DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Data Engineering and Analytics

Improved Exploration with Stochastic Policies in Deep Reinforcement Learning

Johannes Pitz

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Data Engineering and Analytics

Improved Exploration with Stochastic Policies in Deep Reinforcement Learning

Verbesserte Exploration mit Stochastischen Strategien in Deep Reinforcement Learning

Author:	Johannes Pitz
Supervisor:	Prof. Dr.-Ing. Darius Burschka
Advisor:	Prof. Dr.-Ing. Berthold Bäuml
Submission Date:	September 15, 2020

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, September 15, 2020 Johannes Pitz

Abstract

Deep reinforcement learning has recently shown promising results in robot control, but even current state-of-the-art algorithms fail in solving seemingly simple realistic tasks. For example, OpenAI et al. 2019 demonstrate the learning of dexterous in-hand manipulation of objects lying on the palm of an upside oriented robot hand. However, manipulating an object from above (i.e., the hand is oriented upside-down) turns out to be fundamentally more difficult to learn for current algorithms because the object has to be robustly grasped at all times to avoid immediate failure. In this thesis, we identify the commonly used naive exploration strategies as the main issue. Therefore, we propose to utilize more expressive stochastic policy distributions to enable reinforcement learning agents to learn to explore in a targeted manner. In particular, we extend the Soft Actor-Critic algorithm with policy distributions of varying expressiveness. We analyze how these variants explore in simplified environments with adjustable difficulties that we designed specifically to mimic the core problem of dexterous in-hand manipulation. We find that stochastic policies with expressive distributions can learn fundamentally more complex tasks. Moreover, beyond the exploration behavior, we show that in not perfectly observable environments, agents that represent their final (learned) policy with expressive distributions can solve tasks where commonly used simpler distributions fail.

Contents

1 Introduction

1.1 In-Hand Manipulation as Motivational Problem

Robot arms have been used in factories all over the world to execute repetitive tasks with superhuman endurance and precision for many years. Over the last decades, the hardware of these systems became so advanced that humanoid robots, such as "Agile Justin" (Bäuml, Hammer, et al. 2014), are physically capable of carrying out tasks such as building scaffolds, a variety of housework, or catching flying balls (Bäuml, Birbach, et al. 2011). However, today the cognitive abilities of robots are less impressive, and without the necessary software, their full physical potential cannot be exploited. Designing complex control pipelines for individual tasks is possible but requires engineering effort. A promising approach to solve many different problems in a unified manner is deep reinforcement learning (DRL). The goal of DRL is to automatically learn to solve a given task via smart trial and error, either on the real robot or in simulation. DRL has shown impressive results on game environments (Mnih et al. 2015, D. Silver et al. 2016) but was also applied successfully to robotics problems, such as opening a valve (Haarnoja, Zhou, et al. 2018) or solving the Rubik's Cube (OpenAI et al. 2019). Robots fit perfectly into the reinforcement learning framework, and one could envision them learning completely autonomous from interactions in the future. However, the algorithms we know today are still limited. In this thesis, we want to investigate one such limitation and design modifications to state-of-the-art algorithms that solve the problem.

Dexterous in-hand manipulation

OpenAI et al. 2019 show that given enough computational resources, a DRL agent can be trained to solve the Rubik's cube, which is an extremely challenging fine manipulation task. However, they place the cube in the hand such that it does not drop unless the fingers push it away. Unfortunately, holding an object from above is fundamentally more difficult to learn for a DRL agent, as Sievers 2020 shows. In Figure 1.1 we depict the setup of his experiments. The DLR Hand II (Butterfaß et al. 2001) is a four-fingered robotic hand with twelve degrees of freedom (DOF) and an advanced sensory system that is capable of precise fine manipulation given an appropriate controller. Thus far, however, we can only train DRL agents that hold the cube and tilt it around the vertical

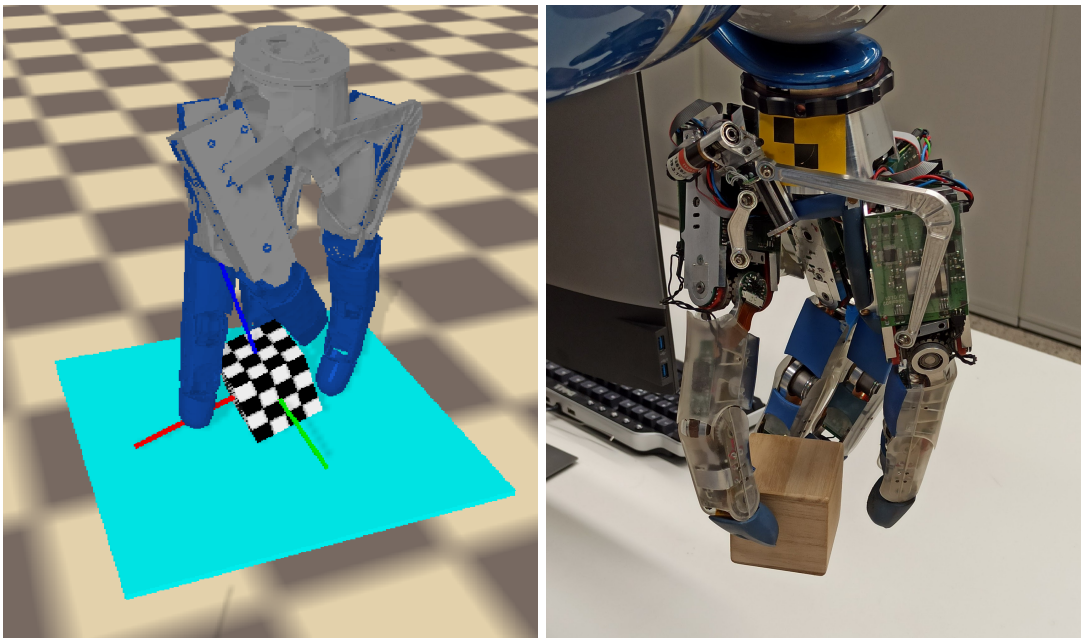


Figure 1.1: The DLR Hand II holding an object from above in simulation (left) and in the real-world (right).

axis. Full rotations, which would require lifting individual fingers and moving them to another side, are not yet possible.

The fundamental difference between holding an object from above or below is, of course, that small mistakes lead to immediate failure. Therefore, a DRL agent has to explore the environment much more carefully to make any progress. Because almost every DRL algorithm for continuous control problems naively explores by adding uncorrelated noise to all joint positions, careful exploration implies inevitably little exploration. Unfortunately, small exploration steps result in slow learning progress and make it increasingly less likely that an agent finds long successful sequences, such as moving fingers to other sides, without being explicitly rewarded for intermediate progress. However, rewarding intermediate progress (reward shaping) is undesirable because it requires additional human design efforts for every new task. Therefore, smarter exploration strategies that shut down the exploration entirely in directions where the agent dies immediately (i.e., the object drops out of the hand) are required.

The solution we pursue in this thesis is to *learn* the exploration strategy, along with the final policy, through environment interactions. This method is commonly used in so-called on-policy algorithms. In contrast, off-policy algorithms usually use predefined exploration strategies, such as epsilon-greedy exploration, which cannot adapt automatically to different tasks. Moreover, the current algorithms which learn

an exploration strategy usually learn simple models that cannot represent correlations between action dimensions. However, correlated exploration may be essential to allow the agent to learn that lifting one finger is safe, as long as the other fingers exert the necessary pressure. Ultimately, we believe that proper dexterous in-hand manipulation will require DRL with learned exploration strategies represented by powerful models and incentives for the agent to continuously explore new states.

Goal of the thesis

The core of this work is the learning of exploration strategies. To allow for that, we model a stochastic *policy*, from which the agent samples its next actions. Stochastic policies are represented by probability distributions (fitted with standard machine learning methods, such as maximum likelihood estimation). Now, since games generally operate in a simple to model discrete action spaces, it is possible to avoid the problem of choosing an appropriate model in many research fields. However, most real-world problems, in particular, robot control problems, naturally have a continuous action space. And there exists an enormous range of models for representing such distributions. Thus far, the most successful algorithms on continuous control benchmarks have relied on extremely simple diagonal normal distributions. As mentioned before, this distribution does not allow correlations between action dimensions. However, in complex environments, such as in-hand manipulation, the agent needs to be capable to explore on a low dimensional submanifold of the whole action space to make progress without instantly failing (i.e., dropping the object) and resetting the environment.

Therefore, we investigate, in controlled environments, how powerful the policy distributions for learned exploration strategies need to be. Unfortunately, we have to leave investigations into incentivizing the finding of new states for future work.

1.2 Fundamentals of Reinforcement Learning

Reinforcement learning has seen a surge of interest since the first successes with the application of deep learning methods. In particular, influential were the Deep Q-Network (DQN), which showed impressive results on challenging Atari games (Mnih et al. 2015), and AlphaGo beating a human grandmaster in the notoriously difficult game of Go (D. Silver et al. 2016). In this section, we briefly introduce the basic mathematical framework of reinforcement learning and a few modern algorithms. Moreover, we also discuss the trade-offs between deterministic and stochastic policies.

Agent-environment interaction

In reinforcement learning, we typically think of agents interacting with different environments, see Figure 1.2.

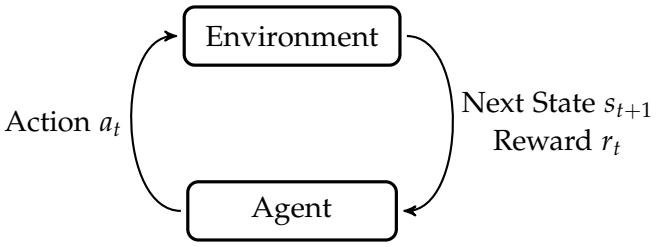


Figure 1.2: The agent-environment interaction in reinforcement learning.

First the agent observes the state s_t and chooses an action a_t to execute. Then the environment steps forward: it computes the next state s_{t+1} and the reward r_t associated with the last action. These are observed by the agent and the procedure repeats. Through multiple interactions, so-called, trajectories emerge.

$$\tau = s_0, a_0, r_0, s_1, a_1, r_1, s_2 \dots \tag{1.1}$$

Note that an ultimate reinforcement agent would find an optimal strategy in any given environment after a small number of interactions. Considering that, for example, all supervised machine learning problems could be phrased as an environment-interaction problem, it becomes clear that a reinforcement learning agent needs to be extremely adaptive.

Markov decision process

The environment of reinforcement learning problems can mathematically be framed by a Markov decision process (MDP), which is defined by:

- The set of states $\mathcal{S}$ (state space).

- The set of actions $\mathcal{A}$ (action space).
- The transition probability $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$, which determines the chance of going to state s_{t+1} when executing action a_t in state s_t.
- The reward probability $r_t \sim R(r_t|s_t, a_t)$, which determines the chance of receiving reward r_t when executing action a_t in state s_t.
- The start state distribution $s_0 \sim P_{\text{start}}(s_0)$.

The goal in an MDP is to find a policy $a_t \sim \pi(a_t|s_t)$ that optimizes the expected discounted sum of returns $\mathbb{E}_{\tau\sim\pi}[G_0]$, where

$$G_t = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_{t'} \tag{1.2}$$

where $\gamma < 1$ is a discount factor, which in combination with bounded returns guarantees that the sum converges.

Assuming full knowledge of the environment, the problem can be solved exactly by value iteration (Bellman 1957). However, value iteration requires expectations over the state space, which are only feasible for small environments with finite state space. Modern deep reinforcement learning algorithms approximate expectations with neural network functions and sampling techniques to admit more complex problems.

Components of an agent

An agent chooses the next action depending on the new state and its internal components. Moreover, after each step, the agent may use the received reward signal to adjust these components.

- **Policy.** Every agent needs a policy $a_t \sim \pi(a_t|s_t)$, as we introduced it before. However, it is not always necessary to represent it explicitly. For example, a DQN agent can deduce its next action immediately from the Q-function, therefore, eliminating the need to represent the policy with an additional neural network.
- **Value Function.** The value function is defined as the expected return at any state given a fixed policy $V^\pi(s) = \mathbb{E}_{\tau\sim\pi}[G_t|s_t = s]$. Similarly the Q-function is defined as the expected return at any state-action-pair given a fixed policy $Q^\pi(s, a) = \mathbb{E}_{\tau\sim\pi}[G_t|s_t = s, a_t = a]$. Most modern algorithms use one of the two functions and approximate them with a neural network.
- **Model.** In some settings, the agent can be equipped with a fixed model, i.e., the rules of the game in AlphaGo. Such a model can be exploited with planning algorithms to improve estimates and learn more quickly. However, a fixed

model limits the applicability of the algorithm drastically. Although recently Schrittwieser et al. 2019 showed that models can also be learned in many settings, we only consider model-free algorithms in this thesis to reduce the complexity.

On-policy algorithms

Reinforcement learning algorithms can be divided into on-policy and off-policy variants. Both have advantages and drawbacks. On-policy algorithms are usually derived from the basic policy gradient update (Sutton et al. 1999) and explicitly learn a policy function (but no Q-function). The main drawback of policy gradient-based algorithms is that the samples used to update the policy must be collected with the same policy. Collecting too little samples before updating the policy leads to high gradient variance (Abdolmaleki et al. 2018) and sometimes catastrophic forgetting. However, collecting new environment samples may be extremely expensive, for example, in real-world robotic tasks or tasks which require complex simulations.

Their advantage is that policy gradient-based algorithms are usually less sensitive to hyperparameter settings and can be applied across many tasks. Moreover, modern algorithms tackle the sample efficiency problem by applying multiple updates to the policy after collecting a sizeable amount of samples. It is important to ensure that the policy does not change too much from the one that collected the samples. Recently the most successful and easily implementable algorithm is Proximal Policy Optimization (PPO) by Schulman et al. 2017. OpenAI et al. 2019 showed that PPO can be effectively applied to robot control tasks given enough computational resources for simulation.

Off-policy algorithms

In contrast to on-policy algorithms, off-policy algorithms (Q-learning algorithms) can reuse old samples. Reusing samples is possible because the gradients of the Q-function do not depend on the policy which collected them. Therefore, off-policy algorithms usually utilize a replay buffer from which they can draw old samples, reducing the need for collecting new ones. The superior sample efficiency has lead to great success in challenging domains, such as Atari 2600 games and classical board games (Badia et al. 2020, Schrittwieser et al. 2019). Moreover, it attracted the interest of the robotics community due to the potential of learning on real-world systems.

The main problem of applying off-policy algorithms to continuous control tasks is that standard Q-learning methods require taking a maximum operation over the action space

$$a_t = \max_{a \in \mathcal{A}} Q^{\pi}(s_t, a) \tag{1.3}$$

In discrete action spaces, the neural network that represents the Q-function can have separate outputs for each possible action, but that is not possible in continuous action

spaces. Lillicrap et al. 2016 solved this problem by passing the action as an input to the Q-function and training an explicit deterministic policy to output actions that maximize the Q-function. This algorithm is called Deep Deterministic Policy Gradient (DDPG), and many variants of it have been proposed. Most notably, the Twin Delayed DDPG algorithm by Fujimoto et al. 2018 introduces the double Q-leaning trick and target smoothing that improves performance in most continuous control benchmarks.

Deterministic policies

Most off-policy algorithms applicable for continuous control tasks, i.e. DDPG and its variants, use deterministic policies which entirely decouples the policy from the exploration behavior.

$$\begin{aligned} a_t &= \Phi(s_t) + \epsilon_t \\ \epsilon_t &\sim \mathcal{N}(0, \epsilon) \end{aligned} \tag{1.4}$$

In principle, it should be an advantage that the exploration strategy can be adjusted during training but it also introduces additional complexity. The simple epsilon greedy exploration in (1.4), requires already multiple hyperparameters ($\epsilon_{\max}$, $\epsilon_{\min}$, and a time step T_ϵ until which the epsilon decays linearly). More complex strategies which achieve deeper exploration, such as time-correlated Ornstein-Uhlenbeck noise (Lillicrap et al. 2016), introduce even more hyperparameters. In practice, off-policy algorithms are sensitive to these hyperparameter and, therefore, difficult to tune (Abdolmaleki et al. 2018).

Stochastic policies

On-policy algorithms, on the other hand, rely on stochastic policies. The distribution that represents the policy needs to exploit the environment reward but also defines the exploration behavior. Empirically, stochastic policies are less sensitive to hyperparameter settings and can be applied to different environments without individual tuning (Abdolmaleki et al. 2018). In practice, researches mostly use categorical distributions in discrete settings and diagonal normal distributions in continuous action spaces.

$$a_t \sim \mathcal{N}(\mu(s_t), \sigma(s_t)^2) \tag{1.5}$$

The reparametrization trick is used to backpropagate the gradients through the neural networks that compute the mean and variance of the distribution.

Partially observable Markov decision process

The partially observable Markov decision process (POMDP) is a generalization of the MDP. Instead of observing the full state of the environment, the agent only receives

observations that are not necessarily sufficient to model the transition and reward distributions. Most modern reinforcement algorithms can still be applied in partially observable settings. However, the key difference between the processes is that only in an MDP there always exists an optimal deterministic strategy. A simple POMDP example, which cannot be solved by a deterministic strategy, is a discretized Labyrinth where the agent can only see if the adjacent squares are walls or free space. If there are two positions in the Labyrinth where all adjacent squares are free space, but the first one requires the agent to move up and the second one requires the agent to move left to solve the task, then a deterministic policy can never be successful.

Most real-world problems are partially observable. In robotics, for example, providing an agent with joint angles but not the joint velocities yields a POMDP. In this case, the problem can be avoided by passing the missing information or a history of states. However, in many scenarios, it might not be realistic to determine how long the history needs to be or which additional observations are required to reduce the problem to an MDP. Moreover, even the smallest measurement noise suffices to technically elevate the problem to a POMDP.

1.3 Related Work

In the following, we discuss previous work related to the exploration problem in deep reinforcement learning and our solution of using expressive stochastic policies.

Expressive policies in reinforcement learning algorithm

Most similar to our work is the Soft-Q algorithm, by Haarnoja, Tang, et al. 2017. They showed that it is possible to work with highly expressive stochastic policies in continuous action spaces. To this end, they propose a method for learning energy-based policies and a modified objective that incentivizes high entropy distributions. Optimizing this objective solves the maximum entropy learning problem (Toussaint 2009). However, while this algorithm theoretically solves the issue we describe in the motivational problem, its performance on benchmark environments was never particularly impressive. Later the authors proposed to parametrize the policy with a diagonal normal distribution and presented the Soft Actor-Critic (SAC) algorithm (Haarnoja, Zhou, et al. 2018). SAC is a state-of-the-art algorithm and superior to the Soft-Q algorithm on standard benchmark environments. The gap in expressiveness between these two models is tremendous and raises the question, whether stochastic policies based on models with intermediate expressiveness could be even more successful than the SAC algorithm. In this thesis, we want to design such variants of the SAC algorithm. We want to find out if there are suitable policy distributions that are sufficiently expressive to allow targeted exploration through correlating action dimensions but are as performant and easy to learn as the original algorithm.

Stochastic policy distribution models

Modeling a reinforcement learning agent's policy is all about modeling a probability distribution of which samples (actions) can be sampled. Such models are called generative models. Recently, deep generative models with millions or billions of parameters have gained much attention. Generative Adversarial Networks (GAN) are capable to produce realistic images (Goodfellow et al. 2014) and auto-regressive neural language models such as BERT or GPT-3 (Devlin et al. 2019, Brown et al. 2020) can generate entire poems which are difficult to discern from genuine texts. However, in the scope of this thesis, we are interested in modeling simple distributions over a low dimensional continuous action space in reinforcement learning. Therefore, we will briefly discuss a range of generative models that might be interesting in this setting.

- **Deterministic models.** In its simplest form, a generative model is a function, such as $f(x) = x^2$ or, for example, an image rendering engine. Given some input the model generates output. Therefore, the previously introduced DDPG algorithm,

which uses a deterministic function for its policy could be viewed as a degenerate special case of the SAC algorithm.

- **Stochastic models.** A stochastic or statistical generative model is usually fitted to a dataset and can then be used to draw new samples that are similar to the training data. Prior knowledge about the target distribution can be added explicitly in the form of distribution families or implicitly by choosing a certain loss function or learning algorithm. The simplest such models are the categorical and the Gaussian distribution in the discrete and continuous setting respectively. Combining the two distributions yields the Gaussian Mixture Model (GMM) which is arguable the simplest proper stochastic generative model.

In general, three main properties are desirable for stochastic generative models. However, depending on the usage it might be perfectly acceptable to give up some of them.

- **Efficient sampling.** Drawing new samples $x_{\text{new}} \sim p(x)$ should be a quick operation.
- **Efficient scoring.** Evaluating the likelihood $p(x)$ of a given sample x should be a quick operation.
- **Interpretable features.** There exists a latent representation for each sample that captures the important features, such as the component in a GMM.

Modeling the policy distribution of the SAC algorithm only requires efficient sampling and computing the likelihood of those samples, i.e. the algorithm does not require scoring arbitrary samples. However, many other reinforcement learning algorithms that could also profit from expressive distributions, such as PPO, require efficient scoring of arbitrary samples. Interpretable features are not of particular importance because it is extremely challenging to analyze the features of millions of actions.

In the following, we quickly discuss several popular generative models:

- **Normalizing flows.** Normalizing flows are convenient invertible functions which are used to map simple distributions to more complicated ones. Many different models can emerge from normalizing flows but simple flows usually allow efficient sampling and efficient scoring while they are significantly more expressive than normal distributions or GMMs (cf. Section 2.1). Haarnoja, Hartikainen, et al. 2018 used normalizing flows as a policy distribution but with a different goal than we have here. They stack multiple policies such that a higher level policy's output is used as the lower level policy's base distribution. This is a promising approach to hierarchical reinforcement learning but they ignore the gained expressiveness

of the policy in their analysis. Since normalizing flows are easy to implement, not well studied in the reinforcement learning setting, and their expressiveness should be more than sufficient to model correlated actions, we decided to use them for our experiments.

