MAX PLANCK INSTITUTE
FOR INTELLIGENT SYSTEMS

Autonomous Learning Group

Master Thesis

Improving Policy-Conditioned Value Functions

Sebastian Griesbach

31/03/2022

Examiners

Dr. Georg Martius
Autonomous Learning
Max Planck Institute for Intelligent Systems

Prof. Dr. Martin V. Butz
Cognitive Modeling
University of Tübingen

Sebastian Griesbach

Improving Policy-Conditioned Value Functions

Master Thesis Computer Science

Eberhard Karls Universität Tübingen

in cooperation with

Max Planck Institute for Intelligent Systems

Thesis period: 01/10/2021 - 31/03/2022

Supervised by:

Maximilian Seitzer

Ph.D. Student

Autonomous Learning

Max Planck Institute for Intelligent Systems

Abstract

Current Actor-Critic Reinforcement Learning algorithms face a fundamental limitation. The critic has to implicitly learn about the current state of the policy embodied by the actor. This in turn leads to a delayed learning reaction of the critic, as it always lacks behind the actor. Policy-Conditioned Value Functions explicitly include a representation of the policy, that has to be evaluated, to the critic. Earlier research has shown that these methods are effective in Reinforcement Learning. In this thesis we build upon the current state of Policy-Conditioned Value Functions by adding several improvements. We explore three different methods for representing policies, which are the flat embedding, the neuron embedding and Network Fingerprinting, and assess their capabilities in different settings. The neuron embedding is our novel approach for policy representations. Furthermore, we establish a base algorithm for Policy-Conditioned Value Functions and explore three algorithmic variants which enable the algorithm to (1) explore multiple policies in one rollout, (2) reframe Reinforcement Learning in continuous space into a binary classification problem and (3) train multiple actors at once with one critic. Our results show that the established policy representation "Network Fingerprinting" in combination with our algorithmic variants improve the overall performance of Policy-Conditioned Value Functions.

Contents

List of Figures **4**

1 Introduction **6**

2 Background **8**

2.1 Markov Decision Processes . . . 8

2.2 Q-Learning . . . 9

2.3 Deep Q-Learning . . . 11

2.4 Actor-Critic Architectures . . . 11

2.5 Online Policy vs Offline Policy learning . . . 13

2.6 The Issue of Actor-Critic Reinforcement Learning . . . 15

2.7 Zero Order Optimization . . . 15

2.8 Policy-Conditioned Value Functions . . . 17

2.8.1 Parameter-Based Value Functions . . . 17

2.8.2 Policy Evaluation Networks . . . 19

2.8.3 What About Inputing Policy in Value Function . . . 20

3 Experiment Settings **22**

3.1 Reinforcement Learning Tasks . . . 22

3.2 Baseline . . . 23

3.3 Policy-Conditioned Actor Critic . . . 25

4 Policy Embedding **28**

4.1 Representation of Policies . . . 28

4.1.1 Flat Embedding . . . 28

4.1.2 Network Fingerprinting . . . 29

4.1.3 Neuron Embedding . . . 31

4.2 Informativeness of Embeddings . . . 32

4.3 Reinforcement Learning performance of embeddings . . . 36

4.4 Scalability of Embeddings . . . 37

4.5 Learned Input States States . . . 39

5 Algorithmic Variations **45**

5.1 N-step Policy-Conditioned Actor Critic 45

5.2 Reinforcement Learning in Continuous Space as a Binary Classification Problem 48

5.3 Multi Actor Learning 52

5.4 Best Results Comparison 57

6 Discussion **59**

7 Conclusion **63**

8 Bibliography **64**

A Appendix **67**

A.1 Algorithms 67

A.2 Hyperparameters 70

A.2.1 Baselines 71

A.2.2 Action Reconstruction 72

A.2.3 Reinforcment Learning Policy Comparison 73

A.2.4 Algorithmic Variations 73

A.3 Code Base 74

List of Figures

1 Reinforcement Learning environments 23
2 Baselines 25
3 Example policy network 28
4 Flat embedding 29
5 Network Fingerprinting 30
6 Neuron embedding 31
7 Action reconstruction 35
8 Embedding comparison Reinforcement Learning 36
9 Embedding methods scaling 38
10 Pendulum-v0 rollout state representation 40
11 Pendulum-v0 fingerprinting learned input states 41
12 Pendulum-v0 neuron embedding learned input neuron embeddings . . 43
13 N-step PCAC 47
14 Comparing PCAC runs 51
15 Multi Actor PCAC 54
16 Mean actor performance Multi Actor PCAC 56
17 Best results comparison 57

Abbreviations

DDPG Deep Deterministic Policy Gradient.

MC Monte Carlo return/sample.

MDP Markov Decision Process.

PAVF Parameter-Based Action Value Function.

PCAC Policy-Conditioned Actor Critic.

PCVF Policy-Conditioned Value Functions.

PPO Proximal Policy Optimization.

PSSVF Parameter-Based Start State Value Function.

PSVF Parameter-Based State Value Function.

TD error Temporal Difference error.

TD3 Twin Delayed Deep Deterministic Policy.

1 Introduction

In Actor-Critic Reinforcement Learning architectures the Value Function (critic) needs to implicitly learn about the actor policy to successfully evaluate, it and thus advance it [1, 12, 20]. As the actor is ever-changing throughout the training process, so is the learning problem for the critic. This leads to a situation in which the critic is chasing the actor, which might be a cause for instability in Reinforcement Learning [5]. Policy-Conditioned Value Functions (PCVF) are a new branch of Deep Reinforcement Learning with the aim to solve this issue. The primary idea behind this approach is to create a value function, which is able to not only evaluate a single policy but all policies in the policy space, or at least all policies along an improvement path [6, 11, 21]. An improvement path describes the sequence of policies an algorithm learns throughout a training run. Opposed to established Reinforcement Learning approaches, the value function here is explicitly informed about the policy it has to evaluate, as an input. PCVF bare the potential to combine advantages of offline policy and online policy learning approaches by being able to reuse Monte Carlo returns for training. This is usually not possible because Monte Carlo returns are only meaningful in combination with the policy they were sampled from [19]. In chapter 2 we explain all necessary background information Information in detail.

PCVF require a policy representation as input, therefore policy representations are a fundamental building block of PCVF [11, 21]. In chapter 4 we compare three different approaches to represent policies. We compare their capability to embed a policy through an action reconstruction task, and compare them in a Reinforcement Learning setting. Furthermore, we analyze the scalability properties of all three approaches in reference to the policy network size. The basic approach takes all the parameters of a policy and concatenates them to a single vector [6]. Network Fingerprinting was originally proposed by Harb et al. in "Policy Evaluation Networks" [11]. Here the representation is made up of the responses of the policy network to a set of learnable probe states. As a third representation we explored a new approach in which each neuron of the policy network is represented by a vector. These vectors depend on the vectors of the previous layer with translation networks in between. The vectors of the input neurons are learnable parameters. The concatenated vectors of the output neurons make up the representation of the policy. We also explore the learnable probe states and compare them to the learnable input neuron vectors of our representation approach.

Beyond the policy representations we establish a base algorithm, called Policy-Conditioned Actor Critic (PCAC), on which we build further algorithm modifications. We compare PCAC to other PCVF algorithms, state-of-the-art algorithms, as well as to three variations of itself. In chapter 5 we present and analyze those algorithms. The first variation (N-step PCAC) addresses the issue that for exploration, a policy has to perform a whole rollout to receive a meaningful Monte Carlo return. N-step PCAC proposes a method to execute multiple policies in a single rollout by using TD errors instead. This benefits sample efficiency and learning speed at the cost of stability.

An entirely new approach is the Comparing PCAC. This approach reframes the continuous Reinforcement Learning problem into a classification problem. The critic is presented with two policies and is tasked to rank their performance. We show that this method is capable to outperform the base PCAC algorithm. Additionally, we present multiple ideas to further improve this algorithm. Lastly, we make use of the generality of the critic with the Multi Actor PCAC variant. In PCVF the policy is an explicit input to the critic. Therefore we are enabled to use multiple actors simultaneously to explore the policy space. This does not require more environment interactions than using a single actor. Multi Actor PCAC does not show improved generalization across the policy but is effective in avoiding local optima. We also study the effects of combining Multi Actor PCAC with the other two variants. Finally in chapter 6 we discuss and interpret some of our findings and draw our conclusion in chapter 7.

2 Background

2.1 Markov Decision Processes

A Markov Decision Process (MDP) describes a framework for taking a sequence of decisions. It is defined by $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma, \mu_0)$. The decisions $a \in \mathcal{A}$ are made (or actions are taken) based on an observed state $s \in \mathcal{S}$. Given the state and action, the state changes to a new state s' with a certain probability given by $P(s'|s,a)$ and a reward is given based on the transition $R(s,s')$. $\mu_0(s)$ gives a probability distribution for the initial state. $\gamma \in [0,1]$ is the discount factor for future rewards, such that the expected discounted reward up to time step t is given by:

$$\mathbb{E}\left[G_t\right] = \sum_{k=0}^{T-t-1} \gamma^k \mathbb{E}_{s' \sim P(s'|s_{t+k+1}, a_{t+k+1})} \left[R\left(s_{t+k+1}, s'\right)\right] \tag{1}$$

Here, T denotes the length of an entire episode. We denote a strategy to choose actions as π, in the following called policy. π can be stochastic such that $\pi(a|s)$ gives the probability to choose action a when observing state s. π may also be deterministic, in which case the probability for some action would be 1 and 0 for all other actions. For the sake of simplicity and the scope of this thesis, we will focus on deterministic policies, the reasoning behind this is explained in more detail in chapter 3.3. If we assume deterministic transitions and a deterministic policy $s' = P(s,a)$ and $a = \pi(s)$ become functions that return the next state and action respectively, instead of a probability. The equation for the accumulated future reward then also becomes deterministic given some start state $s_0 \sim \mu_0$.

$$G_t^\pi = \sum_{k=0}^{T-t-1} \gamma^k R\left(s_{t+k+1}, P\left(s_{t+k+1}, a_{t+k+1}\right)\right), \tag{2}$$

where $a_{t+k+1} = \pi\left(s_{t+k+1}\right)$. The goal in such a setting is to find an optimal policy π^* which will always return the maximal possible future rewards G [1].

Value functions

Value functions simplify the use of MDPs. There are mainly two different types of value functions used in the context of Reinforcement Learning. First, the state value function v, which determines the value (future return) of a state s when following a specific implicit policy π: $v(s) = \mathbb{E}[G_t^\pi | S_t = s]$. Second, the Q-value function q, which additionally to the state and policy also takes an action. The returned value is the value when taking action a in state s and following implicit policy π thereafter: $q(s, a) = \mathbb{E}[G_t^\pi | S_t = s, A_t = a]$. Ultimately we are interested in finding the optimal policy π^*. This policy is the one that optimizes the value for all states according to the value function that implicitly follows the optimal policy v^*.

$$\forall s \in \mathcal{S} : \pi^* = \arg\max_{\pi} v^*(s) \tag{3}$$

The optimal policy can also be defined with the Q-value function:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} q^*(s, a) \tag{4}$$

However, neither the optimal policy nor the optimal value functions are initially known to us.

2.2 Q-Learning

Q-Learning is an iterative algorithm to solve the aforementioned optimization problem in a MDP setting in a model-free fashion [23]. Model-free means that we do not need the transition P and reward function R of the MDP, thus we do not require the environment model. Information about these processes are purely gained through observations. In Q-learning we initialize a table Q with a cell for each state and action combination. The entries of this tables are supposed to approximate the the Q-value $Q(s, a) \approx q(s, a)$ for that state action pair. For this to be possible we require finite states and actions. Initially this table is filled with zeros or random numbers. To fill this table with meaningful values, the state and action space is explored.

Each time a state transition occurs, the according cell in the table is updated as follows:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right), \tag{5}$$

where α is the learning rate and r_t is the experienced reward for this state transition. Algorithmically, this means that we update the current value closer to the experienced value considering that we are taking the best known action thereafter, due to the max operator. This difference between the current value estimation and an updated value estimation with one more experienced time step is called the Temporal Difference error or TD error $TDError = r_{t+1} + \gamma Q(s_{t+1}, \pi(s_{t+1})) - Q(s_t, a_t)$. It can also be defined using a state-value function: $TDError = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$. To effectively fill the Q-table and converge to stable values, a balance of exploration strategy and greedy exploitation strategies is needed. In Q-learning this is usually done with the ϵ-greedy strategy. The ϵ-greedy strategy, takes the best known action at a specific state $\arg\max_{a \in \mathcal{A}} Q(s, a)$ with probability $\epsilon \in [0, 1]$, an with probability $1 - \epsilon$ it takes a random action. Here, the policy is encoded directly through the Q-value function itself. Since initially there is no information about the actual Q-values, there is no benefit for taking greedy actions over random ones. Later, when the environment has been explored and its Q-values and thus the policy has made learning progress, it becomes more meaningful to select the best action for further exploration. This balance between exploration and exploitation can be handled by starting out with $\epsilon = 1$ and decreasing it asymptotically towards 0. This could be done for example by updating ϵ in each exploration step by $\epsilon \leftarrow c\epsilon$,where $c \in [1, 0]$ regulates how fast ϵ is declining. This algorithm is guaranteed to converge to the optimal Q-value function $Q = q^*$ [13]. The optimal policy π^* is then taken from the table by choosing the action with the maximal Q-Value in each state $\pi^*(s) = \arg\max_a(Q(s, a))$.

2.3 Deep Q-Learning

Q-Learning is a strong optimization algorithm for Markov Decision Processes, however it comes with a major limitation. It requires a table across all states and actions. In order to converge, the algorithm needs to iterate multiple times over the same state-action pairs. This means that Q-learning is limited to problems with finite states and actions. In a continuous state space, the probability of hitting the same state twice is zero. To overcome this limitation, Deep Q-learning replaces the Q-Table by a deep neural network parametrized by θ such that $Q^\theta(s,a) \approx Q(s,a)$. The input to the network is one state and there are as many outputs as there are actions. This way each output represents the Q-value of one specific action in that input state [14]. The Deep Neural Network can be trained with the following objective:

$$J(\theta) = \mathbb{E}_{s,a}\left[\left(y_t - Q^\theta(s_t,a_t)\right)^2\right] \tag{6}$$

$$y_t = \mathbb{E}_{s_{t+1}}\left[r + \gamma \max_{a'} Q^\theta(s_{t+1},a')\right], \tag{7}$$

where y_t is the target Q-value, and s_{t+1} is the resulting next state after taking action a. This is conceptually similar to the standard Q-learning update rule with the difference that a mean squared error is taken between the old Q-value and the updated Q-value. The network can be trained on this objective by taking the derivative and using gradient descent optimizers to train the parameters θ. The policy itself again emerges from taking the best action in each states according to the deep neural network Q-Table approximation, which can be looked up once the state is observed: $\pi(s) = \arg\max_{a\in\mathcal{A}} Q^\theta(s,a)$.

2.4 Actor-Critic Architectures

Q-Learning can only handle discrete finite state spaces and discrete finite action spaces, Deep Q-Learning extends the use case to continuous state spaces by replacing the Q-table with a deep neural network. But it still can only handle discrete finite action spaces. Actor-Critic architectures enable the use fo continuous state-spaces and continuous action-spaces by splitting the network into two networks. Actor-Critic architectures are not limited to continues space, however in the following we assume that they are used in such. The actor network takes a continuous state as input and outputs a continuous action. In the case of stochastic policies, a distribution over the actions is the output, but here we focus here on deterministic policies. For the critic there are different possibilities. In the case of the Deep Deterministic Policy Gradient (DDPG) algorithm, the critic takes both state and action as input to output the Q-value of that state-action pair [12]. This also can be seen as an approximation to a table but with infinite and continuous columns and rows. The training process consists of two steps, the critic update and the actor update, in which the according network is updated. In the case of DDPG the critic objective is the same as the object in Deep Q-Learning with a slightly different target given by the Bellman-equation:

$$J_c(\theta_c) = \mathbb{E}_{s,a}\left[(y_t - Q^{\theta_c}(s_t, a_t))^2\right] \tag{8}$$

$$y_t = \mathbb{E}_{s_{t+1}}\left[r_{t+1} + \gamma Q^{\theta_c}(s_{t+1}, \pi^{\theta_a}(s_{t+1}))\right] \tag{9}$$

where y_t is the target Q-value for time step t, r_{t+1} is the experienced reward for transition $s_t \to s_{t+1}$, γ is the discount factor for future rewards, Q^{θ_c} is the parametrized critic with parameters θ_c and π_{θ_a} is the parametrized actor with parameters θ_a. Here, the use of a max operator is no longer possible due to the infinite action space, therefore the next action a' is chosen by the actor π^{θ_a}. The critic parameters θ_c are then updated according to the gradient of the object $\nabla_{\theta_c} J_c(\theta_c)$ and some optimizer.

The actor update in turn uses the estimate of the critic network to calculate the objective and the according gradient:

$$J_a(\theta_a) = \mathbb{E}_s\left[Q_{\theta_c}(s, \pi_{\theta_a}(s))\right] \tag{10}$$

With this objective, the actor parameters are updated via gradient ascent as the approximated Q-value is subject to maximization.

Unlike Q-learning, this algorithm no longer holds any guarantees for finding the optimal policy and suffers from instability due to cyclic dependencies of the critics target [2]. However in practice these algorithms displays high performance [12]. There are methods to further stabiles this algorithm, like TD3 [7], which is beyond the scope of this thesis.

2.5 Online Policy vs Offline Policy learning

Model-free Reinforcement Learning algorithms are divided into online policy learning (on-policy) and offline policy learning (off-policy). On-policy methods may only learn from their current policy or a very similar one. In the case of actor-critic models this means that only experiences from the current actor may be used to advance the critic and the actor. An example for a state-of-the-art on-policy algorithm is the Proximal Policy Optimization (PPO) [20]. PPO does multiple update steps on an exploration batch performed by the current actor. This is possible by limiting the update step size on the actor, such that the policy does not move too far away from the exploration policy during those update steps. PPO is a stochastic Reinforcement Learning algorithm, which means that the actor does not directly output actions but a probability distribution over actions from which the actual action is sampled. Besides some other loss components, the objective mainly consists of an advantage function $A(s, a) = Q(s, a) - V(s)$, where Q is the Q-value function and V the state value function of an implicit policy. When the Action a has been executed and the resulting reward has been observed, the Q-value function may also be defined as $Q(s, a) = r + \gamma V(s')$, where γ is the discount factor and s' the next state [8]. Therefore, we can write the advantage function as $A = r + \gamma V(s') - V(s) = TDError$ which is exactly the formulation of the TD error using the state-value function, instead of the Q-value function.

