Bachelor Thesis in Computer Science

at Freie Universität Berlin

Biorobotics Working Group

Towards solving the RoboFish Leadership Problem with Deep Reinforcement Learning

Julian Stastny
julian.stastny@fu-berlin.de

Supervisor: Prof. Dr. Tim Landgraf

Berlin, 12.09.2019

Abstract

The RoboFish project investigates interactions of swarm agents by using a robot fish to interact with real fish. This work explores the use of Deep Reinforcement Learning to train a robot policy that is able to lead real agents. To this end, a simulation with handcrafted swarm agents as well as an appropiate observation and action space for the robot is set up. Then, Neural Networks with Convolutional layers and Long-Short Term Memory cells are trained to maximize reward via Proximal Policy Optimization. The reward function is defined such that it measures the ability to make agents follow the robot to different places in the environment. Domain Randomization is used on the simulation in order to obtain a policy that is robust against different agent dynamics. The method is evaluated on its ability to lead one agent in a randomized simulation where interestingly, the robot seems to display a form of meta learning by being able to dynamically adapt its policy by ad-hoc inference about the agent dynamics. Its success in randomized simulations provides reason to be optimistic about a transfer from simulated to real environment in future work.

Acknowledgements

This thesis would not exist without the support of the many people involved in the RoboFish project and the Biorobotics Lab in general. From the latter, I would like to thank Leon Sixt, Benjamin Wild and David Dormagen for helpful discussion and feedback as well as Moritz Maxeiner and Mathis Hocker for their help with hardware and other technical aspects of the thesis.

I also thank Jan Peters for hosting me at the IAS in Darmstadt for a very productive week. Thanks to Hany Abdulsamad, Boris Belousov, Carlo D'Eramo and Fabio Muratore from IAS for helpful discussion and ideas.

Lastly, many thanks go to Gregor Gebhardt and Tim Landgraf for their continuous support, feedback, their suggestions and the fact that they were an absolute pleasure to work with.

Contents

1 Introduction

RoboFish is a joint project between Freie Universität Berlin (FU), Humboldt-Universität zu Berlin (HU) and Leibniz-Institut für Gewässerökologie und Binnenfischerei (IGB) [Landgraf et al., 2013]. Its goal is to investigate swarm interactions of fish. To this end, it is not only of interest to observe the interactions of (incontrollable) real fish but also to introduce controllable agents in order to investigate the boundary conditions of swarm behavior.

One way to find out more about the dynamics of swarm behavior and especially the properties of leadership is to trying to lead a swarm of real agents, a challenge for which a water tank with a tracking system and a controller for robot fish has been set up [Moenck et al., 2018], allowing us to observe interactions between real and a realistic looking robot fish.

When it comes to the task of leading live fish, we have little knowledge of what is needed to make other fish accept the leader and how leadership can be retained. It is known that Guppies prefer to follow large leaders and that it is generally possible to lead real fish with a robot [Bierbach et al., 2018]. Previous work in the project also indicates that simple robot behavior based on handcrafted heuristics is sufficient to lead a fish in some instances[1]. However, it is not known whether a different, more complex behavior would have been successful in the instances where the handcrafted behavior was insufficient. Designing more sopisticated behaviors by hand would be quite complicated, especially for potential future experiments involving multiple fish, multiple robots or even other kinds of swarm agents. Past experience also indicates that autonomous learning is a more powerful family of approaches than the one of handcrafted design [Sutton, 2019].

For this reason, the following work investigates the possibility of using an artificial neural network, trained with Reinforcement Learning (RL) [Sutton and Barto, 1998], to control a robot fish to lead a single Guppy in the aforementioned tank.

[1] The corresponding paper is in preparation for submission.

2 Preliminaries

In traditional Supervised Learning, a function approximator is optimized to predict labels given features. For example, given pictures of animals and their corresponding species, a neural network can be trained to predict the species given just the picture. In a similar fashion, one could train a neural network to predict the actions of an agent like a fish given trajectories of real fish recordings. However, because we do not want to train the robot to just fit in but to actually lead the swarm, this approach has a fundamental problem: A neural network that optimizes for realistic behavior of some fish does not necessarily optimize for leadership.

RL, on the other hand, does not learn from labels, but from interactions with an environment. Instead of a regression or classification loss, the feedback consists of a reward by the environment. A neural network is then optimized such that its outputs correspond to actions that maximize reward. The key idea is that we can choose this reward to be such that high scores can only be achieved via leadership.

A challenging problem within the context of RL is that it is generally not very data efficient. Because of this, it is necessary to train the neural network in a simulation where data can be collected quickly. But since we care about our ability to lead real agents, we are then faced with the fact that most likely, the dynamics of the simulated environment are not the same as those of the real world. Therefore, a robot that is able to achieve high rewards in simulation might not be able to perform well in the task we actually care about, which introduces the problem of Sim-to-Real Transfer.

Subsection 2.1 gives a brief introduction to RL, while Subsection 2.2 then deals with our approach to Sim-to-Real Transfer.

2.1 Reinforcement Learning

The following paragraphs formalize the RL setting and introduce the algorithm I use in this work.

2.1.1 Markov Decision Processes

A Markov Decision Process (MDP) is a tuple $(\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R})$ where$\mathcal{S}$ is a space of states that an agent can be in, $\mathcal{A}$ is a space of possible actions that an agent can take. $\mathcal{P}$ refers to the transition function, which returns the next state s' for any action a in any state s. Finally, the reward function $\mathcal{R}$ assigns a reward for each transition from s to s'. A core property of MDPs is that transition likelihoods from a state s do not depend on the past trajectory of states and actions leading up to the agent being in state s.

While MDPs assume that agents have full observational access to $s \in S$, Partially Observable Markov Decision Processes (POMDP) relax this assumption. States are split into an observable and a hidden part. Since $\mathcal{P}$ still uses the full state, it is in the interest of the agent to learn a (possibly implicit) model of the hidden state. For notational simplicity, I will use $\mathcal{S}$ to refer to the observable part of the state, keeping in mind that actually, the full state involves a hidden part as well.

Given a (PO)MDP, consider the set of functions $\pi_\phi : \mathcal{S} \to \mathcal{A}$ parameterized by a neural network's weights ϕ. The goal in reinforcement learning is to maximize the return $\mathcal{R} = \sum_{t=1}^{n} \lambda^t r_t$ with $\lambda < 1$, which constitutes the discounted sum of rewards over an episode. The choice of reward function will be discussed in Section 4.

2.1.2 Proximal Policy Optimization

One commonly used family of RL algorithms is Policy Gradients (PG) [Sutton et al., 2000], where the aim is to optimize the parameters of the behavior, called policy, directly. Given a function approximator with parameters ϕ, PG algorithms search for a locally optimal policy $\pi_\phi : \mathcal{S} \to \mathcal{A}$ by gradient ascent on the objective $J(\phi) = E_t \left[\Psi_t \log \pi_\phi(a_t|s_t)\right]$, where in the most basic version, $\Psi_t = \mathcal{R}$. Since the reward function is not differentiable, the gradient with respect to J has to be estimated. This is done by sampling whole trajectories from π_ϕ and computing a gradient estimate $g = E_t \left[\Psi_t \nabla_\phi \log \pi_\phi(a_t|s_t)\right]$.