- **Variational inference.** Variational methods are used to approximate intractable integrals. In variational inference a simple distribution Q is used to approximate the posterior distribution P over the unobservable variable $\mathbf{Z}$ given some data $\mathbf{X}$: $P(\mathbf{Z} \mid \mathbf{X}) \approx Q(\mathbf{Z})$. The variational distribution Q is fitted to the posterior by minimizing the KL-Divergence between them (cf. Kingma and Welling 2014). The variational distribution can usually be sampled and scored efficiently but it is only an approximation of the true posterior. Additionally, a low dimensional latent variable could yield interpretable features. Fellows et al. 2019 propose a variational inference framework for reinforcement learning (VIREL). They derive theoretical results and show impressive empirical performance. However, we decided against using variational inference for our experiments because they are more complicated to implement than normalizing flows and it is not obvious that the additional expressiveness would yield any benefit.

- **Generative adversarial networks.** Generative adversarial networks (GANs) are density free models that are trained via alternating optimization of a discriminator and a generator (cf. Goodfellow et al. 2014). They are not immediately applicable to reinforcement learning because density free implies that it is not possible to score samples.

- **Energy-based models.** Energy-based models assign an unnormalized scalar ("energy") to each input which represents its probability (Du et al. 2019). Samples from such a model can be drawn through iterative processes or by training an additional generative model (similar to a GAN). Haarnoja, Tang, et al. 2017 show that energy-based models can be used for Q-learning. They propose the Soft-Q algorithm which uses a generator network (presumably because it is faster than an iterative process). However, the authors later realized that the additional complexity is largely unnecessary and presented the SAC algorithm (with a diagonal normal distribution) as a superior successor of the Soft-Q algorithm. Therefore, we decided to not use energy-based models for our experiments.

Reducing the problem with classical control methods

Classical non-differentiable policies can be improved by model-free deep reinforcement learning. T. Silver et al. 2018 propose the Residual Policy Learning framework, which can be orders of magnitude more sample efficient than plain deep reinforcement learning if a reasonable initial controller exists. Moreover, they claim that in this setting

agents can solve long-horizon, sparse reward problems, which would be impossible to learn from scratch.

Instead of composing commands, Li et al. 2019 learn a hierarchical policy based on classical control methods and reinforcement learning. Manipulating an object while holding it is an extremely challenging task for reinforcement agents because many of the possible actions result in abrupt failure. To circumvent the problem, the authors employ two separate controllers. On the lower level is a traditional model-based controller, which can execute manipulation primitives without dropping the object. And on the higher level, a reinforcement agent is trained to orchestrate these primitives.

Both approaches are promising and could potentially solve our motivational problem, but designing the required classical controller is non-trivial and limits the use case to one specific system. In this thesis, we focus solely on deep reinforcement learning algorithms that learn only from environment interactions. We aim to understand how algorithms in the general framework learn and if we can modify them to exhibit these desired properties automatically.

Curiosity-driven exploration

Here, we describe methods that incentivize the exploration of new states. They aim to achieve deep exploration by explicitly rewarding it. In contrast to our work, they work in discrete action spaces and, therefore, mostly ignore the problem of cautious exploration. However, we believe that expressive distributions will be a necessary foundation for the successful application of curiosity-driven methods to complex control problems.

Recently researchers started investigating methods to incentivize the agent to explore previously undiscovered states with higher probability. Simple approaches count how often the agent visited certain states and reward actions that lead to novel states. Ostrovski et al. 2017, for example, use neural density models to facilitate counting in high dimensional state spaces. More complex methods include but are not limited to predictability (Pathak et al. 2017) or reachability (Savinov et al. 2019) measures, and also skill or goal based approaches (Bougie et al. 2020).

Curiosity-driven exploration has great potential for enabling deep exploration, and might also be applicable in settings where exploration in most directions leads to certain death. However, the mentioned methods always require an additional function, usually a neural network, to score the merit of states or entire trajectories. This additional component and the weighting between curiosity reward and environment reward makes it often difficult to tune the algorithm. Therefore, we decided to leave the application of curiosity-driven exploration for future work. Instead, we try to solve the thus far largely ignored problem of cautious exploration with a simple and more robust method.

1.4 Outline and Contributions

In this thesis, we provide an in-depth investigation of the effects of policy distributions with varying expressiveness in deep reinforcement learning. While previous works have experimented with highly expressive policy distributions (cf. Section 1.3), to the best of our knowledge, an incremental analysis, such as ours, has not been conducted before.

In Chapter 2, we analyze normalizing flows and develop an extension to the Soft Actor-Critic algorithm.

- We provide a detailed background of normalizing flows and validate three simple models in a density fitting experiment. Moreover, we introduce conditional normalizing flows and expose novel empirical generalization properties that are crucial for their efficient utilization in policy distributions.
- We devise two extensions to the Soft Actor-Critic algorithm. We replace the originally used diagonal normal distribution in the policy with more expressive alternatives, where one of them relies on the previously discussed conditional normalizing flow model.

In Chapter 3, we conduct innovative experiments on environments with various exploration difficulties. The experiments convincingly show that under certain conditions, expressive policy distributions are necessary to solve the given task.

- We validate the implementation of our algorithms on standard benchmark environments.
- We introduce a simple robot navigation environment designed to showcase the simplest setting in which the original diagonal normal policy distribution fails to explore the environment. While more expressive policies can exploit correlations between action dimensions and solve the task.
- We present a 2D fine manipulation environment, designed to translate the results from the previous section to a more realistic robotic task. We show that the results carry over, but higher-dimensional freedoms can reduce the effects.
- We show that expressive policies may be of fundamental importance in partially observable environments, beyond the improved exploration. In this setting also the final policy can require correlated actions that are not representable by a naive normal distribution.

2 Deep Reinforcement Learning with Stochastic Policies

In this chapter, we develop the methods used in experiments in the following chapter. First, we introduce normalizing flows and study their applicability for modeling stochastic policy distributions for reinforcement learning agents. Then, we present the Soft Actor-Critic algorithm and show how it can be extended with expressive policy distributions, such as normalizing flows.

2.1 Analysis of Normalizing Flows

In this section, we give a more detailed introduction to normalizing flows and present density fitting experiments to show how simple flows are much more expressive than a normal distribution. Moreover, we introduce conditional normalizing flows (Trippe et al. 2018) that are key to utilizing normalizing flows for reinforcement learning. Finally, we empirically expose basic generalization properties of conditional normalizing flows that, to the best of our knowledge, have not been explored before.

2.1.1 Background

Normalizing flows are used to map simple distributions to more complicated ones (see Figure 2.1 for an example). They were first introduced by Tabak et al. 2013, who showed that using a differentiable, monotonic bijection as a mapping allows us to force the output to be a proper probability distribution. Composing a series of simple invertible mappings then results in more complex transformations. Using such composed transformations, Rezende et al. 2015 showed that a standard normal distributions can be morphed into rich approximate posteriors in the context of variational inference and Dinh et al. 2017 popularized the use of normalizing flows for density estimation. For a more detailed summary of normalizing flows in general and their applications, we refer to the review by Kobyzev et al. 2020.

Change of variable

To sample from a complex distribution that is approximated by an initial distribution and a chain of transformations is trivial. First, a sample of the initial distribution, i.e.,

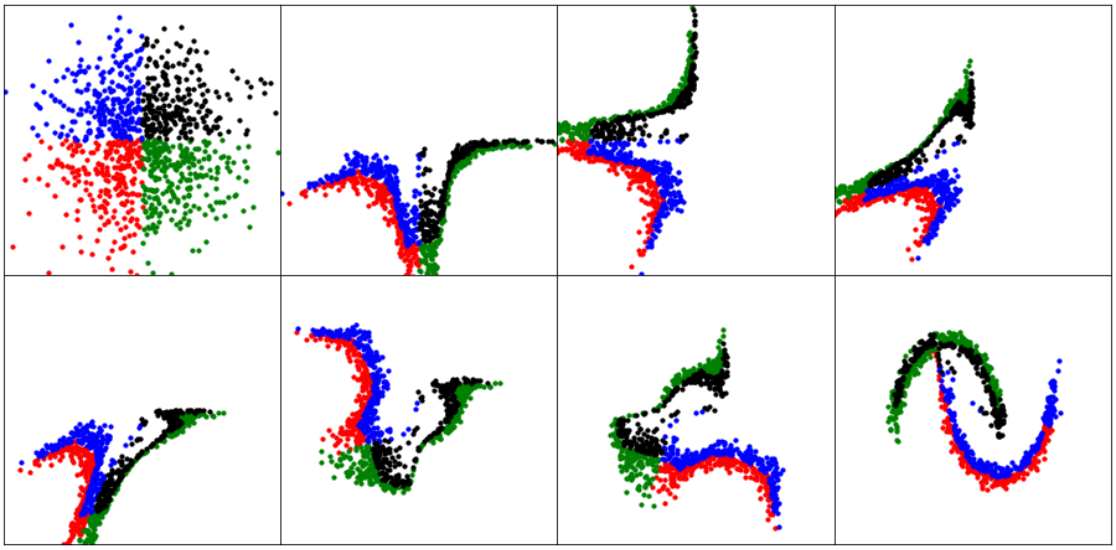


Figure 2.1: A two dimensional standard normal distribution (top left) is mapped to a distribution which explains the two moons dataset (bottom right). From left to right, top to bottom, we show the transformed samples after each layer of the normalizing flow. The points are colored by the quadrant they originate from.

a standard normal or uniform distribution, is drawn. Then the transformations are applied in sequence. However, computing the density of a sample requires to first transform it back to the initial distribution. Then we can compute the probability density of this inverse-transformed sample under the initial distribution and apply the change of volume induced by the inverse transformations. The change of variables formula determines that the change of volume is the product of the absolute values of the determinants of the Jacobians for each transformation (cf. Kobyzev et al. 2020).

Let $\mathbf{Z} \in \mathbb{R}^D$ be a random variable with probability density $p_{\mathbf{Z}} : \mathbb{R}^D \to \mathbb{R}$. Let $\mathbf{Y} = g(\mathbf{Z})$ be the output of an arbitrary invertible and differentiable function g, with inverse $f = g^{-1}$. Then the probability density of $\mathbf{Y}$ is determined by the change of variable formula.

$$\begin{aligned} p_{\mathbf{Y}} &= p_{\mathbf{Z}}(f(y))|\det \mathcal{D}f(y)| \\ &= p_{\mathbf{Z}}(f(y))|\det \mathcal{D}g(f(y))|^{-1} \end{aligned} \tag{2.1}$$

Where

$$\mathcal{D}f(y) = \frac{\partial f}{\partial y}, \quad \mathcal{D}g(z) \quad = \frac{\partial g}{\partial z} \tag{2.2}$$

are the Jacobians of f and g respectively.

Bogachev et al. 2007 proofed that under reasonable assumptions arbitrarily complex functions g can generate any distribution on $\mathbf{Y}$ from any base distribution on $\mathbf{Z}$. How-

ever, constructing arbitrarily complex functions with the required properties at once is difficult. Therefore, we chain multiple simpler functions together to create more complex ones while preserving the desired properties.

Let g_i be bijective functions which are easy to compute, invert, and to calculate the determinant of their Jacobian. Then the function

$$g = g_N \circ g_{N-1} \cdots \circ g_1 \tag{2.3}$$

is also bijective with the inverse

$$f = f_1 \circ \cdots \circ f_{N-1} \circ f_N \tag{2.4}$$

and the determinant of the Jacobian is

$$\det \mathcal{D}f(y) = \prod_{i=1}^{N} \det \mathcal{D}f_i(x_i) \tag{2.5}$$

where $\mathcal{D}f_i(x_i) = \frac{\partial f_i}{\partial x}$ is the Jacobian of f_i, and $x_i = g_i \circ \cdots \circ g_1(z) = f_{i+1} \circ \cdots \circ f_N(y)$, such that $x_N = y$.

Density estimation

Given a dataset $\mathcal{D} = \{y^{(i)}\}_{i=1}^{M}$ of samples from a complex distribution. We choose a simple base distribution on $\mathbf{Z}$ and a possibly composed flow function g, parametrized by parameters Θ. The data likelihood can be computed by

$$\begin{aligned} \log p(\mathcal{D}|\Theta) &= \sum_{i=1}^{M} \log p_{\mathbf{Y}}(y^{(i)}|\Theta) \\ &= \sum_{i=1}^{M} \log p_{\mathbf{Z}}(f(y^{(i)}|\Theta)) + \log \left| \det \mathcal{D}f(y^{(i)}|\Theta) \right| \end{aligned} \tag{2.6}$$

where the first term is the likelihood of the inverse-transformed sample under the base distribution and the second term is the volume correction for the change of variable. In case of a composed function g the product in (2.5) becomes a sum over the individual correction terms because we are working in log space.

The parameters Θ can be adjusted by gradient descent to maximize the likelihood. Note that that requires many applications of the inverse f and computing its log determinant. Sampling, on the other hand, only requires the forward computation of g. Therefore, depending on the application, one may choose less efficiently computable inverse or forward functions to obtain more complexity (cf. Kobyzev et al. 2020).

Autoregressive property

One commonly used technique to efficiently increase the expressiveness of the final distribution are autoregressive models (Papamakarios et al. 2017, Oord et al. 2016). The joint density of the random vector is decomposed into one-dimensional conditional densities such that each dimension only depends on the previous ones

$$p(y) = \prod_{i=1}^{D} p(y_i | y_{1:i-1}). \tag{2.7}$$

Typically the conditional densities are modeled as normal distributions with learnable parameters. In particular, a neural network Ψ which computes the mean and standard deviation of a normal distribution is used

$$\begin{aligned} p(y_i | y_{1:i-1}) &= \mathcal{N}(y_i | \mu_i, \exp(\beta_i)^2) \\ \mu_i &= \Psi_{\mu_i}(y_{1:i-1}) \\ \beta_i &= \Psi_{\beta_i}(y_{1:i-1}). \end{aligned} \tag{2.8}$$

The autoregressive property guarantees triangular determinants of which we can efficiently evaluate the determinant. The strong inductive bias that the first dimensions do not depend on the latter ones can be bypassed by permuting the vector between layers (cf. Jang 2018).

Sampling from the complex distribution is simple in this example:

$$\begin{aligned} z_i &\sim \mathcal{N}(0, 1) \\ y_i &= z_i \exp(\beta_i) + \mu_i. \end{aligned} \tag{2.9}$$

And note that scoring arbitrary samples y does not require inverting the neural networks. We can compute the corresponding base distribution sample $z = f(y)$ by reversing the scale and shift of the forward pass

$$z = \frac{y - \Psi_\mu(y)}{\exp(\Psi_\beta(y))}. \tag{2.10}$$

Simple flows

We present a selection of normalizing flows that we then evaluate in the following experiment.

- **Masked Autoregressive Flow (MAF)**. The MAF does exactly what we described above. Germain et al. 2015 introduce a clever network masking scheme that allows computing all μ_i and β_i in a single pass. Therefore, scoring samples and fitting parameters are extremely fast, while sampling still requires sequentially computing the individual dimensions of the output sample in each layer.

- **Real-Valued Non-Volume Preserving Flow (Real-NVP)**. Real-NVP (Dinh et al. 2017) is a simplification of the MAF. Instead of conditioning each dimension on the previous ones the authors simply condition one part of the vector (usually half) on the other. Since there is only one conditional step the forward and inverse computations are similarly fast, and by stacking multiple layers complex distribution can still arise.
- **Neural Autoregressive Flow (NAF)**. NAF (Huang et al. 2018) is a generalization of the MAF. Instead of computing the parameters μ and β from $y_{1:i-1}$, the authors compute multiple pseudo parameters, which parametrize another neural network. This network then transforms y_i to z_i. The method only marginally increases the computational requirements but yields a richer family of possible distributions. Since it is not possible to invert the transforming network in the same way as the trivial shift and scale, it is not possible to sample from the resulting distribution.

Inverse flows

Often it is significantly slower to compute the inverse than the forward flow. In the case of the NAF, it is even impossible to so. Thus, workloads should ideally consist of mostly scoring given data samples and not sampling new data. However, it is always possible to apply the same flow in the inverse direction. Therefore, in settings where frequent sampling is required but scoring arbitrary samples is not, one can still use the methods described above. In fact, our application of normalizing flows in reinforcement learning will only require the sampling of new data.

Conditional flows

Conditional normalizing flows can be implemented in various ways depending on the dimension of the conditional variable. Trippe et al. 2018 train a neural network that takes a low dimensional conditional variable as input and outputs the parameters of the normalizing flows. To cope with high dimensional conditional variables Lu et al. 2020 propose to pass the conditional variable through an encoder network ψ and append the resulting output to the input of the flow network Ψ.

As we will explain in Section 2.2 in a reinforcement learning setting, the conditional variable will usually be the environment state of the agent. Since training robots with camera inputs is a potential use case, we implement the method proposed by Lu et al. 2020 although we are working with relatively low dimensional conditional variables in this thesis.

Finally, let $\mathcal{D} = \{(y^{(j)}, t^{(j)})\}_{j=1}^{M}$ be a dataset with labels $t^{(j)}$ then the density of one

sample in the MAF is computed as

$$\begin{aligned} p(y_i|y_{1:i-1}) &= \mathcal{N}(y_i|\mu_i, \exp(\beta_i)^2) \\ \mu_i &= \Psi_{\mu_i}(y_{1:i-1}, h) \\ \beta_i &= \Psi_{\beta_i}(y_{1:i-1}, h) \\ h &= \psi(t). \end{aligned} \tag{2.11}$$

2.1.2 Density Fitting Experiment

The goal of this experiment is to find out which normalizing flows are best suited for representing expressive policies in reinforcement learning for robotics. The action space of a robot usually has as many dimensions as the robot has joints. For example, with the four fingers of the DLR Hand II, we have an action space of 12 dimensions. Entire humanoid robots can have 50 joints or more (Bäuml, Hammer, et al. 2014, Tassa et al. 2012) but compared to the number of pixels in an image this is still a small number. Note that the dimension of the environment state (the observations) could still be high resolution images but teh distribution only models the actions. Therefore, we will focus our comparison on the three commonly used flows presented before without the additional modifications that were designed for high dimensional distributions (Kingma and Dhariwal 2018).

Dataset

We use 2000 samples of the well known two moons dataset (Figure 2.2). The dataset is two dimensional that has the advantage that we can easily visualize the distributions. The labels of the points are ignored in this experiment.

Figure 2.2: The two moons dataset.

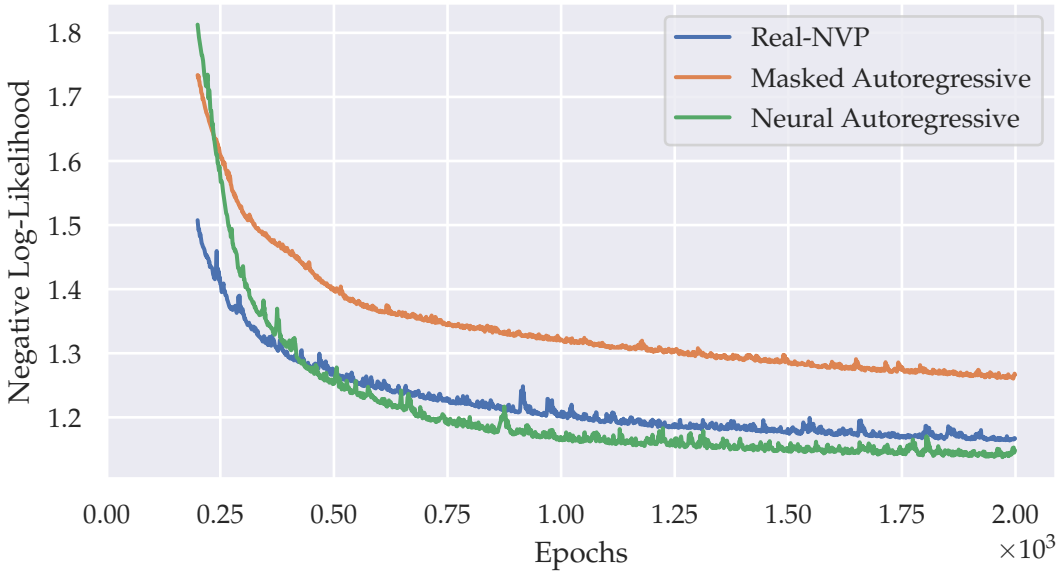


Figure 2.3: Training curves of the two moons density fitting experiment. The MAF and the Real-NVP flow show a similar trend, however, the Real-NVP flow has consistently lower negative log-likelihood. The NAF starts out with extremely high negative log-likelihood but performs best after around 500 episodes. Note we do not show the first 200 episodes because the high values would spoil the scale.

Setup

We estimate the probability density of the data points as described in (2.6) and maximize the likelihood via gradient descent. For this we use the Adam optimizer and split the training data in two batches. Preliminary experiments showed that increasing the number of layers and, therefore, the number of parameters improves the result significantly at first but yields diminishing returns above 1000 parameters. We also found out that batchnorm layers are not necessary and sometimes lead to high variance in the training loss. Therefore, we report only the comparison of the three flows with fixed hyperparameters. We use eight layers without batchnorm and a two-layer MLP with 10 neurons each for the network Ψ.

Results

Figure 2.3 shows that the Real-NVP flow and the NAF perform similarly after a few hundred episodes. However, the MAF seems to learn slower and has 0.1 higher negative log-likelihood than the Real-NVP flow. Since the Real-NVP flow performs best with very little training and is the simplest model of the three, we decided to work with this

(a) Learned (b) Original

Figure 2.4: The final distribution of the Real-NVP flow compared to the original samples.

model in the remaining experiments of this thesis.

In the comparison in Figure 2.4, one can clearly see that the model finds a good fit for the data. Particularly interesting is the thin tail connecting the two moons in the learned distribution. This connection shows that it is difficult for the model to split the continuous probability mass of the base distribution into separate regions. However, stretching and wrapping the density layer by layer works impressively well (cf. the trained Real-NVP flow in Figure 2.1). Since the Real-NVP flow can efficiently be inverted, we can also visualize the distribution in a density plot (Figure 2.5). Generating such a plot requires scoring a sample for each pixel, but the result gives a much more complete picture of the distribution than only sampling from it.