One of the main advantages of on-policy methods is that observed returns, or Monte Carlo returns, may be used at the end of an episode as the current policy has been executed and therefore we have a sample of the value function for all visited states, opposed to only using estimates. However, this only works as long as the actor policy does not deviate too far from the policy that experienced this rollout. Thus, data that has been gathered in the past may not be reused in the future.

Off-Policy on the other hand may reuse any experiences made by any policy. An example of a state-of-the-art off-policy algorithm is the Twin Delayed Deep Deterministic Policy Gradient (TD3) [7]. It is an extension of the Deep Deterministic Policy Gradient (DDPG) [12] which addresses mostly stability concerns by reducing overestimation of the approximated Q-values of the critic. Off-policy methods also use the TD error, but they do not make use of the Monte Carlo return. Instead they approximate it with a parametrized Q-value function, typically a neural network, as described in chapter 2.4. While those approximations are less reliable than the Monte Carlo returns, they can be reused at a later point in training. In the TD error with the Q-value function $TDError = r + Q^{\theta}(s', a') - Q^{\theta}(s, a)$ the only assumption is that the reward r will occur when action a is executed in state s. If the environment is deterministic, this is always true regardless of the underlying policy. If the environment is stochastic environment the expected reward would be would always be the same regardless of the underlying policy. Therefore the gathered experiences may be reused to train any other policy or to be more specific, their according value functions. However, Monte Carlo returns for these state action pairs with respect to the current policy are not available, therefore both instances of the value functions are merely approximations. This may lead to instability during the training process [2].

Both on- and off-policy approaches have advantages and disadvantages. PPO and in general on-policy approaches are known to have robots training behavior due to the availability of Monte Carlo returns, with makes the use of self referential approximations as targets obsolete. Off-policy algorithms on the other hand are often more sample efficient as they reuse previously made experiences and thus need less interactions with the environment. In scenarios, where interacting with the environment is costly, this may be beneficial.

2.6 The Issue of Actor-Critic Reinforcement Learning

In Reinforcement Learning, within the actor-critic framework, the value function has to approximate the value of a specific implicit policy. This implicit policy is usually a past version of the actor combined with exploration mechanisms. However this policy is constantly changing during the training process thus, the learning problem for the value function is changing as well. The paper "The Value-improvement Path: Towards Better Representations for Reinforcement Learning" [5] describes this issue in depth. The Reinforcement Learning process is seen as a sequence of learning problems instead of a single learning problem. In some sense the critic has to chase the actor as the critic has to implicitly model the current behavior of the actor. This sequence of learning problems is called the value-improvement path.

As the policy is produced or updated by the value function, we are interested in the optimal value function. They propose a method to achieve better generalization across the improvement path, which potentially could result in faster progression of the value function. Their proposed method consists of splitting the value function into a learnable representation and a linear transformation. The linear transformation may be optimally calculated for the current experienced state, action, reward data. The representation after the learnable component however, is supposed to be more consistent over the whole value improvement path such that it may be useful for future value functions further along the path (training progress). Different auxiliary losses are employed to achieve this effect [5].

While our approach follows a different idea, Dabney et al. [5] gives a foundation for the problem that Policy-Conditioned Value Functions try to solve, namely a generalized value function across the space of policies.

2.7 Zero Order Optimization

Zero order optimization methods are black box optimization methods which do not assume anything about the process that is being optimized. They only require a fitness function $f(z)$. This fitness function evaluates how well the input z performs in a given task or process. Such methods do not require the fitness function to be differentiable. One very successful branch of those methods are evolution strategies [17]. Those strategies represent the inputs z to the fitness function f as a batch of genomes, the so-called population. Genome here simply refers to a representation of an individual, an instance of z. Each individual in the population is evaluated by the fitness function. The best performing candidates are kept while the rest is discarded. For the next population, the survivors of the last population are multiplied and mutated. This simple idea is based on biological evolution. It is effective but not necessarily efficient. Instead of modelling the populations by individuals, one can also model the population by probability distributions of the genomes [10]. If those probability distributions are parametrized $P(z|\theta)$ and differentiable with respect to the parameters θ, then instead of randomly mutating the best individuals the whole population distribution may be adjusted with a gradient ascent methods towards a higher performance. However, Wiersta et al. [24] observed that these methods are instable as the step size of this learning problems depends on the uncertainty in an undesirable manner. High uncertainty leads to tiny steps, thus slowing down the learning progress. Low uncertainty exaggerates the step size near the optimum and therefore often oversteps. The step size is also dependent on the euclidean distance in the parameter space of the probability distributions. Since this can vary between different parameterizations and probability distributions, it is also undesirable. Wiersta et al. proposes nature evolution strategies [24] which instead scale the step size base on the Kullback-Leibler divergence between the start and end distribution of an update step. This way directions with high uncertainty are diminished while directions with low uncertainty are enforced. This also fixes the instability issue near the optimum [24].

Reinforcement Learning may also be seen as a zero order optimization. The fitness function f is the rollout of the environment and the input to that function is a policy π. The analogy to natural evolution strategies comes with the Policy-Conditioned Value Function. Natural evolution strategies try to model a probability distribution over populations and extract an improvement direction from the gradient [24]. Policy-Conditioned Value Functions are very similar in the sense that they take in a policy and try to predict its value when executed in the environment [6]. As the Value function itself is fully differentiable, when modeled by a neural network, we also take the gradient to extract a proposed improvement direction for the policy. In this sense, the Policy-Conditioned Value Function takes on the same role as the parametrized probability distribution in natural evolution strategies. Both PCVF and Natural Evolution Strategies are improved by samples of the real fitness function.

2.8 Policy-Conditioned Value Functions

As previously established, value functions approximate the value of a state with respect to some implicit policy π:

$$V^{\pi}(s_t) \approx \mathbb{E}\left[\sum_{k=0}^{T-t-1} \gamma^k R\left(s_{t+k+1}, P\left(s'|s_{t+k+1}, \pi\left(s_{t+k+1}\right)\right)\right)\right] \tag{11}$$

We denote this implicit dependency by V^{π}. As the policy changes, the value function also has adapt, as described in chapter 2.6. The concept of Policy-Conditioned Value Functions (PCVF) is instead to learn a sequence of value functions $V^{\pi_1}(\cdot), V^{\pi_2}(\cdot), V^{\pi_3}(\cdot), \dots$ to include the policy itself in the inputs to the value function $V(\cdot, \pi_1), V(\cdot, \pi_2), V(\cdot, \pi_3), \dots$. This way, the Value function has the potential to generalize across different policies. The foundation for PCVF approaches was laid by the papers "Parameter-Based Value Functions" [6] and "Policy Evaluation Networks" [11]. Besides these and a third paper "What About Inputing Policy in Value Function" [21], only little research has been conducted so far on Policy-Conditioned Value Functions.

2.8.1 Parameter-Based Value Functions

Parameter-Based Value Functions assume that the policies are parametrized by θ. Three different variations are proposed, which we will explain in the following.

Parameter-based Start State Value Function, PSSVF The critic receives only the parameters $V(\theta) = \mathbb{E}\left[G_t^{\pi_\theta}|s_t = s_0\right]$. In the PSSVF variation, Monte Carlo returns are included in the objective $J_{PSSVF}(\theta_c) = \mathbb{E}_{\theta_a}\left[(V^{\theta_c}(\theta_a) - V^{MC}(\theta_a))^2\right]$. This means a policy is executed once in the environment such that a Monte Carlo return for this policy is sampled. If the environment does not have a static initial state, the critic is not informed of the starting conditions and has to implicitly model the start state distribution. One rollout yields only one data point to train on, consisting of the policy and Monte Carlo return. In terms of sample efficiency measured by the time steps in the environment this is worse compared to other algorithms which make use of each time step in the rollout. However, the collected data can be reused indefinitely, although after a certain point the reuse of the data has no effect because the critic has learned the information it provides. It has been shown that in practice the PSSVF performs well despite this limitation [6].

Parameter-Based State Value Function, PSVF The critic receives states and parameters $V(s, \theta) = \mathbb{E}\left[G_t^{\pi_\theta} | s_t = s, \theta\right]$. The PSVF opposed to the PSSVF version generates one data point for each time step of the environment. During training The critic receives the policy parameters, the state and the TD error as its target. The TD error here is based on the value function and not on the Monte Carlo return. The objective is the same as for the DDPG update step seen in equation 8 with the difference of using a state-value function instead of a Q-value function and having the policy parameters as an additional input to the value function.

$$J_{PSVF}(\theta_c) = \mathbb{E}_{s,\theta_a}\left[(y_t - V^{\theta_c}(s_t, \theta_a))^2\right] \tag{12}$$

$$y_t = \mathbb{E}_{s_{t+1}}\left[r_{t+1} + \gamma V^{\theta_c}(s_{t+1})\right] \tag{13}$$

As each time step yields one data point, more data is available to train the critic, although instability due to the TD error might be an issue here. Here the critic is informed about the state and can take this information into account for the value estimation [6].

Parameter-Based Action Value Function, PAVF In this setting the critic receives state, action and parameters $Q(s, a, \theta) = \mathbb{E}\left[G_t^{\pi_\theta} | s_t = s, a_t = a, \theta\right]$. The state action variant is similar to the state variant but additionally to the state the action is provided as input to the value function. Since the parameters determine which action a deterministic policy choses this is redundant information. The idea is that this additional input, to the critic, helps the critic to learn how to interpret the policy parameters as it gives an additional gradient flow channel thorough these actions [6].

To represent the policy, Faccio et al. [6] use a vector including all parameters of the policy network. This approach works on shallow policies and small multi-layered-policies. It outperforms augmented random search and is competitive to DDPG in some environments. It has been shown that one shot learning is possible with this method. This means with a trained critic, a newly initialized policy, may be updated without interacting with the environment. The paper shows that this may lead to similar results as training policy and critic simultaneously and even outperforms it sometimes[6].

This thesis is mostly based on the approaches of Parameter-Based Value Functions but also makes use of the fingerprinting policy representation which originates from Policy Evaluation Networks.

2.8.2 Policy Evaluation Networks

Taking the parameters of a policy as a vector, intuitively seems to be an unscalable approach. Policies which display the same behavior may have vastly different representations and the needed parameters in the critic grow rapidly with the parameters of the policy when using a fully connected layer. Policy Evaluation Networks propose a different representation [11]. Here, a set of input states are learned, in the paper called probe states. The resulting actions, when these input states are fed into a policy, are used as representation of the policy. This method is called Network Fingerprinting. In our approach we also took advantage of this representation method and will be further discussed in chapter 4.1.2. The framing of the algorithm of Harb et al. [11] is differs from Parameter-Based Value Functions. A distributional prediction as a classification problem for the value of a policy is used, instead of a prediction of the actual value. In regards to one-shot learning, it has been shown, that a critic trained purely on a spectrum of random policies enables the critic to produce new policies which outperform any policy it has seen so far. The difference here is that the critic has not been trained along an actor but the policies have been randomly initialized and rolled out to gather data. Therefore the generalization capability of this approach has been demonstrated [11]. However these experiments are limited to small scale policy networks and environments which can also be solved with random search. For our experiments we used fingerprinting and found it to be very effective but we did not use the distributional prediction.

2.8.3 What About Inputing Policy in Value Function

In "What About Inputting Policy in Value Function: Policy Representation and Policy-extended Value Function Approximator" [21] Tang et al. build upon the same bases as Policy Evaluation Networks [11] and Parameter-Based Value Functions [6] with some key differences. The focus is on developing one specific algorithm rather than exploring the space. As an input setting they only use the Q-value version where the value function receives a representation of the policy, a state as well as an action, similar to the PAVF approach of Parameter-Based Value Functions. They propose different policy representation strategies. The raw representation is the same as the one Faccio et al. [6] used, simple vectorizing of all policy parameters. The Second representation is the Surface Policy representation, which is a set of state action pairs of a policy. It differs from fingerprinting because here the input states are not learned but stem from a rollout of that specific policy. A third representation is the Original Policy Representation which uses all parameters of a policy as well as state action pairs. Here, the input is passed through a permutation invariant transformation and a feed forward network. They also propose different strategies on training this method:

- Action prediction, where the representation network learns to predict the action of a policy in a specific state, based on that policy's representation.
- Contrastive Learning, similar policies should be close to one another in their

representation while different policies should have a large (euclidean) distance.

- End-to-End Learning, the representation is trained along side the critic based on the critic objective in a Reinforcement Learning setting.

The aim of their approaches is also to generalize within the improvement path as referred to in "The Value-Improvement Path" [5] rather than across the whole policy space [21].

The paper shows results of their approach outperforming a PPO baseline with up to 40% better performance [21]. In the HalfCheetah-v1 [22] (see chapter 3.1 for description) their PPO benchmark reaches a mean score of 2621. Their best approach on that environment reaches a mean score of 3725. Unfortunately we cannot verify this benchmark as HalfCheetah-v1 is an outdated version of the environment. Executing the PPO baseline of the RL Baselines3 Zoo package [16] on HalfCheetah-v2 reaches an average score of 5616. It is unclear whether the different version of the environment has significant impact on these scores.

3 Experiment Settings

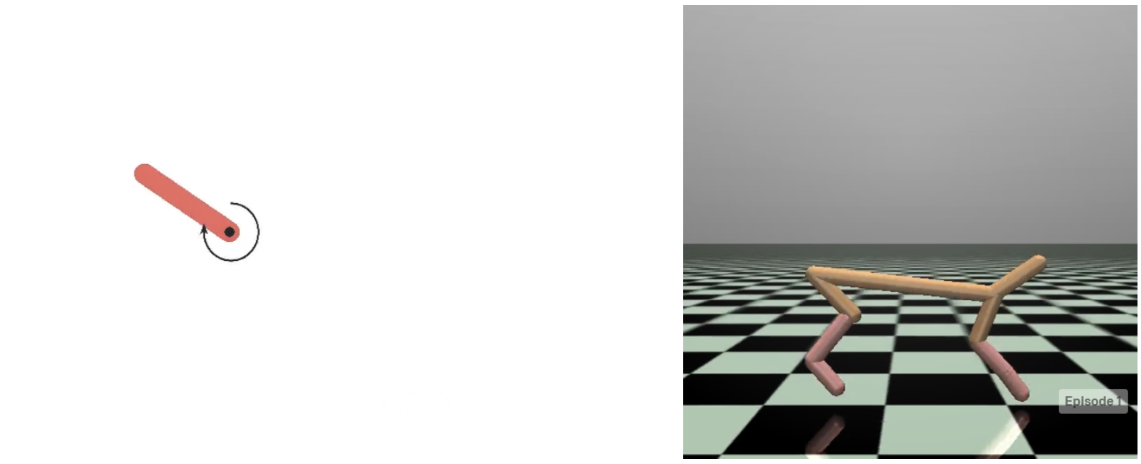
3.1 Reinforcement Learning Tasks

In this thesis we explore new algorithmic ideas. As each of these algorithms requires hyperparameter optimization for fair comparison, we opted to only use two different environments to limit the computational cost. We use well-established OpenAI Gym [3] environments. These environments are commonly used, such that most relevant algorithms have been tested on these environments. We decided on using the Pendulum-v0 (figure 1a) and the HalfCheetah-v2 (figure 1b) environments. Both are physics-driven tasks, Pendulum implemented with the Box2d engine [4] and HalfCheetah with the MuJoCo engine [22]. As outlined below the pendulum is meant to be a simple task and HalfCheetah is supposed to be a more advanced challenge for our algorithms. Both environments are continuous in their action- and observation-space. The following is a description of the tasks the agents have to learn in those environments.

Pendulum-v0 Pendulum-v0 is an environment in which the agent is provided with the velocity and angel of a pendulum. The goal is to balance this pendulum upright. To achieve this, the agent can apply angular force onto the pendulum. However, the force is not strong enough to just push the pendulum up. It has to swing the pendulum upwards and then keep it stable at the top to solve the environment (See figure 1a). The Observation dimensionality is 3 and the action dimensionality is 1. The environment terminates after 200 time steps.

HalfCheetah-v2 In HalfCheetah-v2 the agent has control of a body that somewhat resembles half of a cheetah. The body may only move in two dimensions and therefore cannot fall on the side. The agent is provided with the position,the velocity and the angle of its body as well as the angle and torque of its joints. The task for the agent is to run as far as possible within one episode by acting force angular on the joints of the body (See figure 1b). The Observation dimensionality is 17 and the action dimensionality is 6. The environment terminates after 1000 time

Figure 1: Reinforcement Learning Environments



(a) A rendered state of the Pendulum-v0 gym environment.

(b) A rendered state of the HalfCheetah-v2 gym environment.

steps, however during training rollout we limit this to 300 time steps. The reasoning behind this is to enable algorithms to explore more different policies with fewer time steps in the environment. Evaluation rollouts are executed with the regular 1000 time steps limit.

3.2 Baseline

To compare the performance of our algorithms we use several benchmarks. As widely used algorithms, we use PPO and DDPG. We outline these algorithms in chapter 2.5. More specifically, we present the differences of online and offline learning with PPO and TD3 as examples. TD3 is an improved variation of DDPG. This thesis does not focus on improving specific approaches, but rather to explore conceptual ideas of general algorithms in the field of PCVF. Since we are attempting to advance the foundation for Policy-Conditioned Value Functions, it seems more suitable to compare to this to a more bare-bones version of an algorithm, hence DDPG. DDPG is a deterministic off-policy algorithm, like all of our approaches. PPO on the other hand is a stochastic on-policy algorithm. Using both algorithms as a baseline should therefore give a good indicator for the overall performance. These algorithms are widely used therefore benchmarks are available for all common Reinforcement

Learning tasks. We did not implement and optimize these algorithms ourselves ,instead we used the RL Baselines3 Zoo package [16]. This package contains many pertained Reinforcement Learning agents of different algorithms for different tasks. See Appendix A.2.1 for details on how these benchmarks were evaluated. While the DDPG and PPO benchmarks give a good indicator for performance comparative to state-of-the-art Reinforcement Learning algorithms, we are specifically interested in advancing Policy-Conditioned Value Functions. This thesis relies on the previous work of Faccio et al. [6]. Therefore, the "Parameter-Based Value Function" [6] will be used as another benchmark to indicate whether our approaches hold an advantage over other PCVF methods. We build our own implementations of PSVF and PSSVF (see chapter 2.8.1) and optimized them alongside our methods. Because, optimizing for the HalfCheetah-v2 environment is computational costly, we opted to only optimize PSVF, as this method is the most similar to our base algorithm State Policy-Conditioned Actor Critic (PCAC). Small scale experiments, which we did not include, show that Start State variants tend to perform worse in HalfCheetah-v2. We suspect this is due to the more complex states, which in turn hold more information. While a pendulum can relatively easy recover from any state, in HalfCheetah for some states a specific policy might be needed. For example if the HalfCheetah lies on its back.