While unbiased, this naturally leads to high variance, which is why several techniqes for variance reduction have been proposed. One of them is to use an advantage function, setting $\Psi_t = \mathcal{A}^\pi(s_t, a_t)$. Using Generalized Advantage Estimation [Schulman et al., 2015b], this requires the estimation of the value function $V^\pi(s_t) = E_{\pi_\phi} \left[\sum_{l=0}^{n} r_{t+l}\right]$. $V^\pi(s_t)$ is estimated by a function approximator that may or may not use a subset of ϕ, called the Critic. The neural network parameterizing π is called the actor, and thus, the family of variance reduction techniques that use value estimates are called Actor-Critic [Sutton et al., 2000] methods.

Vanilla PG methods, even when augmented with a critic, suffer from instability during training. This is because small gradient steps in parameter space can result in big updates in policy space, leading us to a policy that might be much worse than the one before the gradient step. This can be irreversible, because the new policy might too degenerate to rediscover the aspects that made the policy from before better.

In order to avoid this, Trust Region Policy Optimization (TRPO) [Schulman et al., 2015a], a variant of PG, enforces a KL Divergence constraint between the old and the new policy. This avoids instability by limiting the step size

in policy space. In practice however, TRPO suffers from being relatively complicated to implement and from having to compute the KL Divergence, which - depending on the implementation - is either expensive or inaccurate.

An alternative to TRPO is Proximal Policy Optimization (PPO) [Schulman et al., 2017]. It achieves the same improvement to Vanilla PG in a simple way that does not require computing the KL Divergence. Let $pr_t(\phi) = \frac{\pi_\phi(a_t|s_t)}{\pi_{\phi_{old}}(a_t|s_t)}$ be the probability ratio of an action (given state) under the new and the old policy. The old policy is the one that produced the given trajectory. PPO optimizes the Clipped Surrogate Objective

$$J^{CLIP}(\phi) = E\left[\min(pr_t(\phi)A_t, \text{clip}(pr_t(\phi), 1-\epsilon, 1+\epsilon)A^t)\right], \tag{1}$$

where ϵ is a hyperparameter that determines how far we are willing to deviate from the old policy. This formulation has the effect of penalizing changes to the policy that move $pr_t(\phi)$ away from 1. On the other hand, by taking the minimum, PPO can recover from accidental changes to the policy into the (according to the advantage estimator) wrong direction. This enables us to make several gradient steps on the same trajectories, which improves data efficiency.

Algorithm 1 Proximal Policy Optimization

1: **while** training **do**
2: Run $\pi_{\phi_{old}}$ for T steps
3: Compute advantage estimates $A_1, .., A_T$
4: Optimize ϕ with respect to J^{CLIP} using K epochs of training
5: $\phi_{old} \leftarrow \phi$
6: **end while**

It should be noted that Deep Q-Networks (DQN) [Mnih et al., 2013] are also considered state of the art for several reinforcement learning problems. However, for RoboFish I only consider the PPO algorithm because DQN often fails on continuous control tasks. In our case, the action space of the robot fish consists of a turn-angle and the forward speed or boost, and is thus continuous.

2.2 Sim-to-Real Transfer

An optimal policy in a simulated environment can still fail when applied to the real world. This is because there might be important differences between the real and the simulated environment dynamics that are not taken into account by a policy that was just trained in silico.

A possible solution to this problem is Domain Randomization [Tobin et al., 2017], which converts the simulation into a distribution of simulations by

randomizing its parameters. The agent is then trained on the whole distribution.

If a policy works on a whole family of environments, this makes it more likely to work in the real environment, provided that it is related to this family closely enough. An interesting question is whether this results in a policy that is merely robust or dynamically adaptive to the instances of the family of the environment. The latter would imply that some kind of meta-learning is taking place. Recent work, where Domain Randomization was used in a difficult Sim-to-Real task, suggests that the latter is actually the case, since long-short term memory networks (LSTM) were needed despite the task being an MDP [OpenAI et al., 2018]. Because MDPs are memoryless, meaning that future states conditional on both past and present states depends only on the present state, this implies that the policy needs the LSTM memory to store estimates about the true nature of the environment that it is running in.

3 Problem Statement

In general, the RoboFish project has the goal of finding policies for interactions between robots and real agents. In this work, I focus on leadership as the interaction of interest, but the approach can be extended to other kinds of interactions by setting the reward function such that it returns high values for the desired behavior.

In order to use RL, I use a simulation to learn the policy, which is then evaluated in the real world. In the case of RoboFish, the real world is a rectangular water tank with a tracking system that extracts the positions of all agents in a time step. It also includes a realistic looking robot which can be controlled by the policy. By using ray casting (RC), the approach is made agnostic to whether observations are made from a bird's eye view, as in the case of RoboFish, or directly from the perspective of the robot, e.g. by having sensors installed on the robot. The technical setup of the real world water tank is described in the next section.

The following sections are concerned with the problem of finding a good reward function that leads to the desired interactions, training a policy in a simulated environment and successfully transferring the policy to the real world. I evaluate the potential of our approach both via its ability to create policies that make a robot lead a Guppy in the aforementioned water tank as well as its ability to lead different kinds of agents in simulation.

4 Environment Setup

Here I specify the setup of the experiment. In RL, this mainly involves the definition of an observation space, an action space and a reward function. They are designed to be relatively independent of the technical setup of the real environment, which is described a the beginning of this section. The end of this section deals with the design of the simulated agents, which are the core component of the training environment. OpenAI Gym was used for the simulated environment.[2]

4.1 Real environment

The water tank is rectangular and of size 1×1 meter. It has two cameras attached, one on top and one below the tank. The top-view camera records the movement of the fishes, while the bottom-view camera tracks the location of the controllable robot fish. The robot fish consists of a 3D-printed realistic looking model of a Guppy and is magnetically attached to a wheeled robot below the tank. The so-called RoboTracker preprocesses each camera frame and puts the result into a behavior module, which outputs desired actions. The RoboTracker interprets these as radians for the turn and centimenters for the movement forward and postprocesses them into low-level control commands that are sent to the wheeled robot. It is possible to set the frequency in which the robot can choose new actions.

In general, my approach is designed to work with tracking setups where the following interface is implemented.

Algorithm 2 Evaluation on a real world environment

```
1: while experiment is running do
2:     Capture the current coordinates of agents or view of robot;
3:     Given the observation, compute the next action;
4:     Send the action to the robot;
5: end while
```

[2] The Guppy Gym environment was jointly developed by Gregor Gebhardt (GG) and me. I developed an initial prototype based on a simulation by Moritz Maxeiner. Based on the prototype and past experience with Gym, GG developed a second version that is easier to extend with different types of simulated fish. Based on the second version, I developed sub-environments for the leadership task with different rewards and several wrappers for the observations and actions. The ray casting wrapper was jointly implemented by GG and me.

4.2 Observation and action space

The RoboTracker is able to extract the coordinates and angles of agents in the tank , which is one option for the input into the neural network. However, I opt for a ray casting representation instead. This way, the robot is provided with similar observations as the live fish, which might be more likely to lead to a biologically plausible leadership policy. It also makes it possible to use Convolutional Neural Networks (CNN), which are partly inspired by the visual cortex of the brain [Hubel and Wiesel, 1968].