Implementation

This experiment is based on the implementation of Jang 2018. We use python as the language and the following packages for utilities: `numpy`, `scikit-learn`, `matplotlib`, and `tqdm`. However, instead of `tensorflow` we rely on `PyTorch` as our deep learning framework. In `PyTorch` there exist *Transforms* which can be applied to *Distributions* but normalizing flows are not yet available. The `pyro` package for deep universal probabilistic programming builds on `PyTorch` and has a class *TransformModule* which inherits from both *torch.distributions.transforms.Transform* and *torch.nn.Module*. This class is ideal to implement normalizing flows and many are available already (*List of Python Packages* 2020).

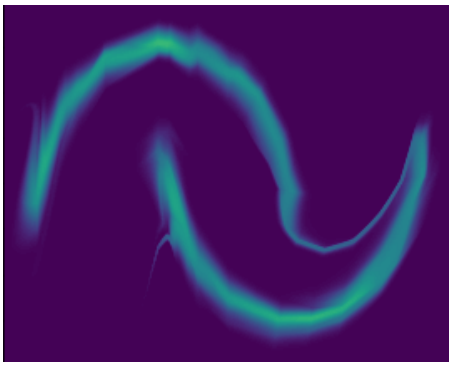


Figure 2.5: The final distribution of the Real-NVP flow visualized in a density plot. Brighter colors indicate higher probability. Note, we can use the same model to generate samples (Figure 2.4) and score arbitrary points (this plot).

2.1.3 Conditional Density Fitting Experiment

In this section, we report on two experiments. Firstly, we want to test our implementation of the conditional normalizing flow before applying them to a complex reinforcement learning algorithm. In the second experiment, we want to test how well the method generalizes to conditional variables that are not in the training data or even far off the training data.

Datasets

For the first part of this experiment, we reuse the two moons dataset and simply pass the label of each point as the conditional variable. Note that the label indicates which of the two moons the point was sampled from. In the second part, we generate semi-circles (moons) around the origin depending on a continuous variable θ, i.e., the angle which the semi-circles are pointing. To test the generalization performance, we only train with points generated at specific angles. In particular, we generate 227 samples for the angles 0.0 radian to 2.1 radian in 0.1 radian increments, totalling 4994 training points. We call this the angle-moon dataset (cf. Figure 2.8).

Setup

The setup for the first part of the experiment remains mostly the same as in the unconditional experiment. However, we only run the Real-NVP flow, and we condition it on the label of the dataset. The entire setup is visualized in Figure 2.6. In the second part we condition on the two dimensional vector of $\sin(\theta)$ and $\cos(\theta)$ to avoid discontinuities after 2π radian. In both cases, the encoder network is a single linear layer with 16 neurons.

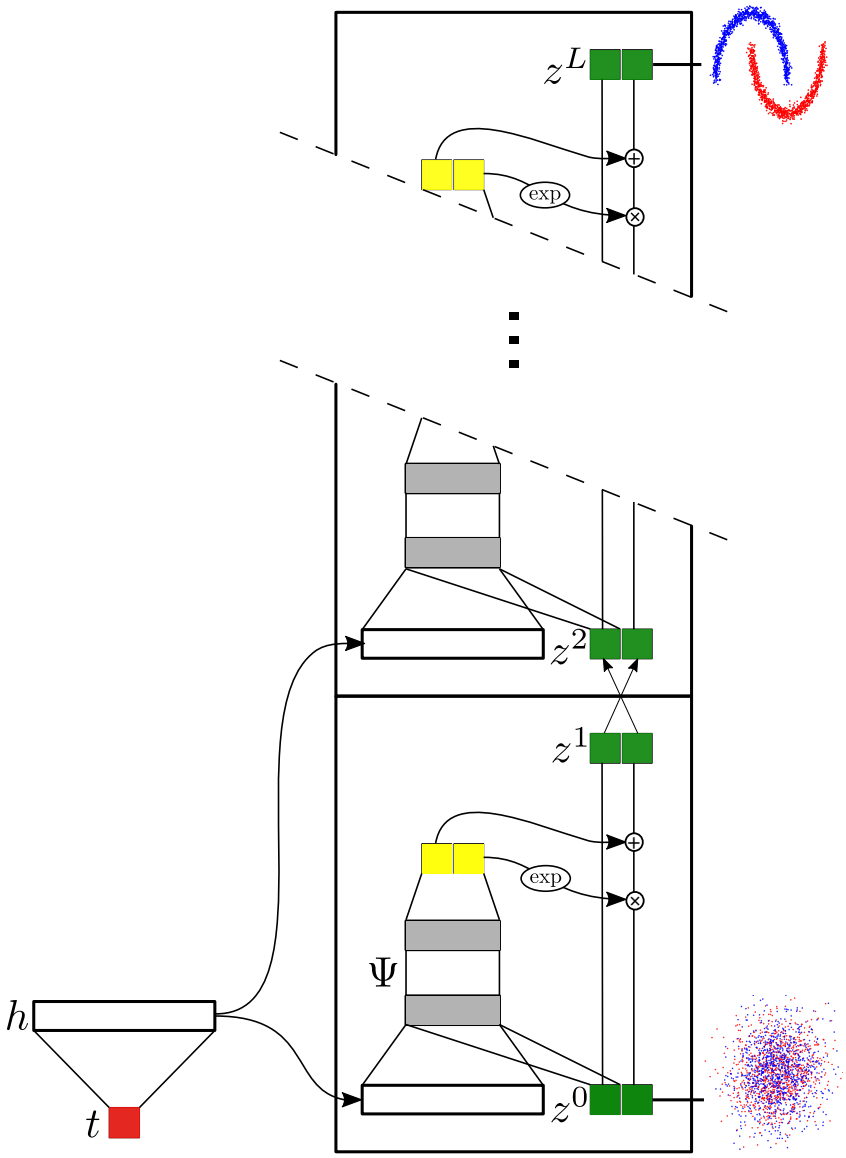
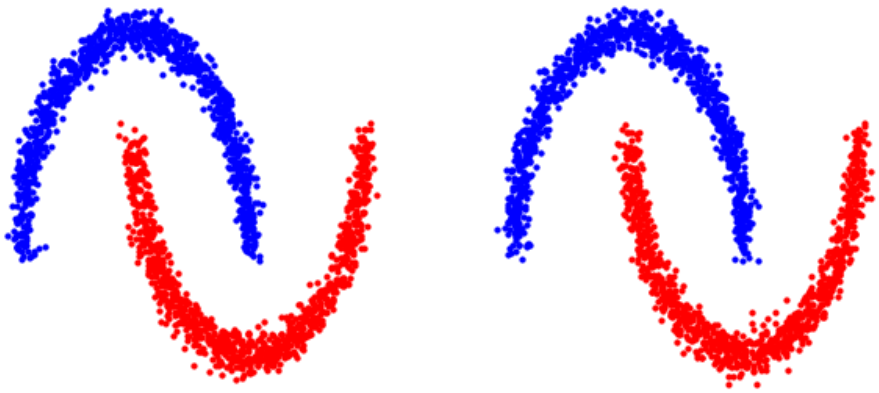


Figure 2.6: The conditional Real-NVP flow model used in the conditional density fitting experiment. On the left, we show the conditional variable t in red and the encoded representation h of it in white. In the center, we depict one of the eight flow layers. Each layer passes the vector h and the first dimension of the sample z through the neural network Ψ (grey), which outputs shift and scale (yellow) for the second dimension. Between flow layers, we permute the coordinates of z. On the right, we exemplarily indicate the initial standard normal distribution and the final two moons distribution.



(a) Learned (b) Original

Figure 2.7: The final distribution of the conditional Real-NVP flow compared to the original two moons dataset.

Condition on label

The final distribution learned in the first part of the experiment looks extremely similar to the original one (Figure 2.7). During training, the negative log-likelihood drops down to 0.5, which less than half of the best performing unconditional flow (compare Figure 2.3). Moreover, the conditional flow converges already after 250 episodes.

The boost in performance shows that it is easier for neural networks to learn separate transformations depending on the label than one transformation that needs to separate the points. Note that here the same neural network switches between two completely different transformations without a problem. On the other hand, the biggest challenge for the unconditional flow seems to be separating the initial normal distribution into two groups (see the faint connection between the moons in Figure 2.4 and Figure 2.5). This challenge, of course, is removed entirely in the conditional flow setting.

Note that the model can easily classify new samples. We can score their probability under both conditions and pick the maximum. Although the model is fairly small, it was easily able to learn the two moons distribution, which shows how powerful even simple normalizing flows are.

Condition on angle

The original and the learned distribution on the angle-moon dataset look very similar (Figure 2.8). Only at the tips of the open circle, the curve of the learned distribution seems to flatten out slightly. This result shows that the neural network can also learn many more separate distributions than just two.

However, far more interesting is the generalization of the model to new angles, which

(a) Learned (b) Original

Figure 2.8: The final distribution of the conditional Real-NVP flow compared to the original angle-moon dataset.

are not in the training data. In Figure 2.9, we show density plots of eight angles evenly spaced between three training samples, i.e., in the range 1.0 radian to 1.2 radian. Although it is difficult to see in the figure, the orientation of the circle changes between every two plots. This can be seen clearly when skipping through the individual plots on a computer screen. Thus, the model does not simply memorize the 21 distributions for the training angles and return the one distribution that is closest to the requested angle. The model does properly generalize to new angles it has never seen before.

In Figure 2.10 we show density plots of eight angles evenly spaced between 0 radian and 2π radian. The first three plots are generated with angles that are within the range of training angles (0.0 radian to 2.1 radian), and all of them show a clear semi-circle structure. The fourth plot was generated with an angle of 2.36 radian. We can already see that the tips of the circle are not as sharp as in the previous plots. The fifth plot generated at 3.14 radian shows that model does not properly generalize further off the training data. Instead of pointing in the opposite direction of the first plot, the semi-circle points in the same direction as the previous one. The sixth and seventh plots show complete failure far away from the training data. The eighth plot generated at 5.50 radian is close enough to 0 radian such that we can vaguely see a semi-circle again.

To the best of our knowledge, there are no published results on this kind of generalization of conditional normalizing flows. Although the generalization far away from the training data can only roughly reproduce the principled shape of the distribution, it generalizes well for values closer to the training data. For the intended use of conditional normalizing flows in this thesis for reinforcement learning tasks, the generalization between training samples is of utmost importance and works perfectly fine.

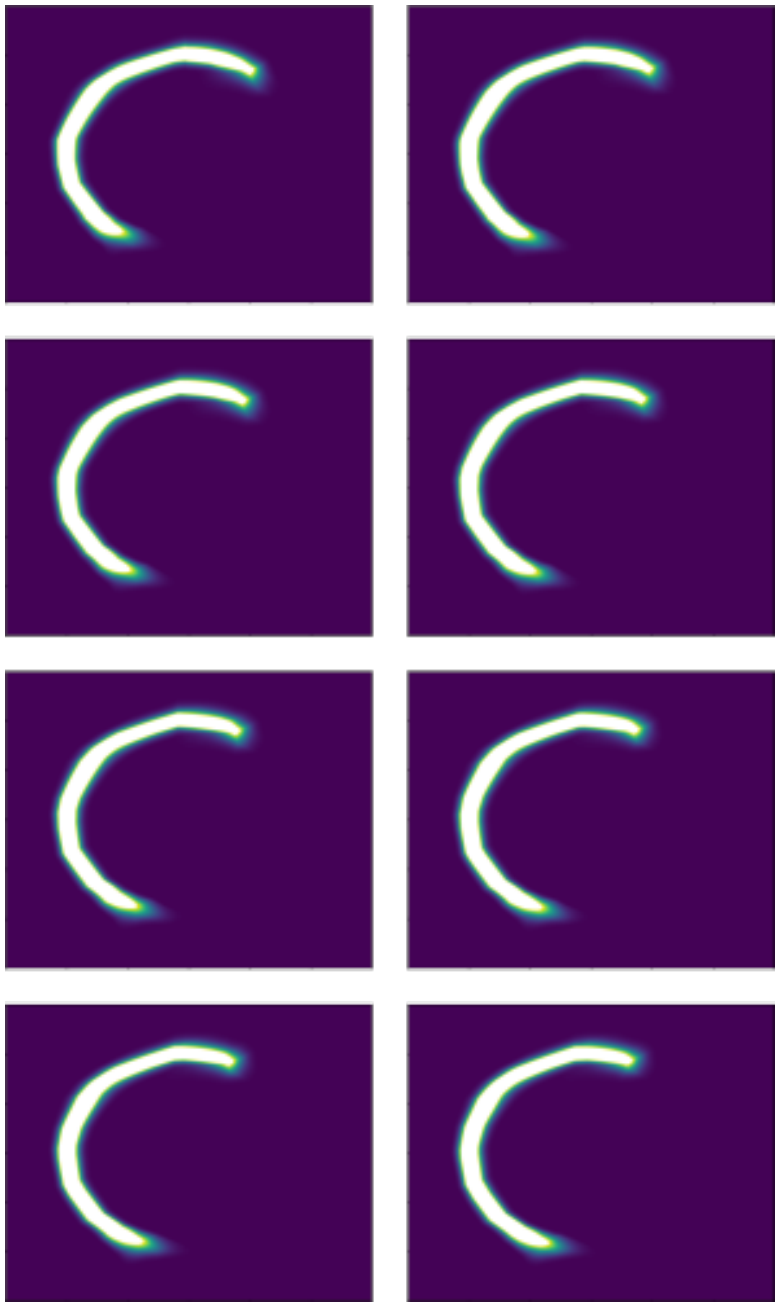


Figure 2.9: Visualization[a] of the generalization between training angles. Brighter colors indicate higher probability. The plots are generated at angles 1.00, 1.03, 1.06, 1.09, 1.11, 1.14, 1.17, 1.20 (in radian) from left to right, top to bottom..

[a] Link to sequence with individual images.
`https://drive.google.com/drive/folders/1TP2xdxGn68etXUN5rlFmac0q-50b8URs?usp=sharing`

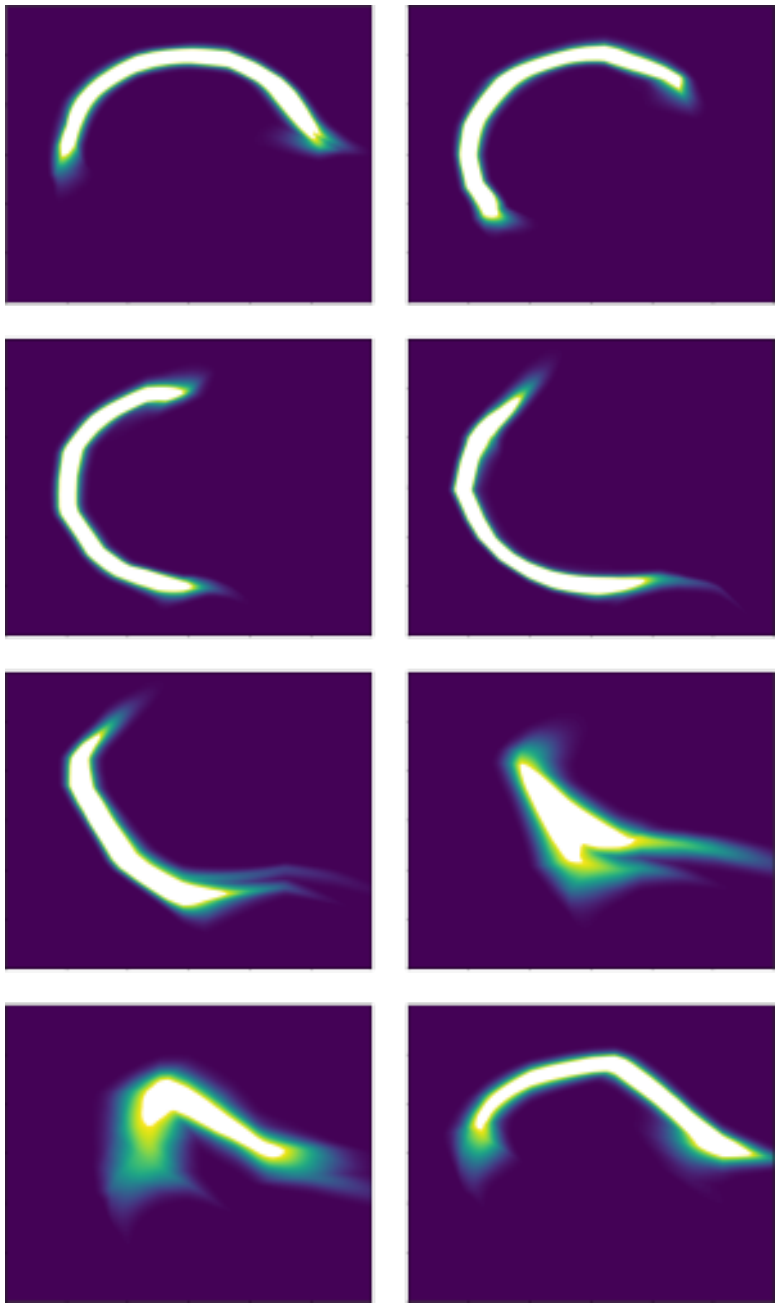


Figure 2.10: Visualization of the generalization far away from training angles (0.0 - 2.1 radian). Brighter colors indicate higher probability. The plots are generated at angles 0.00, 0.79, 1.57, 2.36, 3.14, 3.93, 4.71, 5.50 (in radian) from left to right, top to bottom.

Implementation

This experiment builds onto the previous experiment. We use the conditional normalizing flows implemented in `pyro`. The datasets are created similar to the two moons dataset in `scikit-learn` (*List of Python Packages* 2020).

2.2 Extending the Soft Actor-Critic Algorithm

In this section, we describe the reinforcement learning algorithm and its variants, which we designed to compare in Chapter 3. Moreover, we explain why we choose different degrees of expressiveness for the policy distributions.

2.2.1 Standard Soft Actor-Critic

Even though SAC was derived as a simplification of Soft-Q learning (Haarnoja, Tang, et al. 2017), the algorithm is extremely similar to the previously known DDPG (Lillicrap et al. 2016) and its variant, TD3 (Fujimoto et al. 2018). For a detailed description and derivations of the algorithm, we recommend reading Haarnoja, Zhou, et al. 2018. Here we will discuss the key features of the algorithm and our implementation in particular. For completeness, Algorithm 1 shows the entire algorithm including the relevant equations. Later we will often refer to this original SAC algorithm as *Gauss-Agent*.

- **Maximum Entropy Framework.** Instead of the standard expected discounted sum of rewards, $\mathbb{E}_{\tau\sim\pi}[\sum_t \gamma^t r_t]$, the SAC algorithm optimizes an objective function which additionally rewards high entropy policies: $\mathbb{E}_{\tau\sim\pi}[\sum_t \gamma^t (\alpha\mathcal{H}(\pi(\cdot|s_t)) + r_t)]$, where the entropy coefficient α is a hyperparameter which determines the importance of high entropy compared to the task reward.

- **Policy.** The policy is the main difference between SAC and DDPG/TD3. Instead of training only the mean and adding explorations noise afterwards, SAC uses the reparametrization trick to train a diagonal normal distribution as the policy. Let D be the action dimension and let $\mathbb{I}_D$ be the identity matrix of dimension D. A neural network Ψ takes in the environment state and outputs an intermediate feature vector $h \in \mathbb{R}^H$. This feature vector is then the input to two linear layers. One computes the mean ($\mathbf{A}_\mu \in \mathbb{R}^{D\times H}$), and the other the log variance ($\mathbf{A}_\beta \in \mathbb{R}^{D\times H}$). The samples of the resulting normal distribution are then squashed with an element-wise tanh operation to be in the range $[-1, 1]$[1].

$$\begin{aligned} \pi(a_t|s_t) &= \tanh\Big(\mathcal{N}(\mu_t, \mathbf{\Sigma}_t)\Big) \\ \mu_t &= \mathbf{A}_\mu h_t \\ \mathbf{\Sigma}_t &= \mathbb{I}_D \exp(\beta_t) \\ \beta_t &= \mathbf{A}_\beta h_t \\ h_t &= \Psi(s_t) \end{aligned} \tag{2.12}$$

[1]The samples of the unbounded normal distribution get squashed because the actions need to be within a finite interval to apply them to joint angles or motor torques.

- **Entropy Estimation.** The algorithm uses Mote Carlo sampling to estimate the entropy $\mathcal{H}(P) = \mathbb{E}_{x\sim P}[-\log P(x)]$. Therefore, a complex distribution without analytical form for its entropy can be used as a policy.
- **Q-function.** The Q-function for the objective of the SAC algorithm is $Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi}[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^t\mathcal{H}\left(\pi(\cdot|s_t)\right)\big|\, s_0 = s, a_0 = a]$. It is represented by a neural network that takes in a state-action-pair and predicts a single scalar value. Similar to TD3, SAC trains two separate Q-function of which we use the minium as the final Q-value (to prevent overestimating the value).
- **Target Network and Polyak averaging.** Just as DDPG/TD3, SAC uses target networks for the Q-functions that are updated by Polyak averaging.
- **Temperature learning.** In Haarnoja, Zhou, et al. 2018, the authors propose to learn the entropy coefficient α, which they call the temperature parameter. It is important to note, however, that learning the entropy coefficient, requires fixing an entropy target of the policy in advance. Therefore, the entropy is not maximized anymore despite the name "maximum entropy framework". It should rather be thought of as a constraint optimization problem. We will experiment both with fixed α (max entropy) and with learned α (fixed entropy).
- **Weight decay.** In reinforcement learning weight decay is not as widely used as in supervised learning. As such SAC is usually implemented entirely without explicit regularization on the networks. We found that especially in our custom environments weight decay often stabilizes the learning progress and prevents catastrophic forgetting. Therefore, we add a small decay term to all weights (directly in the optimizer without having to modify the loss equations).

