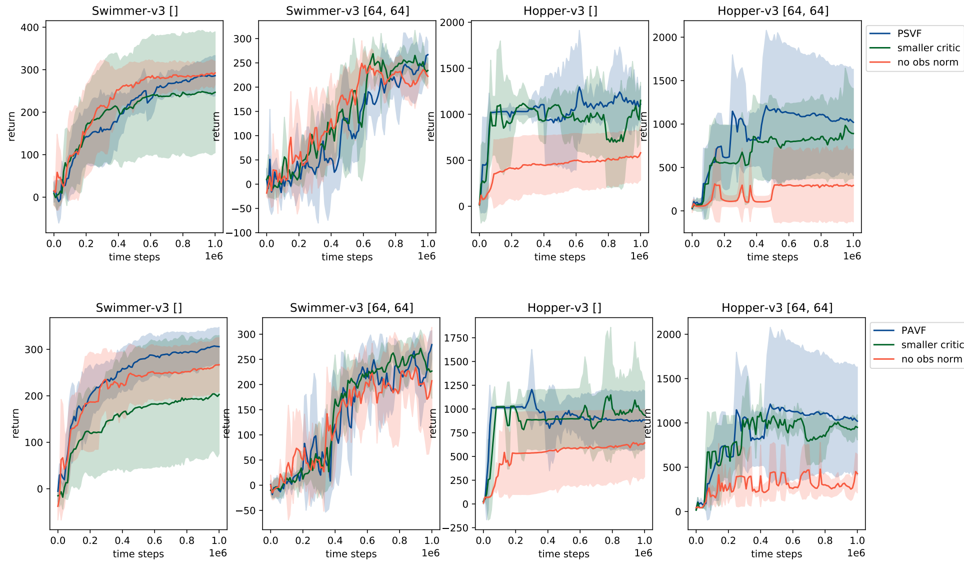


Figure A.3: Learning curves for PSVF and PAVF for different environments and policies removing observation normalization and using a smaller critic. We use the hyperparameters maximizing the average return. Learning curves are averaged over 5 seeds. For this ablation we use deterministic policies.

A.3 Experimental details

A.3.1 LQR

PSSVF We use a learning rate of $1e-3$ for the policy and $1e-2$ for the PSSVF. Weights are perturbed every episode using $\sigma = 0.5$. The policy is initialized with weight 3.2 and bias -3.5. All the other hyperparameters are set to their default. The true episodic $J(\theta)$ is computed by running 10,000 policies in the environment with parameters in $[-5,5] \times [-5,5]$. $V_w(\theta)$ is computed by measuring the output of the PSSVF on the same set of policies. Each red arrow in figure 3.1 represents 200 update steps of the policy.

PSVF and PAVF Using the exact same setting, we run PSVF and PAVF in LQR environment and we compare learned $V(s_0, \theta)$ and $Q(s_0, \pi_\theta(s_0), \theta)$ with the true PSVF and PAVF over the parameter space. Computing the value of the true PSVF and PAVF requires computing the infinite sum of discounted reward obtained by the policy. Here we approximate it by running 10,000 policies in the environment with parameters in $[-5,-5] \times [-5,5]$ for 500 time steps. This, setting $\gamma = 0.99$, provides a good approximation of their true values, since further steps in the environment

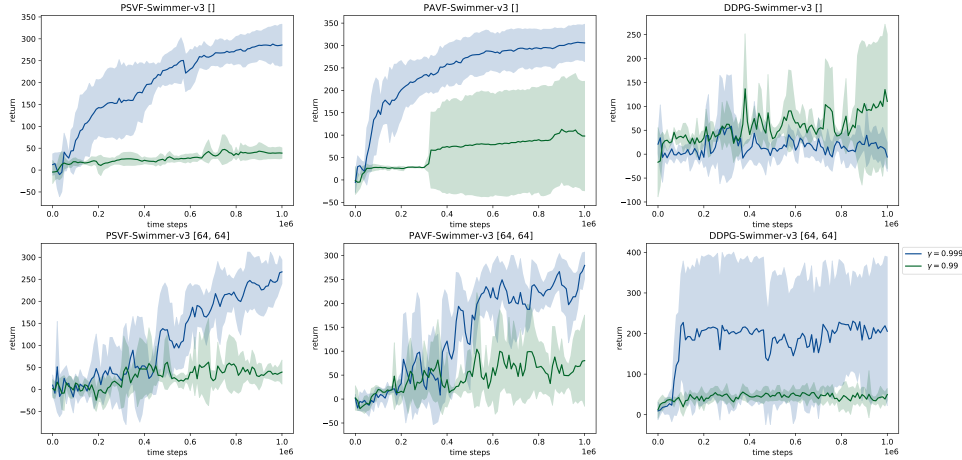


Figure A.4: Effect of different choices of the discount factor in Swimmer-v3 for PSVF, PAVF and DDPG, with shallow and deep deterministic policies. We use the hyperparameters maximizing the average return. Learning curves are averaged over 5 seeds

result in almost zero discounted reward from s_0. We use a learning rate of $1e-2$ for the policy and $1e-1$ for the PSVF and PAVF. Weights are perturbed every episode using $\sigma = 0.5$. The policy is updated every 10 time steps using 2 gradient steps; the PSVF and PAVF are updated every 10 time steps using 10 gradient updates. The critic is a 1-layer MLP with 64 neurons and tanh nonlinearity.

A.3.2 Main Experiments

Methodology In order to ensure a fair comparison of our methods and the baselines, we adopt the following procedure. For each hyperparameter configuration, for each environment and policy architecture, we run 5 instances of the learning algorithm using different seeds. We measure the learning progress by running 100 evaluations while learning the deterministic policy (without action or parameter noise) using 10 test trajectories. We use two metrics to determine the best hyperparameters: the average return over policy evaluations during the whole training process and the average return over policy evaluations during the last 20% time steps. For each algorithm, environment and policy architecture, we choose the two hyperparameter configurations maximizing the performance of the two metrics and test them on 20 new seeds, reporting average and final performance in table A.2 and A.3 respectively.

Figures A.5 and A.6 report all the learning curves from the main results and for a small non linear policy with 32 hidden neurons.

A.3.3 Sensitivity analysis

In the following, we report the sensitivity plots for all algorithms, for all deterministic policy architectures and environments. In particular, figure A.7, A.8, A.9, A.10 and A.11 show the performance of each algorithm given different hyperparameters tried during training. We observe that in general deep policies are more sensitive and, apart for DDPG, achieve often a better performance than smaller policies. The higher sensitivity displayed by ARS is in part caused by the higher number of hyperparameters we tried when tuning the algorithm.

A.3.4 Table of best hyperparameters

We report for each algorithm, environment, and policy architecture the best hyperparameters found when optimizing for average return or final return in tables A.4, A.5, A.6, A.7, A.8 and A.9.

Policy: []	MountainCar Continuous-v0	Inverted Pendulum-v2	Reacher -v2	Swimmer -v3	Hopper -v3
ARS	63 ± 6	886 ± 72	-9.2 ± 0.3	228 ± 89	1184 ± 345
PSSVF	85 ± 4	944 ± 33	-11.7 ± 0.9	259 ± 47	1392 ± 287
DDPG	0 ± 0	612 ± 169	-8.6 ± 0.9	95 ± 112	629 ± 145
PSVF	84 ± 20	926 ± 34	-19.7 ± 6.0	188 ± 71	917 ± 249
PAVF	82 ± 21	913 ± 40	-17.0 ± 7.7	231 ± 56	814 ± 223
Policy:[32]					
ARS	37 ± 11	851 ± 46	-9.6 ± 0.3	139 ± 78	1003 ± 66
PSSVF	60 ± 33	701 ± 138	10.4 ± 0.5	189 ± 35	707 ± 668
DDPG	0 ± 0	816 ± 36	-5.7 ± 0.3	61 ± 32	1384 ± 125
PSVF	71 ± 25	529 ± 281	-11.9 ± 1.2	226 ± 33	864 ± 272
PAVF	71 ± 27	563 ± 228	-10.9 ± 1.1	222 ± 28	793 ± 322
Policy: [64,64]					
ARS	28 ± 8	812 ± 239	-9.8 ± 0.3	129 ± 68	964 ± 47
PSSVF	72 ± 22	850 ± 93	-10.7 ± 0.2	158 ± 59	922 ± 568
DDPG	0 ± 0	834 ± 36	-5.5 ± 0.4	92 ± 117	767 ± 627
PSVF	80 ± 9	580 ± 107	-10.7 ± 0.6	137 ± 38	843 ± 282
PAVF	73 ± 10	399 ± 219	-10.7 ± 0.5	142 ± 26	875 ± 136

Policy: []	Acrobot-v1	CartPole-v1
ARS	-161 ± 23	476 ± 13
PSSVF	-137 ± 14	443 ± 105
PSVF	-148 ± 25	459 ± 28
Policy:[32]		
ARS	-296 ± 38	395 ± 141
PSSVF	-251 ± 80	463 ± 18
PSVF	-270 ± 113	413 ± 61
Policy: [64,64]		
ARS	-335 ± 35	416 ± 105
PSSVF	-281 ± 117	452 ± 34
PSVF	-397 ± 71	394 ± 71

Table A.2: Average return with standard deviation (across 20 seeds) for hyperparameters optimizing the average return during training using deterministic policies. Square brackets represent the number of neurons per layer of the policy. [] represents a linear policy.

Policy: []	MountainCar Continuous-v0	Inverted Pendulum-v2	Reacher -v2	Swimmer -v3	Hopper -v3
ARS	73 ± 5	657 ± 477	-8.6 ± 0.5	334 ± 34	1443 ± 713
PSSVF	84 ± 28	970 ± 126	-10.0 ± 1.0	350 ± 8	1560 ± 911
DDPG	0 ± 1	777 ± 320	-7.3 ± 0.4	146 ± 152	704 ± 234
PSVF	76 ± 36	906 ± 289	-16.5 ± 1.6	238 ± 107	1067 ± 340
PAVF	68 ± 42	950 ± 223	-17.2 ± 15.4	298 ± 40	720 ± 281
Policy:[32]					
ARS	54 ± 20	936 ± 146	-9.2 ± 0.4	239 ± 117	1048 ± 68
PSSVF	89 ± 22	816 ± 234	-10.2 ± 1.0	294 ± 41	1204 ± 615
DDPG	0 ± 0	703 ± 283	-4.6 ± 0.6	179 ± 150	1290 ± 348
PSVF	84 ± 31	493 ± 462	-11.3 ± 0.8	290 ± 70	1003 ± 572
PAVF	92 ± 7	854 ± 295	-10.1 ± 0.9	307 ± 34	967 ± 411
Policy: [64,64]					
ARS	11 ± 30	976 ± 83	-9.4 ± 0.4	157 ± 54	1006 ± 47
PSSVF	91 ± 16	898 ± 227	-10.7 ± 0.6	224 ± 99	1412 ± 691
DDPG	0 ± 0	943 ± 73	-4.4 ± 0.4	196 ± 151	1437 ± 752
PSVF	93 ± 1	1000 ± 0	-10.6 ± 1.0	257 ± 26	1247 ± 344
PAVF	93 ± 2	827 ± 267	-10.6 ± 0.4	232 ± 42	1005 ± 155

Policy: []	Acrobot-v1	CartPole-v1
ARS	-126 ± 26	499 ± 2
PSSVF	-97 ± 6	482 ± 53
PSVF	-100 ± 18	500 ± 0
Policy:[32]		
ARS	-215 ± 97	471 ± 110
PSSVF	-116 ± 33	500 ± 0
PSVF	-244 ± 151	488 ± 36
Policy: [64,64]		
ARS	-182 ± 45	492 ± 18
PSSVF	-233 ± 139	500 ± 0
PSVF	-406 ± 51	499 ± 2

Table A.3: Final return with standard deviation (across 20 seeds) for hypermarameters optimizing the final return during training using deterministic policies.