Ray casting divides the field of perception of the robot into b equally sized bins. Every bin consists of a 2-tuple of the proximity to the tank's wall and the proximity to the nearest agent in that bin, if there is one. Using the length of the diagonal of the quadratic tank, the proximities are scaled into the range $[0, 1]$. The observations get represented as a matrix $o \in [0, 1]^{2\times b}$. Because this representation risks losing important information when two or more fish are in the same bin, I build something analogous to a short-term memory by actually feeding the network with the last k matrices, where I drop the proximities to the wall in the old matrices as they do not matter. The neural network input then becomes $s \in [0, 1]^{(k+1)\times b}$, which practically contains the same amount of information as before, but in a way that can be biologically justified.

Ray casting also has several technical advantages.

1. The input shape does not depend on the number of fish in the tank, which makes the representation easier to scale and to evaluate on the transferability of policies learned on n agents to scenarios with m agents.

2. It adds inductive bias, because the network does not have to learn equivariances between the locations of the tank. For example, when the robot is in the center, pointed at the fish that is in the upper-left corner, this situation is equivalent to the fish being in any other corner. With RC, this equivalence is built in by default and should increase data efficiency when learning.

3. Using CNNs reduces the number of parameters, which generally reduces overfitting.

4. It uses only the coordinates and does not need additional information like IDs or angles, which makes the approach more general.

It should be noted that there exist other options that have some of these advantages. Using polar coordinates relative to the robot's pose would be

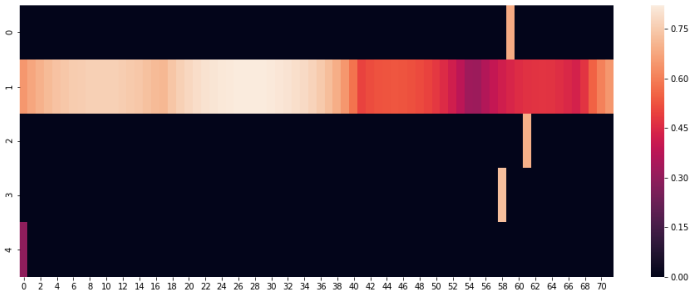

Figure 1: An example ray casting input for the Neural Network. Row 0 is the agent at time t, row 1 the walls at time t, rows 2 and 3 the agent at times t-1 and t-2. Row 4 is the view of the current goal corner, only used when using the Corner Reward (see Section 4.4).

slightly more precise, but lose both biological plausibility and the shape invariance to the number of fish. Another option would be a learned swarm embedding as in Gebhardt et al. [2018]. However, since I empirically found the RC representation to work well, I avoid the introduction of this additional layer of complexity in this work. Note that the corner reward, introduced in Section 4.4, changes the input slightly.

The action space is defined as continuous and two-dimensional. The first dimension is interpreted as the desired turn, the second as a movement forward. This has the advantage of being general and relative to the agent's position. The maximum turn and speed are adjusted to the hardware limitations given by the real robot, which is lower than that of the real and simulated fish.

4.3 Simulation design

Couzin et al. [2002] introduced a simple model of collective animal behavior. It assumes the existence of three concentric zones around every agent in a swarm: The Zone of Repulsion (ZOR), Zone of Orientation (ZOO) and Zone of Attraction (ZOA). In sum, they consititute the Zone of Interaction (ZOI). The agent is repelled by close neighbors, aligns with others in an intermediate distance and approaches those that appear in its zone of attraction. While the ordering of the zones is fixed, the zone sizes are hyperparameters. A relevant factor for the difficulty of the RL problem is that fish with big ZORs are more difficult to lead than ones with small ZORs. In the original Couzin model, the agents move with constant velocity.

The Couzin model can be extended in ways that are plausible with respect

to past observations of Guppy behavior.[3] While the first two extensions are specific to guppies, we expect the last one to be generalizable to other kinds of agents.

1. The constant velocity is replaced by randomly sampled forward bursts. This more closely resembles the movement style displayed by Guppies.

2. A bias against moving towards the center of the tank can be introduced, which is especially useful when it comes to training the robot to maximize r^{center} in the real world.

3. The zone sizes are made dynamic. When an agent A's ZOR is entered by some agent B, this increases the relative size of A's ZOR towards B. Over time, the zones turn back to their default as long as the ZOR is not entered again.

Based on this agent, which I call the BiasedAdaptiveCouzin (BAC), I randomize the size of the ZOR, ZOO and ZOA as well as the growth and shrink factors of the adaptive ZOR and the magnitude of the bias against the center. This constitutes the RandomizedBAC. The parameters are sampled from (clipped) normal and half-normal distributions that were chosen heuristically. See Appendix B for details.

An episode consists of a fixed number of steps. At the start of an episode, the location of fish and robot are sampled. The robot location is sampled from $\mathcal{N}(\mathrm{c}_{\mathrm{center}},\sigma^2)$, where c_{center} refers to the coordinates of the tank's center and σ is a small value, resulting in the robot being very close to the center at the beginning. The agent location is sampled like this: First, one of the four corners is sampled uniformly. Then, the horizontal and vertical position relative to the corner is sampled from two independent zero-mean Gaussians. Simply put, the agent is sampled close to one of the corners, which matches realistic starting positions of fish and prevents the robot from being in the ZOR before it can bas moved.

[3] They were jointly developed by GG and me, then implemented by the former.

Algorithm 3 RandomizedBAC(BAC)

function ADAPTIVEZONES(adaptiveZoneFactor, zoneRadius, zooFactor)
    zor ← zoneRadius · adaptiveZoneFactor
    zoo ← (zoneRadius - zor) · adaptiveZoneFactor * zooFactor
    zoa ← zoneRadius - zoo - zor
    return zor, zoo, zoa
end function

function INITIALIZE
    BAC.initialize()
    sample: adaptiveZoneFactor, zoneRadius, zooFactor, f_{shrink}, f_{growth}, b_{center}
end function

function ACTION(robot, c_{self})
    zor, zoo, zoa ← adaptiveZones(adaptiveZoneFactor, zoneRadius, zooFactor)
    a ← BAC.action(robot, c_{self}, zor, zoo, zoa)
    a ← action + BAC.bias(b_{center}, $||c_{self} - c_{center}||_2$)
    if robot in zor **then**
        adaptiveZoneFactor ← adaptiveZoneFactor · f_{growth}
    else
        adaptiveZoneFactor ← adaptiveZoneFactor · f_{shrink}
    end if
    return a
end function

4.4 Reward function

Defining reward functions is generally considered a challenging aspect of RL [Irpan, 2018]. This is because broadly because of two reasons:

1. Reward Overfitting: RL agents are trained to maximize reward and usually have no prior knowledge about the behavior indended by the designer. Therefore, if the reward does not capture *exactly* what we want, it will most likely lead to undesired behavior.

2. Local optima: At any point during training, a RL agent has to make a trade-off between taking actions that maximize reward given current knowledge, and taking actions that gain more knowledge. The more reward has to be sacrificed for new knowledge, the more likely the agent is to stay in its current local optimum. Therefore, to formulate it in terms of the optimization landscape, a good reward is designed such that it does not have deep and/or long "valleys" between local optima and the global optimum.