Implementation

We use the `PyTorch` variant of OpenAi's `spinningup` repository as our base code. We added temperature learning and basic parallelization via `mpi4py` and OpenAi's *VectorEnv* to the SAC implementation. Additionally, we want to point out that the latex code of Algorithm 1 is taken from the `spinningup` repository with modifications regarding the temperature learning (*List of Python Packages* 2020).

2.2.2 Low-Rank Decomposition Policy

The simplest possible changes to the policy to increase the expressiveness are training the full covariance matrix, training a low-rank decomposition of the covariance matrix, or training a Gaussian mixture model (GMM).

Algorithm 2 (Source: Achiam 2018)

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$, (initial) temperature α, OPTIONALLY entropy target $\mathcal{H}^{\text{target}}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$.
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$.
  Execute a in the environment.
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal.
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$.
  If s' is terminal, reset environment state.
  if update_iteration **then**
    for j in range(update_many) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$.
      Compute targets for the Q-functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s').$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2.$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right) \Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt. θ via the reparametrization trick.
      OPTIONALLY Update temperature by one step of gradient decent using

$$\nabla_\alpha \frac{1}{|B|} \sum_{(s,a) \in B} \Big(- \alpha \log \pi_\theta\left(a|s\right) - \mathcal{H}^{\text{target}} \Big).$$

      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2.$$

    end for
  end if
until convergence

GMM. In fact, the authors of the original SAC paper first used a GMM policy but later realized that it did not improve their results (Openreview 2018). We believe that one major reason for the missing improvements is the simplicity of standard benchmark tasks. As we will show in Chapter 3, we can inject certain difficulties into a task that pronounce the advantage of more expressive policies, but most simple benchmark tasks do not exhibit such problems. Moreover, in most cases, we expect that a deterministic policy can solve the problem and the inherent multimodality of GMMs could potentially hinder the agent from converging to one. If, on the other hand, the means of the components converge quickly then most of the additional expressiveness is lost. We even conjecture that the variances of the components would also converge because they receive similar gradient signals, resulting in a simple diagonal normal distribution. Therefore, we decided to focus on other methods and not to implement a GMM policy.

Covariance Matrix. On robot control tasks such as in-hand manipulation, we want to allow the agent to explore on a submanifold of the action space. If it is possible to linearize this submanifold at individual environment states, then a multivariate normal distribution with full covariance matrix should be sufficient to model the required exploration behavior. Therefore, we implemented such a policy distribution. Since a covariance matrix $\mathbf{\Sigma}$ needs to be positive definite it is sufficient to learn a (lower) triangular matrix $\mathbf{L}$ such that $\mathbf{\Sigma} = \mathbf{L}\mathbf{L}^T$. Note that this matrix still scales quadratically with the action dimension. Because the action dimension is usually low, we implemented this setup. However, we consistently had numerical problems despite clipping the off-diagonal values and learning the log of the diagonal as usual.

Low-Rank Decomposition. Because we explicitly want to model the linearization of a *sub*-manifold and not the whole space, a low-rank decomposition of the covariance matrix should also be sufficient to model the intended exploration behavior. Therefore, we also implemented a policy based on this method. `PyTorch` has a *LowRankMultivariateNormal* distribution which suggests learning a covariance diagonal d and a covariance factor $\mathbf{V} \in \mathbb{R}^{D\times K}$ where D is the dimension of the action space and K the rank of the decomposition, such that $\mathbf{\Sigma} = \mathbf{V}\mathbf{V}^T + \mathbb{I}_D d$. After clipping the entries of $\mathbf{V}$ to enforce that $\mathbf{V}\mathbf{V}^T$ has no entries large enough to cause numerical problems due to ignoring the value of the diagonal d this method runs without numerical issues.

The final policy that we use for our *LR-Agent* applies the same neural network and final linear layers as the *Gauss-Agent*, but additionally has one more linear layer $\mathbf{A}_\mathbf{V} \in \mathbb{R}^{(DK\times H)}$ which takes the intermediate features and, after reshaping (denoted by $\text{vec}^{-1}(\cdot)$, the inverse of the vectorization), outputs the covariance factor $\mathbf{V} \in \mathbb{R}^{D\times K}$. We

use $K = 2$ for all experiments.

$$\begin{aligned} \pi(a_t|s_t) &= \tanh\Big(\mathcal{N}(\mu_t, \mathbf{\Sigma}_t)\Big) \\ \mu_t &= \mathbf{A}_\mu h_t \\ \mathbf{\Sigma}_t &= \mathbf{V}_t \mathbf{V}_t^T + \mathbb{I}_D \exp(\beta_t) \\ \beta_t &= \mathbf{A}_\beta h_t \\ \mathbf{V}_t &= \mathrm{vec}^{-1}_{D\times K}(\mathbf{A}_\mathbf{V} h_t) \\ h_t &= \Psi(s_t) \end{aligned} \tag{2.13}$$

2.2.3 Normalizing Flow Policy

The most expressive policy we investigate in this thesis uses normalizing flows to represent complex distributions. The setup of our *NF-Agent* is very similar to the conditional density fitting experiment in Section 2.1. The base distribution is a D dimensional standard normal distribution. We use $L = 4$ layers of the Real-NVP flow and condition on the intermediate feature vector h, which is generated by the same neural network Ψ as in the *Gauss-Agent*. The flow layers consist of two-layer MLPs with 10 neurons each, just like in the previous section (cf. Figure 2.6). We denote the one layer of the conditional normalizing flow as the function Γ. The output of the final flow layer gets squashed with a tanh transformation as usual in the SAC algorithm.

$$\begin{aligned} \pi(a_t|s_t) &= \tanh\left(z_t^L\right) \\ z_t^{i+1} &= \Gamma(z_t^i, h_t) \\ z_t^0 &= \mathcal{N}(0, \mathbb{I}_D) \\ h_t &= \Psi(s_t) \end{aligned} \tag{2.14}$$

One peculiarity that arises from normalizing flow policies is the test behavior of the agent. Usually, at test time, researches turn off the exploration noise such that the policy is deterministic. If the policy is a normal distribution this results in executing the mean action. With a normalizing flow policy choosing the mean of the final policy distribution is not possible without sampling hundreds of actions. Therefore, the most reasonable alternative seems to be passing the mean of the base distribution into the flow transformation. Arguably this strategy is more sensible than picking the mean of a complex distribution because it could have zero probability density at its empirical mean (for example, a half-moon distribution). However, we have no guarantee that a large portion of the probability mass is close by the mean (as it is the case with a normal distribution). Thus, the performance of the deterministic policy of the *NF-Agent* could be worse than the stochastic one in some cases.

3 Experimental Validation and Analysis

In this chapter, we conduct innovative experiments on environments with various exploration difficulties. First we validate our algorithms. Then, we present two environments which we designed to show that under certain conditions, expressive policy distributions are necessary to solve the given task.

3.1 Validation on Standard Benchmark Environments

The main experiments of this thesis are conducted in two custom environments, which we introduce in Section 3.2 and Section 3.3 respectively. Therefore, we first validate our algorithms and their implementation on standard benchmark environments for continuous control reinforcement learning. We opted for `PyBullet` environments over `MuJoCo` because they are open source and do not require an expensive license (*List of Python Packages* 2020).

Hyperparameters

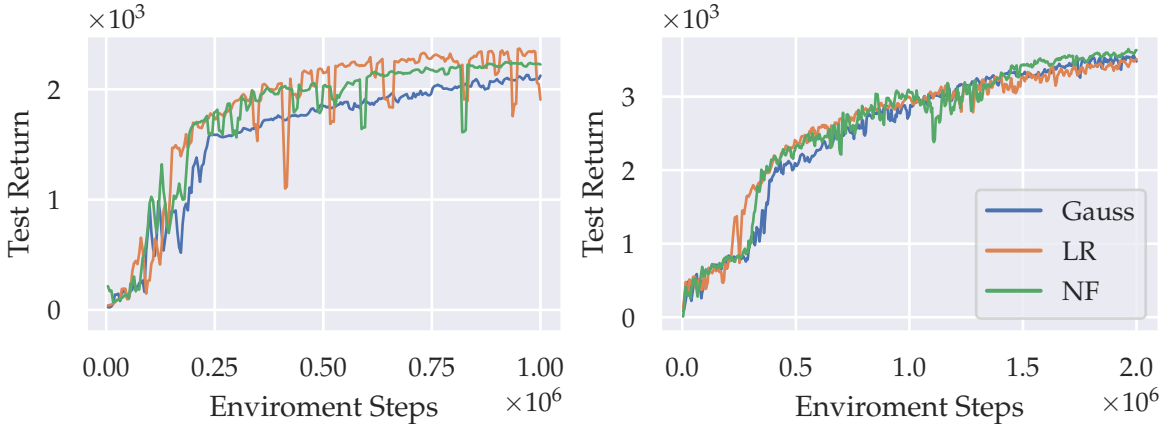
We use the hyperparameters reported in Table 3.1 for all the algorithms on both `PyBullet` environments. We selected them based on the values in Haarnoja, Zhou, et al. 2018 except for our addition of addition of the weight decay term. We did not attempt to tune them. The hyperparameters introduced by our modifications are set as described in Section 2.2.

Results

Figure 3.1 shows the results for the two environments *Walker2DBulletEnv-v0* and the higher dimensional *AntBulletEnv-v0*. We did only a single run and the results are smoothed via a moving window average filter. Of course a single run does not allow for in detail comparison of the three agents but the results clearly show solid learning progress. Moreover, the final performance of all three agents is in both environments on par with the tuned results reported in Raffin 2018 (3485 on the Ant environment and 2053 on Walker2D). From this we can conclude that the SAC algorithm, including temperature learning, is correctly implemented and that our modifications to the policy distribution do not prevent the agent from solving standard continuous control benchmark tasks.

Parameter	Value
network units	(256, 256)
non-linearity σ	ReLU
discount γ	0.98
polyak ρ	0.99
optimizer	Adam
learning rate	3e-4
entropy coefficient α	auto
entropy target $\mathcal{H}^{\text{target}}$	$-\dim(\mathcal{A})$
batch size	256
update every	256
update many	256
replay buffer size	1e6
update after	2000
initial random steps	1e4
weight decay	2.5e-4

Table 3.1: SAC Hyperparameters



(a) Walker2D (b) Ant

Figure 3.1: Training curves of the validation experiment. We show a single smoothed (window average filter) run of each agent on two `PyBullet` environments. All three agents solve the task and have a similar performance during the whole learning process.

3.2 Analysis of Basic Exploration Properties

In this section we first introduce a simple navigation environment (Section 3.2.1) and then analyse three experiments conducted on this environment (Section 3.2.2 - 3.2.4). Each experiment is designed to expose specific differences between the agents with more expressive policies and the basic *Gauss-Agent*. Additionally, we present insights into the learning dynamics of the SAC variants which emerge clearly in this simple environment.

3.2.1 The *Tunnel* - A Simple Navigation Task

Motivation

The goal of the *Tunnel* environment is to be a minimal environment that can be configured to require different degrees of coordination between action dimensions.

Setup

- There is a tunnel of adjustable width connecting the possibly randomized starting position with the goal region. The agent must stay within this tunnel at all times. Otherwise, the episode is reset. Usually, the tunnel width w is multiple times smaller than the agent's reach r, such that it requires coordination between the action dimensions to stay within the tunnel (otherwise, the agent can only make very small steps).
- In this environment, we do not simulate dynamics. At each time step, the agent chooses a position within its reach and is then deterministically moved to that position before the next step.
- In all experiments reported in this thesis there exists a fixed goal region. The agent can be incentivized to reach this region by a sparse reward for arriving, a dense reward for approaching, or a combination of both.

The simplest form of the environment allows specifying the *total dimension* count and the *coupled dimension* count. The tunnel is then a straight line along the *coupled dimensions* (see Figure 3.2 for an example). Additionally, we have the option to specify *free dimensions*, which are not constrained to the tunnel. In principle, we could work with arbitrarily large dimensions. However, to allow for good visualization, we use only two dimensional examples. To increase the complexity despite this choice, we also implemented the option to use the circumference of a circle or a two-dimensional linear spline as a tunnel (Figure 3.3).

Details

The action space $\mathcal{A} \in [-1, 1]^D$, where D is the number of *total dimensions*, is normalized within the reach. Therefore, the agent's movement in each step is computed by multiplying the action with the reach.

The state space $\mathcal{S} \in [-1, 1]^D$ does not need to be normalized because we choose the world coordinates to be within -1 and 1. However, when using the circle path, we add the radius of the circle and the norm of the agent's current position to the state.

Implementation

The environment is implemented entirely in python and `numpy`. It inherits OpenAI's ubiquitous `gym` interface. We use the windowing library `pyglet` (based on OpenGL) for visualization. And we use `shapely` and `scipy` to compute the distance from and generate complex linear splines (*List of Python Packages* 2020).

3.2.2 Fixed Entropy Coefficient Leads to Collapsing Entropy

Motivation

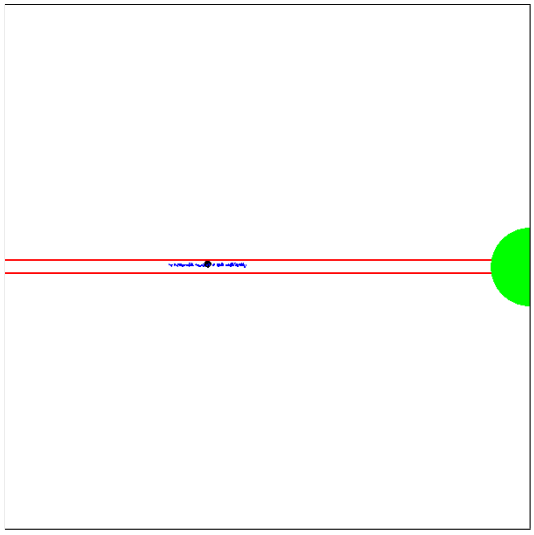
In this experiment, we want to gain an understanding of how the original SAC algorithm, without temperature learning, learns in a simple environment with dense rewards. Moreover, we construct the environment such that the *Gauss-Agent* will have to decrease its entropy much further than the other agents. That should give insights into how more expressive policy distributions can lead to better performance in complex environments.

Setup

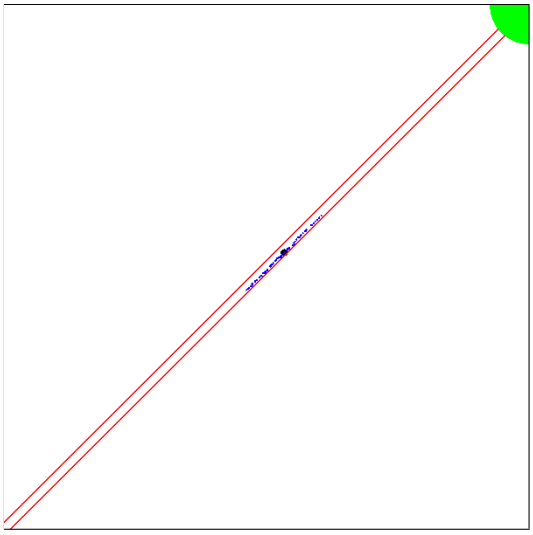
We run this experiment on the *Tunnel* environment using the circle path. The agent starts two thirds around the circle away from the goal region and receives a reward of 0.1 for reaching it. Along the way, we provide a dense reward for reducing the distance to the goal region. There is no penalty for leaving the path, but the episode is reset immediately. The agent's reach is 4 times the path width ($r = 4w$), and it takes at least 6 steps to reach the goal from the starting position. In Table 3.2, we provide the learning algorithms' hyperparameters.

Insights

To gain a deeper understanding of the algorithm, we visualize policies during the training process. In this simplified setting and, in particular, due to the dense reward, we can see how the agent learns new steps one at a time. Figure 3.4 shows an agent that already learned the first few steps. At the next step, its best guess is to apply

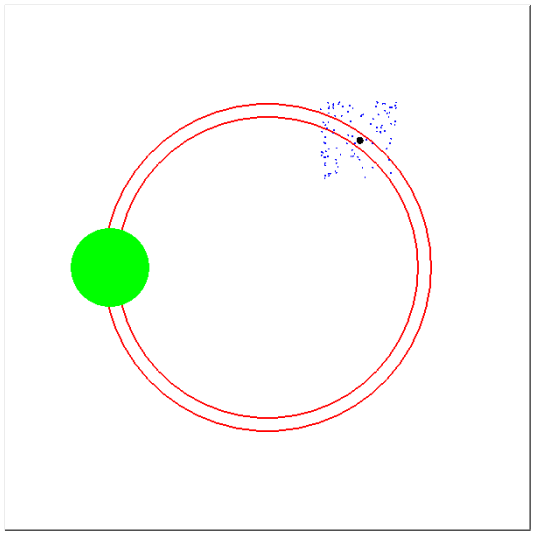


(a) *Coupled dimension* = 1

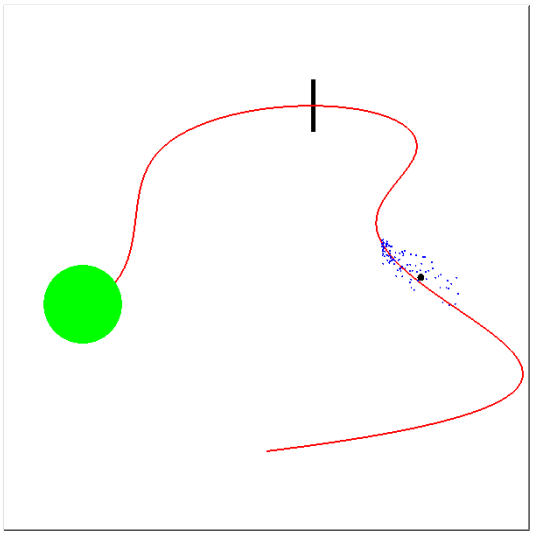


(b) *Coupled dimension* = 2

Figure 3.2: Simple *Tunnel* environment. The green area is the goal region. The red lines indicate the tunnel. The black circle represents the agent. The blue dots are action samples drawn from a trained policy.



(a) Circle



(b) Linear spline

Figure 3.3: The *Tunnel* environment with more complicated paths. Here we show not fully trained policies to clearly see the reach of the agent compared to the width of the tunnel. Note: In the linear spline case, we only plot the centerline of the tunnel and specify the constant width with the black bar.

the same distribution that worked for the previous step (presumably, caused by the generalization properties of the neural network). However, arguably more interesting is that with every training iteration without seeing additional environment reward (i.e., before an action on the path was sampled by chance), the entropy of the distribution at the last step increases due to the maximum entropy term in the objective. Once the entropy is large enough, it is only a matter of time until one action on the path is sampled. The agent sees that this action generates more return, and the networks learn to output that action with higher probability, decreasing the previously built up entropy. Once this step is burnt into the neural networks the next step can be learned. This simplified procedure already gives an intuition for the inner workings of the algorithm. Moreover, it shows how powerful the maximum entropy framework can be. For example, in an environment with an equivalent of a straight tunnel, a sudden change in direction could be extremely difficult to learn for ϵ-greedy algorithms such as DDPG. If ϵ is set too small and the network reliably outputs actions that are far away from the tunnel, the agent will never see the new direction and, therefore, have no learning signal that could help to find it.

Discussion

In Figure 3.5, we can see that all agents reach a similar average return eventually. However, the *Gauss-Agent* requires more steps, and its entropy drops deeper than the

Parameter	Value
network units	(128, 128)
non-linearity σ	ReLU
discount γ	0.8
polyak ρ	0.99
optimizer	Adam
learning rate	3e-4
entropy coefficient α	0.01
batch size	256
update every	1
update many	1
replay buffer size	1e4
update after	1000
initial random steps	2000
weight decay	2.5e-5

Table 3.2: Hyperparameters for experiments on the *Tunnel* environment.

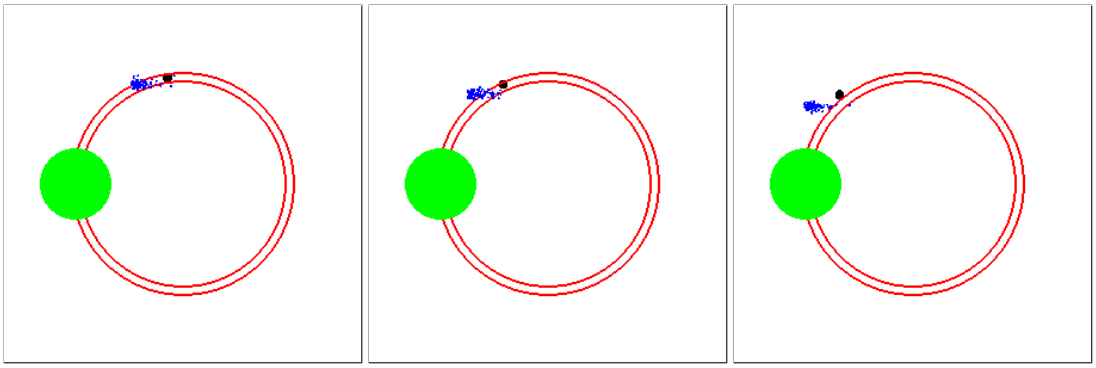


Figure 3.4: Visualization of the learning process of the *Gauss-Agent*. From left to right we see consecutive time steps. The action samples (blue) show that the agent has a high probability of success in the first image but in the second image most of the probability mass lies outside of the tunnel. In the last image we see that after stepping outside of the path the neural networks produce a similar distribution to the step before.