Figure 2 shows all benchmarks in the two environments. For the agents from RL Baselines3 Zoo, we only display the final average return. For the methods we implemented ourselves we also show the training progress. Our implementations of PSVF reaches a low score compared to the scores reported in the original paper. However, the original paper did not include these environments in its experiments. Therefore, we cannot say with certainty whether this is due to our optimization or if these tasks are more challenging for these algorithms. In the approaches we optimized ourselves, policy networks that are trained on the Pendulum-v0 environment have one hidden layer with 64 neurons. All policies trained on the HalfCheetah-v2 environment have two hidden layers with 128 neurons each. The DDPG and PPO benchmarks follow the setting of the RL Baseline3 Zoo [16]. We will refer back to

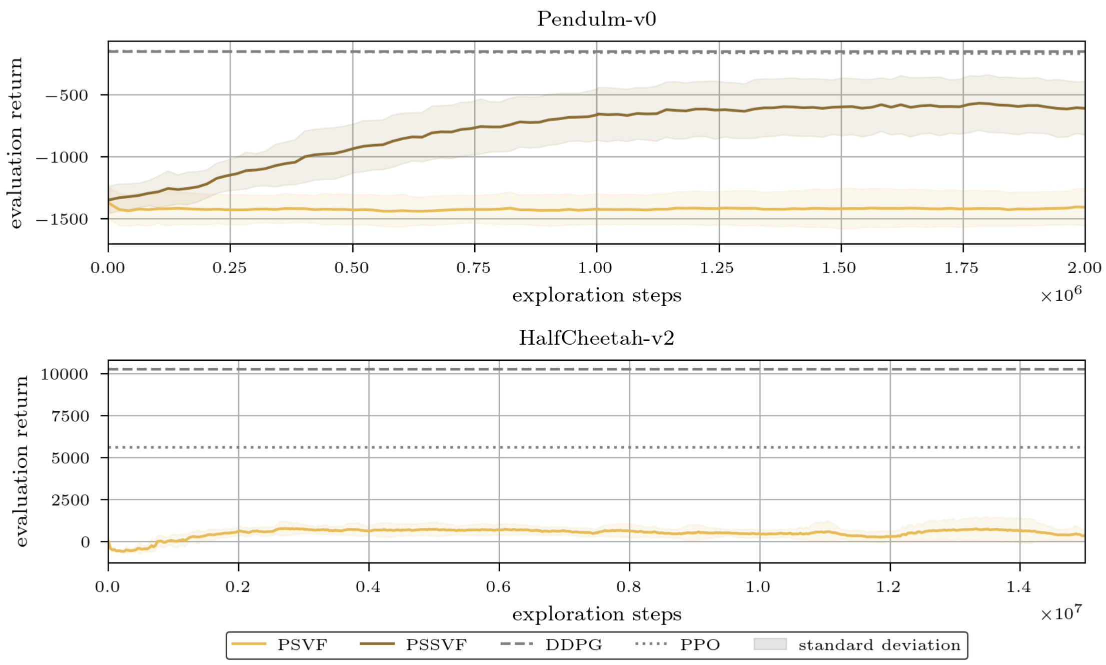


Figure 2: Training results of Parameter-Based Value Functions, DDPG and PPO benchmarks.

these benchmarks in chapter 5.4. They are not included in other experiments to reduce clutter in the plots.

3.3 Policy-Conditioned Actor Critic

We call our base algorithm Policy-Conditioned Actor Critic (PCAC). Similar to Parameter-Based Value Functions, there are different variations of this algorithm. State PCAC is our analog to PSVF. The algorithm does not assume a specific representation for the policy. In the following Pseudocode (see algorithm 1) the specific representation is omitted by using the policy embedding function $\phi(\pi)$. In contrast to the similar PSVF algorithm, we do not use the TD error but instead the Monte Carlo return (MC) for each time step. In PCVF we do not need to approximate the Monte Carlo return with a TD error, as it is available after a rollout and may be reused for training in combination with the policy that achieved this return. This allows to circumvent the instability issues of self-referential TD errors. Using deterministic policies and deterministic environments, exact data of state, policy and

MC is produced. Conceptually, we do not need to revisit any data point as the MC return should always be the same due to determinism. However, this is not entirely true, if the environment does not have a specific terminal state but rather terminates after a set amount of time. In this case the state might not be informative about the remaining time, thus data becomes stochastic. Both environments we use, terminate after a set amount of time. Due to this property, we additionally use a discount factor on the MC returns. This effectively results in a reduced deviation of the experienced state-values, which we suspect eases the task of learning them. Another difference is that we made slight modifications to the exploration process. PSVF always uses a noisy version of the current policy for exploration. Thus, the critic never experiences the actual policy of the current actor. We found that during some training runs, the critic overestimates the performance of the current policy. To counteract this effect, we used a variant of ϵ-greedy exploration: With a probability of $\epsilon \in [0, 1]$ the current policy is used for exploration without the addition of noise. This way the critic obtains data regarding the current policy which corrects the overestimation error. Also we introduced another parameter $\mathit{random\,policy} \in [0, 1]$, such that with probability *random policy* a newly initialized policy network is used for exploration. The idea behind this to achieve better generalization capabilities across the whole policy landscape, by using policies outside of the current improvement path. Both parameters (ϵ, *random policy*) were subject to optimization in our experiments. They were found be be useful sometimes but mostly they were not used after optimization (set to 0). The experiment settings in appendix A.2 show in which experiments these parameters are used. Besides these differences (MC and exploration) State PCAC is identical to PSVF. Algorithm 1 shows the pseudocode. The pseudocode of other mentioned algorithms can be found in appendix A.1. Start State PCAC is identical to PSSVF with exception of the aforementioned exploration mechanism. The PAVF equivalent, state action PCAC has not been explored in this thesis.

Algorithm 1: State Policy-Conditioned Actor Critic

```
Input: Differentiable Critic $V_w(s, \phi(\pi)) : \mathcal{S} \times \Pi \to \mathcal{R}$ with parameters $w$ and
  policy embedding function $\phi$; deterministic actor $\pi_\theta : \mathcal{S} \to \mathcal{A}$ with
  parameters $\theta$, state space $\mathcal{S}$ and action space $\mathcal{A}$.;
Output: Learned $V_w(s, \phi(\pi)) \approx V^\pi(s) \forall s \in \mathcal{S}, \forall \pi \in \Pi$, learned
  $\pi_\theta(s) \approx \pi^*(s) \forall s \in \mathcal{S}$;
initialize critic and actor parameters $w, \theta$;
while Stop criteria not met do
    create exploration policy $\pi_{\theta_e} = f_e(\pi_\theta)$;
    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 $\pi_{\theta_e}$;
    $t \leftarrow 0$;
    while $t < T$ do
        $mc_t \leftarrow \sum_{k=t}^{T} \gamma^{k-t} r_t$;            // discounted return with $\gamma \in [0, 1]$
        Store $(s_t, mc_t, \pi_{\theta_e})$ in replay buffer $D$;
        $t \leftarrow t + 1$;
    end
    for some steps do
        Sample a batch $B = (s, mc, \pi)$ from D;
        Update critic by stochastic gradient descent
          $\nabla_w \mathbb{E}_{(s,mc,\pi) \in B} [mc - V_w(s, \phi(\pi))]^2$
    end
    for some steps do
        Sample a batch $B = (s)$ from D update actor by gradient ascent:
          $\nabla_\theta \mathbb{E}_{s \in B} [V_w(s, \phi(\pi_\theta))]$
    end
end
```

4 Policy Embedding

4.1 Representation of Policies

In Policy-Conditioned Value Functions, the critic expects to receive a representation of a policy. The three papers discussed in chapter 2.8 propose different representations [6, 11, 21]. Additionally to the vectorization approach of "Parameter-Based Value Functions" [6] and fingerprinting [11] we introduce a new representation method to embed a policy. To conceptualize and explain all embedding methods, we present them in unified graphics. To illustrate, we use a small policy with one hidden layer (see figure 3) and show how this it would be embedded by the three different approaches. For the experiments in this chapter we used a policy network with one hidden layer of 64 neurons for Pendulum-v0, HalfCheetah-v2 and random networks.

4.1.1 Flat Embedding

The flat embedding is the straight forward idea to represent the parameters of a policy in a vector, as shown in figure 4. This method has been used in Parameter-Based Value Functions [6]. However, this comes with several drawbacks.First, it scales in an undesirable manner with the policy network. We discuss this in detail in chapter 4.4. Second, this method is not invariant to neuron permutation. If two

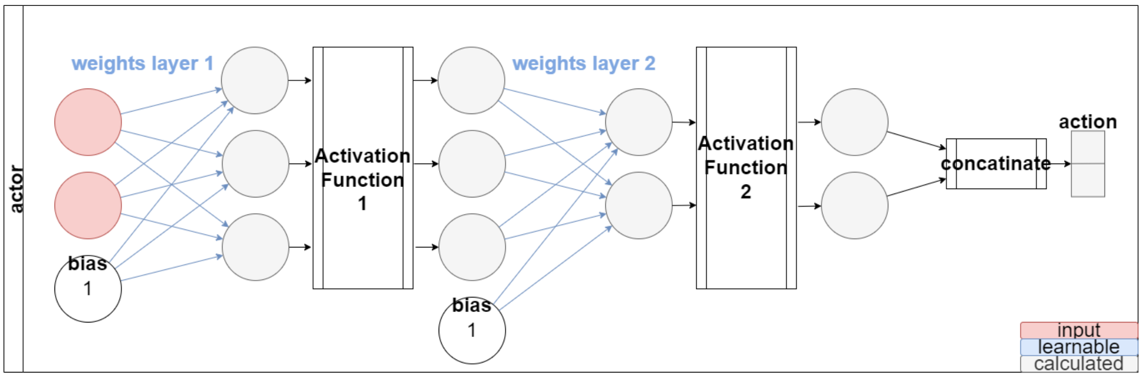


Figure 3: Small example policy network with two input neurons, one hidden layer with three neurons and two output neurons. The output neurons are concatenated to build the action of this policy network.

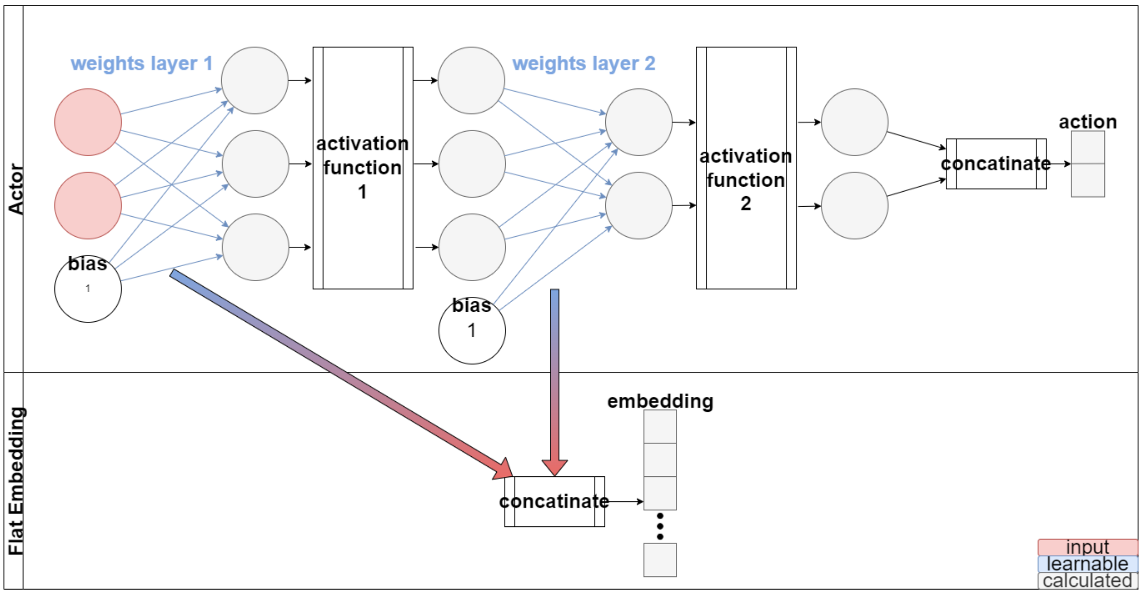


Figure 4: The flat embedding concatenates all learnable parameters of the policy into one vector.

neurons in the same layer switch their position, including all their connections, the resulting network will behave identically, but the order of parameters will change and therefore the resulting policy representation as well. Third, this representation method requires all policies, that are to be evaluated, to have the exact same architecture. Otherwise, the dimensionality and order of the resulting embedding vector would change, providing the critic with inconsistent data. Nevertheless, Faccio et al. [6] have shown that this policy representation fed to a critic is effective in Reinforcement Learning tasks with shallow and small multilayer policies [6].

4.1.2 Network Fingerprinting

The fingerprint embedding, or Network Fingerprinting as originally called in ”Policy Evaluation Networks”, avoids many issues of the flat embedding [11]. Figure 5 shows how this approach works. This representation uses a learnable set of input states, which are passed through the policy network. The resulting actions are concatenated into the embedding vector. Since the embedding method can be considered a component of the critic, and the process is fully differentiable, the input states may be optimized alongside the rest of the critic parameters. This allows to adjust

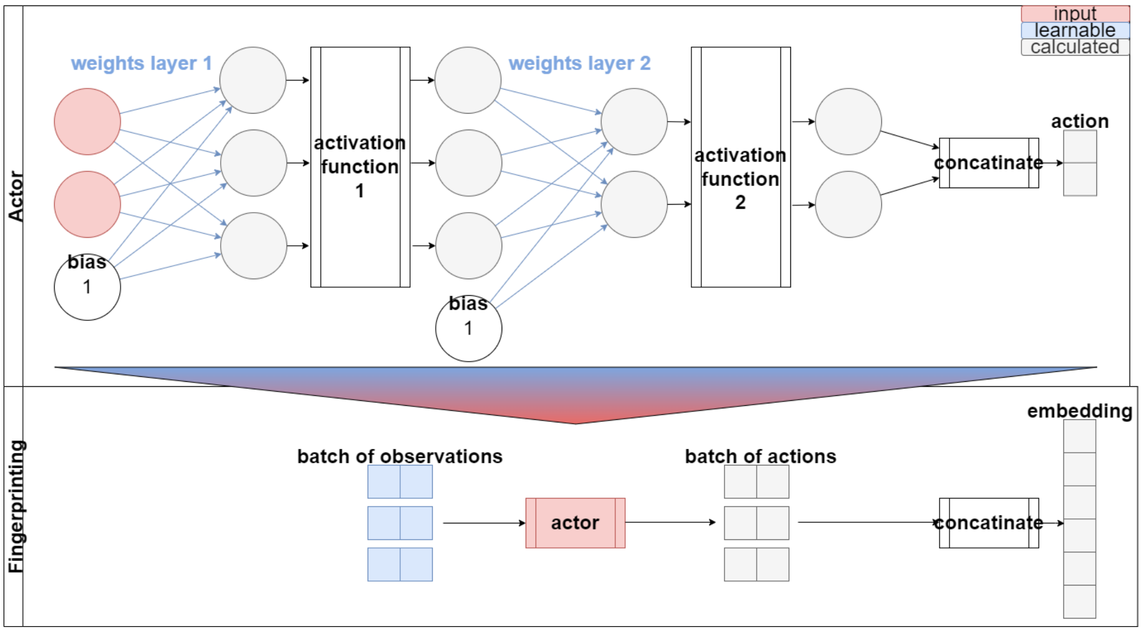


Figure 5: The fingerprint embedding learns a set of input states and concatenates the response actions given by the policy.

the input states through gradient optimization following the critic objective during training. In chapter 4.5 we show how these states are distributed after training. Fingerprinting has several advantages over the flat embedding. First, number of learnable input states is a hyperparameter which also controls the size of the embedding. Second, even though larger policies potentially require more embedding dimensions to minimize information loss, they still can be a lot smaller than the combined size of all policy parameters. In chapter 4.5 we present an example with a small policy representation. See chapter 4.4 for details of scalability. Third, policies which are similar in their behavior are similar in their embedding, as the embedding is just the response of the policy to a set of states. This also makes fingerprinting invariant to neuron permutations. Finally, this method does not require policy networks to have the same architecture. As long as input and output layers of the policies have the same dimensionality, they will result in an embedding vector of the same structure. However, whether a critic could actually deal with this setting is unclear. All experiments following chapter 4 are conducted using fingerprint as policy embedding, due to the results outlined in chapters 4.2, 4.3 and 4.5.

