Bachelor's thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Reinforcement learning for manipulation of collections of objects using physical force fields

Dominik Hodan

Supervisor: doc. Ing. Zdeněk Hurák, Ph.D.
Field of study: Cybernetics and Robotics
May 2020

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Hodan Dominik**	Personal ID number:	**474587**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Control Engineering**		
Study program:	**Cybernetics and Robotics**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Reinforcement learning for manipulation of collections of objects using physical force fields

Bachelor’s thesis title in Czech:

Posilované učení pro manipulaci se skupinami objektů pomocí fyzikálních silových polí

Guidelines:

The goal of the project is to explore the opportunities that the framework of reinforcement learning offers for the task of automatic manipulation of collections of objects using physical force fields. In particular, force fields derived from electric and magnetic fields shaped through planar regular arrays of 'actuators' (microelectrodes, coils) will be considered. At least one of the motion control tasks should be solved:
1. Feedback-controlled distribution shaping. For example, it may be desired that a collection of objects initially concentrated in one part of the work arena is finally distributed uniformly all over the surface.
2. Feedback-controlled mixing, in which collections objects of two or several types (colors) - initially separated - are blended.
3. Feedback-controlled Brownian motion, in which every object in the collection travels (pseudo)randomly all over the surface.

Bibliography / sources:

[1] D. Bertsekas, Reinforcement Learning and Optimal Control. Athena Scientific, 2019.
[2] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, “An Introduction to Deep Reinforcement Learning,” FNT in Machine Learning, vol. 11, no. 3–4, pp. 219–354, 2018, doi: 10.1561/2200000071.
[3] F. L. Lewis and D. Vrabie, “Reinforcement learning and adaptive dynamic programming for feedback control,” IEEE Circuits and Systems Magazine, vol. 9, no. 3, pp. 32–50, Third 2009, doi: 10.1109/MCAS.2009.933854.
[4] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. Cambridge, Massachusetts: A Bradford Book, 2018.

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Zdeněk Hurák, Ph.D., Department of Control Engineering, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **07.02.2020** Deadline for bachelor thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

doc. Ing. Zdeněk Hurák, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

Acknowledgements

First and foremost, I would like to express my great appreciation to my caring parents and my whole family, whose support has allowed me to fully focus on my passions and my studies.

I would also like to thank my friends, who have kept me going during the past years. Special thanks go to Vít Fanta for proofreading this thesis.

Thanks are also due to my supervisor, Zdeněk Hurák, for his support, expertise, and the entire AA4CC team. In particular, I would like to thank Martin Gurtner and Filip Richter, who have, without exception, patiently responded to my oftentimes silly questions and who have always been incredibly friendly with the “new kid” in the lab.

Last but not least, I wish to acknowledge the help provided by the Research Center for Informatics at CTU Prague that let me use the RCI cluster to train most of the neural networks presented in this thesis.

Declaration

I declare that I wrote the presented thesis on my own and that I cited all the used information sources in compliance with the Methodical instructions about the ethical principles for writing an academic thesis.

In Prague, 22. May 2020

Abstract

This bachelor's thesis deals with the simulation of a magnetic manipulation platform called Magman and methods of reinforcement learning that utilize the implemented simulator for training. Both of the implemented algorithms, namely Deep Deterministic Policy Gradient (DDPG) and Soft Actor-Critic (SAC), belong to the group of offline reinforcement learning methods based on Q-learning. The first part of this thesis is dedicated to relevant reinforcement learning theory and a description of the above-mentioned algorithms. The following chapter consists of a description of the developed simulator, including its function from the point of view of an end-user. In the last part, several problems solved by these methods are presented. In particular, the tasks were ball position control, coil control, shaping of a distribution of a collection of balls and finally feedback-controlled mixing of two groups of balls.

Keywords: reinforcement learning, Q-learning, MagMan, machine learning, Soft Actor-Critic

Supervisor: doc. Ing. Zdeněk Hurák, Ph.D.