When it comes to leading Couzin-like agents, an additional problem is introduced: The distance in policy space between good policies and bad policies might be very short, because the ZOR is right next to the ZOO.

A state $s_t \in S$ consists of the coordinates $c_{t,i}$ of agents identified by IDs i at time step t. Let 0 be the robot's ID. In order to capture our intuitive notion of leadership, it is necessary to specify a formal reward function where high values can only be achieved by leading the agents. I define several reward functions to study their effects on the learned leadership behavior and the stability of convergence. They can roughly be divided into *direct* and *indirect* measures of leadership as well as *auxiliary* rewards that only have the purpose of stablizing training.

One option for a direct reward function is to observe whether the agents follow the robot when moving. Let $\mathbf{f}_{t,i} = c_{t+1,i} - c_{t,i}$ be the moving vector of agent i from time t to $t+1$ and let $\mathbf{g}_{t,i} = c_{t,robot} - c_{t,i}$, which is the location vector from agent i to the robot at time t. Intuitively, between any t and $t+1$, we would like the agent to move towards the robot, meaning that we want the angle between $\mathbf{f}_{t,i}$ and $\mathbf{g}_{t,i}$ to be as small as possible. This would be captured by the cosine similarity of the two terms. Since a plausible notion of following also takes into account the magnitude of the movement towards the robot, the cosine similarity is scaled by the norm of the agent's movement vector.

As a result, I define the reward function[4]

[4]This reward function is used as an evaluation metric in a paper from the RoboFish project that is currently in preparation for submission.

$$r_t^{follow} = \frac{1}{M}\sum_{i=1}^{M} \frac{\mathbf{f}_{t,i}^{\mathsf{T}}\mathbf{g}_{t,i}}{||\mathbf{g}_{t,i}||_2 \cdot ||\mathbf{f}_{t,i}||_2} \cdot ||\mathbf{f}_{t,i}||_2 = \frac{1}{M}\sum_{i=1}^{M} \frac{\mathbf{f}_{t,i}^{\mathsf{T}}\mathbf{g}_{t,i}}{||\mathbf{g}_{t,i}||_2}. \tag{2}$$

I also consider the reward $r_t^{follow\geq 0} = \max(r_t^{follow}, 0)$, which clips negative rewards to 0. The rationale behind this, which is confirmed in experiments, is that this stabilizes training at the start by avoiding the penalization of explorative actions that risk repulsing the agents.

One option for an indirect reward funtion is to reward the agent for the fish being close to the center of the tank. This would indirectly measure leadership because past observations show that the guppies prefer not to be in the center of the tank. Formally, define

$$r_t^{center} = \frac{diag}{2} - \frac{1}{M}\sum_{i=1}^{M} ||c_{t,i} - c_{center}||_2 \ , \tag{3}$$

where M refers to the number of agents and $||\cdot||_2$ to the euclidean norm. *diag* refers to the tank's diagonal length and is used to keep the reward positive.

As an alternative to the center-reward, I introduce a reward function that measures whether the robot is able to lead the agents from one corner to the opposite corner of the tank. Since it depends on a state, it is easiest expressed in terms of an algorithm. Let c_{llc} and c_{urc} be the coordinates of the tank's lower left corner and upper right corner.

Algorithm 4 Corner Reward

Require: hyperparameter ϵ, goal = random.choice(c_{llc}, c_{urc})

function r_t^{corner}
  $r_{candidate} \leftarrow \frac{1}{M}\sum_{i=1}^{M} \mathbf{g}_{t,i}$
  if goal = c_{urc} **then**
    $r \leftarrow \max(r_{candidate}, 0)$
    if $\frac{1}{M}\sum_{i=1}^{M} ||c_{t,i} - c_{urc}||_2 \leq \epsilon$ **then**
      goal $\leftarrow c_{llc}$
    end if
  else
    $r \leftarrow \max(-r_{candidate}, 0)$
    if $\frac{1}{M}\sum_{i=1}^{M} ||c_{t,i} - c_{llc}||_2 \leq \epsilon$ **then**
      goal $\leftarrow c_{urc}$
    end if
  end if
  return r
end function

In a nutshell, we randomly choose one of the corners as the goal at the beginning of the episode. After every step, the robots gets reward if the agents got closer to the goal. Whenever the agents are within some average distance ϵ to the goal, the goal switches to the opposite corner. Beyond being a measure of whether the robot is able to lead the agents, the reward can also be interesting as a measure for how difficult it is to lead the agents through the shortest path between corners, which includes the center. When using r^{center}, the current goal is indicated in the robot's observation by a ray casting to the current goal corner.

I also define several auxiliary reward-extensions that can be wrapped around the base rewards.

One is

$$r^{wall-avoidance}(r_t) = \begin{cases} 0 & d^{wall}(c_{t,robot}) \leq \epsilon \\ r_t & else \end{cases}, \tag{4}$$

where ϵ is a hyperparameter and $d^{wall}(c_{t,robot})$ refers to the smallest distance of the robot to the wall at time t. It is used for the practical reason of avoiding crashes of the real robot into the wall.

One measure I take to stabilize training is to use

$$r_t^{clipped-proximity-bonus}(r_t) = r_t \cdot (1 + \frac{1}{M} \sum_{i=1}^{M} ZOI(i) - \max(||c_{t,i} - c_{t,robot}||_2,\, ZOI(i))). \tag{5}$$

By using the proximity bonus, we can incentivize the robot to interact with the agents. This is necessary because especially early in training, interactions are likely to result in repulsion, which usually leads to lower rewards than not interacting at all. This often results in the locally optimal policy of avoiding interactions. I clip this bonus with the Zone of Interaction to avoid influencing the policy further after the robot has entered it. The reason for multiplication with r_t instead of using the arguably more standard choice of addition is that for the latter, it would probably be necessary to weight the bonus with another hyperparameter that depends on the choice of non-auxiliary reward.

Lastly, I introduce

$$r^{repulsion}(r_t) = \begin{cases} 0 & \exists i : ||\mathbf{g}_{t,i}|| \leq ZOR(i) \\ r_t & else \end{cases}. \tag{6}$$

This also has stabilizing effects on the training as it avoids locally optimal policies that rely on chasing the agents or shepherding, given that those are

undesired. Note that theoretically, this reward makes stronger assumptions about the structure of the real agent's behavior, since it assumes the existence of a repulsion zone. However, this can be justified by the fact that especially after using Domain Randomization on the zone size, the robot agent has no way of knowing whether it is in the ZOR except via observing whether the agent has repulsed behavior. I would postulate that online adaptation of the policy on the basis that the agents show signs of repulsion is desirable for the interaction with real agents as well.

The main reward can then be composed from the other rewards for example as

$$r_t^{main} = r_t^{repulsion} \circ r_t^{wall-avoidance} \circ r_t^{clipped-proxmity-bonus} \circ (r_t^{follow \geq 0} \cdot r_t^{corner}), \tag{7}$$

where direct rewards such as $r_t^{follow \geq 0}$ can be used as training stabilizers as well.