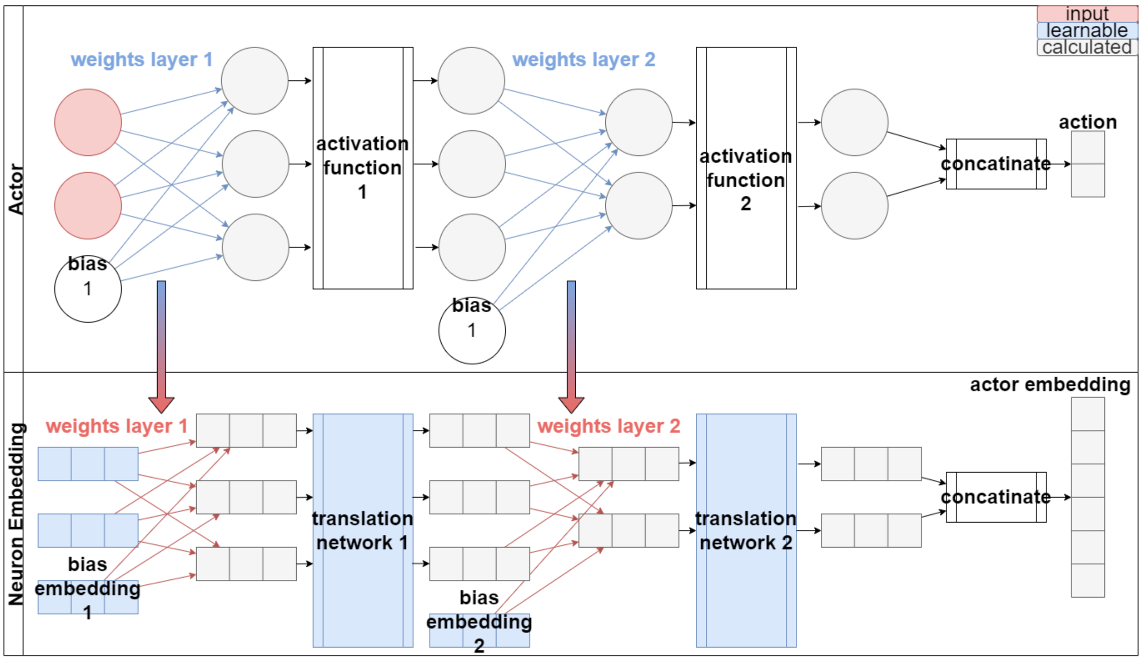


Figure 6: The neuron embedding aims to learn a meaning representation for each neuron of the policy. It imitates the forwards pass of the policy with additional small shared networks in between the layers. Other learnable parameters are the input neuron embeddings as well as bias vectors.

4.1.3 Neuron Embedding

The neuron embedding is our proposal for policy representations. It acts similar to the fingerprinting, but was developed independently and originally inspired by ”Permutation-equivariant Neural Networks applied to dynamics prediction” [9]. Guttenberg et al. [9] use neural networks to approximate the interaction between particles. Particle representations are put as pairs through small shared neural network. These networks are trained to model the interaction between two particles. Pooling operations, grouped by individual particles, create the next set of representations. This is a general approach but needs to consider n^2 pairs, as each particle has to interact with every other particle and only two particles are considered at a time. In contrast to modelling the interactions between particles, we know the exact interaction of a set of neurons in a policy network. It is simply a linear (affine) transformation, which already includes pooling (summation over inputs). The idea of introducing small shared neural networks between intermediate representations inspired the neuron embedding method.

Figure 6 shows a visualization of the neuron embedding. Similar to fingerprinting, the neuron embedding has a set of learnable input vectors. However, these vectors do not represent states of the environment but embeddings of the policy input neurons. Here, we do not choose the number of input states, but the dimensions for the initial embedding for these input neurons. This can be interpreted as being the same up to a transposition. These initial embeddings are then transformed according to the weights of the policy network which results in a new set of vectors, with one vector for each neuron in the hidden layer. Additionally, a learnable bias embedding is added and scaled according to the bias values of the policy network. In the next step, each of these intermediate vectors is passed separately through a small neural network. The idea behind this is to enable the network to translate the neuron embeddings from the first layer to a set of neuron embeddings referring to the second layer. Different layers in a neural network tend to take on different tasks. Therefore, this translation might be helpful to learn meaningful embeddings. This process continues until the output neurons of the policy network are reached. Then the output neuron embeddings are concatenated to create the policy representation.

In the neuron embedding the size of the policy representation is controlled by the input neuron embedding dimensionality. Identical policies with permuted neurons will have identical embeddings but the closeness of similar policies is not guaranteed in with neuron embeddings. This method may embed different architectures of policy networks, but besides the number of input and output neurons, the number of hidden layers also has to be the same. The number of neurons within those layers may vary.

4.2 Informativeness of Embeddings

In this section we compare the previously presented with respect to their informativeness in two settings. In the first setting, an embedding is used to reconstruct the action of a policy in a specific state. In the second setting, we compare the three embeddings in our two chosen Reinforcement Learning tasks with our State

Policy-Conditioned Actor Critic algorithm.

To determine whether the three policy representations contain information, we set up the following experiment: First, we created a dataset which contains policy networks, environment states and actions according to the policy networks. All policies are deterministic. Secondly, we used the dataset to train a decoder network, alongside the embedding networks. The decoder receives the policy representation and a state to predict the action of a policy. The decoder and embedding network are trained in a supervised manner.

To obtain the dataset, we run the TD3 [7] algorithm on Pendulum-v0 and HalfCheetah-v2 environments. After some training steps the trained policy network is saved and several rollouts are performed and recorded. Then training continues and the process repeats. Each individual policy is only trained for a limited amount of time. On regular intervals, the policy is newly initialized before continuing training. This diversifies the set of policies as they are not all part of the same improvement path. Another part of the dataset are random networks. Here, no environment or training algorithm is used. We simply initialize random policies, put in normal distributed noise as states and record the outputs as actions. The data from HalfCheetah-v2, Pendulum-v0 and random networks then is split into a test- and validation set.

To ensure a fair comparison, all methods are restricted to the same embedding dimensionality. Since the flat embedding has no possibility to control its embedding size, we use a linear embedding instead. This means that the policy parameters are passed through a single neural network layer with the desired output dimensionality. In a setting where the flat embedding is used, this would happen anyway, thus it does not change the concept of the flat embedding.

Comparing reconstruction losses gives an indicator about the informativeness of policy representations. However, this does not indicate how well the decoder distinguishes between different policies through their representations. To address this, we introduce another comparison, the embedding information advantage: After the action reconstruction training finished, the decoder is tested again. We pass through all available data points (training and validation data) and calculate the reconstruction loss. However, we do this in two settings. In the first setting, the true policy embeddings of the data points are passed to the decoder alongside the states. In the second setting, the average over all policy embeddings is passed to the decoder instead of the actual embeddings. So the decoder receives the same embedding for all data points. This reveals how much information is gained through the embeddings and how much information is implicitly learned by the decoder and the embedding network. If the performance with the true embeddings is better, this means that the embedding holds useful information about the specific policies. If the difference is small, this means that much of the information is learned on average and the actual policy representation is effectively not taken into account.

Figure 7 shows the results of all embedding methods on all datasets. For Pendulum-v0, we see that the linear embedding is overfitting and that the training loss is worse than the validation loss of the other two methods. Fingerprinting and neuron embedding generalize well, with fingerprinting outperforming the neuron embedding. Looking at the embedding information advantage, the linear method benefits significantly from true embeddings within the training data but less within the validation set. This is in line with overfitting. Both the neuron embedding, as well as fingerprinting, gain much performance through the true embeddings. HalfCheetah-v2, which is depicted in the middle row, is a more complex environment. Here, the linear method also overfits but reaches a lower training loss than the other methods. Neuron embedding and fingerprinting perform similar. Both generalize to a lesser extent than they did for Pendulum-v0. The embedding information advantage shows that the linear method performances even better on the validation set with the mean embeddings instead of the true embeddings, which indicates extreme overfitting. Further, the neuron embedding seems to be more depend on the policy representation than the Fingerprinting in this scenario. The results of the random networks, depicted in the bottom row, are similar to the results to the pendulum environment. The linear embedding is overfitting again, neuron embedding and Fingerprinting have perfect generalization, but fingerprinting is out-

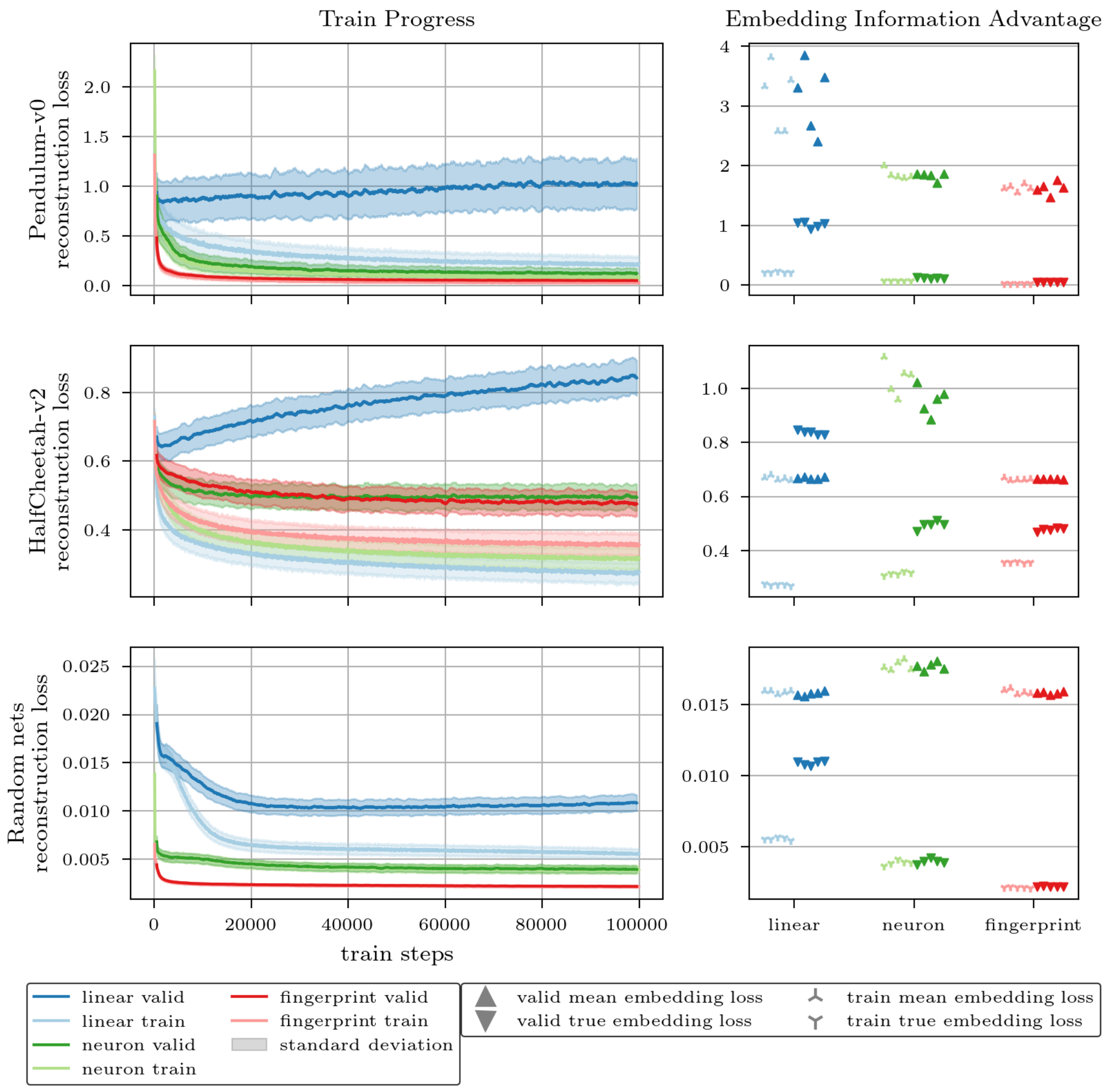


Figure 7: The left side shows the training progress of all methods across three different settings. The progress of validation error and train error are displayed separately. The filled area around the lines shows one standard deviation over all runs. Five runs have been executed per method. The right side shows the advantage of the reconstruction gained through the embedding after training. We compare how well the decoder can reconstruct the action given the true embedding of the policy compared to the reconstruction given only the mean embedding across all policies. All five runs are displayed individually here.

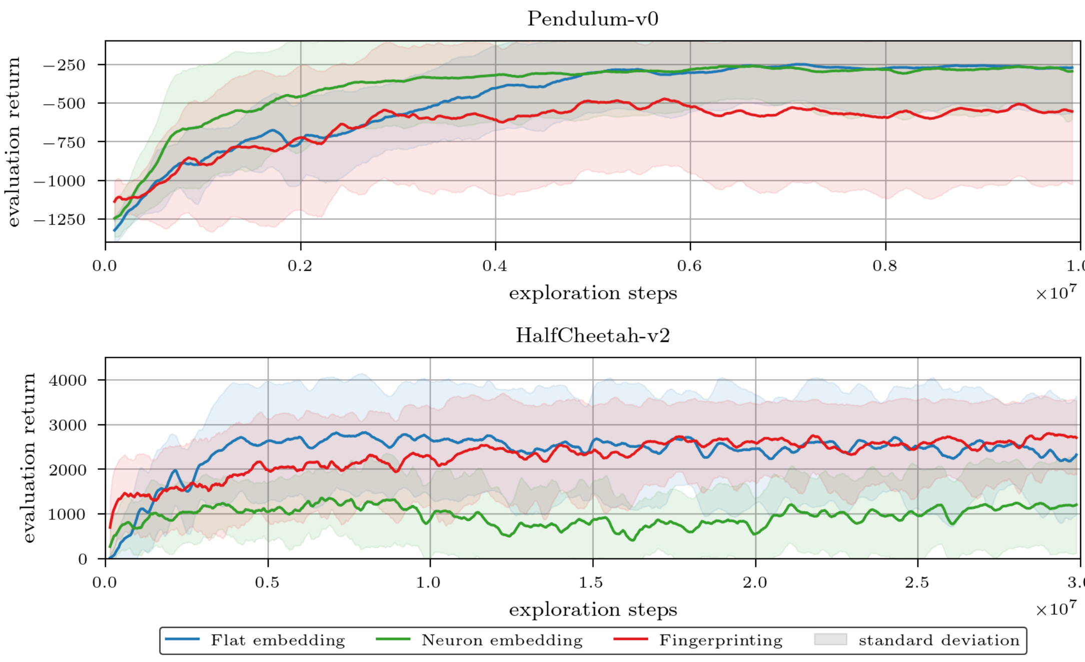


Figure 8: Mean evaluation returns of the State PCAC algorithm using different policy embeddings across 10 runs with standard deviation.

performing neuron embedding by a margin. This is also reflected by the embedding information advantage. Overall fingerprinting show consistently the best results in action reconstruction.

4.3 Reinforcement Learning performance of embeddings

Ultimately, we are interested in the Reinforcement Learning capabilities of the policy representations. Therefore, we train our State PCAC algorithm with all three embeddings and compare the results. The methods have individually been optimized by a small scale grid search for the pendulum environment and a distributional optimization for the HalfCheetah environment. Experiments for algorithmic approach in following chapters have been optimized more extensively. Consequently, The results here should not be seen as an indicator of the capability of the algorithm, but rather a comparison of different policy representations in a common setting. Used hyperparameters can be found in appendix A.2. We use the two aforementioned environments Pendulum-v0 and HalfCheetah-v2. The graphs in figure 8 show the mean and standard deviation of 10 optimized runs. All embeddings show learning behavior, although the overall performance is low. In Pendulum-v0 we can see that both neuron and flat embedding are consistently learning to solve the environment, although the solution is not always efficient (The agent takes longer than necessary to balance the pendulum upwards). Fingerprinting is volatile. When inspecting single runs, most of the neuron embedding and flat embedding runs solve the environment and few runs stagnate at a certain point and make no further learning progress. The behavior of fingerprinting runs is different. These runs are volatile to an extent that progress is often undone, but they are not as likely to stagnate around the same return value. Overall fewer runs are successful but the best fingerprinting run beats the performance of the best runs of the other two methods.

For the HalfCheetah-v2 environment the results are different. Whereas the neuron embedding is the fastest learner in the pendulum task, it performs poorly in the HalfCheetah environment. However, the neuron embedding has more hyperparameter options. It can change embedding sizes of specific neuron layers and include further non-linear layers in between. With more optimization it might be able to better suit this task, but being too depended on hyperparameters is an undesirable trait as it makes optimization more difficult. Fingerprinting and the flat embedding behave similarly. The flat embedding is learning slightly faster, but displays the highest volatility in this setting. The best single run is again achieved by Fingerprinting.

In summery, no single method holds a clear advantage over the others, in the Reinforcement Learning setting. Flat embedding appears to be most reliable across tasks. The performance of the neuron embedding is heavily depending on the task. Fingerprinting is less reliable but shows the highest potential for good performance.

4.4 Scalability of Embeddings

Complex Reinforcement Learning tasks tend to require larger policy networks. In order for Policy-Conditioned Value Functions to be a valid option in these tasks, policy embedding networks need to be scalable. Each of the three presented methods

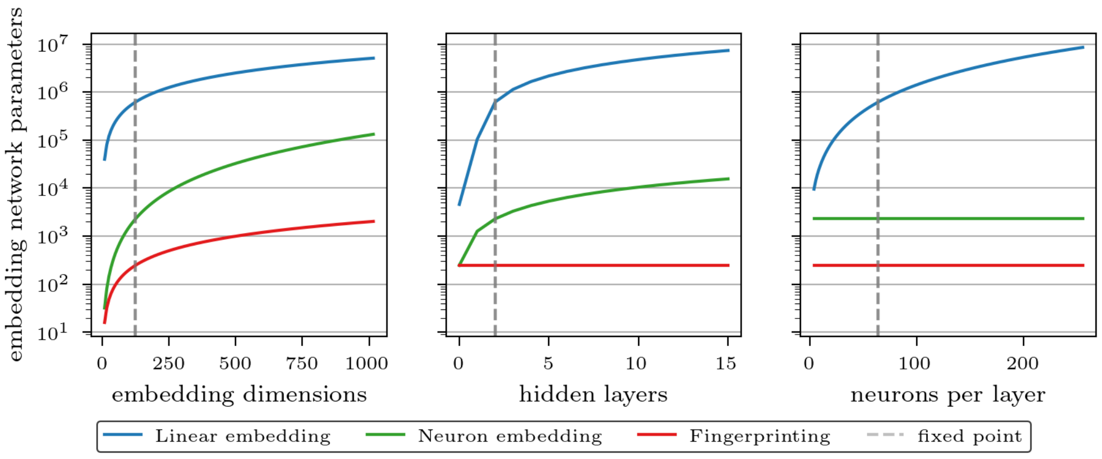


Figure 9: The three graphs show how the number of learnable parameters in the embedding networks scale with the policy network and the embedding dimensionality. Note the logarithmic scale on the y-axis. The vertical line indicates the particular value of the hyperparameter that has been used in the other two graphs that have a different hyperparameter on their x-axis.

scales with different properties. Figure 9 shows the with respect to three different hyperparameters of the policy network and the embedding itself. The y-axis shows the number of learnable parameters in the embedding network. For this comparison we use a fictional policy network with 8 input neurons and 4 output neurons. In the graphs where embedding dimensions, hidden layers, and neurons per layers are not on the x-axis, they are constant at 124 embedding dimensions, two hidden layers and 64 neurons per layer respectively. This is also indicated by the gray vertical line on those plots where the individual hyperparameter is on the x-axis. As a example: The gray vertical line on the left plot is at 124, meaning the other two graphs use an embedding dimensionality of 124.