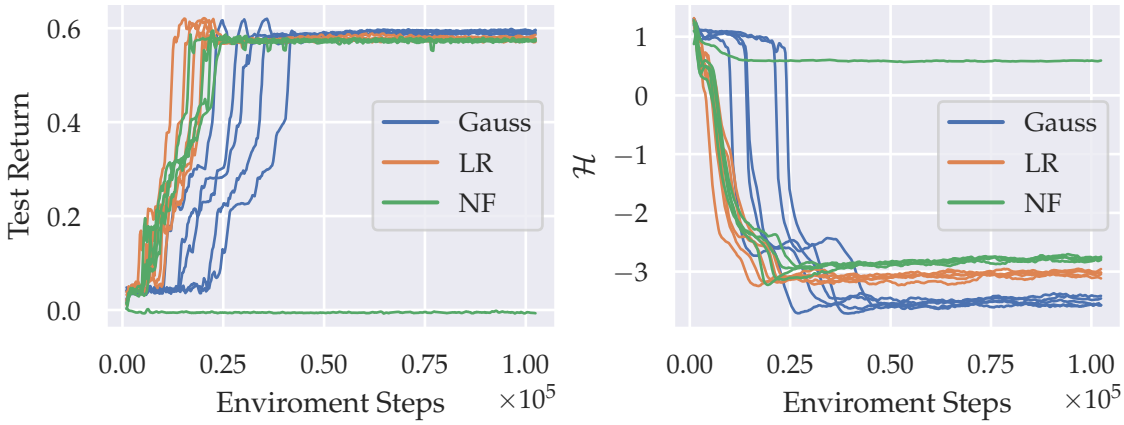
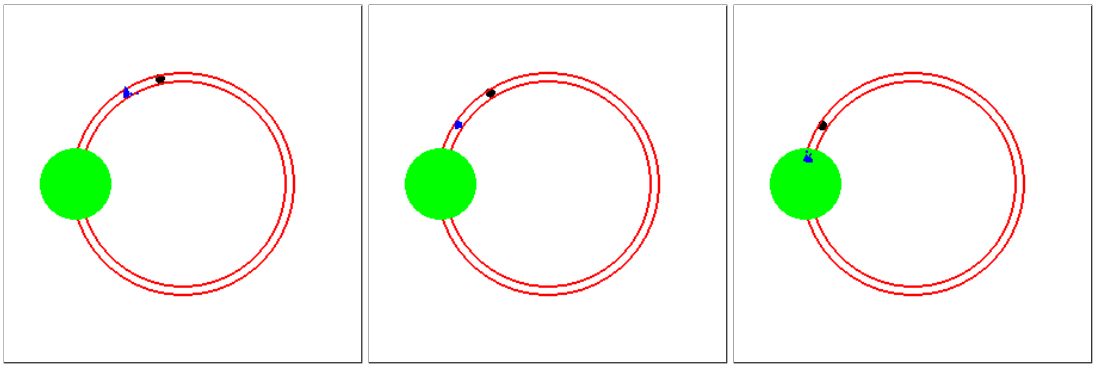
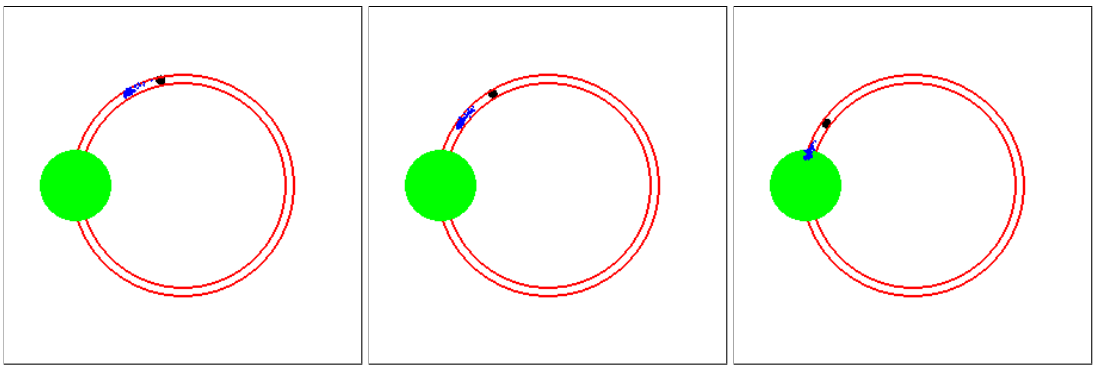


Figure 3.5: Training curves on the *Tunnel* environment with fixed entropy coefficient α. We show five smoothed individual runs of each agent. The *Gauss-Agent* requires more environment steps to solve the task and decreases its entropy much further than the other agents. Note one of the five *NF-Agent* runs did not learn anything useful but kept the entropy high instead.

other agents. The *NF-Agent* holds more entropy than the *LR-Agent*, and both increase their entropy slightly after solving the task. In contrast the *Gauss-Agent* remains at a constant entropy. Since the delta in entropy might seem small compared to that of the uniform distribution the agents are initialized with, we provide a visualization of the final policy distributions in Figure 3.6.



(a) Gauss Policy



(b) Normalizing Flow Policy

Figure 3.6: Visualization of trained policies. The *NF-Agent* can adapt its policy distribution to the curvature of the circle and increase the entropy while the *Gauss-Agent* cannot.

Now in this environment, there is no additional reward available. Therefore, maximizing the return by exploiting the current policy seems to be the optimal behavior. However, for example, a humanoid robot may be equipped with a reward such that the robot's priority is to stand upright (or rather, not fall to the ground). Once that task is solved, the agent may be able to discover new rewards through walking forward. As such, walking is a new layer or level in the environment that unlocks only after the prioritized task is solved. The agents cannot rule out the possibility of an additional level in our environment, and, in order to maximize the chance of finding more reward, it is essential to explore the boundaries of what is possible. Therefore, the preferred behavior is generally to increase the policy's entropy, which we may only be possible with an expressive policy distribution.

A draw back ist that the NF-Agent not always succeeds but gets stuck in a minimum with high entropy, i.e., is learning nothing reasonable. We see these outliers of the

NF-Agent often throughout our experiments. We suspect that the neural network structure is sensitive to initialization. Possibly we could achieve more consistent results by applying different initialization schemes. However, the small networks in the flow layers might require new methods altogether because modern initialization schemes are designed for large networks (cf. He et al. 2015). Therefore, we leave investigations into outliers and network initialization for future work and focus on the general trend of the agreeing runs (nevertheless we report all outliers in the figures).

3.2.3 Fixed Entropy Leads to Oscillating Policy

Motivation

More expressive distributions can learn higher entropy policies in the original SAC setting, and that might be a great advantage in complex environments. However, the most recent, and on benchmarks most successful, version of SAC employs temperature learning. Since temperature learning basically fixes the entropy over the entire learning process, one might conjecture that more expressive policy distributions are useless in this setting. In this experiment, we show that is not the case because too little expressiveness can, for example, lead to strong oscillations.

Setup

For this experiment we first use exactly the same environment setup and hyperparameters (cf. Table 3.2) as in Experiment 3.2.2 except that we learn the entropy coefficient α. As entropy target we use the standard $\mathcal{H}^{\text{target}} = -\dim(\mathcal{A})$. To amplify the effects seen in the first run, we increased the final reward from 0.1 to 1.0 and narrowed the path width from $1/4$ of the agents reach to $1/5$. The results are reported in Figure 3.7 and Figure 3.9 respectively.

Insights

In the runs with the original setup (cf. Figure 3.7) there is a clear difference in the performance of the agents in the training score, but their test performance is almost equivalent. Visualizing the policy, see Figure 3.8, allows us to understand what causes this gap. The *Gauss-Agent* increases the entropy at the first steps excessively, such that the agent sees these states disproportionally often and can compensate for low entropy distributions on the following steps. I.e., the agent artificially increases its policy's entropy at states where it could easily find better distributions to meet the entropy target. We expect that such a policy can hamper the learning progress in more complex environments due to collecting a lot of useless data of the start state and not exploring the rest of the world in detail.

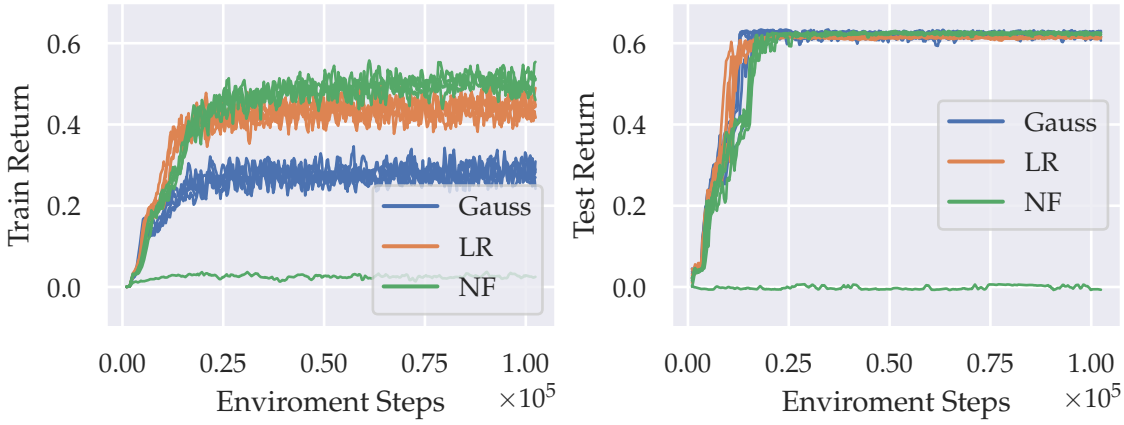


Figure 3.7: Training curves on the *Tunnel* environment with fixed entropy target $\mathcal{H}^{\text{target}}$. We show five smoothed individual runs of each agent. The *NF-Agent* performs around twice as good as the *Gauss-Agent* in terms of training performance but the test performance is almost identical. Note the outlier *NF-Agent* run stems from the same seed as in the previous experiment.

Discussion

Knowing about the "frontload-entropy-trick" we can also generate catastrophic behavior. We incentivize the agent to reach the goal region more often by increasing its relative importance, and we force the agent to collapse the entropy further by reducing the path width. The results can be seen in Figure 3.9. The *Gauss-Agent* struggles to find a policy that solves the task and contains the required entropy. The average return, the entropy, and the entropy coefficient α start oscillating together. If the agent finds the final reward, it collapses the entropy to reach it in every episode. In response to the

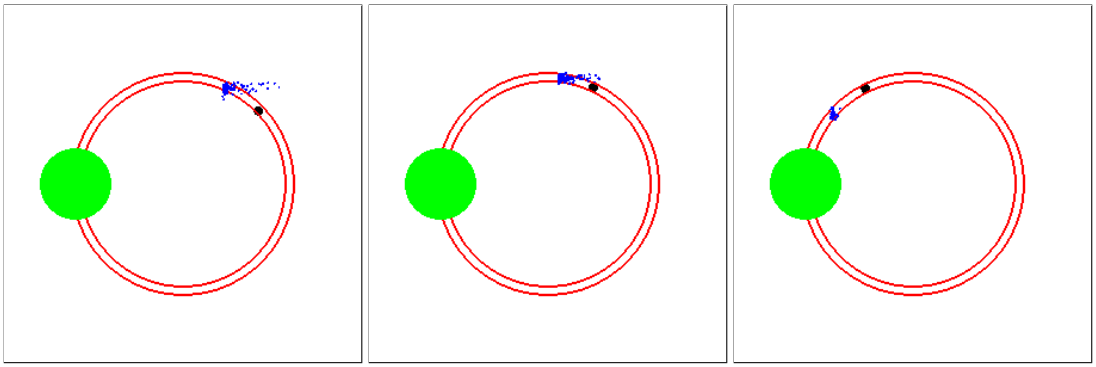


Figure 3.8: Visualization of the *Gauss-Agent's* policy which increases the entropy in the first few steps to compensate for lower entropy distributions later on.

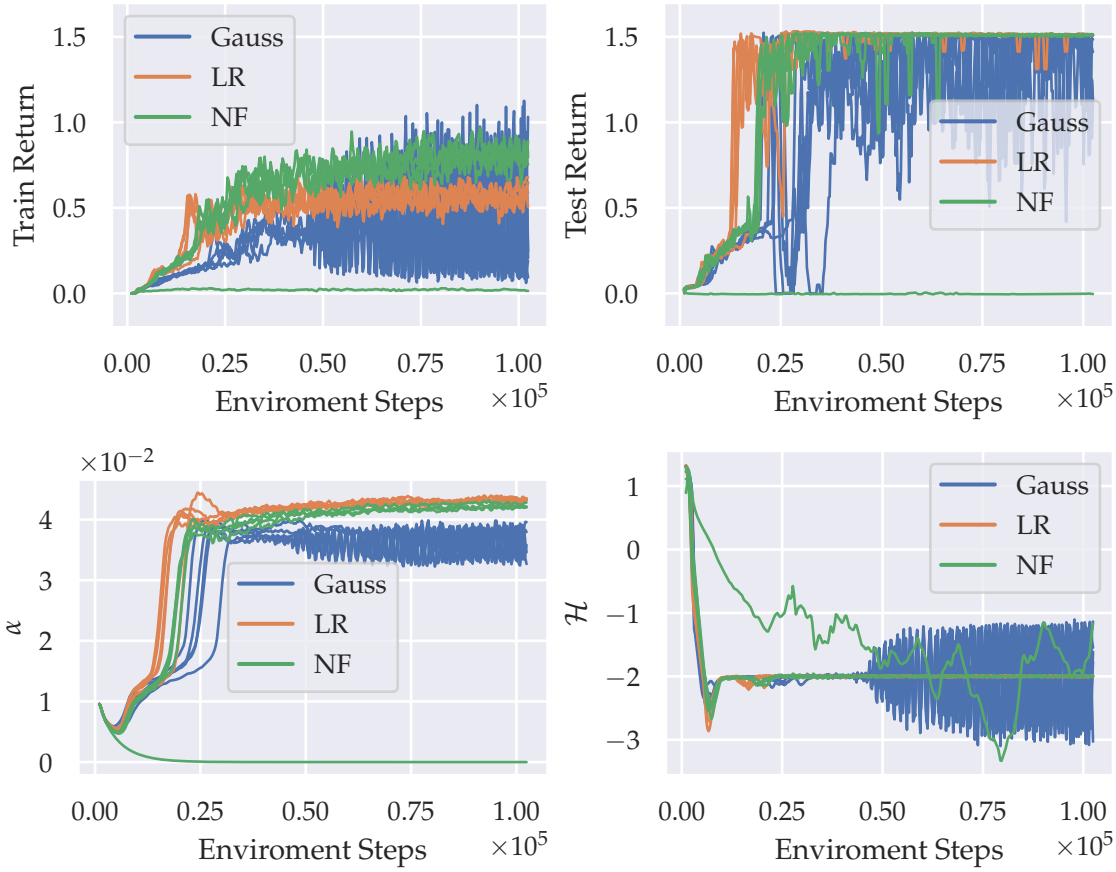


Figure 3.9: Training curves on the *Tunnel* environment with fixed entropy target $\mathcal{H}^{\text{target}}$ and modified path width/reward. We show five smoothed individual runs of each agent. The *Gauss-Agents* exhibit strong oscillations once they reach the goal region for the first time. Every single run exhibits these oscillations. Note the outlier *NF-Agent* run stems from the same seed as in the previous experiment.

dropping entropy, the coefficient α increases, until the agent favors adding entropy over reaching the goal. In this case, the oscillations lead not only to poor training scores but also to a loss of test performance.

Oscillations, as we see here, should be easily detectable, assuming sufficiently small inspection intervals. However, they will probably not occur in more complex environments because the policy would either forget everything it learned as described in the next paragraph. Or there will be a dimension in which the agent can increase the entropy without a huge drop in performance. The latter gives an intuition of how SAC with temperature learning can make learning progress as long as it finds

good sequences with the amount of entropy used (does not get stuck with an already sufficiently high entropy policy). Moving the entropy that was in a certain dimension somewhere else could be interpreted as focusing on a specific part of the problem.

In general, temperature learning introduces some added complexity that is not always desirable. Often it is difficult to understand the resulting effects and their interactions. In the original SAC setting the Q-Values are increasing almost monotonically during training, as it should be expected. However, with temperature learning, there can be sudden drops. In particular, before the networks find an equilibrium around the entropy target $\mathcal{H}^{\text{target}}$, the Q-Values always jump violently. Sometimes one can also observe a drop in entropy (caused by the agent finding out a useful sequence of actions) many hours into training that leads to a sudden spike of the entropy coefficient α, which in turn immediately rises the Q-Values. Once the policy adapts and α decreases, the Q-Values drop, which may result in catastrophic forgetting (cf. Wang et al. 2019).

We expect the difference between more and less expressive policies to be smaller with temperature learning than without in environments with dense reward, i.e., continuous control benchmarks. However, it is not always a good idea to use temperature learning. Often it works with much less tuning effort, but it potentially introduces the problems described above. And most importantly, it breaks the maximum entropy idea and can, therefore, easily get stuck in difficult to explore environments if the entropy target is too small (or not make any progress if the entropy target is too large). In particular, for environments where the exploration requires a high degree of correlation, it might still be indispensable to allow the policy to express these correlations.

3.2.4 Sparse Reward Leads to Complete Failure

Motivation

The first two experiments showed that there are potential advantages that come from more expressive policies, but so far, they did not have a substantial effect on the final test performance. In this experiment, we demonstrate the fundamental shortcoming of an insufficiently expressive policy in an environment with sparse rewards.

Setup

We again run this experiment on the *Tunnel* environment but choose the even simpler straight-line task to allow switching between correlated and uncorrelated exploration tasks. Here the agent spawns in the middle of the world. In the uncorrelated setting, the goal region is to the right of the agent, while in the correlated setting, the region is diagonally above/right of the agent. The path is a straight line connecting the starting position with the goal region and extending to the other side of the world (cf. Figure 3.2). The width is $1/8$ of the agents reach, and it takes 6 or 7 precise steps to reach the goal

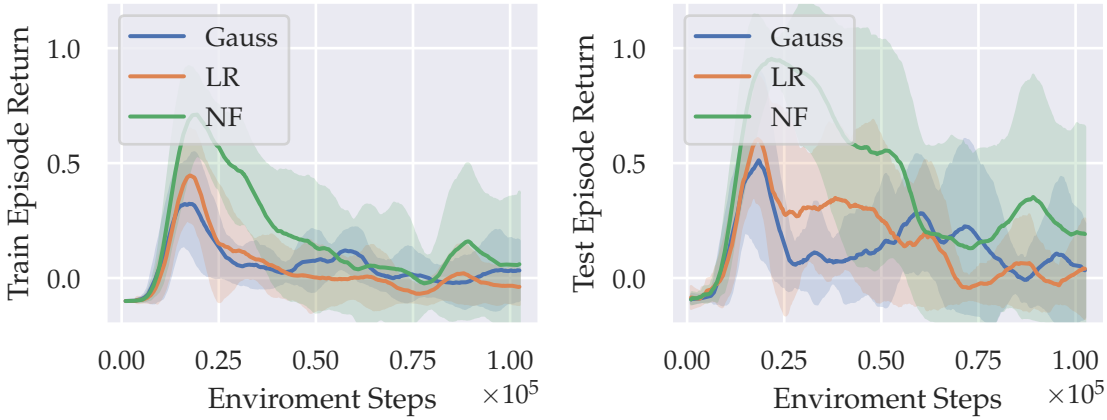
(but usually the agents use more). There is no reward for making progress, only a penalty of -0.1 for stepping outside of the path (which triggers a reset as usual) and a positive reward of 1.0 for reaching the goal. We stick to the hyperparameters reported in Table 3.2 except for the network sizes. We use only two layers of 16 units instead of 128. For the *NF-Agent*, we reduce the size of the feature extractor in the policy and the flow network to two layers of 4 units to remain at a comparable number of parameters as the other agents. Moreover, the entropy coefficient, also known as reward scaling, needs to change with the new reward function to $\alpha = 2.5\text{e-}3$. We run this experiment in both the original SAC setting and with temperature learning, in which case we still use the standard $\mathcal{H}^{\text{target}} = -\dim(\mathcal{A})$ as entropy target.

Discussion

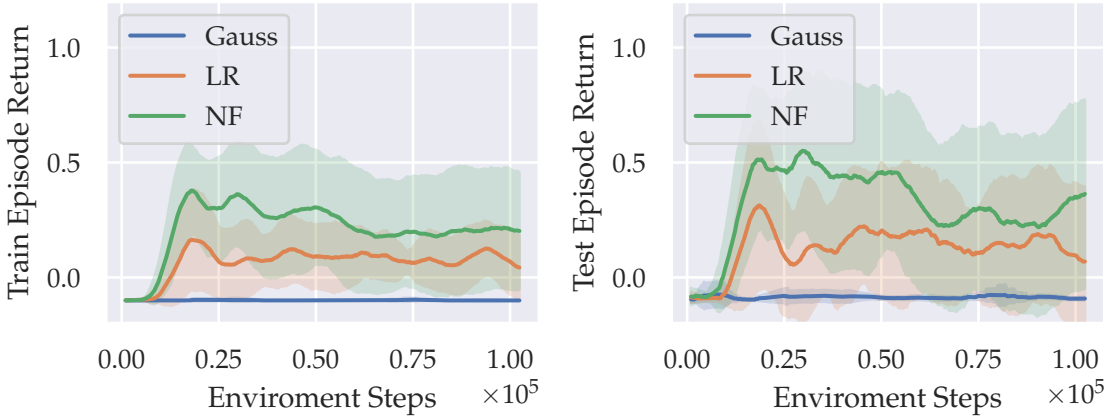
Figure 3.10 shows the results in the original SAC setting. As expected, there is no significant difference between the agents on the uncorrelated exploration task. However, on the correlated exploration task, the *Gauss-Agent* fails to learn a useful policy. Two runs occasionally survive to the end of the episode (0 reward instead of -0.1) or rarely even reach the goal, but that is only possible because of the extreme simplicity of this environment. For example a policy, which independent from the input, returns an action where $a_1 \approx a_2$ can survive until the end of the test episode, or $a_1 \approx a_2 > 1/40$ can find the reward (40 is the maximum episode length). Therefore, even if the agent has never been anywhere close to the goal during training, the networks sometimes output actions which are useful in this degenerate case. However, it is extremely unlikely that a real-world problem would ever be solved by such a simple policy since we could easily solve that problem with classical algorithms.