The approaches to reward function design are compared in Section 6.

5 Agent Setup

In order to train the policy, we need to specify its parameterization and a training algorithm. I use PPO as the algorithm for the reasons outlined in Section 2. The following paragraphs are thus focused on the the neural network that parameterizes the policy.

It can be assumed that most interesting agents, like Guppies, have memory. This means that we do not have full access to the state, which makes the problem a POMDP. This is one reason why I include LSTM-cells in the network, with the other being the usage of Domain Randomization. Those can be used to store implicit estimates about the hidden part of the state.

The ray casting input first gets processed by a stack of 1D-Convolutions. When using a 360° view, I use circular padding instead of zero padding. The resulting features get flattened, then they go through a stack of LSTM-cells surrouded by dense layers. I use ReLU as the activation function.

In the case of the actor, the output is μ_{turn} and $\mu_{forward}$. In order to squash the output into the range of possible actions, I use a scaled tanh activation. During data collection, standard deviations σ_{turn} and $\sigma_{forward}$ are used to sample the action from Gaussians. Note that the standard deviations are learned, but they do not depend on the input. During evaluation, the two μ are used as actions directly, because sampling is only used for expoloration. The Critic outputs a single scalar representing the value of the input state.

I use TensorFlow [Abadi et al., 2016] and TFAgents[Guadarrama et al., 2018] for implementation.

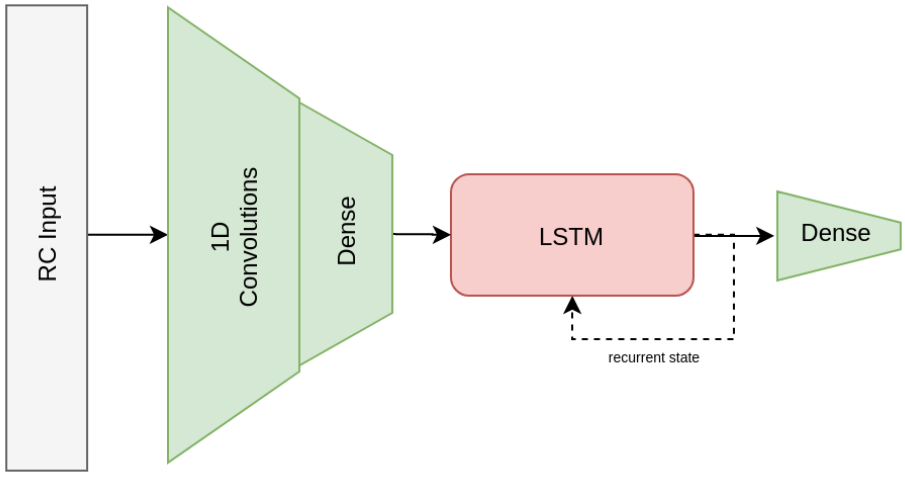

Figure 2: The basic architecture of both actor and critic. They do not share weights.

6 Evaluation

I evaluate the success of our method in simulation. Evaluating in simulation has the advantage that informative statistics can be allocated on a large scale by running the trained model on different environment configurations. The evaluation on the real environment will be done in future work.[5]

6.1 Reward function design

The main hyperparameter I investigated was the effect of different reward functions. As indicated in Section 4.4, this consumed a significant fraction of the invested time. It turned out that when just optimizing one of the indirect rewards, the robot tends to learn a kind of shepherding behavior, where it deliberately enters the ZOR to repulse the agent towards the center or, respectively, into the corner. When just optimizing r_t^{follow}, the robot tends to avoid interaction, which is probably caused by accidental repulsions early in training. The problem here is that this local optimum is very hard to escape because interaction is necessary to explore better policies. When using $r_t^{follow \geq 0}$, I encountered the issue that a policy where the agent gets chased is a more likely to reach local optimum than learning to lead the agent. This problem gets amplified as Domain Randomization, which increases the difficulty of leading but not that of chasing, is introduced. It should be noted that one possible solution for this would be to change the Couzin dynamics such that repulsion is more difficult to handle by the robot. Instead however, I added $r_t^{repulsion}$ as auxiliary reward, which solved the problem. At this point however, due to the penalization of repulsion, there reappears a reason for not interacting with the agent, which is why I add the clipped proximity bonus.

In general, if the robot is able to lead the agent to a predefined place (like the center or corner), this is a stronger signal for leadership abilities than a pure follow-reward. Actually, a policy that is able to maximize the follow-reward but not able to lead the agent to any specific place should make us very suspicious. Therefore, I evaluate on a policy trained to maximize the reward from Equation 7.

[5] At the time of writing this thesis, a hardware problem (high and varying latency between the RoboTracker and the wheeled robot) prevented a proper evaluation in the real environment. The model that I analyze in Section 6.2 and 6.3 was not able to transfer to the real world, probably due to the latency problem. Since the interesting part of this work lies in the transfer to different *fish dynamics*, evaluation in simulation, where transfer to different *robot dynamics* is not necessary, is still very informative as a prior for whether the approach can transfer to the real world.

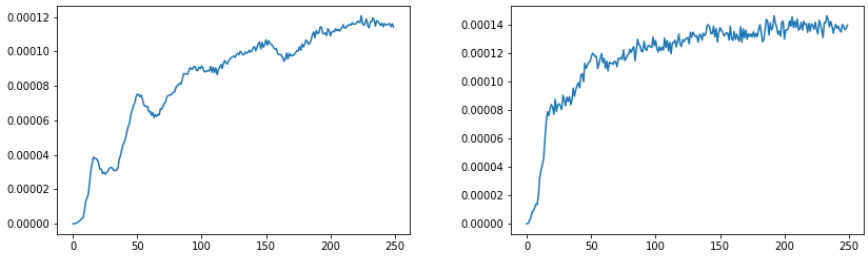

Figure 3: Average reward per step on the left, standard deviation on the right.

6.2 Performance in simulation

Looking at rendered videos of episodes, I find that the robot is able to consistently lead the randomized Couzin agent from corner to corner. Importantly, it is able to adapt to relatively strong variation in zone size and other randomization parameters. For a subjective assessment of these results, I have uploaded exemplary videos[6].

I evaluate the robot quantitatively as well. For this, the saved policy is run it in a simulation for 10000 episodes. The initial state of robot and agent are drawn from the same distribution as in training, but the same initial state is used for all episodes to ensure comparability.[7] At the start of every episode, the parameters of the randomized agent are sampled from the same distribution as in training.

Figure 3 shows the average reward and standard deviation per step. I would interpret the initial rise of reward within the first ˜25 steps as the reward gained by the robot increasing its proximity to the agent. It can be observed that after over time, reward continuously rises, which indicates that as the robot learns more about the agent's dynamics, it gets better at leading, leading to higher rewards.The standard deviation also rises over time, but seems to converge after ˜150 steps, which could imply that the rewards converge to the potential optimum given randomization parameters and trained policy. Appendix A shows more plots for means and standard deviations for different tracked metrics that are not included here because they did not lead to more conclusive results. Evaluating a more stable policy might be more informative.

[6] https://cutt.ly/kwFE5gm

[7] Although I did not quantify this systematically, anecdotal evidence suggests that the policy generalizes to out-of-distribution starting states.