The graph shows that all three methods are scaling with the embedding dimensionality. Although, all methods are scaling linearly, the scaling factor is different. For fingerprinting the scaling factor is the dimensionality of the input state (here 8). For the neuron embedding it is the dimensionality of the embedding vector (here 124). For the linear embedding, the scaling factor is the number of parameters in the policy network (here 4996).

The middle graph shows the shows how the embedding networks scale with the number of hidden layers of the policy network. Fingerprinting is not affected by this while the other two scale linearly. The scaling factor for the neuron embedding depends on the network size of the layer translation networks within the neuron embedding. For each layer in the policy network one of these translation networks is added. Another part of the scaling factor is the embedding dimensionality, as for each layer a learnable bias vector is added. See figure 6 as a reminder of these components. The scaling factor for the linear embedding is the number of added weights and biases per layers.

The right graph shows the scaling with respect to the number of neurons in each layer of the policy network. Each hidden layer has the same number of neurons. Both fingerprinting and neuron embedding are not affected by this. The linear embedding scales linearly with the scaling factor of $number\,of\,hidden\,layers \times embedding\,dimensions$.

Summing it up, the linear embedding scales with all shown aspects of the policy network with large scaling factors and will quickly become infeasible for larger policies. The neuron embedding does not scale with the number of parameters of the policy network, but with the number of layers. However, if choosing the embedding dimensionality accordingly, large policy networks can be embedded with this method. Fingerprinting displays the best scalability. It does not scale with any aspects of the policy network besides the number of input neurons. The number of learnable parameters is almost entirely dictated by the embedding dimensionality, which can be chosen independently of the policy network size. Therefore, fingerprinting is the best choice to embed large policy networks under the aspect of required learnable parameters.

4.5 Learned Input States States

Fingerprinting learns a set of input states during training (see figure 5 as a reminder). Here we examine these learned states with a specific visualization tailored to the Pendulum-v0 environment. Figure 10 shows a successful rollout in this visualization as an example of how it is to be understood. Successful refers to the

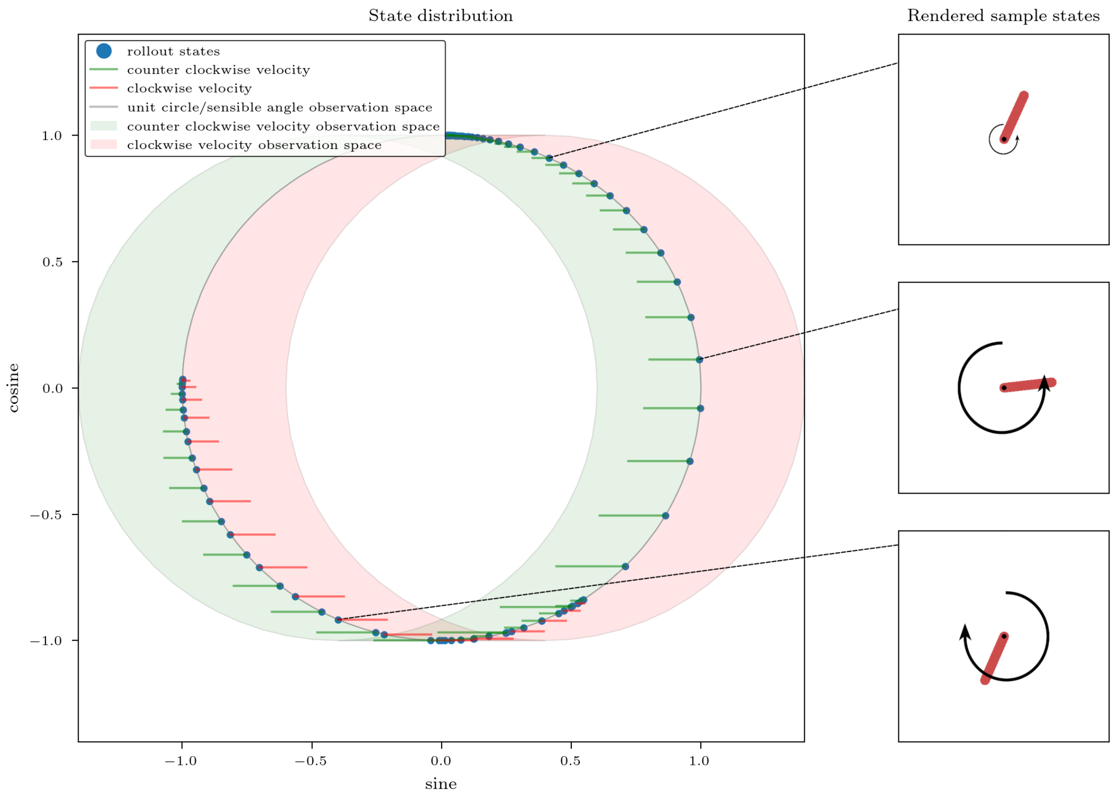


Figure 10: States of a successful rollout in the Pendulum-v0 environment. The scatter plot on the left shows the observed states of this rollout. The axes are the sine and cosine of the pendulums angle, thus the points lie on the marked unit circle. The red and green lines coming out of the points depict the rotational speed of the pendulum in that position. Red shows clockwise velocity and green counterclockwise velocity. The green and red areas around the unit circle show the possible range of velocities according to the observation space. On the right, some sample states are rendered. The size of the arrow around the pendulum indicates the velocity.

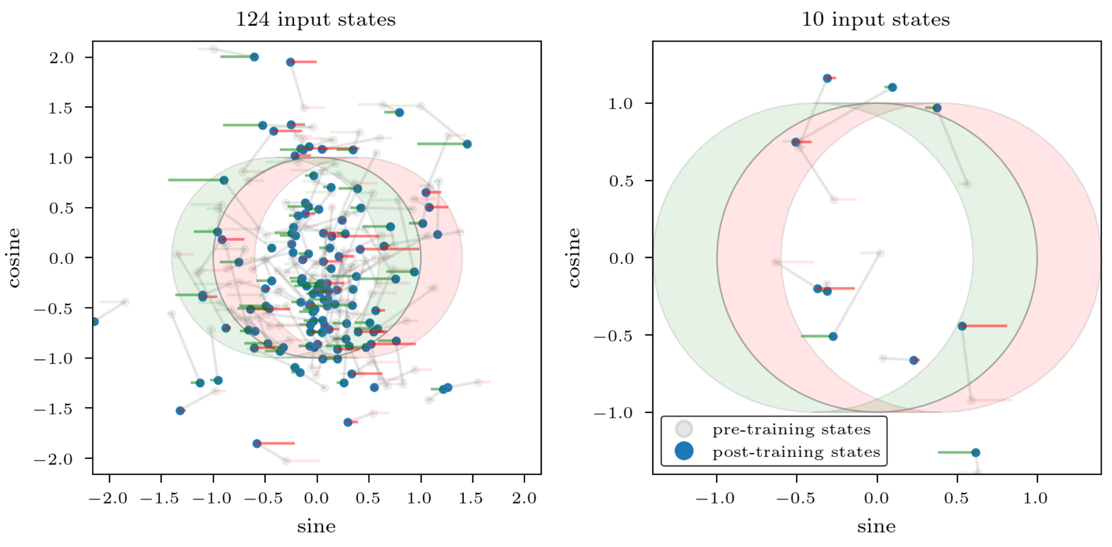


Figure 11: Learned input states of the fingerprint embedding. The gray transparent points show the initial positions of the states before training. The connected blue points are the same states after training. Two different training runs can be seen. One on the left with 128 learnable input states and one on the right with 10 learnable input states. Both runs successfully learned a policy which solves the environment reliably.

policy managing to swing the pendulum up and balancing it in place. The policy was trained by the State Policy-Conditioned Actor Critic algorithm using the fingerprint embedding. The rollout starts at the bottom and ends at the top. The unit circle together with the green and red shaded areas show the sensible observation space of the environment. These are the states that are possible to occur during rollout. However, programmatically, the observation space of Pendulum-v0 is not limited to sensible sine-cosine pairs. Therefore, this environment grants the opportunity to examine whether the learned input states of the fingerprint embedding are characteristic states of the environment. The input states of Fingerprint are not limited to the observation space of an environment. We expect the embedding to learn characteristic states of the environment to query a policy in order to evaluate its performance. As a consequence, the learned input states should be close to the sensible observation space. The following experiments show that this is not the case.

With the same visualization as in figure 10 we present the learned input states of fingerprinting in figure 11. As a reminder, the input states are the only learnable parameters of the fingerprint embedding. The resulting policy of the training run, in which these input states were learned, solves the Pendulum-v0 environment reliably. Since the policy and critic only receive normalized observations, we need to de-normalize them here to interpret them as environment states. Note that both pre- and post-training states were de-normalized using the observed mean and variance after training has terminated. This is the reason why the initial points do not appear to be normal distributed. Looking at the distribution of these states, it becomes apparent that these do not coincide with the sensible observation space of the environment, as marked by the unit circles and red and green patches. Barely any points lie reasonably close to the unit circle and some points are even outside of the value range of sine and cosine $[-1, 1]$. The movement throughout the training seems random and these states were never observed during the training process, as they cannot occur. To show that this does not happen because 128 input states are enough, such that the network can afford nonsensical input states, we repeated the same experiment with only 10 input states. This is visualized on the right side of figure 11. Here, the learned states seem nonsensical as well. However, even with only 10 learnable input states the algorithm learned a successful policy. Fingerprinting does not seem to learn characteristic states of the environment. Instead it learns a more abstract concept which still is effective in evaluating policies.

The neuron embedding is not intended to learn environment states, but the input neuron embeddings (see figure 6 as a reminder) can be interpreted as such. To get an idea of whether the neuron embedding works similar to the fingerprint embedding, we run the same experiment and extract the input neuron embeddings. We use a input neuron embedding dimensionality of 124. Transposing them gives us 124 vectors with the observation space dimensionality (3). Figure 12 shows the results of this experiment. Both training runs were successful. As the input neuron embeddings are considered as parameters similar to those of a linear layer, they are usually initialized by a Xavier uniform distribution. This is the default initialization for linear layers in PyTorch [15]. The left plot shows the results of this initialization. The input neuron embeddings are clustered around a small area. This is due

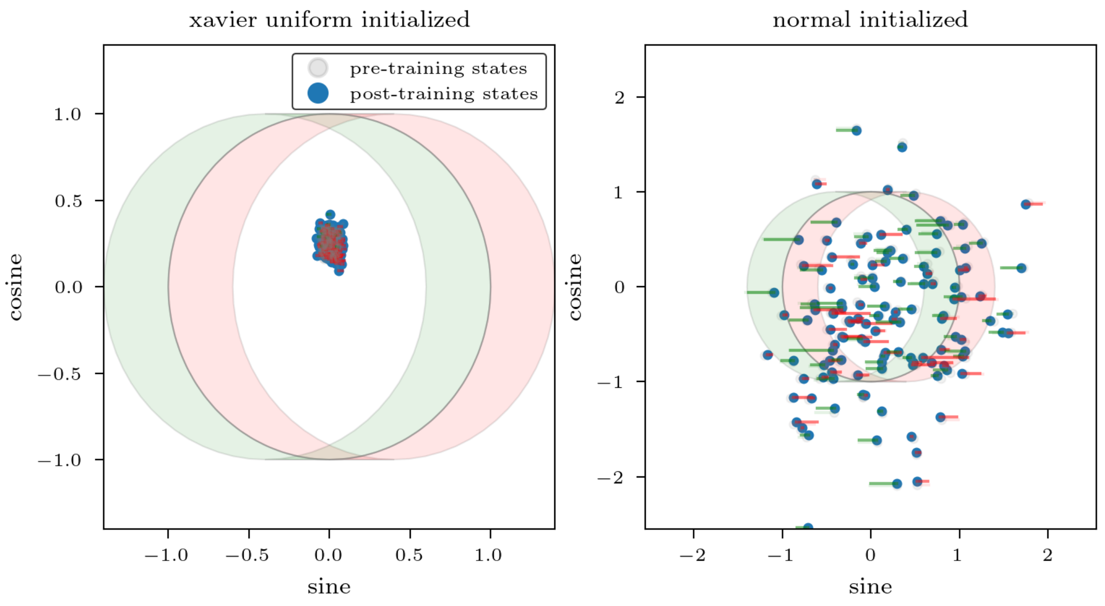


Figure 12: Learned input neuron embeddings of the neuron embedding method. The left image shows a run where the input neuron embeddings were initialized with a Xavier uniform distribution, the right side shows a training run where the input neuron embeddings were initialized with a standard normal distribution.

to the scale of the initialization itself. During the learning process the input neuron embeddings may be shifted, but opposed to fingerprinting, these are not the only parameters of the neuron embedding. Adjusting weights of translation networks might be more effective for learning progress. We rerun the experiment with a normal distribution initialization for these parameters to see whether the input neuron embeddings behave similarly to the input states of fingerprinting. We did not change the initialization of any other learnable parameters. The plot on the right of figure 12 shows the result. The distribution of learned input neuron embeddings interpreted as states of the environment does look similar to those of the fingerprint embedding method. However, over the training run these points did not change noticeably. Perhaps the input neuron embeddings suffer from a vanishing gradient due to more layers of this method. Another explanation would be that it is not effective to move these points because most of the work is done by the layer translation networks. Regardless, the learned policy solves the environment and the resulting distribution is similar to the one of Fingerprinting. This is evidence that the neuron embedding and the fingerprint embedding behave similarly and the intended purpose of the neuron embedding is already fulfilled through fingerprinting by simpler means. Due to these findings (combined with the other embedding analyses 4.2 and 4.3) all following experiments from here on will be conducted using the fingerprint embedding.

5 Algorithmic Variations

In the previous chapter we set our focus on policy representations, which is an essential component of Policy-Conditioned Value Functions. In this chapter we explore new algorithmic ideas which are enabled by PCVF. This is not meant to find a new refined Reinforcement Learning algorithm but rather as an indicator for promising future research directions. Along with the algorithmic variations we propose further improvement strategies, which we did not test extensively. These could be used for future experiments. All algorithms that are shown in the following use the fingerprinting policy representation. In the following all policy networks that are trained on the Pendulum-v0 environment have one hidden layer with 64 neurons. All policies trained on the HalfCheetah-v2 environment have two hidden layers with 128 neurons each, same as our benchmarks (see chapter 3.2). In chapter 5.1 we propose a method to explore multiple policies in a single rollout. Chapter 5.2 explains our algorithmic variation to reframes continuous reinforcement learning into a binary classification problem. Finally chapter 5.3 shows an extension that makes use of the generality of PCVF to train multiple actors at once.

5.1 N-step Policy-Conditioned Actor Critic

The N-step Policy-Conditioned Actor Critic variant is a modification that arose from the difficulty of effective exploration. The Start State PCAC variants creates only a single data point per rollout. The State PCAC variant creates n data points for n time steps in the environment (see algorithm 1). However, all data points from a single rollout contain the same policy. To achieve generalization across policies the critic should be trained on as many policies as possible. In the N-step PCAC variant we perform what is essentially the TD error. Within one rollout we explore multiple policies. To give an example: Say we run an environment for 200 time steps. We could replace the policy at different points in time. If we switch after 50 time steps we may explore 4 different policies while still having 50 time steps to approximate the Monte Carlo return on for each policy. To calculate the Monte Carlo return we use the discounted experienced future rewards. If the rollout did not terminate we approximate the remainder with the critic. In this example we would create 4 data points with 4 different policies per rollout. We do not calculate the value for each time step, but only for the first time step each policy is executed. This extended sequence of actions for a single state should give the critic more information about the true Monte Carlo return. Algorithm 2 shows the detailed pseudocode. Striking

Algorithm 2: N-step Policy-Conditioned Actor Critic

```
Input: Differentiable Critic $V_w(s, \phi(\pi)) : \mathcal{S} \times \Pi \to \mathcal{R}$ with parameters $w$ and
  policy embedding function $\phi$; deterministic actor $\pi_\theta : \mathcal{S} \to \mathcal{A}$ with
  parameters $\theta$, state space $\mathcal{S}$ and action space $\mathcal{A}$.;
Output: Learned $V_w(s, \phi(\pi)) \approx V_\pi(x) \forall s \in \mathcal{S}, \forall \pi \in \Pi$, learned
  $\pi_\theta(s) \approx \pi^*(s) \forall s \in \mathcal{S}$;
initialize critic and actor parameters $w, \theta$;
while Stop criteria not met do
    create exploration policy $\pi_{\theta_e} = f_e(\pi_\theta)$;
    Generate $n$ steps of an episode
      $s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, r_{t+2}, \ldots, s_{t+n-1}, a_{t+n-1}, r_{t+n}$ with policy $\pi_{\theta_e}$;
    calculate discounted partial return $\hat{mc}$ until time step $n$ for initial state
      $s_t$ with discount factor $\gamma \in [0, 1]$
    $\hat{mc} \leftarrow \sum_{k=0}^{n} \gamma^k r_k$
    $done \leftarrow 0$;
    if episode has terminated then
        reset environment;
        $t \leftarrow 0$;
        $done \leftarrow 1$;
    Store $(s_0, s_n, \pi_{\theta_e}, \hat{mc}, done)$ in replay buffer $D$;
    for some steps do
        Sample a batch $B = (s_0, s_n, \pi, \hat{mc}, done)$ from D;
        calculate target $y = \hat{mc} + (1 - done)\gamma V_w(s_n, \phi(\pi))$;
        Update critic by stochastic gradient descent
          $\nabla_w \mathbb{E}_{(s,\pi,y)\in B} [y - V_w(s_0, \phi(\pi_\theta))]^2$;
    end
    for some steps do
        Sample a batch $B = (s_0)$ from D;
        update actor by gradient ascent: $\nabla_\theta \mathbb{E}_{s\in B} [V_w(s_0, \phi(\pi_\theta))]$
    end
end
```

the right balance on how many policies are explored in one rollout is the main challenge. If chosen right the learning progress might be accelerated, as the critic