An, at first sight, unexpected result is that all agents lose most of their performance in the fixed entropy coefficient/horizontal path setting after successfully solving the task. The reason is simply that the entropy coefficient is slightly too high. The agents slowly increase the entropy until they do not reach the goal anymore. On the correlated task, there are more possible actions that move in the correct direction and are inside the path because the agents reach is a square and the diagonal of a square is longer than the sides. Apparently, this additional space to place probability mass already stabilizes the behavior. We verified the claim by running this experiment again with a smaller entropy coefficient. We show here the original plot because it does not affect the statement we want to make and allows for a consistent comparison between the other parts of the experiment.

In the temperature learning setting (Figure 3.11) all agents perform slightly worse than with fixed entropy coefficient α. We assume that could be counteracted by setting a lower entropy target. Nevertheless, it shows that in a sparse reward setting, the entropy target is an important hyperparameter just as the entropy coefficient α, which it replaces. Qualitatively the results are in accordance to the original setting with fixed



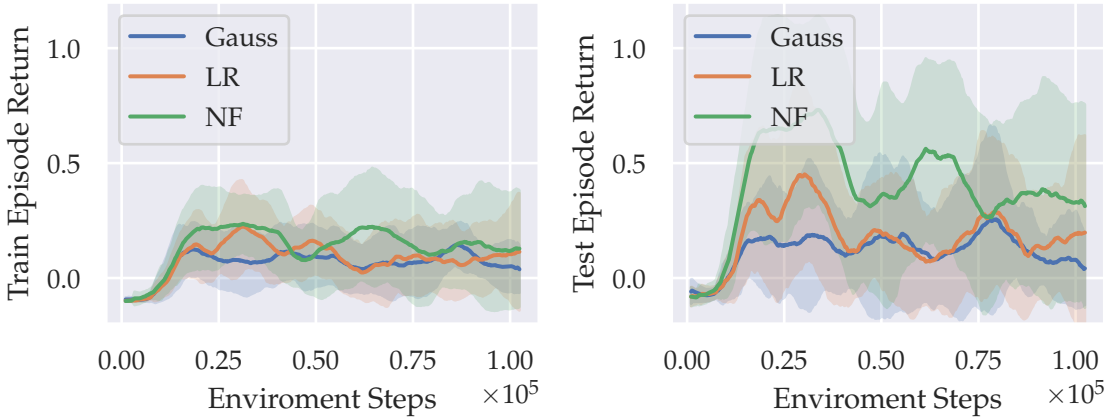
(a) Uncorrelated exploration - horizontal path



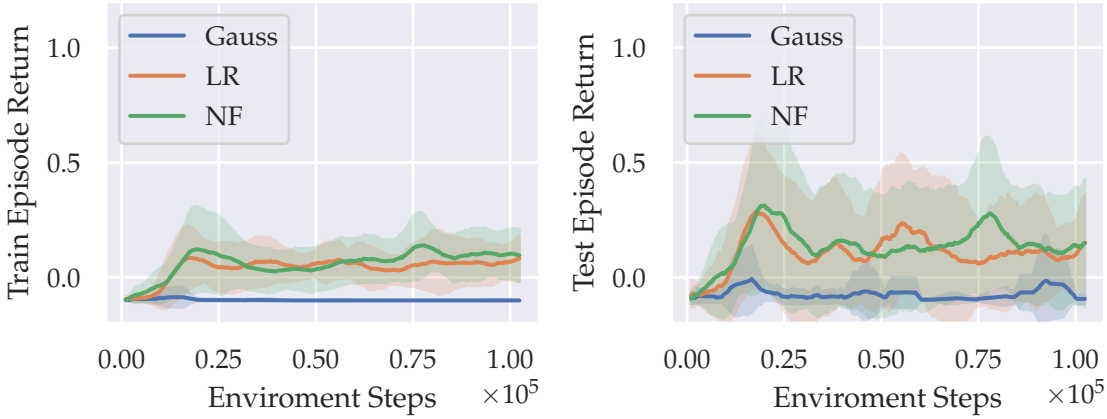
(b) Correlated exploration - diagonal path

Figure 3.10: Training curves on the *Tunnel* environment with sparse reward and fixed entropy coefficient α. We show the average and standard deviation across ten seeds. The *Gauss-Agent* performs on par with the others on the uncorrelated exploration task but fails completely to solve the correlated exploration task. Moreover, the *NF-Agent* consistently outperforms the *LR-Agent* on the correlated path.

α. Still, the *Gauss-Agent* performs similar to the others on the uncorrelated exploration task and fails to solve the problem in the correlated exploration task. The occasional successes of the *Gauss-Agent* are occurring more often in this setting. That is probably due to a similar chain of events as the oscillations in the circle path experiment. If the entropy drops and the networks output a useful mean at chance, the agents can see reward during training, which then improves the chance for seeing rewards during



(a) Uncorrelated exploration - horizontal path



(b) Correlated exploration - diagonal path

Figure 3.11: Training curves on the *Tunnel* environment with sparse reward and fixed entropy target $\mathcal{H}^{\text{target}}$. We show the average and standard deviation across ten seeds. Again the *Gauss-Agent* performs similar to the others on the uncorrelated exploration task but there is clear difference to the more expressive policies on the diagonal path.

testing as well. Here these spikes and collapses of the entropy do not occur as regularly as in the previous experiment but all *Gauss-Agent* runs show them to some extent (on the correlated exploration task).

3.2.5 Summary

The experiments confirm that using a more expressive policy distributions can lead to better performance than the simple diagonal Gaussian. We showed that more expressive policies can hold more entropy and explained how that results in better exploration of the world and, in particular, exploration of the boundaries of what is possible. Furthermore, in environments with sparse rewards, additional expressiveness can make the difference between solving the task and complete failure.

We also gained deep insights into the learning dynamics of the SAC algorithm with and without temperature learning. We discussed how temperature learning breaks the maximum entropy concept but keeps many of the useful properties by exploring new dimensions after finding useful sequences somewhere else. However, when using temperature learning, in particular in environments with sparse reward, there is always the danger of getting stuck in states where steadily increasing the entropy could solve the problem.

Although all the experiments were performed in a simplistic environment, we believe that the insights are nevertheless valuable and could help in understanding the learning dynamics in more realistic higher-dimensional problems.

3.3 Experiments in Complex Environments

In this section we first introduce a 2D fine manipulation environment (Section 3.3.1) and then analyse two experiments conducted on this environment (Section 3.3.2, 3.3.3). We show that the main results transfer from the simple environments in the previous section to this more realistic environment.

3.3.1 The *Square* - A 2D Fine Manipulation Task

Motivation

The goal of the *Square* environment is to model the core elements of a robotic fine manipulation task. Compared to a physically precise simulation of a real robotic hand, we want the environment to be simpler to learn, faster to step, and easy to configure different tasks or settings. Our resulting environment is a 2D caricature of the realistic simulation of the advanced impedance-controlled DLR Hand II of Sievers 2020.

Setup

- The environment is a two-dimensional world with a square in the center, and four *fingers* are positioned around it (see Figure 3.12). The *fingers* are simply points in the plane, which can exert forces on the square. Each one has its own reach, fixed at its initial position.
- We model the square with second-order dynamics. For the *fingers*, we use first-order dynamics because the real robot fingers are mostly limited by their maximum velocity and inertia plays a minor role. The agent interacts with the environment by requesting target positions for a proportional-derivative (PD) controller. This is similar to setting target joint positions for the impedance controlled DLR Hand II.
- In this environment, we learn different tasks, and the specific reward will be explained in each separate experiment section.
- The difficulty shared by all tasks is that the agent may not drop the square after a certain duration of warm-up time. Otherwise, the episode restarts and the agent usually receives a high negative reward. Instead of a gravitational force, we compute the force applied to the square by all *fingers* and multiply by some friction coefficient. If this force is less than the weight force of the square after the warm-up time, the episode is reset.

The environment is designed to be extremely versatile and easily configurable. We can specify weights, inertia, size, starting positions, starting angle, controller parameters and

finger reach, most of which can be passed as a distribution from which the environment samples at each episode reset. Additionally, we can specify which observations should be included in the environment state and how much observation noise we want to add to each one of them. We run all our experiments with 100 Hz simulation frequency to ensure stable collision behavior, but the agent interacts with the environment at a much lower rate, usually 4 Hz. Typically episodes are timed out after 10 or 20 seconds, and the warm-up time is set to 2 to 4 seconds. Currently, we have three different tasks to choose from. In the simplest case, the agent only needs to hold the cube and can then be further instructed to move it to the center or keep the angle steady by applying appropriate penalties (negative reward).

Details

The action space $\mathcal{A} \in [-1, 1]^8$ is normalized within the reach of the fingers. For each of the four fingers, we have x and y coordinates. Note that while the environment is two-dimensional, the action space is eight-dimensional.

The dimension of the state space $\mathcal{S} \in [-1, 1]^D$ depends on multiple settings. Always included are x and y coordinate of the square, not normalized because the world bounds are -1 and 1. Also $\cos(\theta)$ and $\sin(\theta)$ of the square's angle θ are included. The x and y coordinates of the finger positions are normalized within their reach. Unless stated otherwise, we include the last action, i.e., normalized x and y coordinates of the target positions, into the state. Additionally, we include a timer observation that starts at 1 for each episode and decreases linearly to 0 during the warm-up phase. Therefore, the square will only drop if this observation is 0. Finally, we use observation stacking. In particular, we stack the two latest states yielding $D = 42$ for all experiments.

Implementation

The environment is implemented entirely in python and `numpy`. It inherits OpenAI's ubiquitous `gym` interface. We use the windowing library `pyglet` (based on OpenGL) for visualization (*List of Python Packages* 2020).

3.3.2 Holding an Object

Motivation

We want to replicate the results of Experiment 3.2.2 and Experiment 3.2.3 in a more complex environment. In the first part, we investigate how significant the entropy delta between more and less expressive policies is, and, in the second part, we examine the influence on the training performance.

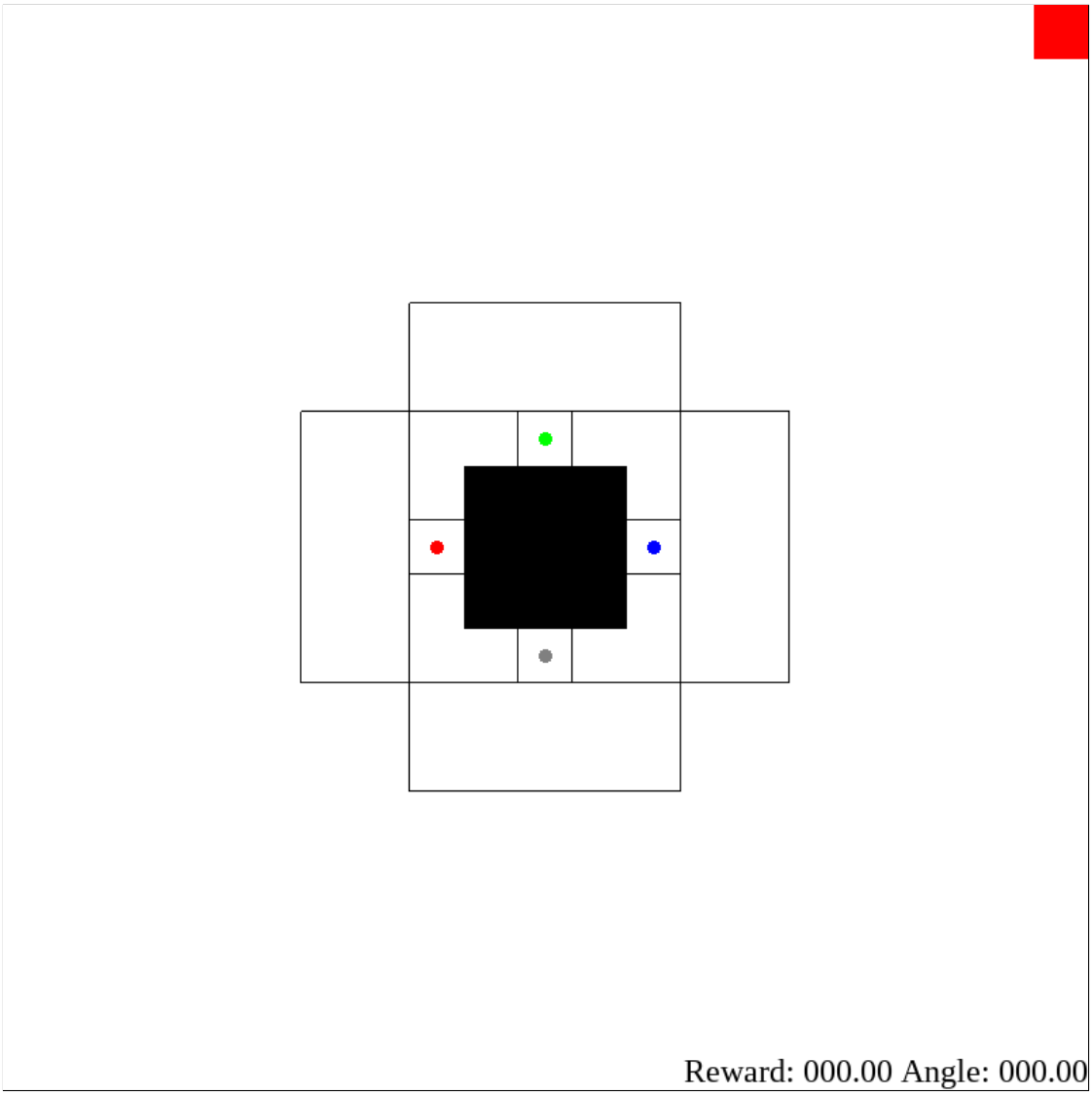


Figure 3.12: The *Square* environment in its initial state without setup noise. The black square in the center needs to be held by the four fingers, indicated by the colored circles. The squared box around each finger shows its respective fixed reach. On the top right, there is an indicator that turns green while the fingers apply enough force to prevent the square from dropping. On the bottom right, there is additional information about the current state of the episode.

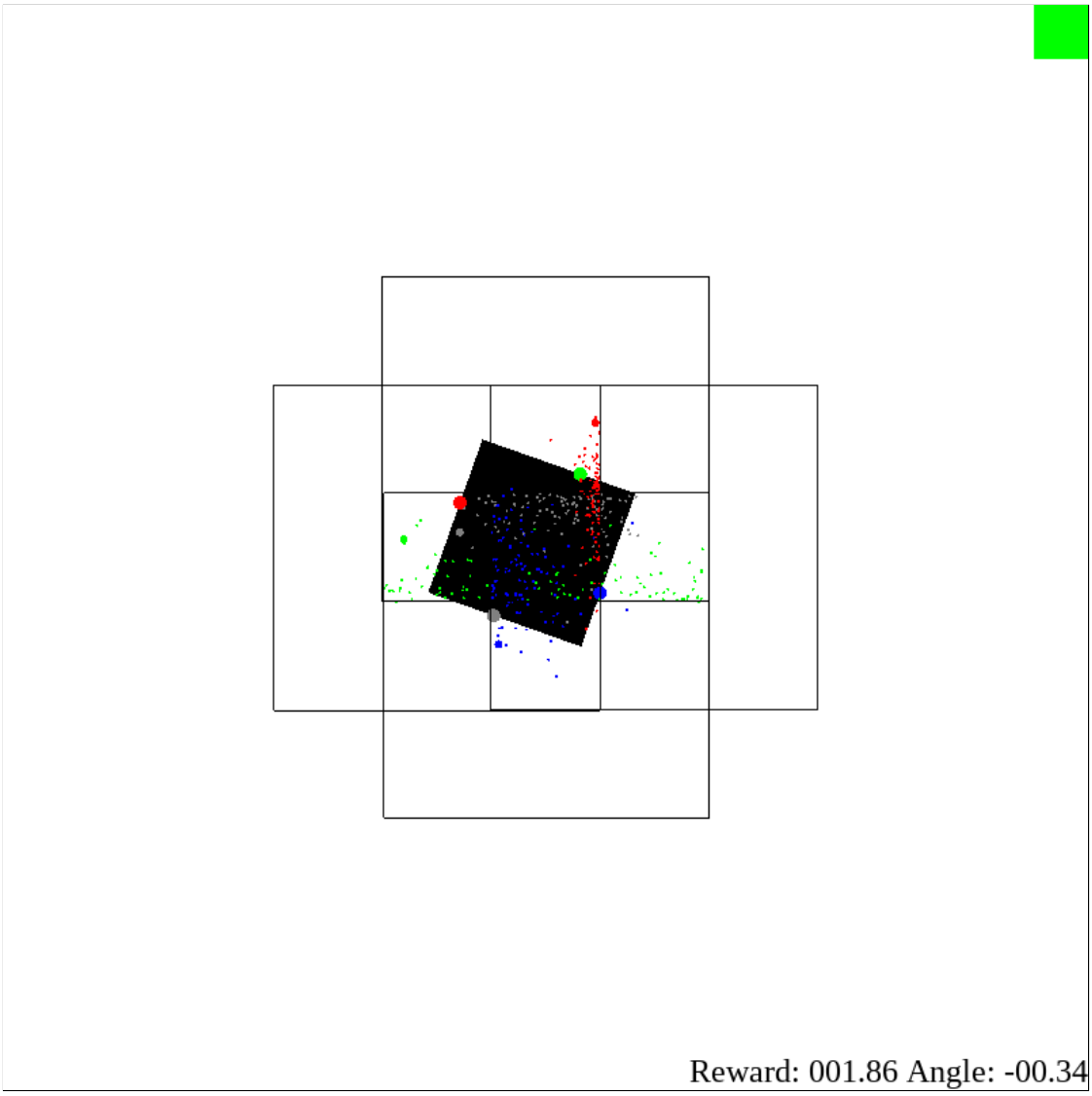


Figure 3.13: The *Square* environment in action. The slight misalignment of the reach boxes is caused by the setup noise. The smaller circles in the respective colors show the current target position that was set in the last step. The single pixels in the respective colors are action samples drawn from the policy given the current observation. Therefore, the targets seen in an image are drawn from the distribution of the last image, not the distribution seen in the same image.

Setup

We use the *Square* environment in its simplest form. The agent is rewarded for applying enough force to the square to prevent it from dropping. This reward can be awarded at each step and is the only positive reward available. We penalize the agent for actions that are close to its maximum reach, simply because that results in policies which are safer to apply on real-world robots. And we penalize the agent for the norm of the squares position and the absolute value of the squares angle. Therefore, the goal of the agent is to hold the square fixed in the center of the world without rotating it around. Additionally, there is, of course, a penalty for dropping the square, i.e., not applying enough force after the warm-up timer of 2 seconds ran out, and for breaking the robot, i.e., leaving the designated reach. Episodes are 10 seconds long (simulated time), and the agent interacts with the environment at 4 Hz. That limits the maximum possible reward to 40. In Table 3.2 we provide the learning algorithms hyperparameters. In the temperature learning setting we report results for $\mathcal{H}^{\text{target}} = -2$ and $\mathcal{H}^{\text{target}} = -\dim(\mathcal{A})$. Note that we do not run the *LR-Agent* in this experiment since it always performed between the other two agents and this experiment requires more computation time.

Fixed entropy coefficient

In Figure 3.14, we show the training curves with fixed entropy coefficient. The results from Experiment 3.2.2 mostly carry over to the new environment. There is a clear

Parameter	Value
network units	(256, 256)
non-linearity σ	ReLU
discount γ	0.9
polyak ρ	0.99
optimizer	Adam
learning rate	3e-4
entropy coefficient α	0.15
batch size	256
update every	100
update many	100
replay buffer size	1e6
update after	200
initial random steps	200
weight decay	2.5e-4

Table 3.3: Hyperparameters for experiments on the *Square* environment.