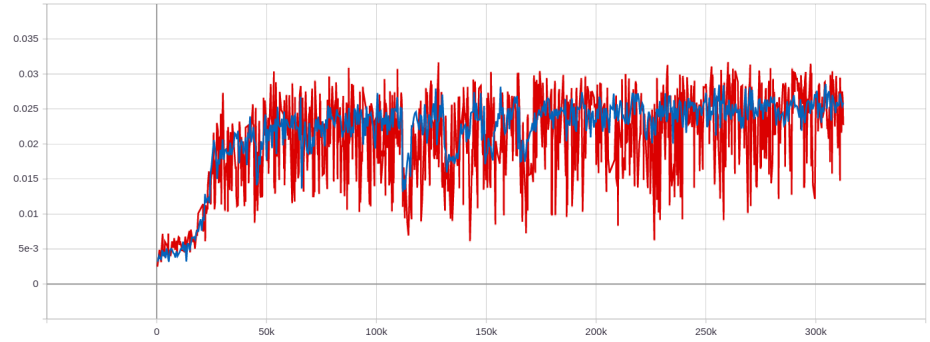

Figure 4: The return over the course of training. Blue is the average performance of the policy during evaluation, red during training. The high variance indicates that training can be improved with better hyperparameters.

6.3 Influence of Domain Randomization

In order to get a better picture of the policy performance, I also track r_t^{follow}, the distance between robot and agent as well as the size of the agent's couzin zones per episode step. The most interesting predictor for the potential of successful Sim-to-Real transfer is the influence of different domain randomization parameters on the return. I quantify their influence by using ordinary least squares (OLS) to predict the return, using the standardized randomization parameters as features. It achieves a R^2 score of 0.2, which implies that a linear view on the randomization parameters can explain 20% of the variance between returns. The noisy behavior of the guppies and mistakes made by the policy probably account for a lot of variance too. The existence of the latter can be confirmed anecdotally by rendering videos and is probably the result of a lack of hyperparameter search at the time of writing the thesis. Table 1 shows the weights computed by OLS.[8] Because the zone radius is the only parameter where high values make the task easier, the signs of the weights are not surprising.

Figure 5 shows the distances between robot and agent for several randomly chosen runs, which provides useful insight in the workings of the trained policy. The lowest leftmost plot ([0.26, 1.25, 0.96]), as one example, demonstrates the ability of policy adaptation even with relatively adverse domain randomization parameters. Further takeaways will be discussed in Section 7, while more plots can be found in Appendix A.

[8] Due to standard error being very close to zero for every weight, it is not necessary to use the t statistic as indicator of parameter importance.

6.3 *Influence of Domain Randomization*

	inter-cept	initial zone factor	grow factor	shrink factor	zone radius	ZOO factor	bias against center
weights (rounded)	0.021	-0.0006	-0.0021	-0.0025	0.0053	-0.0001	-0.0007
relative importance (%)	-	5.25	18.85	22.13	46.67	0.76	6.34

Table 1: Weights computed by OLS

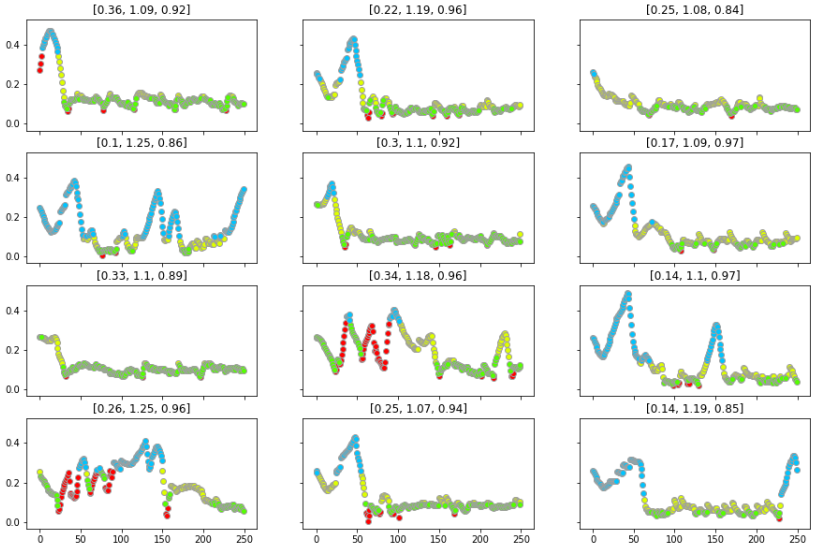

Figure 5: Distance between robot and agent over time for several randomly chosen runs. The colors indicate in which zone the agent is at a given step, where Red≡ZOR, Green≡ZOO, Yellow≡ZOA and Blue≡Outside of ZOI. The numbers above each plot indicate [zone radius, grow factor, shrink factor].

7 Future Work

This thesis should be understood as a preliminary groundwork for many interesting topics surrounding the RoboFish project and the science of swarm interactions in general. The first step in future work should be to evaluate the method on real agents. Another exciting avenue is to try leading several agents. Since that would be a much more complicated task, this will pose many technical challenges. Fortunately, there is a lot of improvement for the approach described in this thesis, which will be described in the following subsections.

Hyperparameter Search

There are many hyperparameters when it comes to this work, most of which had to be neglected due to time constraints. One is the set of hyperparameters surrounding network architecture, where the Convolutions, LSTM-cells and dense layers introduce a lot of space for hyperparameter optimization. Furthermore, PPO seems to be relatively sensitive to hyperparameter settings as well [Henderson et al., 2017, Ilyas et al., 2018]. At the time of writing, many training plots look like Figure X, where at some point during training, performance suddenly breaks down. Ilyas et al. [2018] suggest that learning rate annealing might be a possibility to avoid this phenomenon.

When it comes to Domain Randomization, it would be interesting to explore the space of randomizations and investigate how much randomization the policy can handle, and whether scaling up the LSTM-architecture can lead to more flexibility towards stronger randomization. This would be very useful for creating policies that are robust against different kinds of real world behaviors. As mentioned in Section 6.1, it probably makes sense to change some aspects of the Couzin model's repulsion dynamics, which would not only decrease the likelihood of local optima based on repulsing the agents, but also make the handcrafted model more realistic.

Learned Domain (Randomization)

One approach toward Sim-To-Real transfer is to do Domain Adaptation (DA), which is the idea of adapting the simulated environment to be more similar to the real world [Weng, 2019]. This can be done by imitation learning with (recurrent) neural networks [Eyjolfsdottir et al., 2016] on the trajectories of real agents, which is an approach taken by related work in the RoboFish project. Until recently, a lack of data made it unlikely for imitation learned models to be realistic enough to be useful as a simulation domain, which made the handcrafted Adaptive Couzin agents necessary.