Abstrakt

Tato bakalářská práce se zabývá simulací platformy pro magnetickou manipulaci zvané MagMan a metodami posilovaného učení, které implementovaný simulátor využívají pro trénink. Oba implementované algorithmy, a sice Deep Deterministic Policy Gradient (DDPG) a Soft Actor-Critic (SAC), patří do kategorie offline metod posilovaného učení na bázi Q-learningu. První část práce je věnována relevantní teorii posilovaného učení a popisu výše zmíněných algoritmů. V té následující je uveden popis vyvinutého simulátoru, včetně popisu funkce pro koncového uživatele. V poslední části je uvedeno několik problémů, které byly těmito metodami řešeny. Jmenovitě jde o řízení polohy kuliček, ovládání cívek, tvarování rozložení kolekce kuliček a konečně řízené směšování dvou skupin kuliček.

Klíčová slova: posilované učení, Q-learning, MagMan, strojové učení, Soft Actor-Critic

Překlad názvu: Posilované učení pro manipulaci se skupinami objektů pomocí fyzikálních silových polí

Contents

Figures

Chapter 1

Introduction

In this thesis, I am exploring the opportunities that the reinforcement learning (RL) framework, which is a subset of machine learning (ML), offers to the field of control engineering. In particular, I focused on multi-object manipulation using arrays of actuators that generate physical force fields (e.g., coils or electrodes). A concrete example of such system is MagMan (described in 1.1), which is a real-life counterpart of the simulator presented in chapter 3.

Motivation and Goals

The term machine learning dates as far back as the late 1950s, when Frank Rosenblatt designed the first neural network called Perceptron[1]. However, this approach has only gained most of its current momentum and public attention in the past decade.

The last five years have brought us neural networks that have achieved super-human level performance in some image recognition datasets, such as ImageNet [3], and (video) games like Dota 2 and StarCraft II[2] alike. Given these astonishing results, it is only natural to wonder whether these techniques might help us develop control algorithms of the future.

The end-goal of this thesis is thus to accomplish the necessary baby steps and lay a basis for further development of neural-network-based control for MagMan (and possibly other similar systems). To this end, I also suggest possible further research directions where applicable.

Structure

In chapter 2, I go over some relevant RL theory, as well as two specific reinforcement learning algorithms — Deep Deterministic Policy Gradient (DDPG) and Soft Actor-Critic (SAC) — that I used. Afterward, in chapter 3, I present the simulation environment dubbed *MagManEnv* that I developed as a part of this thesis. The environment then served to train neural networks

[1]`https://www.doc.ic.ac.uk/~jce317/history-machine-learning.html`

[2]`https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii`

using algorithms introduced in the previous chapter in a variety of tasks. These include position tracking (4.2), distribution shaping (4.4), controlled mixing (4.5) and controlling current in coils in order to exert a desired force on an object (4.3). Results, as well as the precise problem formulations, can be found in chapter 4.

1.1 The MagMan Experimental Platform

MagMan is a modular platform for magnetic manipulation developed by the reseach group Advanced Algorithms for Control and Communications (AA4CC) at the Faculty of Electrical Engineering, Czech Technical University in Prague. Its goal is to allow for experimental evaluation of multi-actuator control algorithms.

Each MagMan module consists of a 2x2 array of coils, as shown in figure 1.1, along with the necessary controller and communication electronics. As for its dimensions, the distance between the centers of two adjacent coils is 25 mm. It is possible to set a current between 0 mA to 440 mA to be passed through each of the coils independently via USB.

These modules can then be interconnected to form "arbitrary" shapes (while respecting bus speed and power limitations, of course). Our current physical realization counts with 16 of such modules that together form a grid of 8x8 coils. A flat glass surface is placed on top of these modules, where objects can be manipulated (see photo 1.2).

For further information about the hardware, please consult [4].

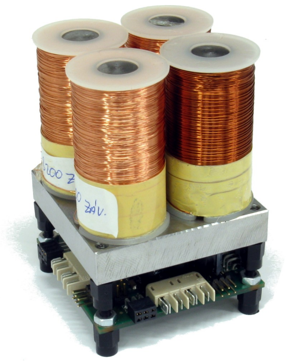


Figure 1.1: A single MagMan module

Examples of items currently used for manipulation are steel balls (colored for easier position detection) and 3D printed triangles with steel balls at each vertex (see render in figure 1.3). After solving the control problem (as described in [1]), we are able to carefully shape the magnetic field around the bodies to move them to the desired position.