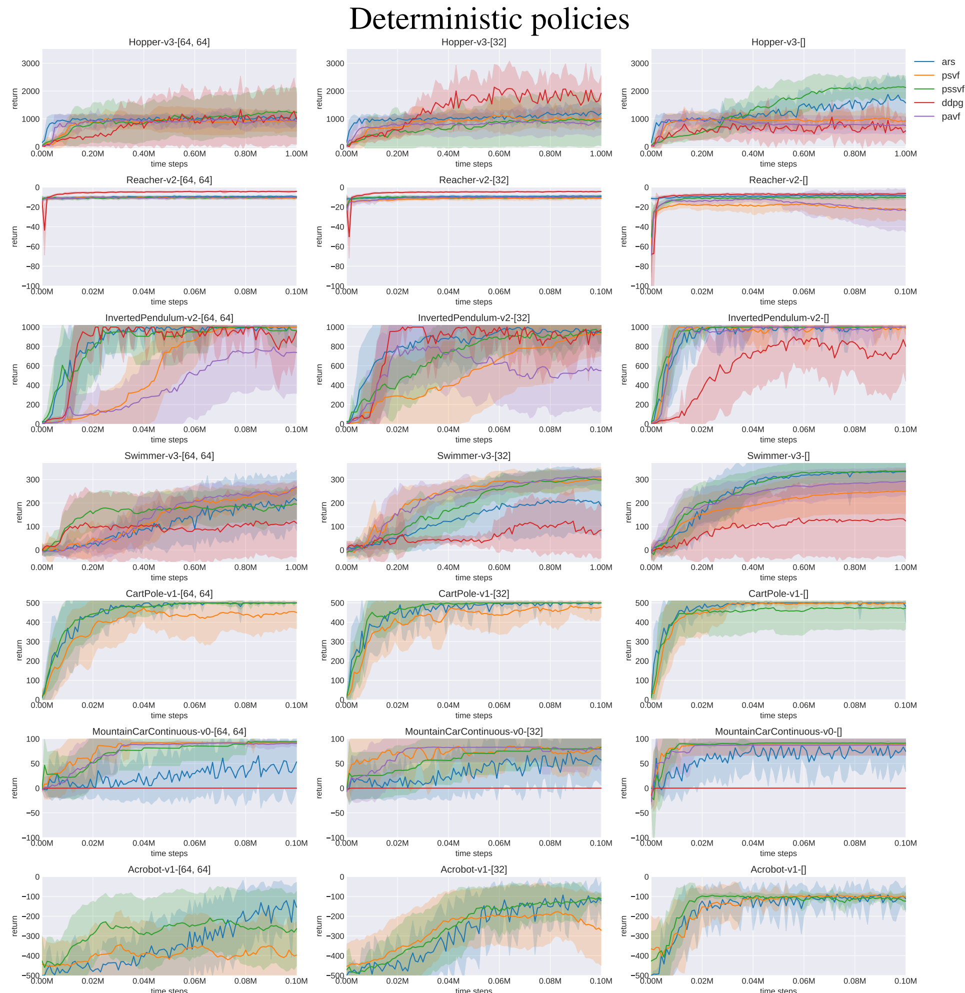


Figure A.5: Learning curves representing the average return as a function of the number of time steps in the environment (across 20 runs) with different environments and deterministic policy architectures. We use the **best hyperparameters found while maximizing the average reward** for each task. For each subplot, the square brackets represent the number of neurons per policy layer. [] represents a linear policy.

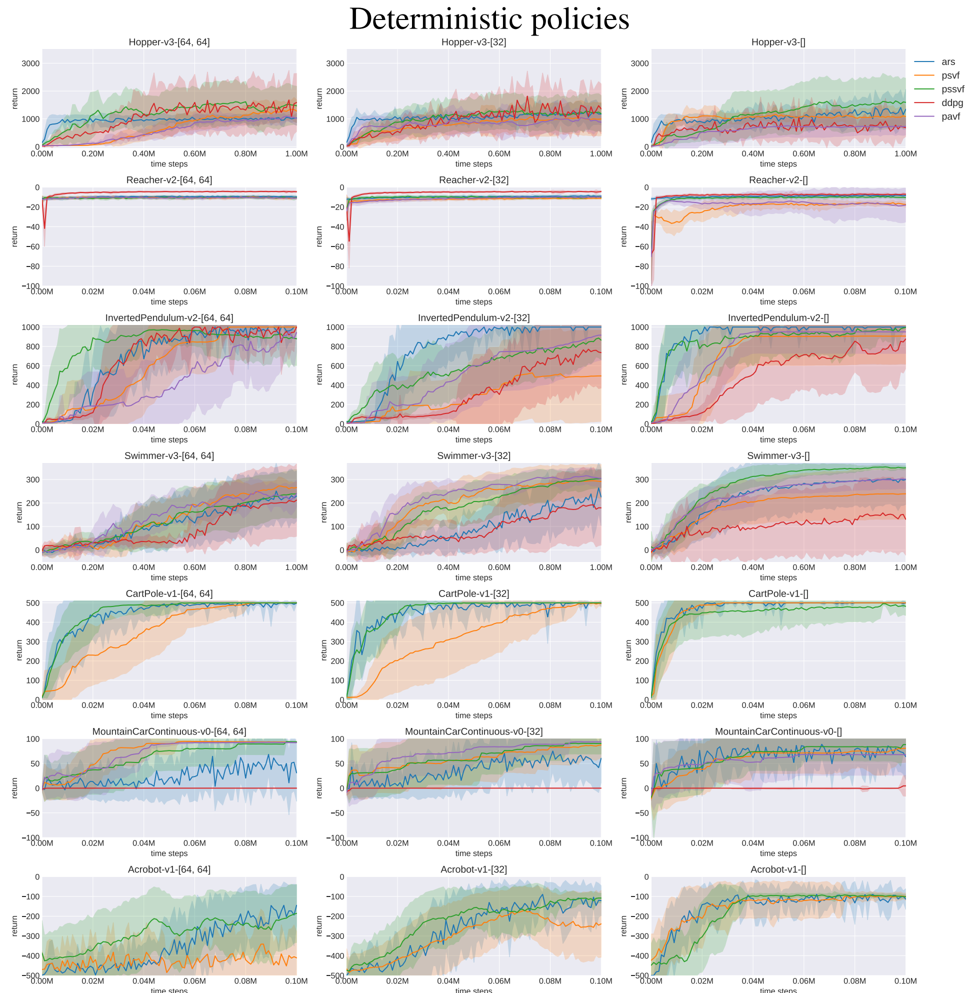


Figure A.6: Learning curves representing the average return as a function of the number of time steps in the environment (across 20 runs) with different environments and deterministic policy architectures. We use the **best hyperparameters found while maximizing the final reward for each task**. For each subplot, the square brackets represent the number of neurons per policy layer. [] represents a linear policy.

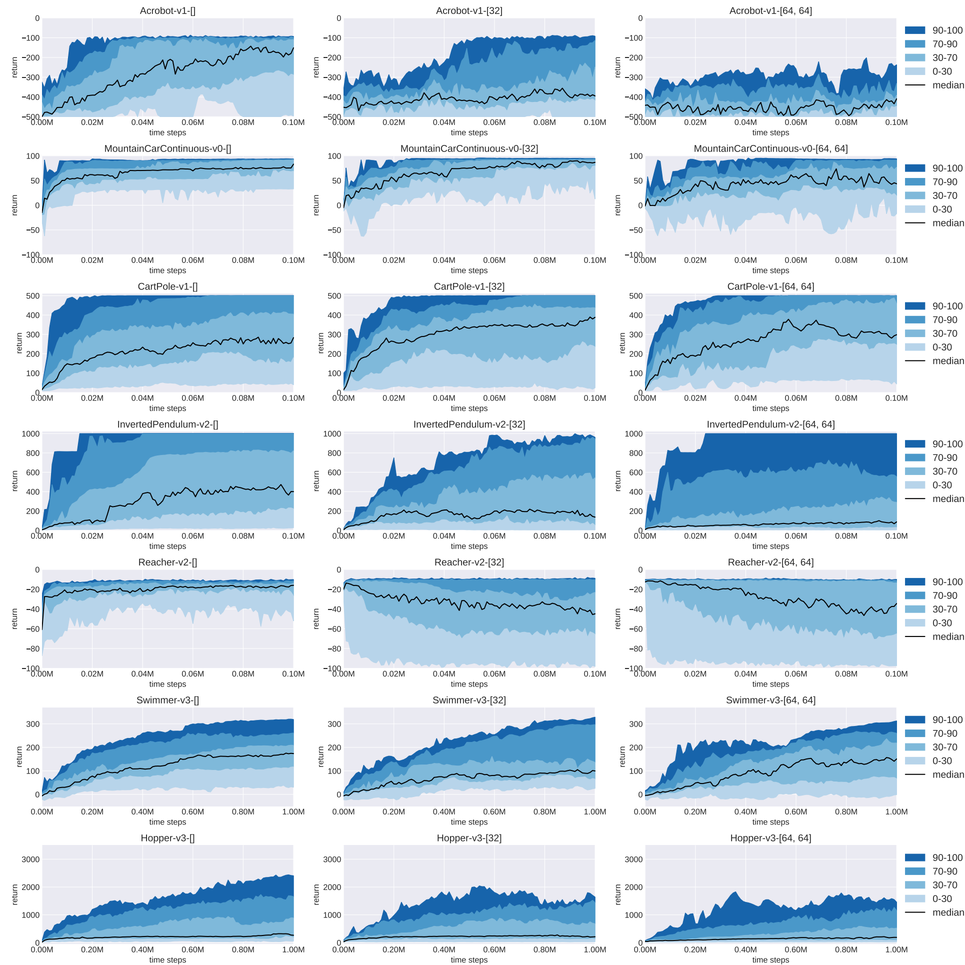


Figure A.7: **Sensitivity of PSSVFs using deterministic policies** to the choice of the hyperparameter. Performance is shown by percentile using all the learning curves obtained during hyperparameter tuning. The median performance is depicted as a dark line. For each subplot, the numbers in the square brackets represent the number of neurons per layer of the policy. [] represents a linear policy.