When it comes to training on imitation learned agents parameterized by neural networks, a promising approach that automatically introduces Domain Randomization, is to represent the neural network weights as distributions instead of point-estimates. The most straightforward variant would be to use independent gaussian distributions, thus representing the parameters with a mean vector $\boldsymbol{\mu}$ and a diagonal covariance matrix $\boldsymbol{\sigma^2}$. In Blundell et al. [2015], so-called Bayesian Neural Networks (BNN) are trained to maximize likelihood while minimizing the KL-Divergence between a fixed prior and the posterior values of the parameters. By using BNNs, we can incorporate epistemic uncertainty into the simulation, where due to the minimization of KL-divergence, some weights get sampled with higher and some with lower variance. Higher uncertainty about a certain aspect thus translates into stronger randomization of that aspect of the dynamics, which means that if it works, the approach is a form of intelligent Domain Randomization without the need for hyperparameters.

Another barrier that currently prevents the usage of imitation learned models is the fact that only a subset of the collected data is informative about leadership dynamics, while the majority is just fish swimming around. It might also be a problem that the data is only from interactions between real fish and not between robot and fish. One way to do targeted exploration of leadership dynamics might be to use a trained policy on a real robot and collect the data from interactions between this robot and real fish. One way to jumpstart this procedure could be to first train the policy in a simulation of Couzin agents like those described in this thesis, and then to finetune on real agents.

Meta Learning

The previous point about finetuning is strongly related to the idea of Meta Learning. As mentioned in Section 2.2, when doing DR on a sufficiently complicated domain, it is not sufficient to find a static policy that is merely robust against randomization; it might be necessary to have a dynamic policy that can adapt online. A recurrent neural network such as an LSTM can be sufficiently dynamic due to its changeable recurrent state, but if not, the next step would be to adapt the weights of the network. (Model Agnostic) Meta Learning [Finn et al., 2017] does just that and is thus a potential avenue to explore. It would also be interesting to explore the connections between the kind of Meta Learning done by recurrent neural networks and the gradient based Meta Learning that is model agnostic.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf. 5

David Bierbach, Hauke Juergen Moenck, Juliane Lukas, Marie Habedank, Pawel Romanczuk, Tim Landgraf, and Jens Krause. Guppies prefer to follow large (robot) leaders irrespective of own size. 2018. doi: 10.1101/320911. URL http://biorxiv.org/lookup/doi/10.1101/320911. 1

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight Uncertainty in Neural Networks. *arXiv:1505.05424 [cs, stat]*, May 2015. URL http://arxiv.org/abs/1505.05424. arXiv: 1505.05424. 7

IAIN D. Couzin, JENS KRAUSE, RICHARD JAMES, GRAEME D. RUXTON, and NIGEL R. FRANKS. Collective Memory and Spatial Sorting in Animal Groups. *Journal of Theoretical Biology*, 218(1):1 – 11, 2002. ISSN 0022-5193. doi: https://doi.org/10.1006/jtbi.2002.3065. URL http://www.sciencedirect.com/science/article/pii/S0022519302930651. 4.3

Eyrun Eyjolfsdottir, Kristin Branson, Yisong Yue, and Pietro Perona. Learning recurrent representations for hierarchical behavior modeling. *arXiv:1611.00094 [cs]*, October 2016. URL http://arxiv.org/abs/1611.00094. arXiv: 1611.00094. 7

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. *arXiv:1703.03400 [cs]*, March 2017. URL http://arxiv.org/abs/1703.03400. arXiv: 1703.03400. 7

Gregor H.W. Gebhardt, Kevin Daun, Marius Schnaubelt, and Gerhard Neumann. Learning Robust Policies for Object Manipulation with Robot Swarms. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7688–7695, Brisbane, QLD, May 2018. IEEE. ISBN 978-1-5386-3081-5. doi: 10.1109/ICRA.2018.8463215. URL https://ieeexplore.ieee.org/document/8463215/. 4.2

Sergio Guadarrama, Ethan Holly Sam Fishman Ke Wang Ekaterina Gonina Neal Wu Chris Harris Vincent Vanhoucke Eugene Brevdo Oscar Ramirez, Pablo Castro, and Anoop Korattikara. *TF-Agents: A library for Reinforcement Learning in TensorFlow*. 2018. URL https://github.com/tensorflow/agents. 5

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep Reinforcement Learning that Matters. *arXiv:1709.06560 [cs, stat]*, September 2017. URL http://arxiv.org/abs/1709.06560. arXiv: 1709.06560. 7

D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 195(1):215–243, March 1968. ISSN 0022-3751. doi: 10.1113/jphysiol.1968.sp008455. 4.2

Andrew Ilyas, Logan Engstrom, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Are Deep Policy Gradient Algorithms Truly Policy Gradient Algorithms? *arXiv:1811.02553 [cs, stat]*, November 2018. URL http://arxiv.org/abs/1811.02553. arXiv: 1811.02553. 7

Alex Irpan. *Deep Reinforcement Learning Doesn't Work Yet*. 2018. URL https://www.alexirpan.com/2018/02/14/rl-hard.html. 4.4

Tim Landgraf, Hai Nguyen, Stefan Forgo, Jan Schneider, Joseph SchrÃ¶er, Christoph KrÃŒger, Henrik Matzke, Romain O. ClÃ©ment, Jens Krause, and RaÃºl Rojas. Interactive robotic fish for the analysis of swarm behavior. In *International Conference in Swarm Intelligence*, pages 1–10. Springer Berlin Heidelberg, 2013. URL http://link.springer.com/chapter/10.1007/978-3-642-38703-6_1. 1

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs]*, December 2013. URL http://arxiv.org/abs/1312.5602. arXiv: 1312.5602. 2.1.2

Hauke JÃŒrgen Moenck, Andreas JÃ¶rg, Tobias von Falkenhausen, Julian Tanke, Benjamin Wild, David Dormagen, Jonas Piotrowski, Claudia Winklmayr, David Bierbach, and Tim Landgraf. BioTracker: An Open-Source Computer Vision Framework for Visual Animal Tracking. *arXiv preprint arXiv:1803.07985*, 2018. 00001. 1

OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning Dexterous

In-Hand Manipulation. *arXiv:1808.00177 [cs, stat]*, August 2018. URL http://arxiv.org/abs/1808.00177. arXiv: 1808.00177. 2.2

John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization. *arXiv:1502.05477 [cs]*, February 2015a. URL http://arxiv.org/abs/1502.05477. arXiv: 1502.05477. 2.1.2

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv:1506.02438 [cs]*, June 2015b. URL http://arxiv.org/abs/1506.02438. arXiv: 1506.02438. 2.1.2

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, July 2017. URL http://arxiv.org/abs/1707.06347. arXiv: 1707.06347. 2.1.2

Richard S Sutton. The Bitter Lesson, March 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html. 1

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0-262-19398-1. 1

Richard S Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In S. A. Solla, T. K. Leen, and K. MÃŒller, editors, *Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press, 2000. URL http://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximati pdf. 2.1.2

Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World. *arXiv:1703.06907 [cs]*, March 2017. URL http://arxiv.org/abs/1703.06907. arXiv: 1703.06907. 2.2

Lilian Weng. Domain Randomization for Sim2real Transfer. *lilianweng.github.io/lil-log*, 2019. URL http://lilianweng.github.io/lil-log/2019/05/04/domain-randomization.html. 7

Appendix A

All of the following plots were created with the policy used in Section 6.2.