In the current version of the platform, a PiCamera is used to capture video at 50 frames per second (FPS). The camera is connected to a RaspberryPi, where the control loop is running.

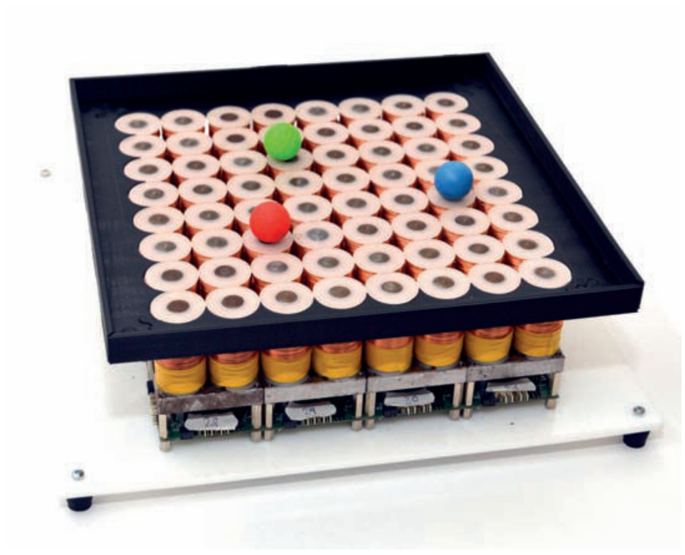


Figure 1.2: A complete MagMan platform (reprint from [1], p. 83)

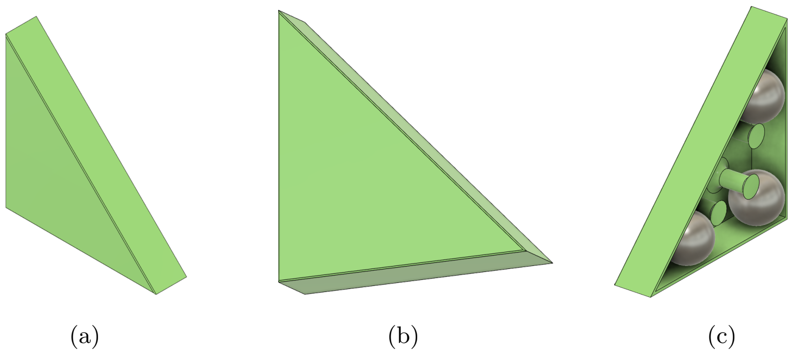


Figure 1.3: A 3D render of the triangle used for manipulation

Before all else, each image is passed to a CPython library, and the position of each of the objects is detected. Next, the data is sent to Simulink, where desired forces are first computed using a P(I)D controller. The list of forces is then passed to a quadratic program solver, which finds an approximate solution to the problem of what current to pass through each of the coils in order to exert the desired forces on the bodies. This action is then transmitted directly to MagMan over USB, as seen in figure 1.4.

System identification, as well as "traditional" control algorithm development, was compiled in [1]. Below, a summary of the parts relevant to the simulation of the platform is presented.