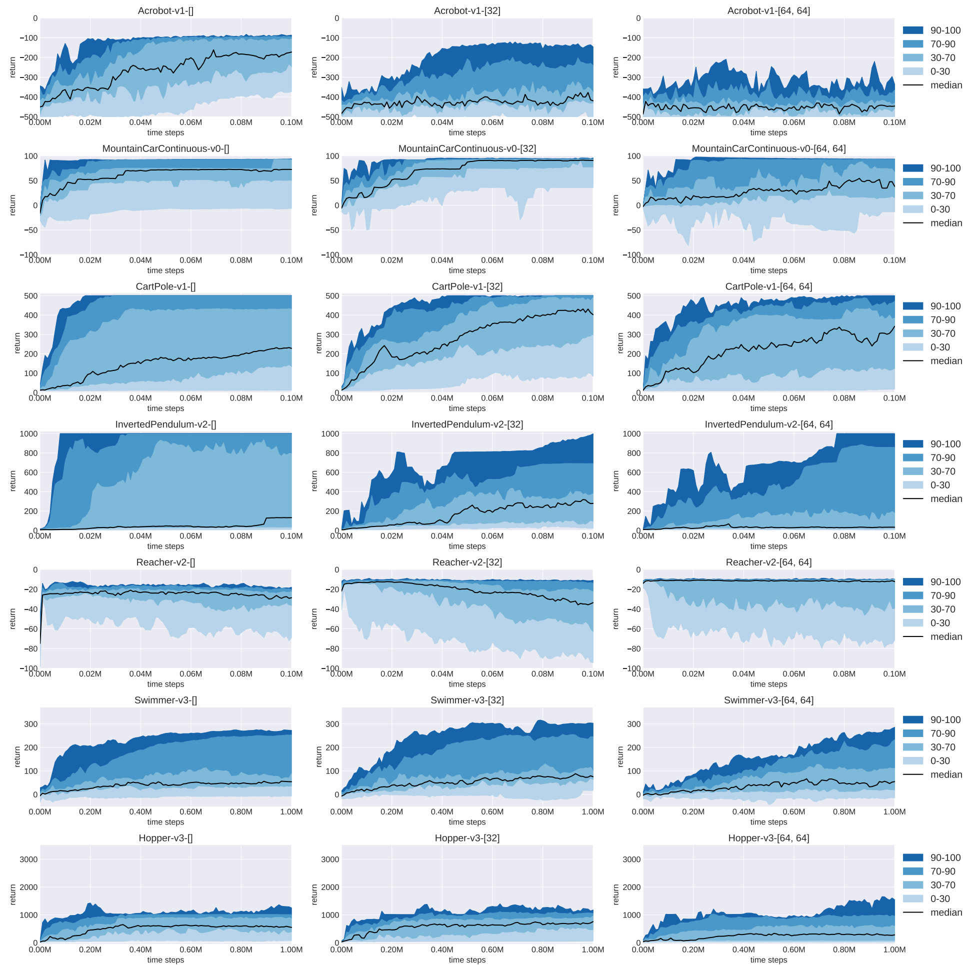


Figure A.8: **Sensitivity of PSVFs using deterministic policies** to the choice of the hyperparameter. Performance is shown by percentile using all the learning curves obtained during hyperparameter tuning. The median performance is depicted as a dark line. For each subplot, the numbers in the square brackets represent the number of neurons per layer of the policy. [] represents a linear policy.

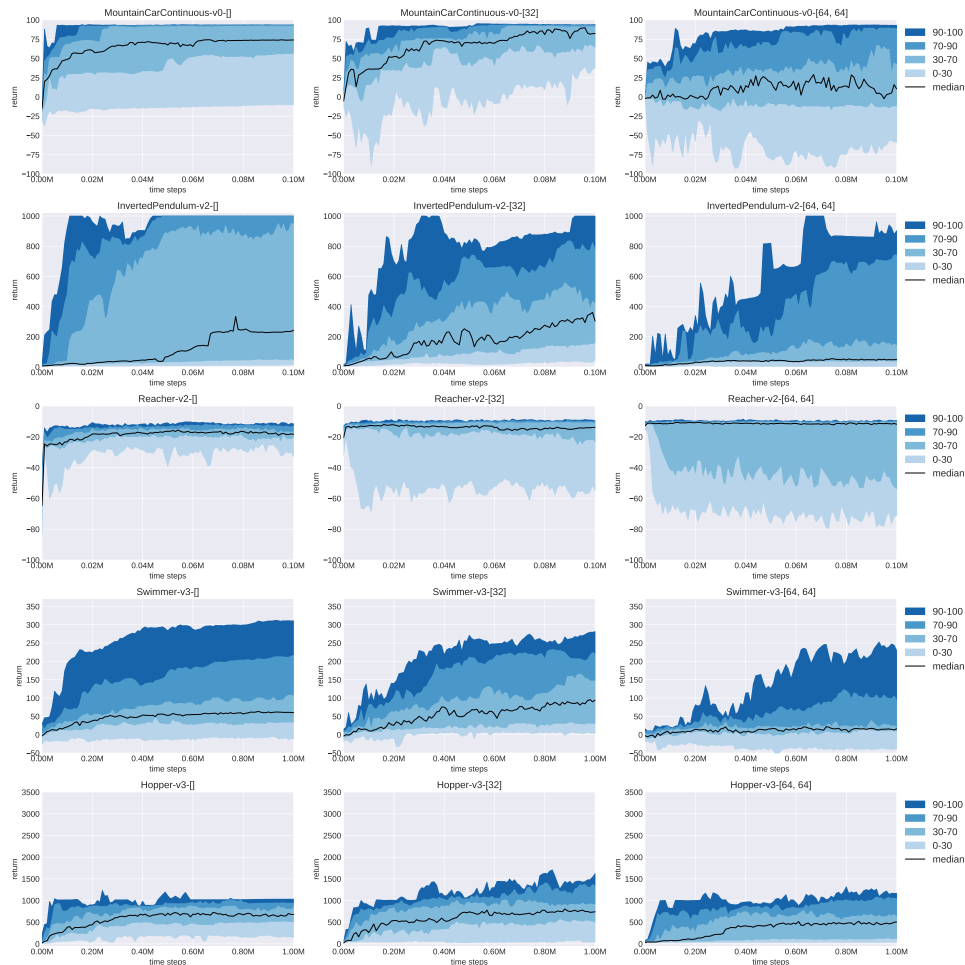


Figure A.9: **Sensitivity of PAVFs using deterministic policies** to the choice of the hyperparameter. Performance is shown by percentile using all the learning curves obtained during hyperparameter tuning. The median performance is depicted as a dark line. For each subplot, the numbers in the square brackets represent the number of neurons per layer of the policy.

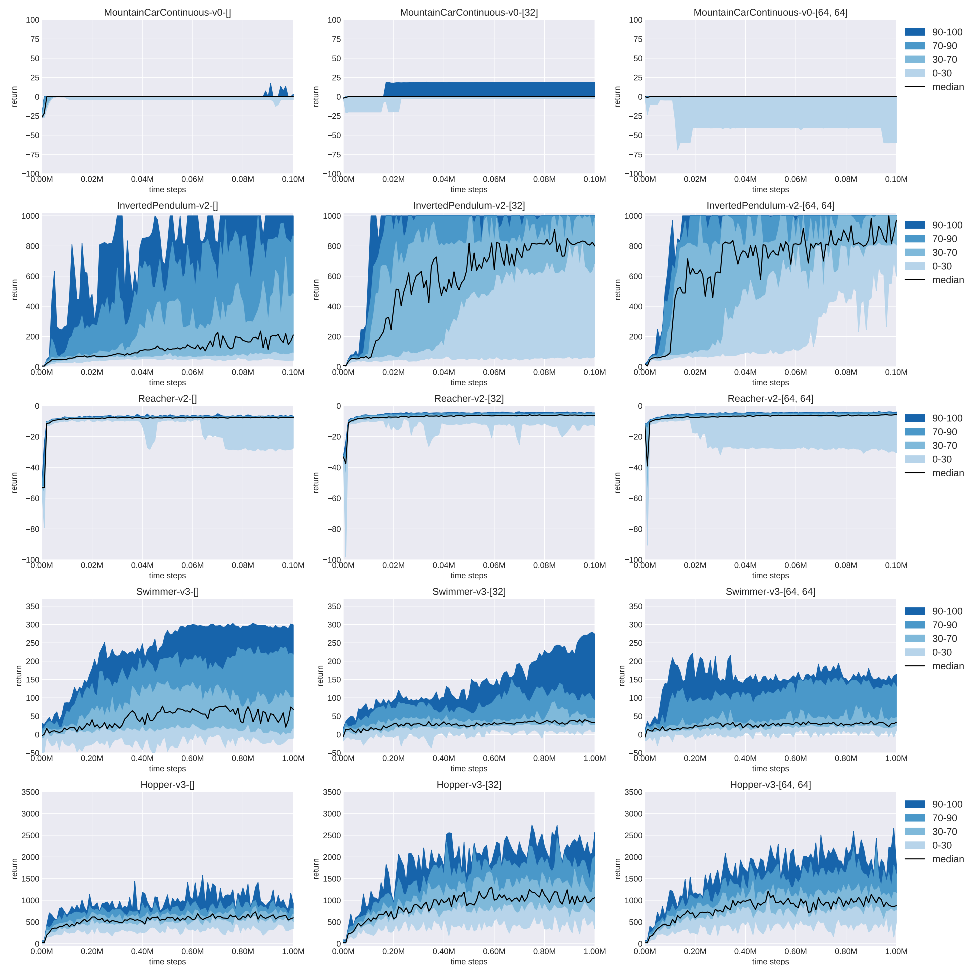


Figure A.10: **Sensitivity of DDPG** to the choice of the hyperparameter. Performance is shown by percentile using all the learning curves obtained during hyperparameter tuning. The median performance is depicted as a dark line. For each subplot, the numbers in the square brackets represent the number of neurons per layer of the policy.