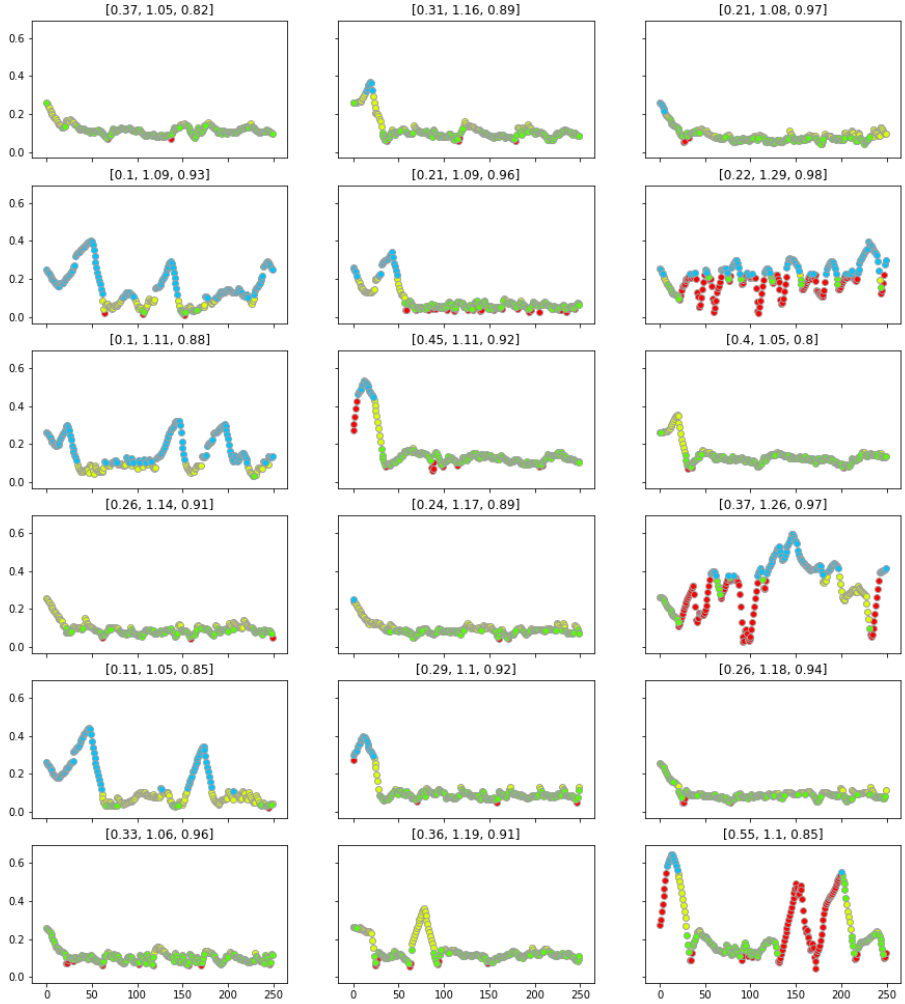

Figure 6: From left to right, average size of ZOR, ZOO and ZOA in the first row, standard deviation in the second row.

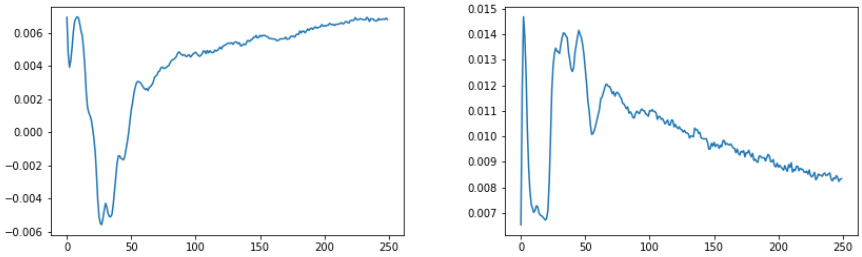

Figure 7: Average r_t^{follow} on the left, standard deviation on the right.

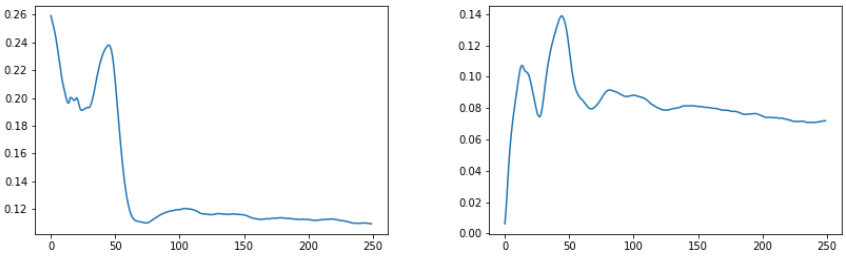

Figure 8: Average distance to agent on the left, standard deviation on the right.

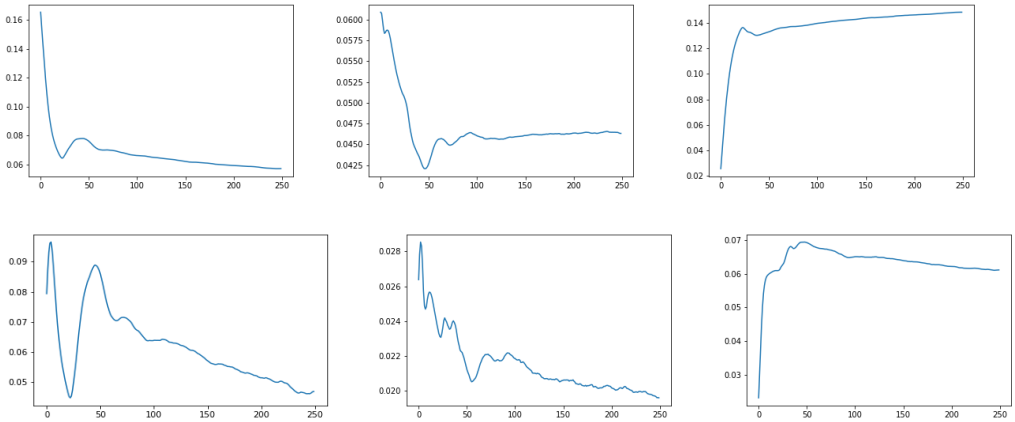

Figure 9: From left to right, average size of ZOR, ZOO and ZOA in the first row, standard deviation in the second row.

Appendix B

Here is the code for the different parts of this thesis:

Guppy Gym:

https://git.imp.fu-berlin.de/bioroboticslab/robofish/gym-guppy

Leadership Gym Enviroment:

https://git.imp.fu-berlin.de/bioroboticslab/robofish/rl/blob/master/envs/_leader_guppy_env.py

Randomized Guppy:

https://git.imp.fu-berlin.de/bioroboticslab/robofish/rl/blob/master/more_guppies/randomized_guppies.py

PPO and network architecture:

https://git.imp.fu-berlin.de/bioroboticslab/robofish/rl/tree/master/PPO

For running a trained policy on the real robot:

https://git.imp.fu-berlin.de/bioroboticslab/robofish/behavior_tensorflow

RoboFish Gym (Deprecated; Working precursor of Guppy Gym):

https://git.imp.fu-berlin.de/bioroboticslab/robofish/gym-robofish