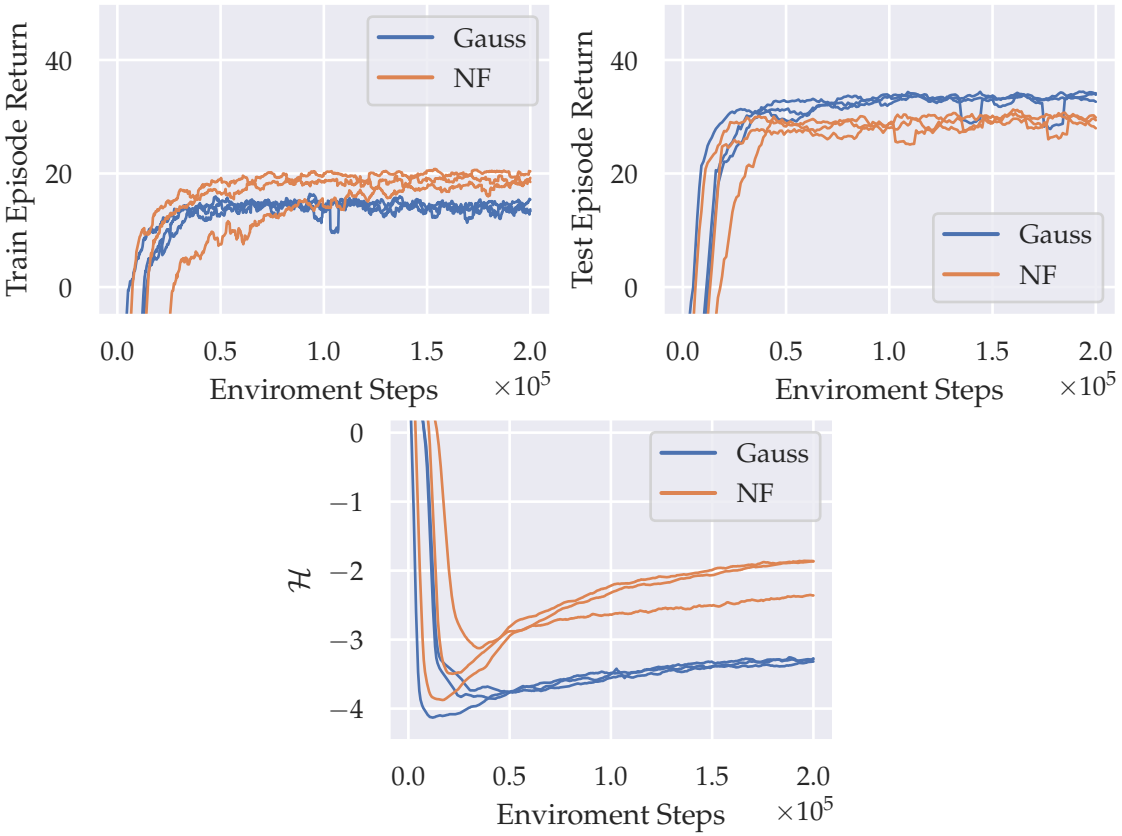
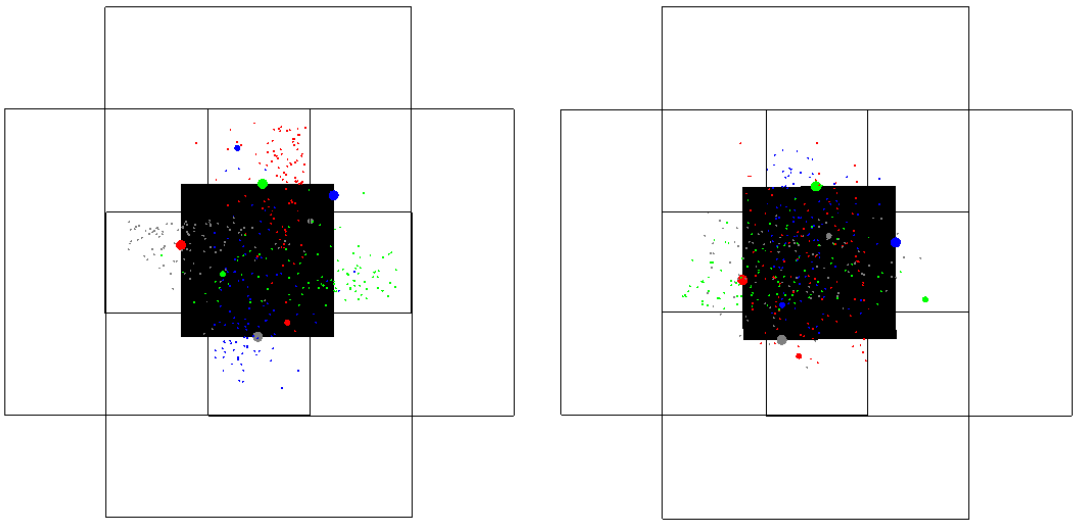


Figure 3.14: Training curves on the holding task of the *Square* environment with fixed entropy coefficient α. We show three smoothed individual runs of each agent. The *NF-Agent's* policy can increase the entropy over 1 point higher than the *Gauss-Agent*. Moreover, the final trainings performance is slightly better.

gap in entropy, and the test episode returns are similar, although the *Gauss-Agent* scores slightly higher. We assume that this is due to the deterministic strategy of the *NF-Agent*, which picks the mean before the flow transformations. Unlike in the *Tunnel* environment, here, the additional expressiveness does not speed up the training process but results in slightly higher training performance.

Visaulizing the entropy

Visualizing the delta in entropy is not as simple as in the previous experiment for two reasons. Firstly, the episodes are longer, and there are some states where the policy distribution is wider and others where the distribution is more focused. Therefore,



(a) *Gauss-Agent* (b) *NF-Agent*

Figure 3.15: Visualization of the agents policies. Both agents are trained with temperature learning and the same fixed entropy target $\mathcal{H}^{\text{target}} = -2$. For the *Gauss-Agent*, the distributions of the individual fingers completely describe the full distribution. Whereas for the *NF-Agent*, due to possible correlations, the per-finger distributions do not cover the full distribution. In particular, the sum of the entropies of the individual finger distributions is only an upper bound for the entropy of the total distribution. Therefore, it looks as if the distribution contains more entropy but in fact there are correlations between the samples which we cannot visualize in two dimensions.

it would be necessary to include a video or a sequence of 40 images to convey how significant the difference is. A second problem is the correlation between dimensions. On one hand, we can take an eight-dimensional diagonal normal distribution, plot it as four two dimensional distributions, and still get an intuition for the total entropy by adding up the individual entropies. On the other hand, for the *NF-Agent*, we still plot the four distributions of the different fingers in their colors but now the sum of the individual entropies is only an upper bound on the total entropy.. Figure 3.15 visualizes this problem by showing the policies of two agents that have the same total entropy.

Now looking at the policy without visualizing the distributions at each step, we would expect to see the fingers sliding across the edges of the square since that intuitively seems to be the highest entropy policy that can hold the square still. And, in particular, we expect to see more sliding from the *NF-Agent* because it requires coordination between the fingers, which the *Gauss-Agent* cannot capture in its policy distribution. It

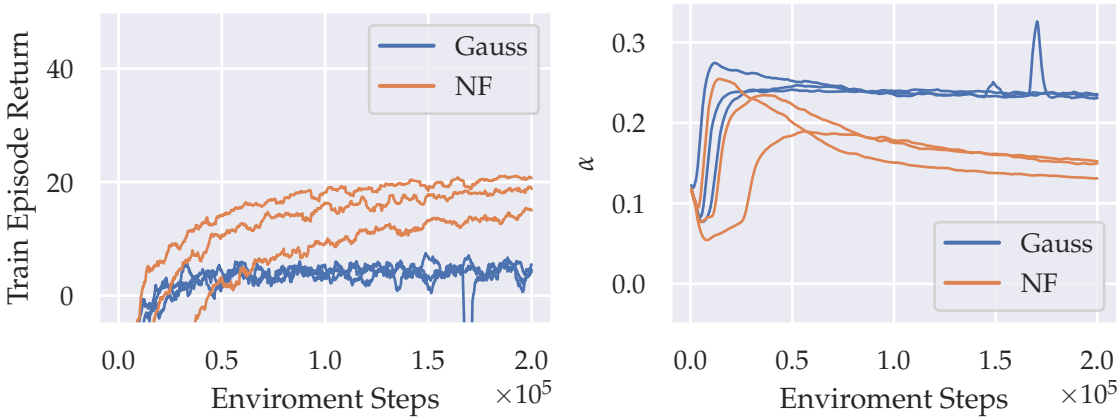
is difficult to quantify but on the videos[1] we can see more fingers sliding around on the *NF-Agent* recordings. However, we also noted that the difference is not as significant as the delta in entropy suggests. An explanation could be that it is easier for the *NF-Agent* to put entropy into coordinated pressing and partial releasing of the square while holding it still rather than sliding along the edges. The big delta in entropy is, of course, comprised of all kinds of exploration that require coordination, but not all of them result in a visible difference.

Temperature learning

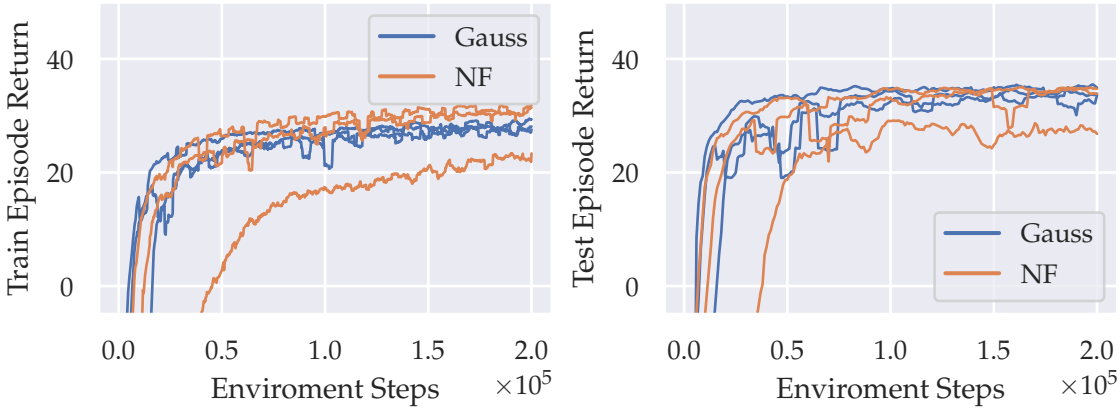
We ran this experiment with two different entropy targets to understand how it affects the learning process. At first we set $\mathcal{H}^{\text{target}} = -2$, which is the average entropy value of the final *NF-Agent* policies trained with fixed entropy coefficient. Therefore, the entropy target is higher than the value the *Gauss-Agent* ended up with, and the experiment tests what happens if the entropy target is (presumably) set too high. In Figure 3.16a we can see exactly the same effects as in Experiment 3.2.3. The *Gauss-Agent* performs worse during training, but, just like in the first part of the experiment, slightly better than the *NF-Agent* in terms of test performance (not shown in the figure because they are very similar to the curves of $\mathcal{H}^{\text{target}} = -8$). Moreover, we can see a major jump in the entropy coefficient curve of one of the *Gauss-Agent* runs and in the episode return during the same time steps. This is probably a milder version of the oscillations we have seen in Experiment 3.2.3, which in this case does not lead to catastrophic forgetting. Apart from these expected effects, we can see that the *NF-Agent* reaches about the same episode return as in the fixed entropy coefficient setting but does need more time steps to do so. And the final entropy coefficient is $\alpha = 0.15$. That is the value used in the first part of the experiment, which we based the selection of our entropy target on.

As the second entropy target we choose $\mathcal{H}^{\text{target}} = -8$ because it is the standard choice of $-\dim(\mathcal{A})$. The results in Figure 3.16b show that both agents can also solve the task with much lower entropy. They even reach similar performance during training compared to their test scores, which are very similar to the test episode returns of the runs with $\mathcal{H}^{\text{target}} = -2$ (therefore not shown in the figure). One of the three *NF-Agent* runs learned much slower than the others. That is concerning if one wants to train in dense reward environments where it is not essential to set high entropy targets, therefore, suggesting it might be necessary to select an appropriate policy distribution for individual tasks. Before drawing definite conclusions on that topic, we would have to run the experiment with many more seeds and investigate different initialization schemes. However, we are ultimately interested in the setting where the agent is looking for a sparse reward, and, without additional incentives (which are outside the scope of

[1] Videos are uploaded to Google Drive.
`https://drive.google.com/drive/folders/1ZODshmp3TV0vLrqEoz4kD-9X1lGunrTz?usp=sharing`



(a) $\mathcal{H}^{\text{target}} = -2$



(b) $\mathcal{H}^{\text{target}} = -8$

Figure 3.16: Training curves on the holding task of the *Square* environment with fixed entropy target $\mathcal{H}^{\text{target}}$. We show three smoothed individual runs of each agent. The *NF-Agent* runs have high variance, but on average, they perform better if the entropy target is high and similar compared to the *Gauss-Agent* if the entropy target is low.

this thesis), high entropy will be required to find it.

3.3.3 Moving an Object

Motivation

In this experiment we want to investigate how the results on improved exploration in sparse reward settings (Experiment 3.2.4) translate to more complex environments.

Setup

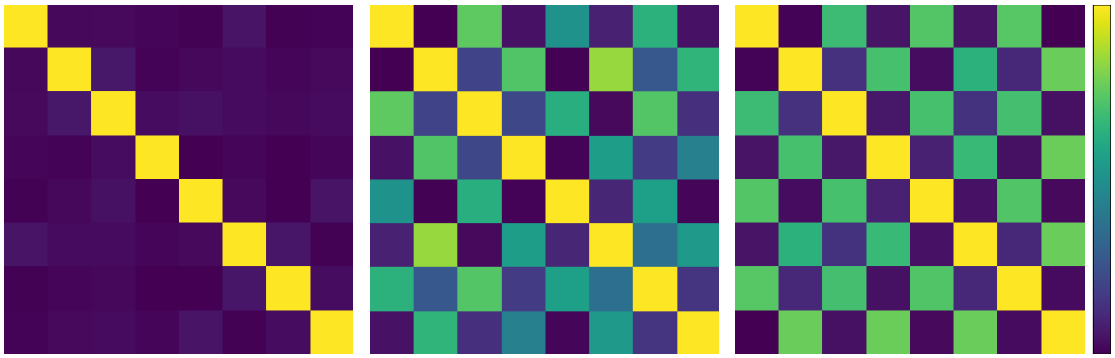
We again use the *Square* environment but with a few modifications. The hyperparameters remain the same (cf. Table 3.3) except for the entropy coefficient, $\alpha = 0.05$, which needs to adapt to the new reward function. Here, we only penalize the square position during the warm-up phase, which lasts 4 seconds. Afterward, the agent has 16 seconds to explore the environment, and we are particularly interested in the resulting movement of the square. There is no designated goal region where the agent receives a sparse reward. Instead, we compare the maximum and the variance of the squares distance to the origin during the exploration phase. To highlight the difference between the agents, we increase the difficulty of the task. We limit the maximum force which the fingers can exert such that the agent requires all 4 fingers to hold the square. Additionally, we add a penalty for the force the fingers try to exert (and drop the reach penalty term because the agent does not go to the edge of its reach as a side effect of our latest modifications). To avoid that runs do not learn to hold the square because the environment is too difficult, we insert some environment interaction into the replay buffer where the agent successfully holds the square still (only at the very beginning of training).

Insights

Given this setup, the difference in the movement of the square was not easy to quantify because the random seed significantly influences the developing policies. However, there are clear qualitative differences. In Figure 3.19 (extends over 2 pages) we can see that the actions of the *NF-Agent* are correlated, i.e. often all 4 targets move jointly left/right or up/down. The *Gauss-Agent* can never learn such a policy. To further investigate the correlations between the action dimensions, we compute the Pearson correlation coefficient

$$\rho_{X,Y} = \frac{\operatorname{cov}(X, Y)}{\sigma_X \sigma_Y} \tag{3.1}$$

of all pairwise action dimensions. Values close to 0 indicate no linear correlation, and values close ± 1 indicate total positive/negative linear correlation. In Figure 3.17 we plot the absolute value of the correlation coefficients in matrix form. The *NF-Agent's* correlation matrix looks distinctly similar to a chessboard. All the evenly and oddly indexed dimensions correlate within two separate groups. In this case, it means that the targets' x-coordinates move together, and the targets' y-coordinates move together. The *LR-Agent's* correlation matrix shows similar traits but not as pronounced, and the *Gauss-Agent* cannot represent any correlations.



(a) *Gauss-Agent* (b) *LR-Agent* (c) *NF-Agent*

Figure 3.17: Visualization of the correlations between action dimensions. The color indicates the absolute value of the pearson correlation coefficient. On the right we show the entire color spectrum from $|\rho| = 1$ (yellow) to $|\rho| = 0$ (purple).

Narrowing the tunnel

Primarily, we were interested in the resulting movement of the square by the different agents. And during the first runs of the experiment difference was not very significant. The problems seems to be the vast freedom the agent has. The more dimensions there are in which the agent can increase the entropy without losing reward, the shallower the exploration becomes. Blindly increasing the total entropy seems to lead to so much noise that different actions start canceling out each other. To test this hypothesis, we penalize the absolute value of the squares y-coordinate, reducing the dimensions in which the agents can explore without punishment.

In the first part of this experiment holding the square was the analog to the tunnel in Experiment 3.2.4. Now we are basically narrowing the tunnel by taking away one of the free dimensions. However, unlike in the previous experiment, we do not have to use a diagonal tunnel because moving the square along the x-axis is already a highly correlated path in the eight-dimensional action space of the agent.

Discussion

Figure 3.18 shows the average over multiple episodes of the maximum and the standard deviation of the square's x-coordinates visited during a single episode. Note that one of the three runs of the *NF-Agent* still learns a significantly different policy. This policy performs worse than all the other runs in terms of average episode return and average Q-values despite Q-values being a low variance quantity. However, the average maximum x-distance and the standard deviation of the x-distance, our metrics for

success in this experiment, are over double that of the other *NF-Agents*. We cannot explain why this seed does not reduce the high variance behavior in favor of episode return. However, due to the large number of environment steps required for this experiment (an order of magnitude more than the previous one), running many seeds is not feasible. Therefore, we leave a more detailed analysis of this peculiarity for future work.

The effect we are mainly interested in can already be observed by comparing the remaining runs. After around 1.5×10^6 environment steps, we can see groupings of the individual runs in the average standard deviation. And as expected, the *Gauss-Agent* moves the square much less around than the *LR-Agent* which in turn has less movement than the *NF-Agent*. In the average maximum x-distance plot, there is more variance between the runs, but we can see that while the *Gauss-Agents* reduce their maximum x-distance over time, the *NF-Agents* increase it. Therefore, the *Gauss-Agent* has to rely on lucky network initialization to find spare rewards early on during the training, while the *NF-Agent* can, under the given circumstances, learn to properly explore the environment and reliably reach potential sparse rewards.

3.3.4 Summary

In this section , we showed that in an environment with a higher dimensional action space, expressive policy distributions can increase the entropy far more than simpler distributions. The entropy delta was larger than in the simpler *Tunnel* experiments of the previous section, but due to the vast freedom the agents have, it did not immediately translate into significantly deeper exploration. However, by reducing the freedoms of the agent we could clearly see the expected effect.

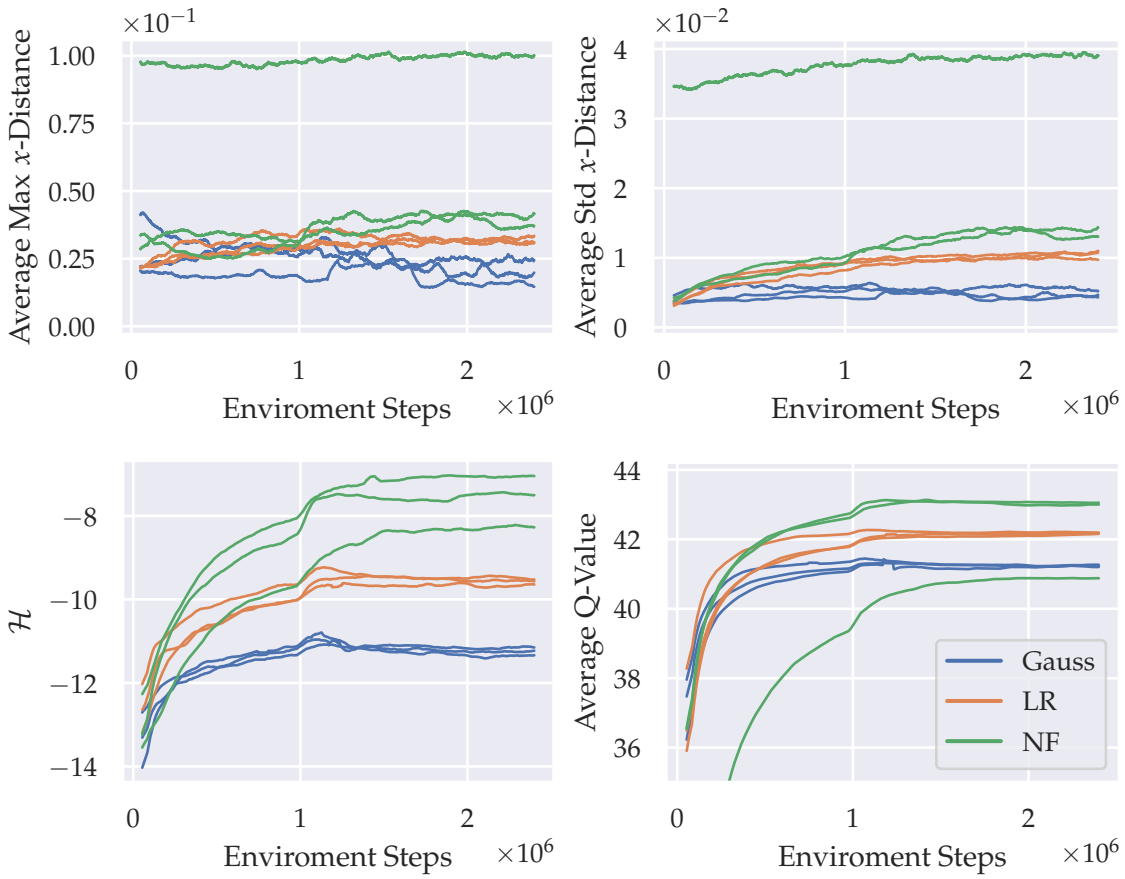
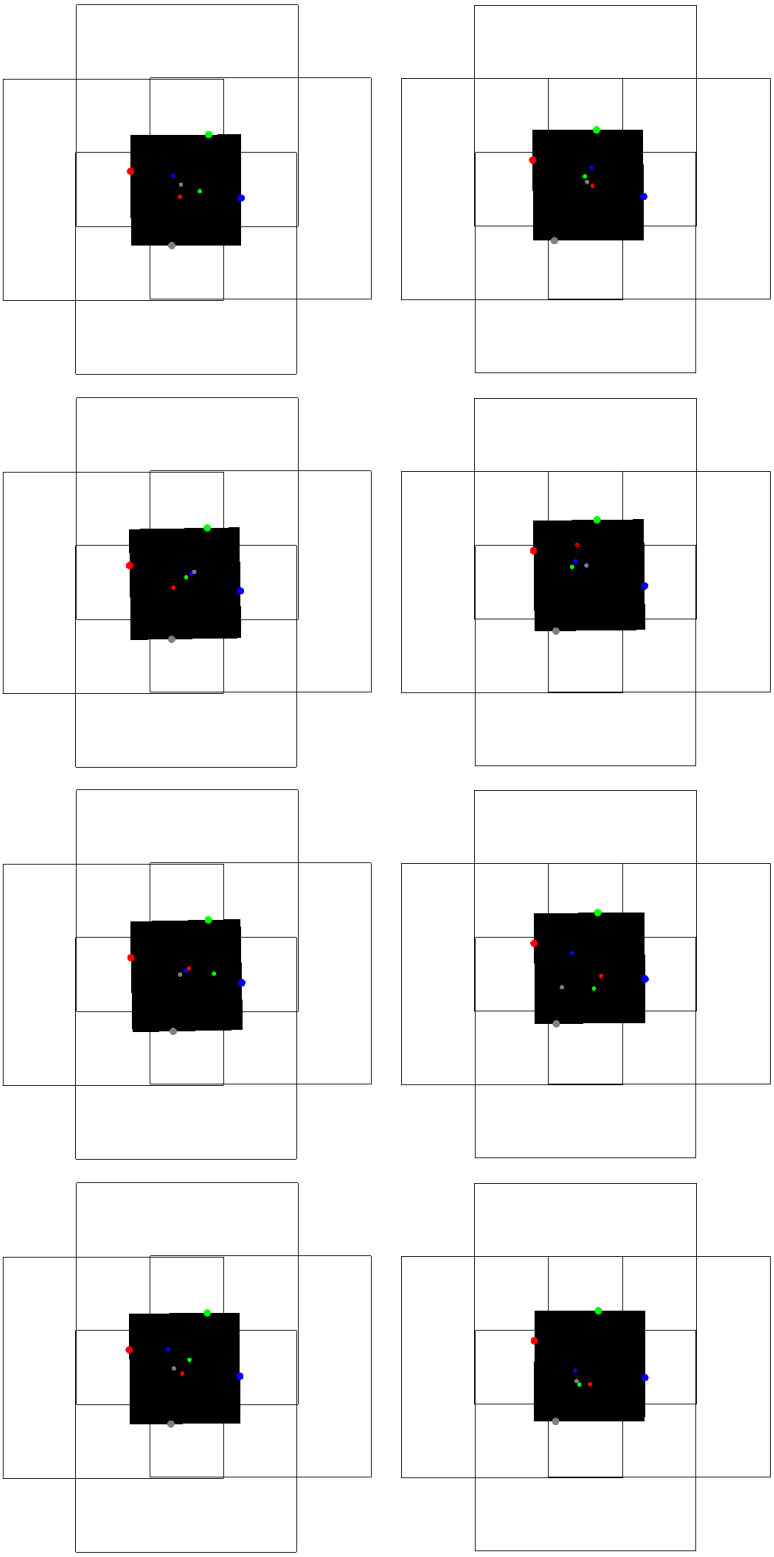
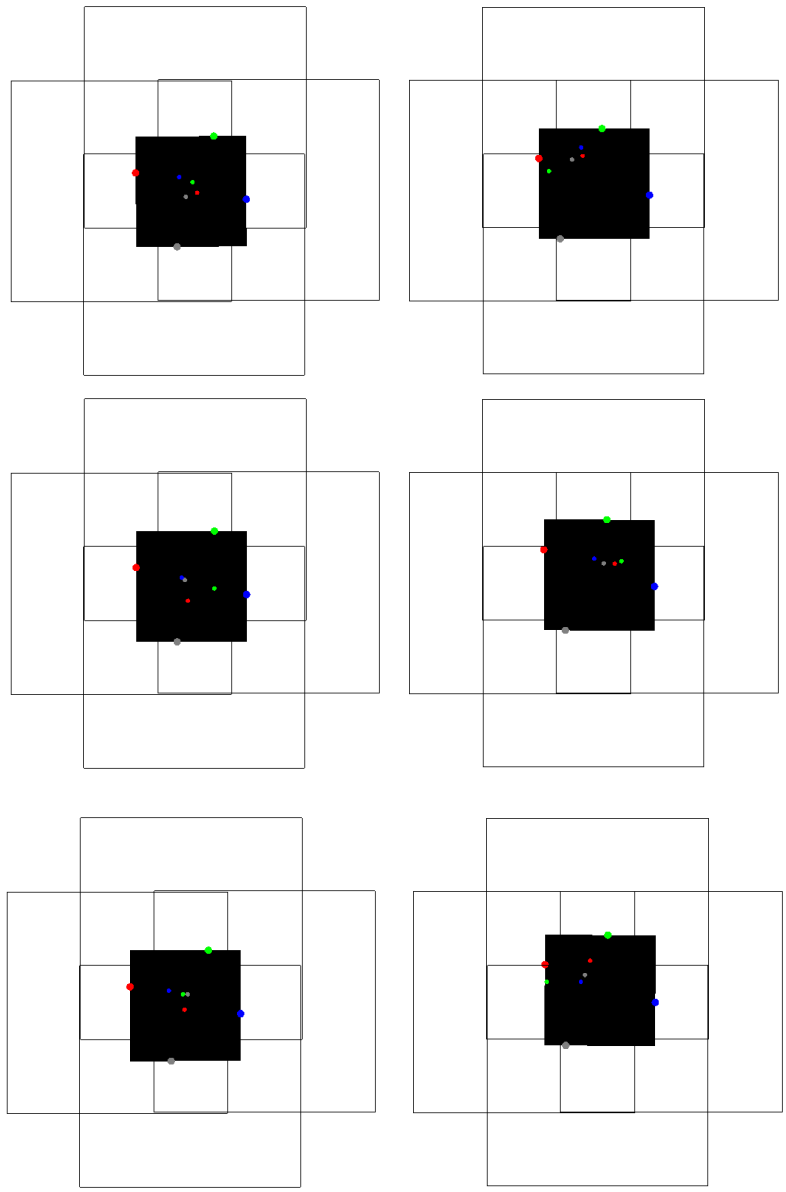