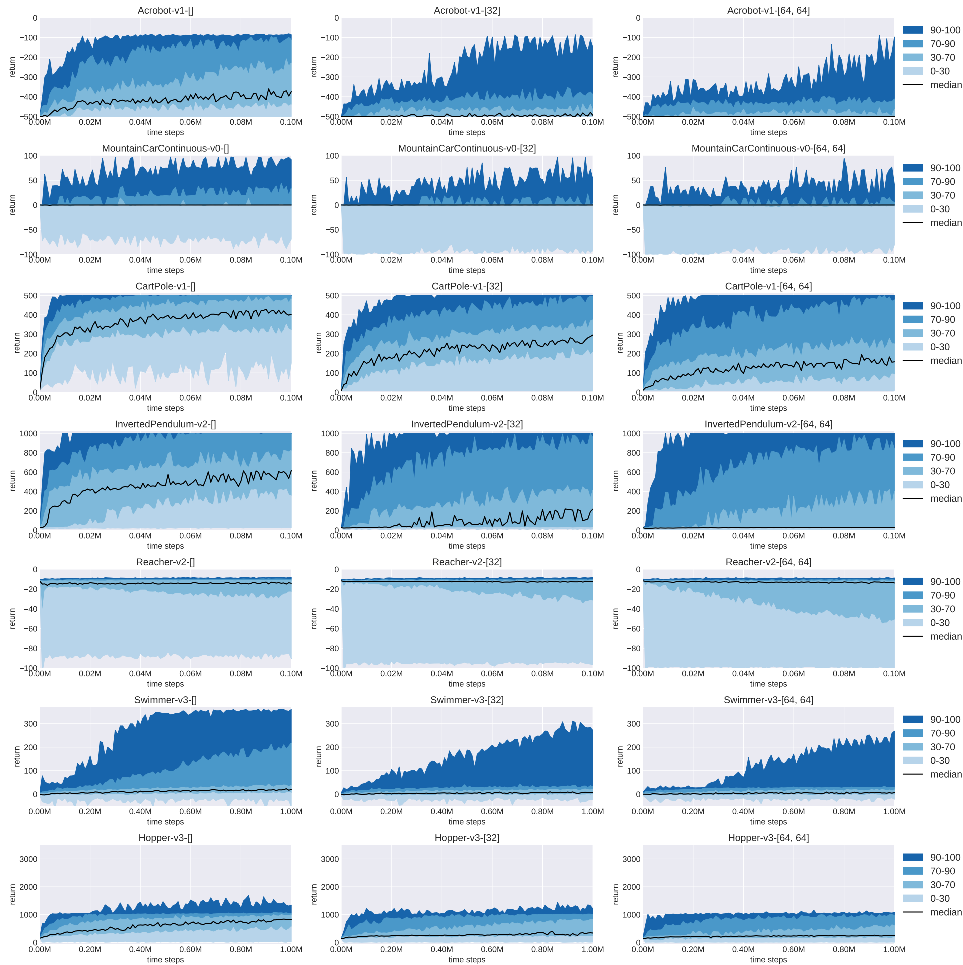


Figure A.11: **Sensitivity of ARS** to the choice of the hyperparameter. Performance is shown by percentile using all the learning curves obtained during hyperparameter tuning. The median performance is depicted as a dark line. For each subplot, the numbers in the square brackets represent the number of neurons per layer of the policy.

Learning rate policy	Policy:	[]		[32]		[64,64]	
	Metric:	avg	last	avg	last	avg	last
Acrobot-v1		1e-2	1e-3	1e-4	1e-4	1e-4	1e-4
MountainCarContinuous-v0		1e-2	1e-3	1e-4	1e-4	1e-4	1e-4
CartPole-v1		1e-3	1e-3	1e-3	1e-3	1e-4	1e-4
Swimmer-v3		1e-3	1e-3	1e-3	1e-3	1e-2	1e-4
InvertedPendulum-v2		1e-3	1e-3	1e-3	1e-3	1e-4	1e-4
Reacher-v2		1e-4	1e-4	1e-4	1e-4	1e-4	1e-4
Hopper-v3		1e-4	1e-4	1e-4	1e-3	1e-4	1e-4
Learning rate critic							
Acrobot-v1		1e-2	1e-3	1e-2	1e-2	1e-2	1e-2
MountainCarContinuous-v0		1e-3	1e-2	1e-3	1e-2	1e-2	1e-2
CartPole-v1		1e-2	1e-2	1e-3	1e-3	1e-2	1e-2
Swimmer-v3		1e-3	1e-3	1e-2	1e-2	1e-3	1e-2
InvertedPendulum-v2		1e-2	1e-2	1e-3	1e-2	1e-3	1e-3
Reacher-v2		1e-3	1e-3	1e-3	1e-3	1e-4	1e-4
Hopper-v3		1e-3	1e-3	1e-2	1e-2	1e-2	1e-2
Noise for exploration							
Acrobot-v1		1.0	1.0	1e-1	1e-1	1e-1	1e-1
MountainCarContinuous-v0		1.0	1.0	1e-1	1e-1	1e-1	1e-1
CartPole-v1		1.0	1.0	1.0	1.0	1e-1	1e-1
Swimmer-v3		1.0	1.0	1.0	1.0	1.0	1e-1
InvertedPendulum-v2		1.0	1.0	1.0	1.0	1e-1	1e-1
Reacher-v2		1e-1	1e-1	1e-1	1e-1	1e-1	1e-1
Hopper-v3		1.0	1.0	1e-1	1.0	1e-1	1e-1

Table A.4: Table of best hyperparameters for PSSVFs using deterministic policies

Learning rate policy	Policy:	[]		[32]		[64,64]	
	Metric:	avg	last	avg	last	avg	last
Acrobot-v1		1e-2	1e-3	1e-2	1e-2	1e-2	1e-2
MountainCarContinuous-v0		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
CartPole-v1		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
Swimmer-v3		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
InvertedPendulum-v2		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
Reacher-v2		1e-2	1e-2	1e-3	1e-2	1e-3	1e-3
Hopper-v3		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
Number of directions and elite directions							
Acrobot-v1		(4,4)	(4,4)	(1,1)	(1,1)	(1,1)	(1,1)
MountainCarContinuous-v0		(1,1)	(1,1)	(1,1)	(16,4)	(1,1)	(1,1)
CartPole-v1		(4,4)	(4,4)	(1,1)	(1,1)	(4,1)	(4,1)
Swimmer-v3		(1,1)	(1,1)	(1,1)	(4,1)	(1,1)	(1,1)
InvertedPendulum-v2		(4,4)	(4,4)	(1,1)	(4,4)	(4,1)	(16,1)
Reacher-v2		(16,16)	(16,16)	(1,1)	(16,4)	(1,1)	(1,1)
Hopper-v3		(4,1)	(4,1)	(1,1)	(1,1)	(1,1)	(1,1)
Noise for exploration							
Acrobot-v1		1e-2	1e-3	1e-1	1e-1	1e-1	1e-1
MountainCarContinuous-v0		1e-1	1e-1	1e-1	1e-1	1e-1	1e-1
CartPole-v1		1e-2	1e-2	1e-1	1e-1	1e-2	1e-2
Swimmer-v3		1e-1	1e-1	1e-2	1e-1	1e-1	1e-1
InvertedPendulum-v2		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
Reacher-v2		1e-2	1e-2	1e-2	1e-2	1e-2	1e-2
Hopper-v3		1e-1	1e-1	1e-1	1e-1	1e-1	1e-1

Table A.5: Table of best hyperparameters for ARS

Learning rate policy	Policy:	[]		[32]		[64,64]	
	Metric:	avg	last	avg	last	avg	last
Acrobot-v1		1e-2	1e-2	1e-4	1e-4	1e-4	1e-2
MountainCarContinuous-v0		1e-2	1e-3	1e-2	1e-4	1e-3	1e-4
CartPole-v1		1e-2	1e-2	1e-2	1e-4	1e-3	1e-4
Swimmer-v3		1e-3	1e-3	1e-3	1e-3	1e-3	1e-3
InvertedPendulum-v2		1e-2	1e-3	1e-4	1e-4	1e-4	1e-4
Reacher-v2		1e-3	1e-2	1e-4	1e-4	1e-4	1e-4
Hopper-v3		1e-3	1e-3	1e-4	1e-4	1e-4	1e-3
Learning rate critic							
Acrobot-v1		1e-3	1e-4	1e-2	1e-2	1e-3	1e-2
MountainCarContinuous-v0		1e-4	1e-3	1e-2	1e-4	1e-3	1e-3
CartPole-v1		1e-2	1e-2	1e-2	1e-3	1e-2	1e-4
Swimmer-v3		1e-4	1e-4	1e-4	1e-4	1e-4	1e-4
InvertedPendulum-v2		1e-3	1e-2	1e-3	1e-4	1e-4	1e-3
Reacher-v2		1e-2	1e-2	1e-3	1e-3	1e-4	1e-4
Hopper-v3		1e-2	1e-2	1e-4	1e-4	1e-2	1e-4
Noise for exploration							
Acrobot-v1		1.0	1.0	1e-1	1e-1	1e-1	1e-1
MountainCarContinuous-v0		1.0	1e-1	1e-1	1.0	1e-1	1e-1
CartPole-v1		1.0	1.0	1.0	1e-1	1e-1	1e-1
Swimmer-v3		1.0	1.0	1.0	1.0	1.0	1.0
InvertedPendulum-v2		1.0	1.0	1e-1	1e-1	1e-1	1e-1
Reacher-v2		1.0	1.0	1.0	1.0	1e-1	1e-1
Hopper-v3		1.0	1.0	1e-1	1e-1	1e-1	1.0

Table A.6: Table of best hyperparameters for PSVFs using deterministic policies

	Algo:	PSSVF		PSVF	
Learning rate policy	Policy:	[]	[64,64]	[]	[64,64]
	Metric:	avg	avg	avg	avg
Acrobot-v1		1e-2	1e-2	1e-2	1e-3
MountainCarContinuous-v0		1e-2	1e-3	1e-2	1e-3
CartPole-v1		1e-3	1e-4	1e-2	1e-3
Swimmer-v3		1e-2	1e-4	1e-3	1e-4
InvertedPendulum-v2		1e-3	1e-4	1e-2	1e-3
Reacher-v2		1e-4	1e-3	1e-2	1e-2
Hopper-v3		1e-4	1e-4	1e-3	1e-4
Learning rate critic					
Acrobot-v1		1e-2	1e-4	1e-4	1e-2
MountainCarContinuous-v0		1e-2	1e-2	1e-3	1e-3
CartPole-v1		1e-2	1e-3	1e-2	1e-2
Swimmer-v3		1e-2	1e-3	1e-3	1e-4
InvertedPendulum-v2		1e-3	1e-3	1e-3	1e-2
Reacher-v2		1e-3	1e-3	1e-3	1e-3
Hopper-v3		1e-3	1e-2	1e-2	1e-4
Noise for exploration					
Acrobot-v1		1.0	1.0	1.0	1.0
MountainCarContinuous-v0		1.0	1e-1	1.0	1e-1
CartPole-v1		1.0	1.0	1.0	1e-1
Swimmer-v3		1.0	1e-1	1.0	1e-1
InvertedPendulum-v2		1.0	1.0	1.0	1e-1
Reacher-v2		1e-1	0.0	1.0	0.0
Hopper-v3		1.0	1e-1	1.0	1e-1