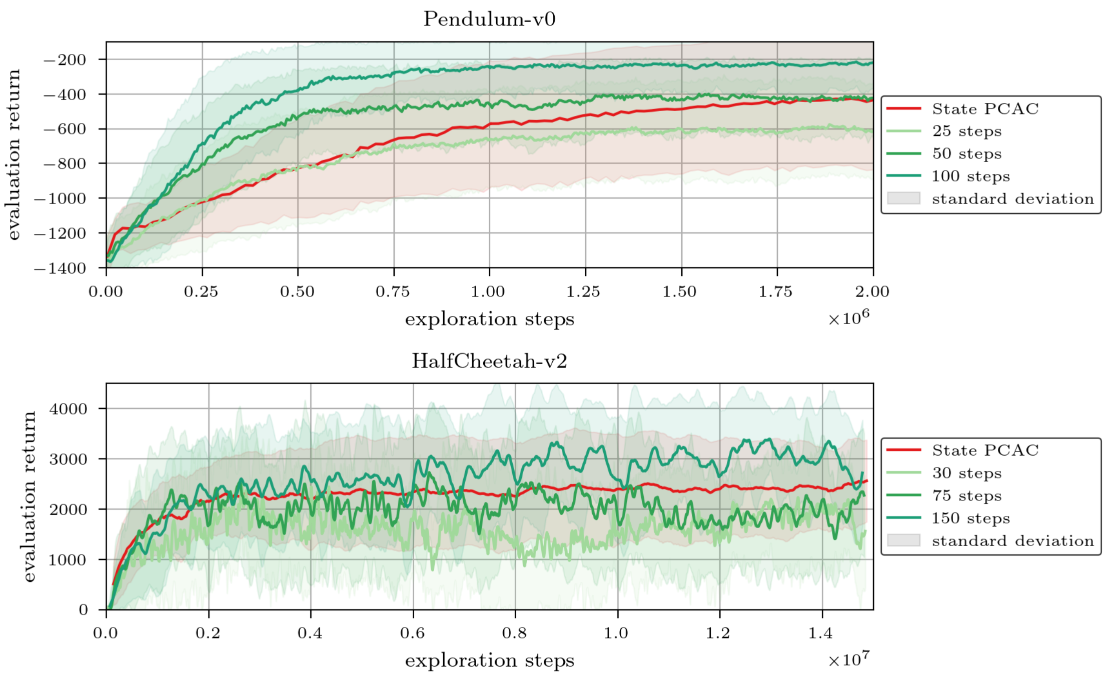


Figure 13: This plot shows the training progress of optimized N-step PCAC variant vs the State PCAC algorithm. The Pendulum-v0 graph shows the mean and standard deviation of 100 runs. HalfCheetah-v2 the mean and standard deviations of 10 runs. Episode length is 200 in Pendulum-v0. Episode length during training is 300 in HalfCheetah-v2 but 1000 during evaluation.

is shown multiple policies per rollout instead of one. The two extremes here are switching the policy in each time step and not switching the policy over the whole rollout. The first gives the critic very little information to learn about the embedding and how specific policies act in the future, the latter would be the same as the Start State PCAC algorithm, with the exception of including the start state explicitly. Exploring too many policies in one rollout will give the critic little information to converge, as it essentially has to predict the value of a set of parameters through possibly only one time step. This is usually not enough for more complex policies and environments.

Figure 13 shows the results of running N-step PCAC in different settings. PCAC is included as our current baseline, which we would like to improve. The Pendulum-v0 environment has a episode length of 200. This means, at $n = 25$ i.e. 25 steps per policy, 8 policies are explored in a single rollout. We limited the training rollout of HalfCheetah-v2 to 300 time steps across all algorithms in this thesis, specifically to increase the explored polices to time steps ratio. We also do this here for a fair comparison even though it would not be necessary for N-Step PCAC. Therefore, 30 steps per policy result in 10 explored policies per exploration rollout. Note that the evaluation rollouts are executed on the default length of 1000 time steps for HalfCheetah-v2 across all algorithms. Due to computational costs, the Pendulum-v0 runs have been executed 100 times while the HalfCheetah-v2 runs have only been executed 10 times. We can see that the training progress in pendulum-v0 is accelerated when setting the n parameter to 50 or 100 steps, which results in 2 or 4 policy explorations per rollout, respectively. In contrast, exploring too many policies grants no benefit or even reduces performance.

In Halfcheetah-v2 results are not as clear. The shown results have been smoothed across 10 evaluation points, at each evaluation the policy is executed 50 times. At $n = 150$ i.e. 2 policies per exploration rollout, we get the best results.However, those runs do not clearly separate their performance from the State PCAC baseline. Setting n too high seems to diminish the returns. N-Step PCAC in general leads to higher volatility. This is especially visible for the HalfCheetah environment as fewer runs have been performed there, despite the smoothing. We suspect this instability is caused by the use of the TD error as instability commonly seen among TD error approaches.

5.2 Reinforcement Learning in Continuous Space as a Binary Classification Problem

Policy-Conditioned Value Functions enable entirely new approaches such as reformulating Reinforcement Learning into a binary classification problem. In PCAC the critic takes in an embedding of a policy and outputs the state-value of this policy. In Comparing PCAC on the other hand, the critic is given two embeddings instead of one. The task is to determine which one of those policies returns a higher value. Unlike regression, here the critic does not have to predict a value. We suspect that this simplifies the problem as the critic no longer needs to determine an exact value for each policy but only needs to be able to rank them. This method opens up a lot of new possibilities and can be implemented in many ways. We choose a bare-bones approach to show the potential of this method. We also propose further improvements of this method in the following. The pseudocode of our Implementation can be seen in algorithm 3. In the following it is also described by words. The code for Comparing Start State PCAC can be found in the appendix (algorithm 6).

The critic receives two policy embeddings (for the start state variant) or two embeddings and two states (for the state variant). Intuitively, it would be desirable to compare two policies on the same state. However, we cannot assume that different policies have any intersections in their experienced rollouts. Therefore, we are lacking the data to compare two policies on the same states. The classification target is 1 if policy one π_1 has received a larger return in that rollout on that state than policy two on its according state and 0 otherwise. If the gained return is exactly the same we set the target at 0.5 to minimize wrong gradient signals. A binary cross entropy loss is then used on the target and the predicted classification of the critic to calculate the loss.

There are many possibilities how to update the actor with this method. We opted for the following implementation: A batch of past actors and states is pulled from the replay buffer. They are compared to the current actor, on the same states, through the critic. On the resulting classifications, gradient ascend is applied to increase the likelihood of the current policy to be classified as performing better, than those of the replay buffer. Another method would be to compare the actor against itself. Small scale experiments indicate that this provides no benefit while reducing stability.

Figure 14 shows the results of two Comparing PCAC variants against the base PCAC algorithm with the fingerprint embedding. The Comparing State PCAC algorithm includes a state for each policy embedding at the comparison and the target is determined based on the return from that state on (as previously described). The Comparing Start State PCAC does not include explicit states but only the policies. The targets are based on the total return of a rollout of that policy. Again,

Algorithm 3: Comparing State Policy-Conditioned Actor Critic

```
Input: Differentiable Critic $V_w(s_1, \phi(\pi_1), s_2, \phi(\pi_2)) : \mathcal{S} \times \Pi \times \mathcal{S} \times \Pi \rightarrow [0, 1]$
  with parameters $w$ and policy embedding function $\phi$; deterministic actor
  $\pi_\theta : \mathcal{S} \rightarrow \mathcal{A}$ with parameters $\theta$, state space $\mathcal{S}$ and action space $\mathcal{A}$.;
Output:
learned $V_w(s_1, \phi(\pi_1), s_2, \phi(\pi_2)) \approx \begin{cases} 1, \text{if } V^{\pi_1}(s_1) > V^{\pi_2}(s_2) \\ 0, \text{if } V^{\pi_1}(s_1) < V^{\pi_2}(s_2) \\ 0.5, \text{otherwise} \end{cases}$
$\forall s_1, s_2 \in \mathcal{S}, \forall \pi_1, \pi_2 \in \Pi$,
learned $\pi_\theta(s) \approx \pi^*(s) \forall s \in \mathcal{S}$;
initialize critic and actor parameters $w, \theta$;
while Stop criteria not met do
    create exploration policy $\pi_{\theta_e} = f_e(\pi_\theta)$;
    Generate an episode $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$ with policy $\pi_{\theta_e}$;
    $t \leftarrow 0$;
    while $t < T$ do
        $mc_t \leftarrow \sum_{k=t}^{T} \gamma^{k-t} r_t$;            // discounted return with $\gamma \in [0, 1]$
        Store $(s_t, mc_t, \pi_{\theta_e})$ in replay buffer $D$;
        $t \leftarrow t + 1$;
    end
    for some steps do
        Sample a batch $B = (s, \pi, mc)$ from D;
        splitting Batch $B$ in half $B_1 = (s_1, \pi_1, mc_1)$, $B_2 = (s_2, \pi_2, mc_2)$;
        calculate targets $y = \begin{cases} 1, \text{if } mc_1 > mc_2 \\ 0, \text{if } mc_1 < mc_2 \\ 0.5, \text{otherwise} \end{cases}$;
        calculate classifications $\hat{y} = V_w(s_1, \phi(\pi_1), s_2, \phi(\pi_2))$;
        Update critic by stochastic gradient descent following binary cross
          entropy loss $\nabla_w \mathbb{E}_{(s,\pi,mc)\in B} [y \log \hat{y} + (1 - y) \log(1 - \hat{y})]$
    end
    for some steps do
        Sample a batch $B = (s, \pi_e)$ from D;
        update actor by gradient ascent $\nabla_\theta \mathbb{E}_{s,\pi_e \in B} [V_w(s, \phi(\pi_\theta), s, \phi(\pi_e))]$
    end
end
```

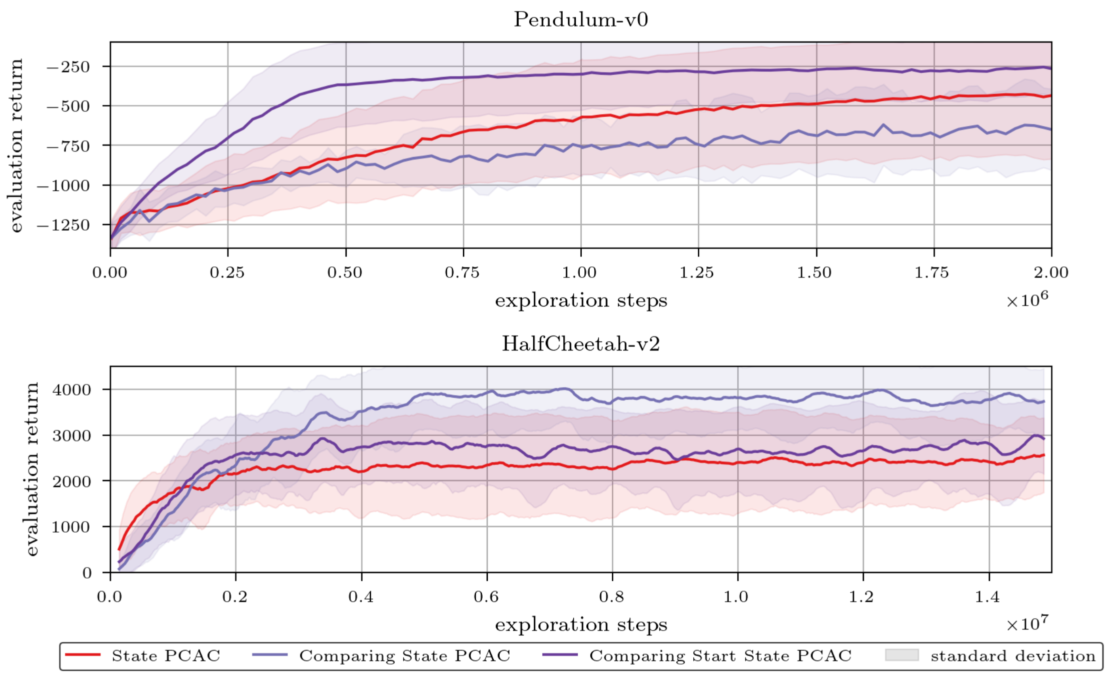


Figure 14: Optimized Comparing State PCAC and Comparing Start State PCAC matched against the base PCAC algorithm using fingerprinting.

we observe that different methods behave differently relative to another in these two environments. In the pendulum environment, the start state variant outperforms the other two methods and Comparing State PCAC does not beat the State PCAC algorithm. However, in the HalfCheetah environment the state variant performs the best while the other two methods perform similarly. This is probably due to the fact that the states of the HalfCheetah environment are more complex than those of the Pendulum environment. Therefore, information about these states have more impact.

Our implementation of Comparing PCAC is basic. Potential improvements for future implementations could be:

- Symmetric use of data for critic update: To enforce Symmetry in the classification data, batch could be mirrored and used again.

- Stronger requirements to be identified as performing better in the targets: Whether a policy is better than another is currently determined by the exact value of a single rollout. Target quality might improve if a margin is required to qualify as better. So instead of

$$V^*(s_1, \pi_1, s_2, \pi_2) = \begin{cases} 1, \text{if } V^{\pi_1}(s_1) > V^{\pi_2}(s_2) \\ 0, \text{if } V^{\pi_1}(s_1) < V^{\pi_2}(s_2) \\ 0.5, \text{otherwise} \end{cases}$$

the optimal value function would be

$$V^*(s_1, \pi_1, s_2, \pi_2) = \begin{cases} 1, \text{if } V^{\pi_1}(s_1) > V^{\pi_2}(s_2) + m \\ 0, \text{otherwise} \end{cases}, \tag{14}$$

where m is the margin. If the policies are roughly the same the target would be 0 not to symbolize that policy two is better but that policy one is *not* better than policy two.

- Using a classification loss for the actor updates: Early experiments show that it might be beneficial to use a cross entropy loss to update the actor instead of maximizing the classification output directly.

5.3 Multi Actor Learning

In common Reinforcement Learning algorithms the critic implicitly learns about the current policy and therefore is only capable of evaluation and improving this specific policy. With Policy-Conditioned Value Functions the critic is explicitly informed about which policy it is evaluating. This way it is possible for the critic to be trained with multiple policies/actors at the same time. Thus, the critic might be better equipped to map the policy landscape when following several improvement paths instead of just one.

Multi actor learning is compatible with any Policy-Conditioned Value Function method as it can be used on top of those. We are interested in whether any methods benefit more from multi actor learning than others. Generalization across the policy space would mean that the information of one successful actor should help to improve all other actors. To see whether this is what happens during multi actor training, we run multiple experiments. First, we compare the results of the learned policies. In the multi actor variant we are taking the best actor of all actors, that were used during training, for evaluation. To assess overall performance we compare the Multi Actor PCAC version to the other two algorithm variations with multi actor on top. Additionally, to assess whether the actors benefit from the experiences of other actors we compare the evaluations across all used actors (instead of just the best) and compare them to the single actor setting with the same amount of actors (multiple runs).

The algorithm modifies the following points when being added on top of another algorithms training process:

- Multiple actors are initialized at the start of training
- At each exploration step a random actor is chosen, based on which the exploration policy is created
- All actors are updated during the actor update step

It is important to note that the same amount of environment exploration interactions are taking place as in the other algorithm, to ensure a fair comparison. The Multi Actor approach does not fundamentally change anything about the algorithm itself, thus we focus our analysis on the affect of the amount of used actors as well as their interactions. Due to computational costs we limit this analyses to the Pendulum-v0 environment, which is faster to execute than the HalfCheetah-v2.

Figure 15 shows the results of combining the multi actors approach with all previous methods. Here the "1 Actor" run is simply the previously introduced methods (PCAC, N-step PCAC, Comparing Start State PCAC) without any modification. All methods seem to benefit from using multiple actors, although some more than others. For all methods the performance rises while the volatility decreases with the number of actors. Again for this evaluation the best actors is always been used for evaluation, when there are multiple to chose from. While no more exploration

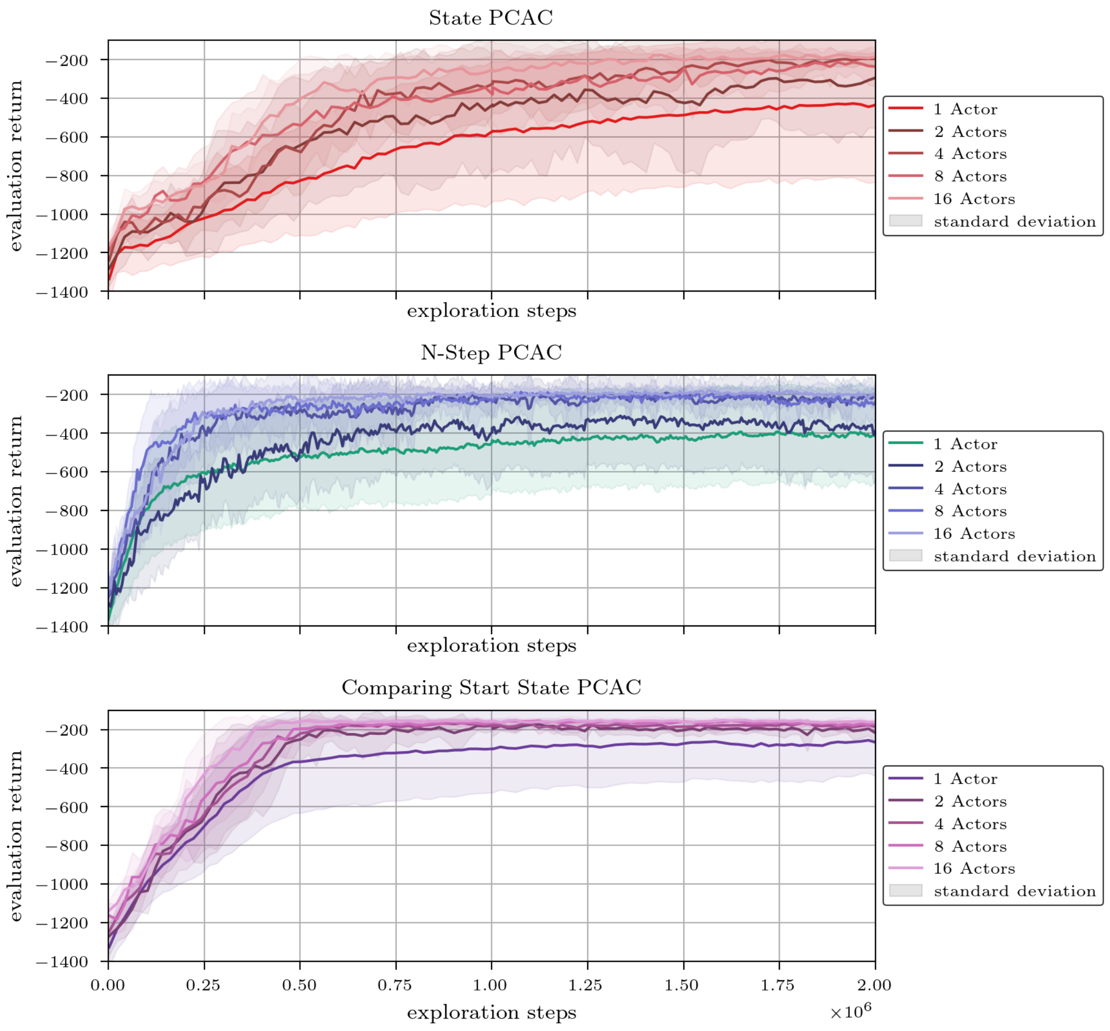


Figure 15: All algorithm variations with different amount of actors trained on the Pendulum-v0 environment. The evaluation is always performed on the best actor within a run.

steps are needed, when using multiple actors compared to single actor, the computational costs still rise linearly with the number of actors, as all actors are updated at each update step. We also tested a variant where only the actor that has last been used for exploration will be updated. This has the same computational costs as the regular methods but performs significantly worse.