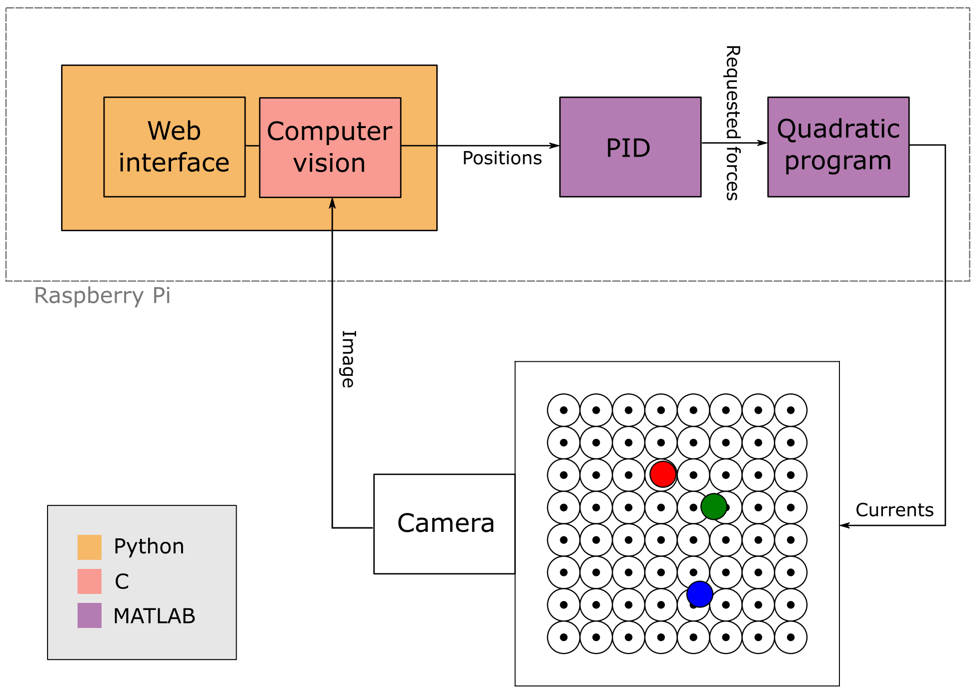


Figure 1.4: Scheme of information flow in MagMan

Magnetic flux density

The coils can be approximated by magnetic monopoles. For a single monopole, we can compute the magnetic flux density as

$$\mathbf{B}_{\mathrm{MP}}(\mathbf{r}) = -\frac{q_m}{4\pi}\nabla\frac{1}{|\mathbf{r}|} = -\frac{q_m}{4\pi}\frac{\mathbf{r}}{|\mathbf{r}|^3} = a\frac{\mathbf{r}}{|\mathbf{r}|^3}, \tag{1.1}$$

where q_m denotes the strength of the monopole located at the origin. In the case of the coils that are used in MagMan, the author estimated that $a = 3.565 \times 10^{-5}$.

From there, we get the magnitude of magnetic flux density caused by a coil as

$$|\mathbf{B}(i,x,y,z)| = \frac{af(i)}{x^2+y^2+z^2}, \tag{1.2}$$

where i is the current flowing through the coil and

$$f(i) = 0.889 \arctan\left(2.51\,|i| + 5.38i^2\right)\operatorname{sgn}(i), \;\; |i| \leq 440\,\mathrm{mA}, \tag{1.3}$$

is a fitted function describing the magnitude of the magnetic field as a function of current.

Coil force

The force exerted on a steel ball by a coil is modeled by the formula

$$\mathbf{F}(i,x,y,z) = k\nabla\mathbf{B}^2 = ka^2 f(i)^2\nabla\frac{1}{(x^2+y^2+z^2)^2}, \tag{1.4}$$

where k is a constant dependent on ball radius and the permeability of both the ball, as well as of the surroundings; x, y, z are relative positions to the monopole (coil).

Constants were then fitted for the final form of the equation

$$F_x(i, x, y) = f^2(i) \frac{cx}{(x^2 + y^2 + d^2)^3}. \tag{1.5}$$

Other components of the force can be found by swapping x for the desired component. Notice that since we are assuming planar motion, the distance in the z axis was replaced by a constant d and $c = -4ka^2$. Numerical values can be found in table 1.1.

ball radius [mm]	m [g]	c [$\times 10^{-10}$]	d [mm]
10	4.09	-2.041	13.3
15	13.76	-5.407	14.3
20	32.64	-16.60	16.6
30	110	-18.5	21.6

Table 1.1: Fitted constants for equation 1.5 for different steel balls

The force exerted by all the coils is then considered to be a superposition of these forces. This was shown to paint an accurate enough picture of the dynamics, even though the fields overlap.