Table A.7: Table of best hyperparameters for PSSVFs and PSVFs using stochastic policies

Learning rate policy	Policy:	[]		[32]		[64,64]	
	Metric:	avg	last	avg	last	avg	last
MountainCarContinuous-v0		1e-2	1e-3	1e-3	1e-4	1e-4	1e-4
Swimmer-v3		1e-3	1e-3	1e-3	1e-3	1e-3	1e-3
InvertedPendulum-v2		1e-2	1e-3	1e-3	1e-4	1e-4	1e-4
Reacher-v2		1e-3	1e-3	1e-4	1e-4	1e-4	1e-4
Hopper-v3		1e-3	1e-4	1e-4	1e-4	1e-4	1e-3
Learning rate critic							
MountainCarContinuous-v0		1e-4	1e-4	1e-4	1e-3	1e-4	1e-3
Swimmer-v3		1e-4	1e-4	1e-4	1e-4	1e-4	1e-4
InvertedPendulum-v2		1e-3	1e-2	1e-2	1e-4	1e-2	1e-3
Reacher-v2		1e-3	1e-3	1e-3	1e-2	1e-3	1e-3
Hopper-v3		1e-4	1e-3	1e-3	1e-2	1e-4	1e-3
Noise for exploration							
MountainCarContinuous-v0		1.0	1e-1	1e-1	1e-1	1e-1	1e-1
Swimmer-v3		1.0	1.0	1.0	1.0	1.0	1.0
InvertedPendulum-v2		1.0	1.0	1e-1	1e-1	1e-1	1e-1
Reacher-v2		1e-1	1e-1	1e-1	1.0	1.0	1.0
Hopper-v3		1.0	1.0	1e-1	1e-1	1e-1	1.0

Table A.8: Table of best hyperparameters for PAVFs using deterministic policies

Learning rate policy	Policy:	[]		[32]		[64,64]	
	Metric:	avg	last	avg	last	avg	last
MountainCarContinuous-v0		1e-2	1e-2	1e-2	1e-4	1e-3	1e-3
Swimmer-v3		1e-3	1e-3	1e-2	1e-2	1e-2	1e-2
InvertedPendulum-v2		1e-4	1e-4	1e-3	1e-3	1e-3	1e-4
Reacher-v2		1e-4	1e-3	1e-2	1e-2	1e-3	1e-3
Hopper-v3		1e-2	1e-2	1e-2	1e-4	1e-2	1e-2
Learning rate critic							
MountainCarContinuous-v0		1e-4	1e-4	1e-4	1e-3	1e-3	1e-3
Swimmer-v3		1e-3	1e-3	1e-3	1e-3	1e-2	1e-3
InvertedPendulum-v2		1e-3	1e-3	1e-3	1e-4	1e-3	1e-3
Reacher-v2		1e-3	1e-3	1e-3	1e-3	1e-3	1e-3
Hopper-v3		1e-3	1e-3	1e-4	1e-4	1e-4	1e-4
Noise for exploration							
MountainCarContinuous-v0		1e-2	1e-2	1e-2	1e-1	1e-1	1e-1
Swimmer-v3		1e-1	1e-1	1e-2	1e-2	1e-2	1e-1
InvertedPendulum-v2		1e-1	1e-1	1e-2	1e-2	1e-2	1e-2
Reacher-v2		1e-1	1e-2	1e-1	1e-1	1e-1	1e-1
Hopper-v3		1e-1	1e-1	1e-1	1e-2	1e-1	1e-2

Table A.9: Table of best hyperparameters for DDPG

Appendix B

General Policy Evaluation and Improvement by Learning to Identify Few But Crucial States

B.1 Implementation details

B.1.1 MNIST Implementation

For our experiments with MNIST we adapt the official code for PSSVF to CNN policies and the MNIST classification problem.

- Policy architecture: The policy consists of two convolutional layers with 4 and 8 output channels respectively, 3×3 kernels and a stride of 1. Each convolutional layer is followed by ReLU activations. The output from the convolutional layers is flattened and provided to a fully connected linear layer which outputs the logits for the ten MNIST classes. The logits are fed into a categorical distribution; the outputs are interpreted as class probabilities.
- Value function architecture: MLP with 2 hidden layers and 64 neurons per layer with bias. ReLU activations.
- Batch size for computing the loss: 1024
- Batch size for value function optimization: 4
- Buffer size: 1000
- Loss: Cross entropy

- Initialization of probing states: uniformly random in $[-0.5, 0.5)$
- Update frequency: every time a new episode is collected
- Number of policy updates: 1
- Number of value function updates: 5
- Learning rate policy: 1e-6
- Learning rate value function: 1e-3
- Noise for policy perturbation: 0.05
- Priority sampling from replay buffer: True, with weights $1/x^{0.8}$, where x is the number of episodes since the data was stored in the buffer
- Default PyTorch initialization for all networks.
- Optimizer: Adam

B.1.2 RL Implementation

Here we report the hyperparameters used for PSSVF and the baselines. For PSSVF, we use the open source implementation provided by Faccio et al. [2021]. For DDPG and SAC, we use the spinning-up RL implementation [Achiam, 2018], whose results are on par with the best reported results. For ARS, we adapt the publicly available implementation [Mania et al., 2018] to Deep NN policies.

Shared hyperparameters:

- Policy architecture: Deterministic MLP with 2 hidden layers and 256 neurons per layer with bias. Tanh activations for PSSVF and ARS. ReLu activations for DDPG and SAC. The output layer has Tanh nonlinearity and bounds the action in the action-space limit.
- Value function architecture: MLP with 2 hidden layers and 256 neurons per layer with bias. ReLU activations for PSSVF and DDPG and SAC.
- Initialization for actors and critics: Default PyTorch initialization
- Batch size: 128 for DDPG and SAC. 16 for PSSVF
- Learning rate actor: 1e-3 for DDPG and SAC; 2e-6 for PSSVF

- Learning rate critic: 1e-3 for DDPG and SAC, 5e-3 for PSSVF
- Noise for exploration: 0.05 in parameter space for PSSVF; 0.1 in action space for DDPG
- Actor's frequency of updates: every episode for PSSVF; every 50 time steps for DDPG and SAC; every batch for ARS
- Critic's frequency of updates: every episode for PSSVF; every 50 time steps for DDPG and SAC
- Replay buffer size: 100k for DDPG and SAC; 10k for PSSVF
- Optimizer: Adam for PSSVF and DDPG and SAC
- Discount factor: 0.99 for DDPG and SAC; 1 for PSSVF and ARS
- Survival reward adjustment: True for ARS and PSSVF in Hopper, Walker, Ant; False for DDPG and SAC
- Environmental interactions: 300k time steps in InvertedDoublePendulum; 3M time steps in all other environments

Tuned hyperparameters:

- Step size for ARS: tuned with values in $\{1e-2, 1e-3, 1e-4\}$
- Number of directions and elite directions for ARS: tuned with values in $\{[1,1],[8,4],[8,8],[32,4],[32,16],[64,8],[64,32]\}$, where the first element denotes the number of directions and the second element the number of elite directions
- Noise for exploration in ARS: tuned with values in $\{0.1, 0.05, 0.025\}$

Hyperparameters for specific algorithms:
PSSVF:

- Number of probing states: 200
- Initialization of probing states: uniformly random in $[0,1)$
- Observation normalization: True
- Number of policy updates: 5

- Number of value function updates: 5
- Priority sampling from replay buffer: True, with weights $1/x^{1.1}$, where x is the number of episodes since the data was stored in the buffer

ARS:

- Observation normalization: True

DDPG and SAC:

- Observation normalization: False
- Number of policy updates: 50
- Number of value function updates: 50
- Start-steps (random actions): 10000 time-steps
- Update after (no training): 1000 time-steps
- Polyak parameter: 0.995
- Entropy parameter (SAC): 0.2

B.1.3 GPU usage / Computation requirements

Each run of PSSVF in the main experiment takes around 2.5 hours on a Tesla P100 GPU. We ran 4 instances of our algorithm for each GPU. We estimate a total of 75 node hours to reproduce our main RL results (20 independent runs for 6 environments).

B.2 Experimental details

B.2.1 Main experiments on MuJoCo

To measure learning progress, we evaluate each algorithm for 10 episodes every 10000 time steps. We use the learned policy for PSSVF and ARS and the deterministic actor (without action noise) for DDPG. We use 20 independent instances of the same hyperparameter configuration for PSSVF and DDPG in all environments. When tuning ARS, we run 5 instances of the algorithm for each hyperparameter configuration. Then we select the best hyperparameter for each environment and carry out a further 20 independent runs. We report the best hyperparameters found for ARS in Table B.1. We report the final return with a standard deviation in Table B.2.

Environment	step size	directions	noise
Walker2d-v3	0.01	[8,8]	0.05
Swimmer-v3	0.01	[8,4]	0.05
HalfCheetah-v3	0.01	[8,4]	0.05
Ant-v3	0.01	[32,16]	0.01
Hopper-v3	0.01	[8,4]	0.05
InvertedDoublePendulum-v2	0.01	[8,8]	0.025

Table B.1: Best hyperparameters for ARS

Environment	PSSVF	ARS	DDPG	SAC
Walker2d-v3	2333 ± 343	1488 ± 961	2432 ± 1330	$\mathbf{5287 \pm 467}$
Swimmer-v3	$\mathbf{349 \pm 60}$	$\mathbf{342 \pm 21}$	129 ± 25	44 ± 1
HalfCheetah-v3	3067 ± 820	2497 ± 611	10695 ± 1358	$\mathbf{13599 \pm 932}$
Ant-v3	1549 ± 240	1697 ± 225	466 ± 716	$\mathbf{5319 \pm 992}$
Hopper-v3	$\mathbf{2969 \pm 165}$	2340 ± 199	1634 ± 1036	$\mathbf{3292 \pm 345}$
InvertedDouble Pendulum-v2	$\mathbf{7649 \pm 2640}$	4515 ± 2733	$\mathbf{7377 \pm 3770}$	$\mathbf{9235 \pm 227}$

Table B.2: Final return (average over final 20 evaluations)

B.2.2 Ablation on weighted sampling

In Figure B.1 we show the benefit of using non-uniform sampling from the replay buffer in Hopper and Walker environments. We compare uniform sampling (no weight) to non uniform sampling with weight $1/x^k$, where $k \in \{1.0, 1.1\}$, and x is the number of episodes since the data was stored in the buffer. We achieve the best results in Hopper and Walker for the choice of $x = 1.1$. It is interesting to take this into consideration when comparing our approach to vanilla PSSVF.