Especially interesting is the multi actor addition on Comparing PCAC. Usually the comparing critic only compares policies that stem from one improvement path and its noisy neighbors (depending on exploration settings), but here the critic has to compare policies from multiple improvement paths, which might be entirely different. The resulting performances for all runs of comparing PCAC, are closer to one another than those of the other methods (Figure 15). This indicates that the method already benefits greatly from only two actors. Further actors do not seem to add a lot of benefit after that. However, as we are only looking at the best actors we cannot assess whether the actors actually benefit from one another. Hence, we run another experiment, which results are shown in figure 16.

Here, we do not chose the best actor per run for evaluation, but all actors are evaluated. The number of used actors is always the same (100), but the amount of runs differs. Thus, for the single actors approach 100 runs have been performed. For the 10 actors approach 10 runs have been performed. If the actors benefit from one another, we would expect to see the average performance across runs with more actors to be significantly better than those with just a single actor. However looking at the runs we can clearly see that this is not the case (figure 16).

We suspect that the policy space is too high dimensional for the critic to generalize across. When a good policy is found the critic does know a path for the other policies towards that good policy. This is not necessarily bad. If the critic found this path, all actors might converge their policies quickly to the currently best policy. This would escape the point of the multi actors approach of having a better exploration through different policies. Earlier approaches like "What About Inputing Policy in Value Function" [21] focus on the local neighbors of a single improvement path. Our experiment is evidence for this approach to be more feasible

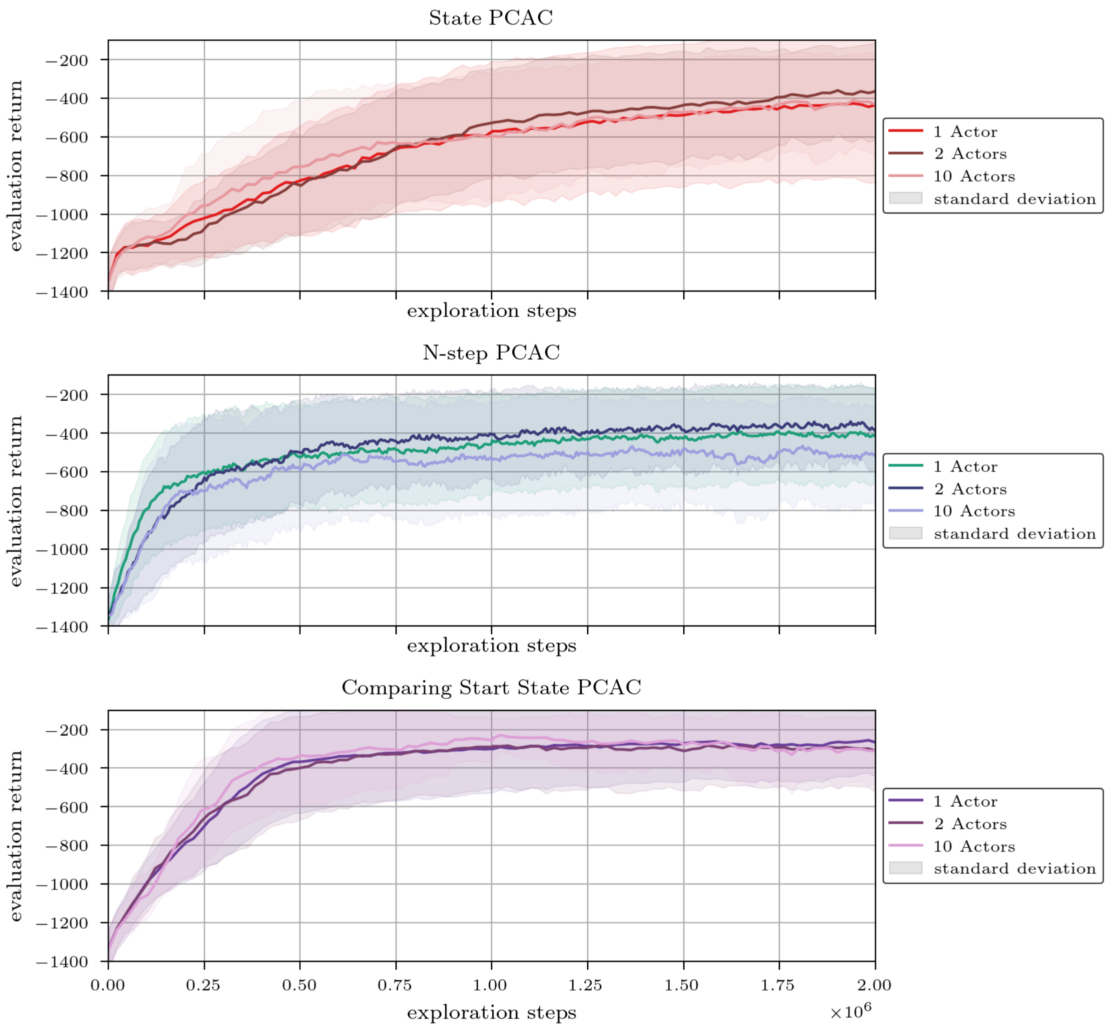


Figure 16: All algorithm variations with different amount of actors trained on the Pendulum-v0 environment. The considered evaluation is the mean across all actors within a run.

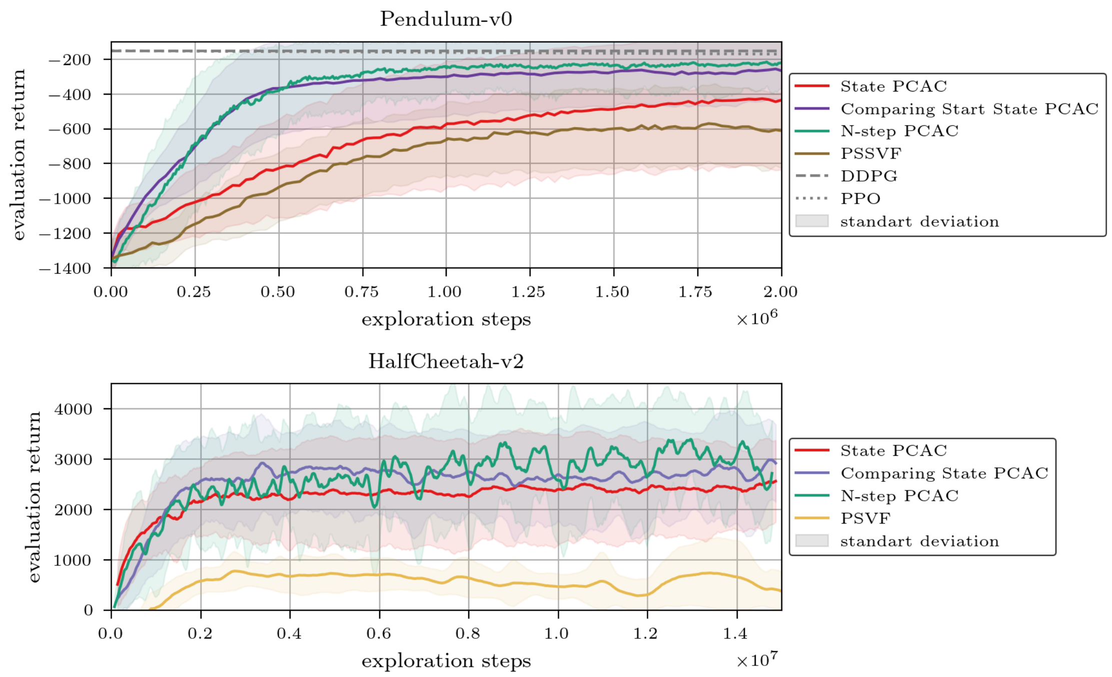


Figure 17: The best algorithm variations by environment, compared to several benchmarks. Multi actor approaches are excluded.

than trying to generalize across all policies. However this multi actor approach is successfully avoiding local optima. As there are more policies, it is more likely that at least one of them finds a good solution. This can be seen by the high returns and low standard deviations of the 16 Actors runs in Figure 15. Beyond that multi actor training does not seem to benefit singular actors.

5.4 Best Results Comparison

Figure 17 shows the best results compared to our base algorithm PCAC and several benchmarks from chapter 3.2. As previously established, Multi Actor PCAC variants do improve performance and reliability. However, they do not represent a new algorithmic idea, but rather a simple mechanic to avoid local optima, therefore we do not include them here.

On Pendulum-v0 our base algorithm State PCAC has a small advantage over PSSVF in the mean, but also a larger variation. PSSVF is the best ”Parameter-Based Value Functions” [6] approach for this environment, following our optimization. Our two additional algorithm variations, Comparing Start State PCAC and N-step PCAC, show a clear improvement over State PCAC and PSSVF. The DDPG and PPO baselines are still the best here. Due to the nature of the Pendulum-v0 environment, the best possible score lies around -150. Both DDPG and PPO are close with a mean score of -151 and -170 respectively.

In the HalfCheetah-v2 environment all our approaches substantially outperform the PSVF baseline, however algorithm variations only grant small improvements if at all. The DDPG and PPO baselines are not visible in the plot, as they would change the scale of the y-axis drastically. DDPG lies at a mean score of 10269 and PPO at 5616. The HalfCheetah-v2 environment holds more potential for improvement than the Pendulum environment. These results show that PCVF do not yet compete with state-of-the-art Reinforcement Learning algorithms but our approaches and modifications were able to improve the performance compared to similar approaches.

6 Discussion

In this thesis, we examined different policy representations and algorithmic variations in the wider framework of Policy-Conditioned Value Functions. In the following, we share our insights and thoughts which we have developed throughout the creation of this thesis.

Policy representations The policy representations are a fundamental building block for all PCVF algorithms and therefore should be chosen with care. Besides the three embeddings we explored (flat embedding, fingerprinting, neuron embedding), there are endless other possibilities how to represent a policy. We found the fingerprint embedding to be the best choice compared to its contestants. It excels in action reconstruction and also shows the best performance in the more challenging environment, of our chosen two. It is clearly the best option of the three, to embed large policies with the fewest learnable parameters. Considering that PCAC was still able to solve the Pendulum-v0 environment using only 10 learnable input states (see chapter 4.5) shows that small policy embeddings are capable of holding sufficient information. Our look into the inner workings of the embedding shows that fingerprinting does not learn characteristic states of an environment. As often in Deep Learning, the mechanism behind this embedding seems too abstract to be simply conceptualized. The neuron embedding approach is also successful, but in contrast to fingerprinting, the input neuron embeddings do not change significantly during the training process. Perhaps the learning aspect of fingerprinting is not necessary and the input states only require to be sufficiently distributed across some space, which does not need to coincide with the observation space. This could be the basis for another, even simpler, representation approach.

Technical challanges PCVF hold some technical challenges over established Reinforcement Learning approaches. Similar to other off-policy methods, a replay buffer is used to store data from the exploration rollouts. Additionally to any data like rewards, states and actions, the policy itself needs to be stored. It does not suffice to store the embedded policy if the embedding shifts during training. Fingerprinting and neuron embedding both have learnable components, therefore they might result in different embeddings at different stages throughout training. The necessity to store policies creates a much larger memory cost than other Reinforcement Learning approaches. This is even worse when not implemented correctly, such that duplicates of the same policy are stored multiple times. Another technical challenge is the implementation of embeddings, which mimic parts or the whole of policy forward passes. This is the case for fingerprinting and the neuron embedding. During training these embeddings need to be able to efficiently embed a batch of policies. In the case of fingerprinting this means that a shared batch of inputs is forwarded through a batch of policy networks. PyTorch [15], the Deep Learning framework we used, does not natively support this operation in a parallelized manner. Frameworks are built with batch forward passes through single networks in mind. For now, custom solutions need to be build to enable this in a parallelized way. We implemented batched matrix multiplication, which imitates the forward passes through a batch of networks. Our current implementation works for sequential fully connected networks. For more advanced network architectures, different solutions would need to be build.

Towards Supervised Learning In PCVF the critic no longer needs to chase the actor during training, as described in chapter 2.6. However, other effects may arise which make training more difficult. The training of PCVF methods is conceptually closer to Supervised Learning. This is especially the case when using Monte Carlo returns, deterministic policies and a deterministic environment. Here, We neglect other stochastic properties of the environment, such as the initial state and state values that are not informed about a time dependent termination. This is, in principle, a desirable shift towards Supervised Learning. In general, Supervised Learning is well optimized and considered to be simpler than Reinforcement Learning. However, the mapping that is the training goal, is extremely complex. In the case of Start State PCAC, which is the simplest setting, the resulting value function should map the space of policies to the space of return values $V(\pi) : \Pi \rightarrow \mathbb{R}$. Considering the complexity of the policy space, this seems to be a nearly infeasible problem. Nevertheless, the proposed PCVF algorithms are effective in learning a successful policy. The value function does not actually generalize across the space of policies but across a local neighborhood of one or multiple policy improvement paths. This is supported by our multi actor experiment, which does not show a higher success rate across all actors, when the value function has access to multiple actors (see chapter 5.3). Also, when using policy representations with learnable components, the learning problem does in fact shift over time, as the policy representations change with the embedding network. Therefore, the problem is not entirely static. Since new data is collected depending on the current actor policy, there is a distributional shift in the data as the actor policy changes over the course of training.

Sample efficiency Sample efficiency was one of the original motivations for combining the advantages of off-policy and on-policy algorithms. We did not highlight this so far, however it became apparent that PCVF require a lot more data to learn effectively than state-of-the-art on- and off-policy algorithms. One of the reasons why PCVF are not sample efficient, is that not only data across the state and actions spaces are required, but additionally also data across the policy space. Samples of states and actions are only meaningful in combination with a policy. This higher dimensionality makes each sample less informative compared to the size of the relevant space. Additionally, only few different policies may be explored in a given budget of environment interactions. With our N-step PCAC approach we counteract this effect by exploring multiple policies within a single rollout. We showed that this does in fact speed up learning progress measured by environment interactions, at the cost of reduced stability. Conceptually, our N-step approach is a step back again towards methods with TD error, as we are unable to use Monte Carlo returns. However, the possibility to use Monte Carlo returns is, in our opinion, a major advantage of PCVF over classical off-policy approaches, as it avoids circular dependencies in the training objectives.

Algorithmic Ideas In addition to exploring policy representations, we have also proposed new algorithmic ideas. As previously mentioned, N-step PCAC was created as to improve sample efficiency. Multi actor approaches are an effective method to avoid local optima by searching along multiple improvement paths simultaneously. Beyond that, Multi Actor PCAC did not show an increased generalization effect. The Comparing PCAC algorithm was born from the idea that the classification, or ranking of policies, might be a simpler task than the regression of exact state values. Comparing PCAC is a construct that is only enabled by the PCVF framework. This is not possible in established Reinforcement Learning structures. We showed its effectiveness in learning successful policies and its increased performance compared to our base PCAC approach. In chapter 5.2, we proposed three further improvements. Although our implementation of this algorithm is straightforward, we see great potential in this approach. There are many more ideas for further improvements or different concepts. An especially interesting setting might be to use Comparing PCAC in a competitive multi agent environment to determine which one of two policies would win.

Comparing optimized runs which have been run often enough to produce significant results is immensely computationally expensive. Therefore, we limited ourselves to the shown results. Many other ideas were tested in smallscale experiments that showed little potential and were therefore not pursued further. PCVF are still unexplored to a large extend. We think there are still many interesting ideas worth exploring.

7 Conclusion

Policy-Conditioned Value Functions are effective in Reinforcement Learning. The field is still largely unexplored and the methods so far are only able to compete with more established algorithms in specific settings. While we did not find a particular algorithm that leverages PCVF into competitiveness to state-of-the-art Reinforcement Learning algorithms, we improved current approaches by several ideas and identified promising research directions.

Our proposed policy representation, the neuron embedding, proved to be effective, however did not show significant benefits over the simpler and more scalable fingerprinting approach. We confirmed the theorized scalability of fingerprinting by using reasonably large policies (two hidden layers of 128 neurons each) in a difficult Reinforcement Learning environment (HalfCheetah-v2). So far, in the original paper [11], fingerprinting had only been tested on smallscale policies. We also showed that learned input states do not reflect observations from the environment.

State PCAC improves performance over "Parameter-Based Value Functions" [6] by using scalable policy representations and the Monte Carlo return instead of the TD error. Multi actor PCAC enables all PCVF algorithms to explore an environment simultaneously with multiple, entirely different, actor policies. This extension shows to be effective in avoiding local optima without requiring more interactions with the environment. N-step PCAC improves sample efficiency and accelerates learning progress at the cost of stability. This trade-off depends on the n parameter. Comparing PCAC transfers Reinforcement Learning in continuous spaces to a classification problem. This approach proves capable of achieving higher scores than our base State PCAC algorithm. Additional research is needed to further refine the presented algorithms to conclude whether Policy-Conditioned Value Functions are a valid alternative to current state-of-the-art Reinforcement Learning approaches.