Figure 3.18: Training curves on the moving task of the *Square* environment with fixed entropy coefficient α. We show three smoothed individual runs of each agent. We go into more detail about the outlier *NF-Agent* run in the discussion. Apart from the outlier, the *LR-Agents* perform in between the more expressive *NF-Agents* and the less expressive *Gauss-Agent*. Of those, the former reaches almost three times more standard deviation in the x-coordinate and double the maximum distance after training.





(a) *Gauss-Agent* (b) *NF-Agent*

Figure 3.19: An action sequence of a trained *Gauss-Agent*[a] and *NF-Agent*[b]. We show the first 7 steps (top to bottom) after the warm-up time ran out. The effect can be seen even better in the videos. Note this figure begins on the previous page.

[a] Video: `https://drive.google.com/file/d/1eef5y8bJciS3xtAq1WxrOAKAJhl7YS7-/view?usp=sharing`
[b] Video: `https://drive.google.com/file/d/1q01nApsX_plIEXqhaYndWkBbHyEsK01n/view?usp=sharing`

3.4 Expressive Policies in Partially Observable Environments

In this section, we shortly present an advantage of more expressive policies in partially observable environments, which is independent of improved exploration, i.e., the main topic of this thesis, but highly relevant for real-world robotic tasks.

Motivation

Most reinforcement learning benchmarks for robotics are fully observable Markov decision problems. And in such an environment, there exists always an optimal deterministic policy. Therefore, it is common practice to use deterministic policies at test time. In this thesis, we have done so as well and, therefore, the additional expressiveness of the policy gives no advantage at test time. However, in real the real world, a robotic system will never fully and accurately observe the entire environment state. Therefore, we are working with POMDPs, in which not always an optimal deterministic policy exists. Thus, solving simple tasks such as a sorting box for children with a deterministic policy would require highly accurate positions of the object and the hole. In cases where it is infeasible to guarantee this level accuracy one will have to resort to stochastic policies.

In this experiment, we show that the *NF-Agent* can learn a stochastic goal-searching policy that finds approximately known goals more successfully than less expressive policies can.

Setup

We use the same hyperparameters (cf. Table 3.3) and environment setup as in Experiment 3.3.3, without penalties for the squares position. Additionally, we provide a sparse reward if the agent moves the square to a specified goal. A new exact goal location is sampled for every episode from a given goal region. In principle, we could pass an approximate goal position as an observation and sample around it. However, for simplicity, we use a fixed goal region, a diagonal line through the origin, for all episodes. That removes the need to pass in an additional goal observation.

Discussion

In Figure 3.20, we show the proportion of episodes in which the agents reached the goal during the training. Test scores are not applicable here because we want to compare stochastic policies and not their deterministic remnants. Note that we sample the starting positions of the square around the origin, and the goals uniformly from the diagonal going through it. Therefore, the agents can find the goal in around 20 % of the episodes by simply holding the square and moving it to the center. Thus, most

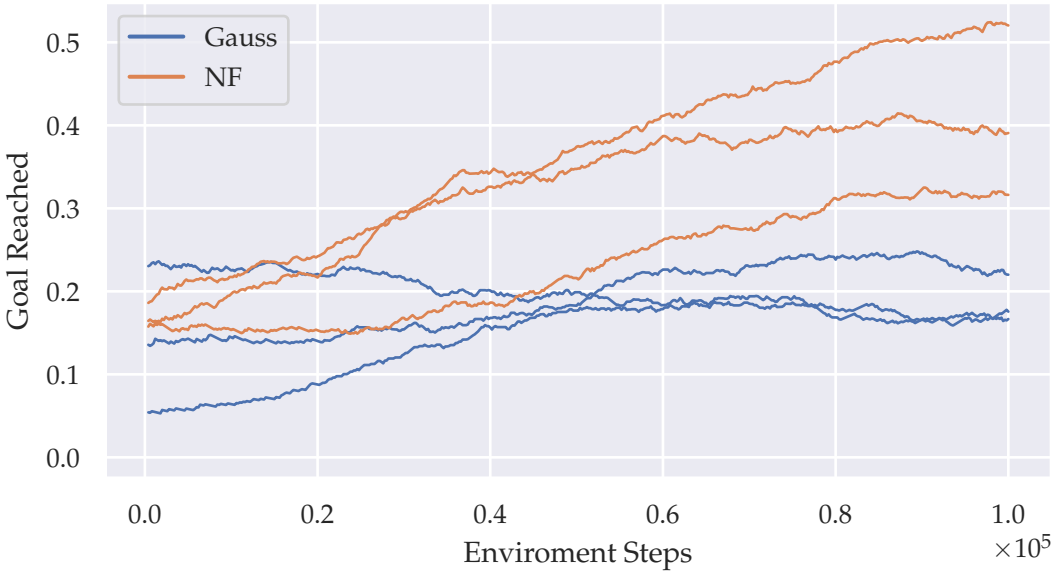


Figure 3.20: Training curves on the *Square* environment with partially observable goal region. We show three smoothed individual runs of both agents. After around 8×10^4 environment steps all *NF-Agent* runs find the goal more often than the *Gauss-Agent* runs.

of the agents immediately have a success rate in that region. However, during the training process, we can see a difference between the two kinds of agents. While there is much variance between the runs, a topic which we leave for future work, it is clear that the *NF-Agents* can improve their performance while the *Gauss-Agents* are stagnating around the default success rate.

Summary

This experiment showed that more expressive policies may become a necessary component of reinforcement learning agents in settings without perfect information of the environment state.

4 Conclusion and Outlook

This thesis is a step towards the applicability of DRL to realistic and relevant robotic tasks. We analyzed the importance of expressive stochastic policies in the context of smart exploration. To this end, we:

- Analyzed normalizing flows and, in particular, the suitability of conditional normalizing flows for the application in stochastic policies.
- Presented the Soft Actor-Critic algorithm along with two extensions with policy distributions of varying expressiveness.
- Designed two versatile environments. The first one allows adjusting the exploration difficulty, and the second one mimics specifically the problem of in-hand manipulation.
- Conducted incremental experiments and an in-depth analysis of the algorithm variants on multiple configurations of the environments.

We successfully showed that even in simple environments, problems may emerge which require expressive policies to solve a given task. As initially conjectured, an agent with the commonly used diagonal normal policy distribution cannot explore environments that require correlations between action dimensions. Additionally, we show that in not perfectly observable environments, agents that represent their final (learned) policy with expressive distributions can solve tasks where commonly used simpler distributions fail.

Implications for on-policy algorithms

In this thesis, we used an off-policy algorithm. However, we assumed that the exploration of the environment is done by the policy. Therefore, our models can also be applied to on-policy algorithms. Moreover, while SAC effectively explores on-policy, it is technically an off-policy algorithm. One could potentially come up with other schemes to facilitate correlated actions by deviating from the policy. For on-policy algorithms, this loophole does not exist. Therefore, our results are especially important to all on-policy reinforcement learning algorithms.

Expressiveness in continuous action spaces

In contrast to simple to model discrete action spaces, modeling continuous action spaces introduces a trade-off between expressiveness and complexity of the distribution. A diagonal normal distribution has only a few parameters, but as we have shown, the naive assumption that the action dimensions are completely uncorrelated can come at a disastrous cost. Choosing more complex models increases the number of required parameters drastically. Due to the limited success of previous works with highly expressive energy-based models, we have focused on relatively simple models, which proved to be an excellent choice for our experiments. These models can represent local correlations very well. And we conjecture that they are sufficient for most robot control problems because there is no reason to believe complex multimodal policies are required to solve such environments.

Future work

Our motivational problem was a robot hand that rotates an object while holding it firmly grasped. So far, we have not solved this task because lifting all fingers, one at a time, and moving them to another side is a long process during which the agents receive no reward. More expressive policies allow the agent to inject more variance in movements that lift a single finger while pressing harder with the others. However, there are still many different possible moves left. Thus, the Brownian motion of sampling new moves at every step makes little progress along any of these paths. The probability of finding the sparse reward signal at the end of one of the paths is certainly higher than with a diagonal normal policy distribution that cannot increase the variance in these movements without dropping the object. However, reliable deep exploration along such paths will require additional modifications to the algorithm. Promising approaches to achieve deep exploration include hierarchical methods and curiosity-driven exploration (both are discussed in the Related Work, Section 1.3). We believe that deep exploration, in combination with expressive policies, will be the key to dexterous in-hand manipulation and many other real word robot control problems.

Bibliography

Abdolmaleki, A., J. T. Springenberg, Y. Tassa, R. Munos, N. Heess, and M. A. Riedmiller (2018). "Maximum a Posteriori Policy Optimisation". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.

Achiam, J. (2018). "Spinning Up in Deep Reinforcement Learning". In:

Badia, A. P., B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, D. Guo, and C. Blundell (2020). "Agent57: Outperforming the Atari Human Benchmark". In: *CoRR* abs/2003.13350.

Bäuml, B., O. Birbach, T. Wimböck, U. Frese, A. Dietrich, and G. Hirzinger (2011). "Catching flying balls with a mobile humanoid: System overview and design considerations". In: *11th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2011), Bled, Slovenia, October 26-28, 2011*. IEEE, pp. 513–520.

Bäuml, B., T. Hammer, R. Wagner, O. Birbach, T. Gumpert, F. Zhi, U. Hillenbrand, S. Beer, W. Friedl, and J. Butterfaß (2014). "Agile Justin: An upgraded member of DLR's family of lightweight and torque controlled humanoids". In: *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*. IEEE, pp. 2562–2563.

Bellman, R. (1957). "A Markovian Decision Process". In: *Indiana Univ. Math. J.* 6 (4), pp. 679–684.

Bogachev, V., A. Kolesnikov, and K. Medvedev (Oct. 2007). "Triangular transformations of measures". In: *Sbornik: Mathematics* 196, p. 309.

Bougie, N. and R. Ichise (2020). "Skill-based curiosity for intrinsically motivated reinforcement learning". In: *Mach. Learn.* 109.3, pp. 493–512.

Brown, T. B., B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei (2020). "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165.

Butterfaß, J., M. Grebenstein, H. Liu, and G. Hirzinger (2001). "DLR-Hand II Next Generation of a Dextrous Robot Hand". In: *Proceedings of the 2001 IEEE International Conference on Robotics and Automation, ICRA 2001, May 21-26, 2001, Seoul, Korea*. IEEE, pp. 109–120.

Devlin, J., M. Chang, K. Lee, and K. Toutanova (2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Ed. by J. Burstein, C. Doran, and T. Solorio. Association for Computational Linguistics, pp. 4171–4186.

Dinh, L., J. Sohl-Dickstein, and S. Bengio (2017). "Density estimation using Real NVP". In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Du, Y. and I. Mordatch (2019). "Implicit Generation and Modeling with Energy Based Models". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. Ed. by H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, pp. 3603–3613.

Fellows, M., A. Mahajan, T. G. J. Rudner, and S. Whiteson (2019). "VIREL: A Variational Inference Framework for Reinforcement Learning". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. Ed. by H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, pp. 7120–7134.

Fujimoto, S., H. van Hoof, and D. Meger (2018). "Addressing Function Approximation Error in Actor-Critic Methods". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by J. G. Dy and A. Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 1582–1591.

Germain, M., K. Gregor, I. Murray, and H. Larochelle (2015). "MADE: Masked Autoencoder for Distribution Estimation". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by F. R. Bach and D. M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 881–889.

Goodfellow, I. J., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio (2014). "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing*

Systems 2014, December 8-13 2014, Montreal, Quebec, Canada. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, pp. 2672–2680.

Haarnoja, T., K. Hartikainen, P. Abbeel, and S. Levine (2018). "Latent Space Policies for Hierarchical Reinforcement Learning". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by J. G. Dy and A. Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 1846–1855.

Haarnoja, T., H. Tang, P. Abbeel, and S. Levine (2017). "Reinforcement Learning with Deep Energy-Based Policies". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by D. Precup and Y. W. Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 1352–1361.

Haarnoja, T., A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine (2018). "Soft Actor-Critic Algorithms and Applications". In: *CoRR* abs/1812.05905.

He, K., X. Zhang, S. Ren, and J. Sun (2015). "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. IEEE Computer Society, pp. 1026–1034.

Huang, C., D. Krueger, A. Lacoste, and A. C. Courville (2018). "Neural Autoregressive Flows". In: *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*. Ed. by J. G. Dy and A. Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, pp. 2083–2092.

Jang, E. (2018). *Normalizing Flows Tutorial*. https://blog.evjang.com/2018/01/nf2.html - accessed 17.08.2020.

Kingma, D. P. and P. Dhariwal (2018). "Glow: Generative Flow with Invertible 1x1 Convolutions". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, pp. 10236–10245.

Kingma, D. P. and M. Welling (2014). "Auto-Encoding Variational Bayes". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. Ed. by Y. Bengio and Y. LeCun.

Kobyzev, I., S. Prince, and M. Brubaker (2020). "Normalizing Flows: An Introduction and Review of Current Methods". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1.

Li, T., K. Srinivasan, M. Q. Meng, W. Yuan, and J. Bohg (2019). "Learning Hierarchical Control for Robust In-Hand Manipulation". In: *CoRR* abs/1910.10985.

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra (2016). "Continuous control with deep reinforcement learning". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Y. Bengio and Y. LeCun.

List of Python Packages (2020).
`https://pypi.org/project/pybullet/`,
`https://pypi.org/project/mujoco-py/`,
`https://pypi.org/project/numpy/`,
`https://pypi.org/project/gym/`,
`https://pypi.org/project/pyglet/`,
`https://pypi.org/project/shapely/`,
`https://pypi.org/project/scipy/`,
`https://pypi.org/project/ruamel.yaml/`,
`https://pypi.org/project/mpi4py/`,
`https://pypi.org/project/scikit-learn/`,
`https://pypi.org/project/tensorflow/`,
`https://pypi.org/project/tqdm/`,
`https://pypi.org/project/matplotlib/`,
`https://pytorch.org/`,
`http://pyro.ai/`.

Lu, Y. and B. Huang (Apr. 2020). "Structured Output Learning with Conditional Generative Flows". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04, pp. 5005–5012.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis (2015). "Human-level control through deep reinforcement learning". In: *Nat.* 518.7540, pp. 529–533.

Oord, A. van den, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu (2016). "WaveNet: A Generative Model for

Raw Audio". In: *The 9th ISCA Speech Synthesis Workshop, Sunnyvale, CA, USA, 13-15 September 2016*. ISCA, p. 125.

OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang (2019). "Solving Rubik's Cube with a Robot Hand". In: *CoRR* abs/1910.07113.

Openreview (2018). *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. `https://openreview.net/pdf?id=HJjvxl-Cb`.

Ostrovski, G., M. G. Bellemare, A. van den Oord, and R. Munos (2017). "Count-Based Exploration with Neural Density Models". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by D. Precup and Y. W. Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 2721–2730.

Papamakarios, G., I. Murray, and T. Pavlakou (2017). "Masked Autoregressive Flow for Density Estimation". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. Ed. by I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, pp. 2338–2347.

Pathak, D., P. Agrawal, A. A. Efros, and T. Darrell (2017). "Curiosity-driven Exploration by Self-supervised Prediction". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by D. Precup and Y. W. Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, pp. 2778–2787.

Raffin, A. (2018). *RL Baselines Zoo*. `https://github.com/araffin/rl-baselines-zoo`.

Rezende, D. J. and S. Mohamed (2015). "Variational Inference with Normalizing Flows". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by F. R. Bach and D. M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, pp. 1530–1538.

Savinov, N., A. Raichuk, D. Vincent, R. Marinier, M. Pollefeys, T. P. Lillicrap, and S. Gelly (2019). "Episodic Curiosity through Reachability". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Schrittwieser, J., I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. P. Lillicrap, and D. Silver (2019). "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model". In: *CoRR* abs/1911.08265.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov (2017). "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347.

Sievers, L. (2020). *In-Hand Manipulation with a Torque Controlled Multi-Fingered Hand Using Deep Reinforcement Learning*.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nat.* 529.7587, pp. 484–489.

Silver, T., K. R. Allen, J. Tenenbaum, and L. P. Kaelbling (2018). "Residual Policy Learning". In: *CoRR* abs/1812.06298.

Sutton, R. S., D. A. McAllester, S. P. Singh, and Y. Mansour (1999). "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*. Ed. by S. A. Solla, T. K. Leen, and K. Müller. The MIT Press, pp. 1057–1063.

Tabak, E. G. and C. Turner (2013). "A Family of Nonparametric Density Estimation Algorithms". In: *Communications on Pure and Applied Mathematics* 66.2, pp. 145–164.

Tassa, Y., T. Erez, and E. Todorov (2012). "Synthesis and stabilization of complex behaviors through online trajectory optimization". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, October 7-12, 2012*. IEEE, pp. 4906–4913.

Toussaint, M. (2009). "Robot trajectory optimization using approximate inference". In: *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*. Ed. by A. P. Danyluk, L. Bottou, and M. L. Littman. Vol. 382. ACM International Conference Proceeding Series. ACM, pp. 1049–1056.

Trippe, B. and R. Turner (2018). *Conditional Density Estimation with Bayesian Normalising Flows*.

Wang, C. and K. Ross (2019). "Boosting Soft Actor-Critic: Emphasizing Recent Experience without Forgetting the Past". In: *CoRR* abs/1906.04009.