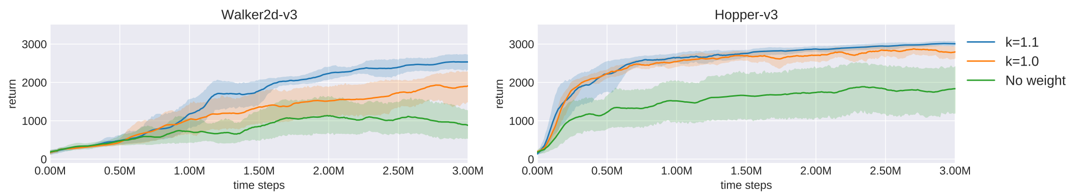


Figure B.1: Comparison between our algorithm without weighted sampling from the replay buffer and with weight $1/x^k$, where $k \in \{1.0, 1.1\}$. Average over 10 independent runs and 95% bootstrapped confidence interval.

B.2.3 Comparison to vanilla PSSVF

Here we compare our PSSVF with policy fingerprinting to vanilla PSSVF. For vanilla PSSVF, we use the best hyperparameters reported in Appendix A.3.4 when optimizing policies with 2 hidden layers and 64 neurons per layer and optimizing over the final rewards. Our algorithm uses the policy architecture of vanilla PSSVF and the hyperparameters of our main experiments, changing only the learning rate of the policy to $1e-4$ and the noise for policy perturbations to 0.1. Figure B.2 shows that while in Swimmer policy fingerprinting is enough to achieve an improvement over vanilla PSSVF, in Hopper non-uniform sampling plays an important role. Note that vanilla PSSVF, learning rates and perturbation noise are tuned for each environment, while in our experiments we keep a fixed set of hyperparameters for all environments to maintain consistency. We expect the performance of our approach to also improve by selecting hyperparameters separately for each environment.

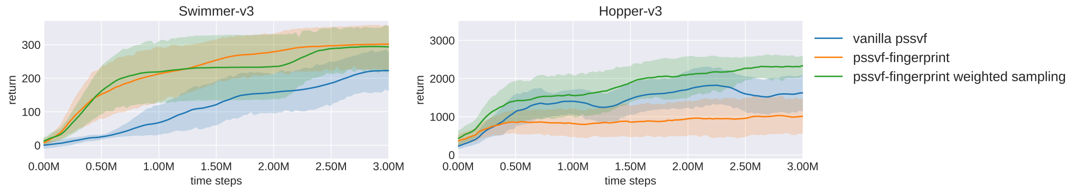


Figure B.2: Comparison between vanilla PSSVF with no weighted sampling and no fingerprinting, PSSVF with policy fingerprinting, and our final algorithm that uses also weighted sampling. The solid line is the average over 10 independent runs; the shading indicates 95% bootstrapped confidence intervals.

Appendix C

Learning Useful Representations of Recurrent Neural Network Weight Matrices

C.1 Experimental details

C.1.1 Hyperparameters

Table C.1 shows the hyperparameters shared by all four encoder types in the experiments. Hyperparameters specific to probing, flattened and neural functional encoders are shown in Tables C.2, C.3 and C.4, respectively.

Hyperparameter	**Value**
A hidden size	256
A #layers	2
z size	16
batch size	64
optimizer	AdamW
learning rate	0.0001
weight decay	0.01
gradient clipping	0.1

Table C.1: General hyperparameters

Hyperparameter	Value
E_R hidden size	256
E_R #layers	2
E_I hidden size	128
E_I #layers	1
E_O hidden size	128
E_O #layers	1
probing sequence length	22

Table C.2: Hyperparameters for probing (interactive and non-interactive) encoders

Hyperparameter	Value
hidden size	128
#layers	3

Table C.3: Hyperparameters for flattened weights encoders

Hyperparameter	Value
#channels	4
#layers	4

Table C.4: Hyperparameters for neural functional encoders

Appendix D

Goal-Conditioned Generators of Deep Policies

D.1 Implementation details

D.1.1 Hyperparameters

Here we report the hyperparameters used for GoGePo and the baselines. For DDPG, TD3, SAC, we use the spinning-up RL implementation [Achiam, 2018], whose results are on par with the best reported results. For ARS, we use the implementation of the authors [Mania et al., 2018], adapted to Deep NN policies.

Shared hyperparameters The table below shows hyperparameters relevant to at least two of the three methods. They stay fixed across environments.

Hyperparameter	ARS	GoGePo	DDPG
Policy Architecture	MLP, 2 hidden layers, 256 neurons each, with bias		
Policy Nonlinearity	tanh		ReLU
Value Function Architecture		MLP, 2 hidden layers, 256 neurons each, with bias	
Value Function Nonlinearity		ReLU	
Initialization MLPs		PyTorch default (for value function)	PyTorch default (for actor & critic)
Batch Size		16	128
Optimizer		Adam	
Learning Rate Actor/Generator		2e-6	1e-3
Learning Rate Value Function		5e-3	1e-3
Exploration Noise Scale	tuned (see below)	0.1 in parameter space	0.1 in action space
Update Frequency Actor/Generator	every batch	every episode	every 50 time steps
Update Frequency Value Function		every episode	every 50 time steps
Number of Actor/Generator Updates		20	50
Number of Value Function Updates		5	50
Replay Buffer Size		10k	100k
Discount Factor	1		0.99
Survival Reward Adjustment	True (for Hopper)		False
Observation Normalization	True		False
Environmental interactions	100k for InvertedPendulum and MountCarContinuous, 3M for all other environments		

Hyperparameters for specific algorithms Fixed across environments:
GoGePo:

- Architecture of the networks H in the generator: MLP with bias, two hidden layers of size 256, ReLU nonlinearity, no output activation function
- Size of learnable hypernetwork embeddings z^j_{mn}: 8
- Size of slices sl^j_{mn} produced by the hypernetwork: 16×16
- Number of probing states: 200
- Initialization of probing states: Uniformly random in $[0,1)$
- Priority sampling from replay buffer: True, with weights $1/x^{1.1}$, where x is the number of episodes since the data was stored in the buffer

DDPG:

- Start-steps (random actions): 10000 time steps
- Update after (no training): 1000 time steps
- Polyak parameter: 0.995

TD3/SAC: We use the default hyperparameter values from the spinning-up RL implementation [Achiam, 2018].

Tuned hyperparameters For ARS, we tune the following hyperparameters for each environment separately using grid search:

- Step size for ARS: tuned with values in $\{1e-2, 1e-3, 1e-4\}$
- Number of directions and elite directions for ARS: tuned with values in $\{[1,1],[8,4],[8,8],[32,4],[32,16],[64,8],[64,32]\}$, where the first element denotes the number of directions and the second element the number of elite directions
- Noise for exploration in ARS: tuned with values in $\{0.1, 0.05, 0.025\}$

Here we report the best hyperparameters found for each environment:

ARS Hyperparameter	Swimmer	Hopper	Inverted-Pendulum	MountainCar-Continuous
Step Size	0.01	0.01	0.001	0.01
Number of Directions, Number of Elite Directions	(8, 4)	(8, 4)	(1, 1)	(1, 1)
Exploration Noise Scale	0.05	0.05	0.025	0.05

UDRL For UDRL we use a previous implementation [Srivastava et al., 2019] for discrete control environments, and implemented additional classes to use it in continuous control tasks with episodic resets (although the original UDRL report [Schmidhuber, 2019] focused on continuous control in single-life settings without resets). We use the previous hyperparameters [Srivastava et al., 2019] and tune learning rate (in $\{1e-3, 1e-4, 1e-5\}$), activation (ReLU, tanh), and their "last_few" parameter (1, 10, 100), which is used to select the command for exploration. For Swimmer, we are not able to reproduce the performance with the original reported hyperparameters. Like for the other algorithms, we use an NN with 2 hidden layers and 256 neurons per layer. Below we report the best hyperparameters found for UDRL.

UDRL Hyperparameter	Swimmer	Hopper	Inverted-Pendulum	MountainCar-Continuous
Nonlinearity	ReLU	ReLU	tanh	ReLU
Learning Rate	1e-3	1e-5	1e-3	1e-5
Last Few	10	10	1	1

D.1.2 Generator implementation details

Generating bias vectors Here we describe how to generate the bias vectors of the policies, which is not explicitly mentioned in section 6.1.4. Analogously to Equations 6.5 and 6.6, the embeddings z^j_{mn} are fed to a dedicated bias-generating network H_χ that produces slices of the shape $f \times 1$, and those slices are concatenated. Since we have a two-dimensional grid of learned embeddings z (see Figure 6.2), we take the mean across the input dimensions of the concatenated slices so that we end up with a bias vector (and not a matrix).

D.1.3 GPU usage / compute

We use cloud computing resources for our experiments. Our nodes have an Intel Xeon 12 core CPU and an NVIDIA Tesla P100 GPU with 16GB of memory. We were able to run four GoGePo experiments on one node in parallel. Our estimate of computation time for the main results is 40 node hours.

D.2 Experimental details

D.2.1 Main experiments on MuJoCo

For ARS and UDRL, the best hyperparameters for each environment are determined by running the algorithm with each hyperparameter configuration across 5 random seeds. The best configurations are those reported in section D.1.1 We use them for the final 20 evaluation runs shown in our main results. For DDPG, TD3, SAC and GoGePo, we use the same hyperparameters for all environments. For 10 episodes, Figure 6.3 evaluates each run every 10000 time steps for Swimmer and Hopper, every 1000 steps for InvertedPendulum and MountainCarContinuous. Table D.1 shows the final return and standard deviation of each algorithm.

	Swimmer	MountainCarContinuous	Hopper	InvertedPendulum
GoGePo	**334 ± 16**	**93 ± 1**	2589 ± 300	**980 ± 40**
ARS	**342 ± 21**	55 ± 33	2340 ± 199	936 ± 42
DDPG	129 ± 25	-1 ± 0.01	1634 ± 1036	**960 ± 175**
UDRL	78 ± 17	-3 ± 0.3	1010 ± 78	219 ± 299
TD3	84 ± 38	0 ± 0	**3156 ± 995**	**948 ± 178**
SAC	44 ± 2	5 ± 21	**3346 ± 604**	**960 ± 181**

Table D.1: Final returns of GoGePo and baselines (mean and standard deviation over 20 evaluations). We highlight all algorithms that are not statistically significantly different from the best one in each task (Welch's t-test with $p<0.05$).

D.2.2 Command strategies

In early experiments, we tried an alternative approach using Importance Sampling [Hesterberg, 1988] estimators. Given a mixture of weights $\beta_i(\theta)$, we considered estimators of the form $\hat{J}(c', w') = \sum_{i=1}^{N} \beta_i(\theta_i) \frac{p(\theta_i|c';w')}{p(\theta_i|c_i;w_i)} r_i$, which provides an unbiased

estimate of the performance of a policy produced by a generator with parameters w' and command c', using past data derived from old generators with different commands. Maximizing $\hat{J}(c', w')$ with respect to the command c' should yield commands encouraging the generator to produce highly performant policies. We tested this using the Balance Heuristic [Veach and Guibas, 1995] estimator for β_k, which is known to have small variance [Papini et al., 2019]. However, in our experiments we observed that generators using such command strategies did not significantly outperform the simple strategy mentioned earlier.