8 Bibliography

[1] Richard Bellman. A Markovian Decesion Process. 6(5):679–684, 1957.

[2] Dimitri P. Bertsekas. *Dynamic programming and optimal control*, volume 1 of *Athena scientific optimization and computation series*. Athena Scientific, Belmont, Massachusetts, fourth edition edition, 2017. ISBN 9781886529434.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. URL `https://arxiv.org/pdf/1606.01540`.

[4] Erin Catto. Box2D, 2021. URL `https://github.com/erincatto/box2d`.

[5] Will Dabney, André Barreto, Mark Rowland, Robert Dadashi, John Quan, Marc G. Bellemare, and David Silver. The Value-Improvement Path: Towards Better Representations for Reinforcement Learning. URL `http://arxiv.org/pdf/2006.02243v2`.

[6] Francesco Faccio, Louis Kirsch, and Jürgen Schmidhuber. Parameter-Based Value Functions. 2021. URL `https://openreview.net/pdf?id=tV6oBfuyLTQ`.

[7] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing Function Approximation Error in Actor-Critic Methods. URL `http://arxiv.org/pdf/1802.09477v3`.

[8] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous Deep Q-Learning with Model-based Acceleration. URL `http://arxiv.org/pdf/1603.00748v1`.

[9] Nicholas Guttenberg, Nathaniel Virgo, Olaf Witkowski, Hidetoshi Aoki, and Ryota Kanai. Permutation-equivariant neural networks applied to dynamics prediction. URL `http://arxiv.org/pdf/1612.04530v1`.

[10] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001. ISSN

1063-6560. doi: 10.1162/106365601750190398. URL http://www.cmap.polytechnique.fr/~nikolaus.hansen/cmaartic.pdf.

[11] Jean Harb, Tom Schaul, Doina Precup, and Pierre-Luc Bacon. Policy Evaluation Networks. URL http://arxiv.org/pdf/2002.11833v1.

[12] 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. URL http://arxiv.org/pdf/1509.02971v6.

[13] Francisco S. Melo. Convergence of Q-Learning: A Simple Proof. URL http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. URL http://arxiv.org/pdf/1312.5602v1.

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[16] Antonin Raffin. RL Baselines3 Zoo, 2020. URL https://github.com/DLR-RM/rl-baselines3-zoo.

[17] Ingo Rechenberg. Evolutionsstrategie — Optimierung technischer Systeme nach Prinzipien der biologischen Evolution. 170 S. mit 36 Abb. Frommann–Holzboog–Verlag. Stuttgart 1973. Broschiert. *Feddes Repertorium*, 86(5):337, 1975. ISSN 0014-8962. doi: 10.1002/fedr.19750860506.

[18] Christoph Roch, Alexander Impertro, Thomy Phan, Thomas Gabor, Sebastian Feld, and Claudia Linnhoff-Popien. Cross Entropy Hyperparameter Optimization for Constrained Problem Hamiltonians Applied to QAOA. URL http://arxiv.org/pdf/2003.05292v2.

[19] G. A. Rummery and M. Niranjan. On-line Q-Learning Using Connectionist Systems. 1994. URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.2539&rep=rep1&type=pdf.

[20] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. URL http://arxiv.org/pdf/1707.06347v2.

[21] Hongyao Tang, Zhaopeng Meng, Jianye Hao, Chen Chen, Daniel Graves, Dong Li, Changmin Yu, Hangyu Mao, Wulong Liu, Yaodong Yang, Wenyuan Tao, and Li Wang. What About Inputing Policy in Value Function: Policy Representation and Policy-extended Value Function Approximator. URL http://arxiv.org/pdf/2010.09536v4.

[22] Saran Tunyasuvunakoll, Yuval Tasse, and Tom Erez. MuJoCo. URL https://github.com/deepmind/mujoco.

[23] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992. ISSN 0885-6125. doi: 10.1007/BF00992698.

[24] Daan Wierstra, Tom Schaul, Tobias Glasmachers, Yi Sun, Jan Peters, and Jürgen Schmidhuber. Natural Evolution Strategies. 15:949–980, 2014. URL http://jmlr.org/papers/v15/wierstra14a.html.

A Appendix

A.1 Algorithms

In the following, we show the pseudocode of other algorithms that were mentioned in the thesis, but their code was not included in the main part.

Algorithm 4: Start State Policy-Conditioned Actor Critic

```
Input: Differentiable Critic $V_w(\phi(\pi)) : \Pi \rightarrow \mathcal{R}$ with parameters $w$ and
  policy embedding function $\phi$; deterministic actor $\pi_\theta : \mathcal{S} \rightarrow \mathcal{A}$ with
  parameters $\theta$, state space $\mathcal{S}$ and action space $\mathcal{A}$.;
Output: Learned $V_w(\phi(\pi)) \approx \mathbb{E}_{s_0 \sim \mu_0}[V^\pi(s_0)] \forall \pi \in \Pi$,
learned $\pi_\theta(s) \approx \pi^*(s) \forall s \in \mathcal{S}$;
initialize critic and actor parameters $w, \theta$;
while *Stop criteria not met* do
    create exploration policy $\pi_{\theta_e} = f_e(\pi_\theta)$;
    Generate an episode $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$ with policy $\pi_{\theta_e}$;
    calculate MC with discount factor $\gamma \in [0, 1]$
    $mc \leftarrow \sum_{k=0}^{T} \gamma^k r_k$;
    Store $(mc, \pi_{\theta_e})$ in replay buffer $D$;
    for *some steps* do
        Sample a batch $B = (mc, \pi)$ from D;
        Update critic by stochastic gradient descent
          $\nabla_w \mathbb{E}_{(mc,\pi) \in B}[mc - V_w(\phi(\pi))]^2$
    end
    for *some steps* do
        update actor by gradient ascent: $\nabla_\theta V_w(\phi(\pi_\theta))$
    end
end
```

Algorithm 5: State Action Policy-Conditioned Actor Critic

```
Input: Differentiable Critic $V_w(s, a, \phi(\pi)) : \mathcal{S} \times \mathcal{A} \times \Pi \rightarrow \mathcal{R}$ with parameters
  $w$ and policy embedding function $\phi$; deterministic actor $\pi_\theta : \mathcal{S} \rightarrow \mathcal{A}$ with
  parameters $\theta$, state space $\mathcal{S}$ and action space $\mathcal{A}$.;
Output: Learned $V_w(s, a, \phi(\pi)) \approx Q^\pi(s, a) \forall s \in \mathcal{S}, \forall a \in \mathcal{A}, \forall \pi \in \Pi$, learned
  $\pi_\theta(s) \approx \pi^*(s) \forall s \in \mathcal{S}$;
initialize critic and actor parameters $w, \theta$;
while Stop criteria not met do
    create exploration policy $\pi_{\theta_e} = f_e(\pi_\theta)$;
    Generate an episode $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T$ with policy $\pi_{\theta_e}$;
    $t \leftarrow 0$;
    while $t < T$ do
        $mc_t \leftarrow \sum_{k=t}^{T} \gamma^{k-t} r_t$;            // discounted return with $\gamma \in [0, 1]$
        Store $(s_t, a_t, mc_t, \pi_{\theta_e})$ in replay buffer $D$;
        $t \leftarrow t + 1$;
    end
    for some steps do
        Sample a batch $B = (s, a, mc, \pi)$ from D;
        Update critic by stochastic gradient descent
          $\nabla_w \mathbb{E}_{(s,a,mc,\pi) \in B} [mc - V_w(s, a, \phi(\pi))]^2$
    end
    for some steps do
        Sample a batch $B = (s)$ from D;
        $a \leftarrow \pi_\theta(s)$;
        update actor by gradient ascent: $\nabla_\theta \mathbb{E}_{s \in B} [V_w(s, a, \phi(\pi_\theta))]$
    end
end
```

Algorithm 6: Comparing Start State Policy-Conditioned Actor Critic

```
Input: Differentiable Critic $V_w(\phi(\pi_1), \phi(\pi_2)) : \Pi \times \Pi \to [0,1]$ with
  parameters $w$ and policy embedding function $\phi$; deterministic actor
  $\pi_\theta : \mathcal{S} \to \mathcal{A}$ with parameters $\theta$, state space $\mathcal{S}$ and action space $\mathcal{A}$.;
Output:
learned $V_w(\phi(\pi_1), \phi(\pi_2)) \approx \begin{cases} 1, \text{if } \mathbb{E}_{s_0 \sim \mu_0}\left[V^{\pi_1}(s_0)\right] > \mathbb{E}_{s_0 \sim \mu_0}\left[V^{\pi_2}(s_0)\right] \\ 0, \text{if } \mathbb{E}_{s_0 \sim \mu_0}\left[V^{\pi_1}(s_0)\right] < \mathbb{E}_{s_0 \sim \mu_0}\left[V^{\pi_2}(s_0)\right] \\ 0.5, \text{otherwise} \end{cases}$
$\forall \pi_1, \pi_2 \in \Pi$,
learned $\pi_\theta(s) \approx \pi^*(s) \forall s \in \mathcal{S}$;
initialize critic and actor parameters $w, \theta$;
while Stop criteria not met do
    create exploration policy $\pi_{\theta_e} = f_e(\pi_\theta)$;
    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 $\pi_{\theta_e}$;
    calculate MC with discount factor $\gamma \in [0,1]$
    $mc \leftarrow \sum_{k=0}^{T} \gamma^k r_k$;
    Store $(mc, \pi_{\theta_e})$ in replay buffer $D$;
    for some steps do
        Sample a batch $B = (mc, \pi)$ from D;
        splitting Batch $B$ in half $B_1 = (mc_1, \pi_1)$, $B_2 = (mc_2, \pi_2)$;
        calculate targets $y = \begin{cases} 1, \text{if } mc_1 > mc_2 \\ 0, \text{if } mc_1 < mc_2 \\ 0.5, \text{otherwise} \end{cases}$;
        calculate classifications $\hat{y} = V_w(\phi(\pi_1), \phi(\pi_2))$;
        Update critic by stochastic gradient descent following binary cross
          entropy loss $\nabla_w \mathbb{E}_{(mc,\pi) \in B}\left[y \log \hat{y} + (1-y) \log(1-\hat{y})\right]$
    end
    for some steps do
        Sample a batch $B = (\pi_e)$ from D;
        update actor by gradient ascent $\nabla_\theta \mathbb{E}_{\pi_e \in B}\left[V_w(\phi(\pi_\theta), \phi(\pi_e))\right]$
    end
end
```

A.2 Hyperparameters

Here, we list the hyperparameters that have been used for all shown experiments. For the Pendulum-v0 environment a simple grid search with few options was used for optimization. For the HalfCheetah-v2 environment a research group internal implementation of the Cross Entropy Hyperparameter Optimization [18] has been used. Not all listed hyperparameters have been optimized, but all relevant hyperparameters for training are displayed. In all cases where a discount factor γ is used, it is set to 0.99 unless stated differently. The hyperparameters are briefly explained in the following:

in states	number of learnable input states/ neuron embedding dimensionality
hidden dims	dimensionality of the hidden layers of decoder or critic network
noise std	standard deviation for noise used to create exploration policy
ϵ	probability to use current actor policy for exploration without noise
random pol	probability to use newly initialized policy as exploration policy
critic rl	learning rate of the critic, this includes the embedding network
actor rl	learning rate of the actor network
rb size	maximal number of entries of replay buffer
batch size	batch size pulled from the replay buffer/dataset per training step
critic steps	number of critic updates per exploration step
actor steps	number of actor updates per exploration step
n-step	special hyperparameter of the N-Step PCAC algorithm (see 5.1)

For all policy networks we used fully connected networks with a tanh activation at the end, and ReLU activations in between. The output of the policy network is scaled according to the action space of the environment. All pendulum-v0 experiments used a policy with one hidden layer of 64 neurons. For HalfCheetah-v2 we used a small policy network with two hidden layers of 64 neurons each for the policy embedding comparisons. For the algorithmic variation comparisons and baselines, we used a larger policy of two hidden layers with 124 neurons each.

A.2.1 Baselines

The following shows the hyperparameters of the baselines we optimized ourselves. S PCAC stand for State PCAC and is our base algorithm, which we presented in chapter 3.3.

Table 1: Optimized Hyperparameters Baselines

	Pendulum-v0			HalfCheetah-2	
	S PCAC	PSVF	PSSVF	S PCAC	PSVF
in states	512	-	-	200	-
hidden dims	[512,512,512]	[512,512,512]	[512,512,512]	[1024,1024,1024]	[1024,1024,1024]
noise std	0.1	0.2	0.4	0.079	0.5
ϵ	0	0	0	0.1	0
random pol	0	0	0	0.1	0
critic lr	0.001	0.0001	0.0001	0.0004	0.00001
actor lr	0.0001	0.01	0.0001	0.00003	0.0003
rb size	200000	2000000	2000000	300000	300000
batch size	32	32	32	32	32
critic steps	10	10	10	8	5
actor steps	10	10	10	7	5

Additionally, we used pre-optimized baselines for DDPG [12] and PPO [20] from the RL Baseline3 Zoo [16]. After installing the package [16], the baselines have been measured as follows:

DDPG, Pendulum-v0 Run command:

```
python enjoy.py --algo ddpg --env Pendulum-v0 -n 20000 --no-render
```

DDPG, HalfCheetah-v2 RL Baseline3 Zoo does not have presets for HalfCheetah-v2, but for HalfCheetah-v3. We copied the hyperparameter from HalfCheetah-v3 to HalfCheetah-v2 in `/hyperparameters/ddpg.yml`. After doing so, run the following the commands to measure the baseline:

```
python train.py --algo ddpg --env HalfCheetah-v2
python enjoy.py --algo ddpg --env HalfCheetah-v2
   -f logs/ --exp-id 1 -n 100000 --no-render
```

PPO, Pendulum-v0 Run command:

```
python enjoy.py --algo ppo --env Pendulum-v0 -n 20000 --no-render
```

PPO, HalfCheetah Here the hyperparameters from HalfCheetah-v3 also need to be copied in `/hyperparameters/ppo.yml`. After Doing so, run following the commands to measure the baseline:

```
python train.py --algo ppo --env HalfCheetah-v2
python enjoy.py --algo ppo --env HalfCheetah-v2
   -f logs/ --exp-id 1 -n 100000 --no-render
```

A.2.2 Action Reconstruction

For the action reconstruction (see chapter 4.2) we used the following policy embedding dimensionality by dataset:

HalfCheetah-v2: 132

Pendulum-v0: 128

Random networks: 130

Since fingerprinting and neuron embedding are limited to multiples of the observation space dimensionality, these numbers cannot be arbitrarily chosen. The hyperparameters for specific embeddings were selected accordingly to create this embedding size. For the neuron embedding each layer always has the same dimensionality to embed individual neurons and translation networks always have no hidden layers. The following hyperparameters were used for the action reconstruction settings:

Table 2: Optimized Hyperparameters for HalfCheetah-v2

	hidden dims	learning rate	batch size
Linear	[66, 66]	0.001	64
Neuron	[132, 132]	0.001	128
Fingerprint	[66, 66]	0.001	64

Table 3: Optimized Hyperparameters for Pendulum-v0

	hidden dims	learning rate	batch size
Linear	[128, 128, 128]	0.001	64
Neuron	[128, 128, 128]	0.001	64
Fingerprint	[128, 128, 128]	0.001	256

Table 4: Optimized Hyperparameters for Random networks

	hidden dims	learning rate	batch size
Linear	[65, 65, 65]	0.00001	64
Neuron	[130, 130, 130]	0.001	256
Fingerprint	[130, 130]	0.001	256

A.2.3 Reinforcment Learning Policy Comparison

For the policy embedding comparison in the Reinforcement Learning setting (chapter 4.3) the following hyperparameters have been used.

Table 5: Optimized Hyperparameters for Reinforcement Learning Embedding comparison

	Pendulum-v0			HalfCheetah-2		
	Flat	Neuron	Fingerprint	Flat	Neuron	Fingerprint
in states	-	512	512	-	92	99
hidden dims	[512,512,512]	[512,512,512]	[512,512,512]	[512,512,512]	[512,512]	[512,512,512]
noise std	0.4	0.2	0.1	0.08	0.1	0.1
ϵ	0	0.1	0	0.1	0.1	0.1
random pol	0.1	0.1	0	0	0	0
critic lr	0.0001	0.0001	0.001	0.00001	0.00008	0.0007
actor lr	0.0001	0.0001	0.0001	0.00004	0.0001	0.00008
rb size	2000000	2000000	2000000	5000000	9000000	6300000
batch size	32	32	32	64	512	64
critic steps	10	10	10	5	5	5
actor steps	10	10	10	5	5	5

A.2.4 Algorithmic Variations

The following hyperparameters were used for our algorithmic variations. All N-Step PCAC experiments use a discount factor of $\gamma = 0.98$ instead of 0.99. Multi actor variations are not listed here, as they always use the hyperparameters of their base algorithm.

Table 6: Optimized Hyperparameters for Algorithmic Variations

	Pendulum-v0			HalfCheetah-2		
	N-Step	Comp Start State	Comp State	N-Step	Comp Start State	Comp State
in states	512	256	512	140	100	220
hidden dims	[512,512,512]	[512,512,512]	[512,512,512]	[1024,1024,1024]	[1024,1024,1024]	[1024,1024,1024]
noise std	0.1	0.2	0.2	0.05	0.1	0.077
ϵ	0	0.1	0.1	0.1	0	0.1
random pol	0	0.1	0.1	0.1	0	0
critic lr	0.001	0.001	0.001	0.000003	0.0002	0.001
actor lr	0.0001	0.0001	0.0001	0.000003	0.00003	0.00005
rb size	5000	1000	200000	5000	1000	300000
batch size	32	32	32	32	32	32
critic steps	10	10	10	11	16	16
actor steps	10	10	10	4	5	1
n-step	50	-	-	100	-	-

A.3 Code Base

By far the most amount of work of this thesis went into developing the necessary code basis to support the shown experiments and many other experiments that were not included in this thesis. The code has been published on GitHub under the following URL: `https://github.com/Sebastian-Griesbach/Improving-Policy-Conditioned-Value-Functions`

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Tübingen, den 31.03.2022

Sebastian Griesbach