Ablation command Figure D.1 shows that when choosing the command for exploration there is a slight advantage for asking the generator for a policy whose return exceeds the best return so far by 20. However, just asking for the maximum return (drive parameter = 0) is also competitive.

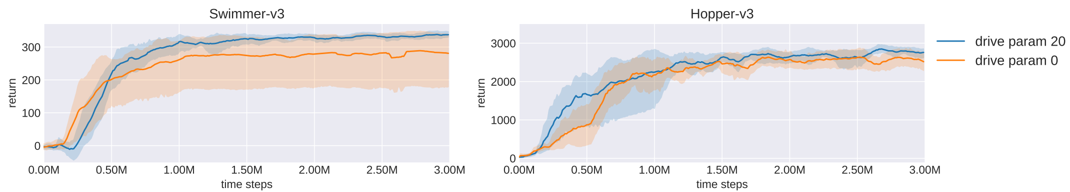


Figure D.1: Comparison of variants of our algorithm with/without drive parameter for command exploration. Average over 5 independent runs and 95% bootstrapped confidence intervals.

D.3 Environment details

MuJoCo [Todorov et al., 2012] is licensed under Apache 2.0.

Bibliography

Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

S. Amari. A theory of adaptive pattern classifiers. *IEEE Trans. EC*, 16(3):299–307, 1967.

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight Experience Replay. In *NeurIPS*, 2017.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Marc G Bellemare, Salvatore Candido, Pablo Samuel Castro, Jun Gong, Marlos C Machado, Subhodeep Moitra, Sameera S Ponda, and Ziyu Wang. Autonomous navigation of stratospheric balloons using reinforcement learning. *Nature*, 588 (7836):77–82, 2020.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

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

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

D. C. Ciresan, U. Meier, J. Masci, and J. Schmidhuber. A committee of neural networks for traffic sign classification. In *International Joint Conference on Neural Networks (IJCNN)*, pages 1918–1921, 2011.

D. C. Ciresan, U. Meier, J. Masci, and J. Schmidhuber. Multi-column deep neural network for traffic sign classification. *Neural Networks*, 32:333–338, 2012.

Corinna Cortes, Yishay Mansour, and Mehryar Mohri. Learning bounds for importance weighting. In *Advances in neural information processing systems*, pages 442–450, 2010.

Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.

Thomas Degris, Martha White, and Richard S. Sutton. Off-policy actor-critic. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML'12, pages 179–186, USA, 2012. Omnipress. ISBN 978-1-4503-1285-1.

Emilien Dupont, Hyunjik Kim, SM Eslami, Danilo Rezende, and Dan Rosenbaum. From data to functa: Your data point is a function and you can treat it like one. *arXiv preprint arXiv:2201.12204*, 2022.

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

Francesco Faccio, Louis Kirsch, and Jürgen Schmidhuber. Parameter-based value functions. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL `https://openreview.net/forum?id=tV6oBfuyLTQ`.

Francesco Faccio, Aditya Ramesh, Vincent Herrmann, Jean Harb, and Jürgen Schmidhuber. General policy evaluation and improvement by learning to identify few but crucial states. *The 15th European Workshop on Reinforcement Learning*, 2022. URL `https://arxiv.org/abs/2207.01566`.

Francesco Faccio, Vincent Herrmann, Aditya Ramesh, Louis Kirsch, and Jürgen Schmidhuber. Goal-conditioned generators of deep policies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 7503–7511, 2023. URL `https://arxiv.org/abs/2207.01570`.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

K. Fukushima. Neural network model for a mechanism of pattern recognition unaffected by shift in position - Neocognitron. *Trans. IECE*, J62-A(10):658–665, 1979.

Kunihiko Fukushima. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics*, 5(4): 322–333, 1969.

Carl Friedrich Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*. 1809.

F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

Dibya Ghosh, Abhishek Gupta, Ashwin Reddy, Justin Fu, Coline Devin, Benjamin Eysenbach, and Sergey Levine. Learning to reach goals via iterated supervised learning, 2019.

F. J. Gomez and J. Schmidhuber. Co-evolving recurrent neurons learn deep memory POMDPs. In *Proc. of the 2005 conference on genetic and evolutionary computation (GECCO), Washington, D. C.* ACM Press, New York, NY, USA, 2005.

A. Graves and J. Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in Neural Information Processing Systems (NIPS) 21*, pages 545–552. MIT Press, Cambridge, MA, 2009.

A. Graves, S. Fernandez, F. J. Gomez, and J. Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural nets. In *ICML'06: Proceedings of the 23rd International Conference on Machine Learning*, pages 369–376, 2006.

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

Karol Gregor. Finding online neural update rules by learning to remember. *arXiv preprint arXiv:2003.03124*, 3 2020.

Aditya Grover, Maruan Al-Shedivat, Jayesh Gupta, Yuri Burda, and Harrison Edwards. Learning policy representations in multiagent systems. In *International conference on machine learning*, pages 1802–1811. PMLR, 2018.

David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 2450–2462, 2018.

David Ha, Andrew Dai, and Quoc V. Le. HyperNetworks. In *International Conference on Learning Representations*, 2016.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018a.

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

Jean Harb, Tom Schaul, Doina Precup, and Pierre-Luc Bacon. Policy evaluation networks. *arXiv preprint arXiv:2002.11833*, 2020.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

Vincent Herrmann, Louis Kirsch, and Jürgen Schmidhuber. Learning one abstract bit at a time through self-invented experiments encoded as neural networks. *arXiv preprint arXiv:2212.14374*, 2022.

Vincent Herrmann, Francesco Faccio, and Jürgen Schmidhuber. Learning useful representations of recurrent neural network weight matrices. In *NeurIPS 2023 Workshop on Symmetry and Geometry in Neural Representations*, 2023. URL `https://openreview.net/forum?id=yqGoKziEvY`.

Timothy Classen Hesterberg. *Advances in importance sampling*. PhD thesis, Stanford University, 1988.

S. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991. Advisor: J. Schmidhuber.

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

Ehsan Imani, Eric Graves, and Martha White. An off-policy policy gradient theorem using emphatic weightings. In *Advances in Neural Information Processing Systems*, pages 96–106, 2018.

Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. A modern self-referential weight matrix that learns to modify itself. In *Deep RL Workshop NeurIPS 2021*, 2021a.

Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. *Advances in Neural Information Processing Systems*, 34:7703–7717, 2021b.

Aleksey Grigorievitch Ivakhnenko and Valentin Grigorievitch Lapa. *Cybernetic Predicting Devices*. CCM Information Corporation, 1965.

Michael Janner, Qiyang Li, and Sergey Levine. Offline Reinforcement Learning as One Big Sequence Modeling Problem. In *NeurIPS*, 2021.

Chi Jin, Akshay Krishnamurthy, Max Simchowitz, and Tiancheng Yu. Reward-free exploration for reinforcement learning. In *International Conference on Machine Learning*, pages 4870–4879. PMLR, 2020.

Leslie Pack Kaelbling. Learning to achieve goals. In *IJCAI*, 1993.

Anssi Kanervisto, Tomi Kinnunen, and Ville Hautamäki. General characterization of agents by states they visit. *arXiv preprint arXiv:2012.01244*, 2020.

J. Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *Annals of Mathematical Statistics*, 23:462–466, 1952. URL `https://api.semanticscholar.org/CorpusID:122078986`.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Louis Kirsch and Jürgen Schmidhuber. Meta learning backpropagation and improving it. *Advances in Neural Information Processing Systems*, 34, 2021.

Louis Kirsch, Sebastian Flennerhag, Hado van Hasselt, Abram Friesen, Junhyuk Oh, and Yutian Chen. Introducing Symmetries to Black Box Meta Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022.

Boris Knyazev, Michal Drozdzal, Graham W Taylor, and Adriana Romero Soriano. Parameter prediction for unseen deep architectures. *Advances in Neural Information Processing Systems*, 34, 2021.

Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Society for Industrial and Applied Mathematics*, 42, 04 2001.

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

Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, and Faustino Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068, 2013.

Alex Krizhevsky, I Sutskever, and G. E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS 2012)*, page 4, 2012.

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

S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's thesis, Univ. Helsinki, 1970.

Yao Liu, Adith Swaminathan, Alekh Agarwal, and Emma Brunskill. Off-policy policy gradient with state distribution correction. *arXiv preprint arXiv:1904.08473*, 2019.

Hamid R. Maei, Csaba Szepesvári, Shalabh Bhatnagar, Doina Precup, David Silver, and Richard S. Sutton. Convergent temporal-difference learning with arbitrary

smooth function approximation. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, NIPS'09, page 1204–1212, Red Hook, NY, USA, 2009. Curran Associates Inc. ISBN 9781615679119.

Hamid Reza Maei. *Gradient temporal-difference learning algorithms*. PhD thesis, University of Alberta, 2011.

Hamid Reza Maei, Csaba Szepesvári, Shalabh Bhatnagar, and Richard S. Sutton. Toward off-policy learning control with function approximation. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 719–726, Madison, WI, USA, 2010. Omnipress. ISBN 9781605589077.

Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search of static linear policies is competitive for reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1800–1809, 2018.

Alberto Maria Metelli, Matteo Papini, Francesco Faccio, and Marcello Restelli. Policy optimization via importance sampling. In *Advances in Neural Information Processing Systems*, pages 5442–5454, 2018.

N. Metropolis and S. Ulam. The monte carlo method. *J. Am. Stat. Assoc.*, 44:335, 1949.

Thomas Miconi, Kenneth Stanley, and Jeff Clune. Differentiable plasticity: training plastic neural networks with backpropagation. In *International Conference on Machine Learning*, pages 3559–3568. PMLR, 2018.

M. Minsky. Steps toward artificial intelligence. In E. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 406–450. McGraw-Hill, New York, 1963.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Ofir Nachum, Bo Dai, Ilya Kostrikov, Yinlam Chow, Lihong Li, and Dale Schuurmans. Algaedice: Policy gradient from arbitrary experience. *arXiv preprint arXiv:1912.02074*, 2019.

Elias Najarro and Sebastian Risi. Meta-learning through hebbian plasticity in random networks. *Advances in Neural Information Processing Systems*, 33:20719–20731, 2020.

Aviv Navon, Aviv Shamsian, Idan Achituve, Ethan Fetaya, Gal Chechik, and Haggai Maron. Equivariant architectures for learning in deep weight spaces. *arXiv preprint arXiv:2301.12780*, 2023.

OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

Aldo Pacchiano, Jack Parker-Holder, Yunhao Tang, Krzysztof Choromanski, Anna Choromanska, and Michael Jordan. Learning to score behaviors for guided policy optimization. In *International Conference on Machine Learning*, pages 7445–7454. PMLR, 2020.

Matteo Papini, Alberto Maria Metelli, Lorenzo Lupo, and Marcello Restelli. Optimistic policy optimization via multiple importance sampling. In *36th International Conference on Machine Learning*, volume 97, pages 4989–4999, 2019.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Jan Peters and Stefan Schaal. Reinforcement Learning by Reward-weighted Regression for Operational Space Control. In *ICML*, pages 745–750, 2007.

Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomput.*, 71(7-9):1180–1190, March 2008. ISSN 0925-2312. doi: 10.1016/j.neucom.2007.11.026.

Doina Precup, Richard S. Sutton, and Sanjoy Dasgupta. Off-policy temporal difference learning with function approximation. In *ICML*, 2001.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Roberta Raileanu, Max Goldstein, Arthur Szlam, and Rob Fergus. Fast adaptation to new environments via policy-dynamics value functions. In *Proceedings of the 37th International Conference on Machine Learning*, pages 7920–7931, 2020.

Paulo Rauber, Avinash Ummadisingu, Filipe Mutz, and Jürgen Schmidhuber. Hindsight policy gradients. In *International Conference on Learning Representations*, 2018.

I. Rechenberg. Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. Dissertation, 1971. Published 1973 by Fromman-Holzboog.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, 1987.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, pages 318–362. MIT Press, 1986.

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

Tom Schaul, Julian Togelius, and Jürgen Schmidhuber. Measuring intelligence through games. *arXiv preprint arXiv:1109.1314*, 2011.

Tom Schaul, Dan Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1312–1320. JMLR.org, 2015.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pages 9355–9366. PMLR, 2021.

J. Schmidhuber. Making the world differentiable: On using fully recurrent self-supervised neural networks for dynamic reinforcement learning and planning in non-stationary environments. Technical Report FKI-126-90, `http://people.idsia.ch/~juergen/FKI-126-90_(revised)bw_ocr.pdf`, Tech. Univ. Munich, 1990.

J. Schmidhuber. Reinforcement learning in Markovian and non-Markovian environments. In D. S. Lippman, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3 (NIPS 3)*, pages 500–506. Morgan Kaufmann, 1991a.

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

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015a. doi: 10.1016/j.neunet.2014.09.003. Published online 2014; 888 references; based on TR arXiv:1404.7828 [cs.NE].

J. Schmidhuber and R. Huber. Learning to generate artificial fovea trajectories for target detection. *International Journal of Neural Systems*, 2(1 & 2):135–141, 1991. (Based on TR FKI-128-90, TUM, 1990).

Juergen Schmidhuber. On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *Preprint arXiv:1511.09249*, 2015b.

Jürgen Schmidhuber. Learning to generate sub-goals for action sequences. In *Artificial neural networks*, pages 967–972, 1991b.

Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992b.

Jürgen Schmidhuber. A 'self-referential' weight matrix. In *International Conference on Artificial Neural Networks*, pages 446–450. Springer, 1993.

Jürgen Schmidhuber. Reinforcement Learning Upside Down: Don't Predict Rewards– Just Map Them to Actions. *arXiv:1912.02875*, 2019.

Jürgen Schmidhuber. Annotated history of modern ai and deep learning. *arXiv preprint arXiv:2212.11279*, 2022.

Konstantin Schürholt, Dimche Kostadinov, and Damian Borth. Hyper-representations: Self-supervised representation learning on neural network weights for model characteristic prediction. 2021. URL `https://api.semanticscholar.org/CorpusID:240070334`.

Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Policy gradients with parameter-based exploration for control. In Véra Kůrková, Roman Neruda, and Jan Koutník, editors, *Artificial Neural Networks - ICANN 2008*, pages 387–396, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-87536-9.

Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, May 2010. ISSN 08936080. doi: 10.1016/j.neunet.2009.12.004.

H. T. Siegelmann and E. D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pages I–387–I–395. JMLR.org, 2014.

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

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat Prabhat, and Ryan P. Adams. Scalable bayesian optimization using deep neural networks. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 2171–2180. JMLR.org, 2015.

Rupesh Kumar Srivastava, Juergen Schmidhuber, and Faustino Gomez. Generalized compressed network search. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO Companion '12, page 647–648, New York, NY, USA, 2012.

ACM, ACM. ISBN 978-1-4503-1178-6. doi: 10.1145/2330784.2330902. URL http://doi.acm.org/10.1145/2330784.2330902.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

Rupesh Kumar Srivastava, Pranav Shyam, Filipe Mutz, Wojciech Jaśkowski, and Jürgen Schmidhuber. Training Agents Using Upside-down Reinforcement Learning. In *NeurIPS Deep RL Workshop*, 2019.

Stephen M. Stigler. Gauss and the Invention of Least Squares. *The Annals of Statistics*, 9(3):465 – 474, 1981. doi: 10.1214/aos/1176345451. URL https://doi.org/10.1214/aos/1176345451.

RL Stratonovich. Conditional Markov processes. *Theory of Probability And Its Applications*, 5(2):156–178, 1960.

Miroslav Štrupl, Francesco Faccio, Dylan R Ashley, Jürgen Schmidhuber, and Rupesh Kumar Srivastava. Upside-down reinforcement learning can diverge in stochastic environments with episodic resets. *The 15th European Workshop on Reinforcement Learning*, 2022a. URL https://arxiv.org/abs/2205.06595.

Miroslav Štrupl, Francesco Faccio, Dylan R Ashley, Rupesh Kumar Srivastava, and Jürgen Schmidhuber. Reward-weighted regression converges to a global optimum. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8361–8369, 2022b. URL https://arxiv.org/abs/2107.09088.

R. S. Sutton. Integrated architectures for learning, planning and reacting based on dynamic programming. In *Machine Learning: Proceedings of the Seventh International Workshop*, 1990.

Richard S Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst, 1984.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, USA, 2018. ISBN 0262039249, 9780262039246.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

Richard S Sutton, Hamid R Maei, and Csaba Szepesvári. A convergent $o(n)$ temporal-difference algorithm for off-policy learning with linear function approximation. In *Advances in neural information processing systems*, pages 1609–1616, 2009a.

Richard S. Sutton, Hamid Reza Maei, Doina Precup, Shalabh Bhatnagar, David Silver, Csaba Szepesvári, and Eric Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 993–1000, New York, NY, USA, 2009b. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553501.

Richard S. Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M. Pilarski, Adam White, and Doina Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '11, pages 761–768, Richland, SC, 2011. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 0-9826571-6-1, 978-0-9826571-6-4.

Richard S Sutton, A Rupam Mahmood, and Martha White. An emphatic approach to the problem of off-policy temporal-difference learning. *The Journal of Machine Learning Research*, 17(1):2603–2631, 2016.

Hongyao Tang, Zhaopeng Meng, Jianye Hao, Chen Chen, Daniel Graves, Dong Li, Changmin Yu, Hangyu Mao, Wulong Liu, Yaodong Yang, et al. What about inputting policy in value function: Policy representation and policy-extended value function approximator. *arXiv preprint arXiv:2010.09536*, 2020.

Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38 (3):58–68, March 1995. ISSN 0001-0782. doi: 10.1145/203330.203343.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

Thomas Unterthiner, Daniel Keysers, Sylvain Gelly, Olivier Bousquet, and Ilya O. Tolstikhin. Predicting neural network accuracy from weights. *ArXiv*, abs/2002.11448, 2020. URL `https://api.semanticscholar.org/CorpusID:211506753`.

Sjoerd van Steenkiste, Francesco Locatello, Jürgen Schmidhuber, and Olivier Bachem. Are disentangled representations helpful for abstract visual reason-

ing? In *Advances in Neural Information Processing Systems*, pages 14222–14235, 2019.

Eric Veach and Leonidas J Guibas. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 419–428, 1995.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, L. Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Caglar Gulcehre, Ziyun Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575:350 – 354, 2019. URL `https://api.semanticscholar.org/CorpusID:204972004`.

Johannes von Oswald, Christian Henning, João Sacramento, and Benjamin F Grewe. Continual learning with hypernetworks. In *8th International Conference on Learning Representations (ICLR 2020)(virtual)*. International Conference on Learning Representations, 2020.

A. Waibel. Phoneme recognition using time-delay neural networks. In *Meeting of the IEICE*, Tokyo, Japan, 1987.

Tongzhou Wang, Jun-Yan Zhu, Antonio Torralba, and Alexei A Efros. Dataset distillation. *arXiv preprint arXiv:1811.10959*, 2018.

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

C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1, 1988.

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

Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural evolution strategies. *The Journal of Machine Learning Research*, 15(1):949–980, 2014.

R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In *Back-propagation: Theory, Architectures and Applications*. Hillsdale, NJ: Erlbaum, 1994.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.

Peter R. Wurman, Samuel Barrett, Kenta Kawamoto, James MacGlashan, Kaushik Subramanian, Thomas J. Walsh, Roberto Capobianco, Alisa Devlic, Franziska Eckert, Florian Fuchs, Leilani Gilpin, Varun Kompella, Piyush Khandelwal, HaoChih Lin, Patrick MacAlpine, Declan Oller, Craig Sherstan, Takuma Seno, Michael D. Thomure, Houmehr Aghabozorgi, Leon Barrett, Rory Douglas, Dion Whitehead, Peter Duerr, Peter Stone, Michael Spranger, , and Hiroaki Kitano. Outracing champion gran turismo drivers with deep reinforcement learning. *Nature*, 62: 223–28, Feb. 2022. doi: 10.1038/s41586-021-04357-7.

W. Zhang, J. Tanida, K. Itoh, and Y. Ichioka. Shift invariant pattern recognition neural network and its optical architecture. In *Proceedings of Annual Conference of the Japan Society of Applied Physics*, volume 6p-M-14, page 734, 1988.

Tingting Zhao, Hirotaka Hachiya, Voot Tangkaratt, Jun Morimoto, and Masashi Sugiyama. Efficient sample reuse in policy gradients with parameter-based exploration. *Neural computation*, 25(6):1512–1547, 2013.

Allan Zhou, Kaien Yang, Kaylee Burns, Yiding Jiang, Samuel Sokota, J Zico Kolter, and Chelsea Finn. Permutation equivariant neural functionals. *arXiv preprint arXiv:2302.14040*, 2023.

Mingchen Zhuge, Haozhe Liu, Francesco Faccio, Dylan R Ashley, Róbert Csordás, Anand Gopalakrishnan, Abdullah Hamdi, Hasan Abed Al Kader Hammoud, Vincent Herrmann, Kazuki Irie, et al. Mindstorms in natural language-based societies of mind. In *NeurIPS 2023 Workshop on Robustness of Few-shot and Zero-shot Learning in Foundation Models*, 2023. URL `https://arxiv.org/abs/2305.17066`.