Ball dynamics

Ball dynamics in two dimensions are modeled by a set of differential equations

$$\ddot{x} = \frac{F_x}{m_{\text{eff}}} \quad \text{and} \quad \ddot{y} = \frac{F_y}{m_{\text{eff}}}, \tag{1.6}$$

where

$$m_{\text{eff}} = \frac{7}{5} m \tag{1.7}$$

is the effective mass that takes into account ball inertia and m is the ball's mass.

Three forces are taken into consideration. First, the force exerted by the coils as described by equation 1.5. Two damping forces are assumed. Rolling resistance is described by

$$\mathbf{F}_{\text{rr}} = -C_{\text{rr}} F_{\text{N}} \frac{\mathbf{v}}{|\mathbf{v}|}, \tag{1.8}$$

where F_N is a normal force, i.e., the sum of gravitational force and the z-component of the force caused by coils, $\mathbf{v}$ is the ball's velocity and C_{rr} is the coefficient of rolling resistance. For a ball with $r = 15\,\text{mm}$, it was measured to be $C_{\text{rr}} = 4.25 \times 10^{-4}$.

The damping force caused by eddy currents was found to be

$$\mathbf{F}_{\text{eddy}} = -C_{\text{eddy}} \left(\mathbf{v} \cdot \nabla |\mathbf{B}|\right) \frac{\mathbf{v}}{|\mathbf{v}|}. \tag{1.9}$$

The value of C_{eddy} was again tested for a ball with $r = 15\,\text{mm}$ and was found to be $C_{\text{eddy}} = 0.115$.

Control

Currently, position control for steel balls is implemented. There are a few issues to keep in mind when developing control algorithms for MagMan.

First, the standard deviation of position measurement is around $0.7\,\text{mm}$ Second, an approximately $2\,\text{FPS} \sim 40\,\text{ms}$ delay was observed in the control loop. Currently implemented algorithms are considerably improved if position prediction is used. It would, therefore, be desirable if a neural network designed for MagMan were able to eliminate this issue and take the raw observations into account.

Chapter 2
Reinforcement learning

The goal of reinforcement learning is to create a program, usually called policy or agent, that accepts sensory data from an environment and based on the data it receives, returns an action to be performed to maximize some reward (or minimize loss).

The problem statement is akin to what traditional optimal control tries to solve. A chief difference between the two is that in RL, we typically do not try to solve the optimization problem at each step but rather to find a policy that "knows" what action to take. Reinforcement learning also does not necessarily require a mathematical model of the environment. There are exceptions to this, as mentioned in 2.1.3.

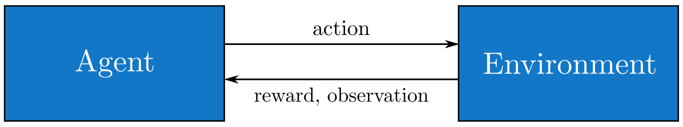


Figure 2.1: Illustration of the RL setting

2.1 Reinforcement learning setting

In the section, I briefly go over the notation and definitions used in the following text.

2.1.1 Markov Decision Process

Let us begin by stating the Markov decision process (MDP). At first, fully observable environment is considered, though this is further generalized in 2.1.1.

Let $\mathcal{A} \subseteq \mathbb{R}^A$ denote a continuous action space and $\mathcal{S} \subseteq \mathbb{R}^S$ a continuous state space. At every timestep t_n, our agent, represented either by a stochastic policy

$$\pi\colon \mathcal{S} \to P(\mathcal{A}), \tag{2.1}$$

where $P(\mathcal{A})$ is a probability density function defined on $\mathcal{A}$, or a deterministic policy

$$\mu\colon \mathcal{S} \to \mathcal{A}, \tag{2.2}$$

takes action $a_n \in \mathcal{A}$. The environment then undergoes a transition to a new state according to its transition function

$$T\colon \mathcal{S} \times \mathcal{A} \to P(\mathcal{S}) \tag{2.3}$$

and returns a new state s_{n+1} and a reward $r_n(s_n, a_n, s_{n+1})$. Sometimes, this notation will be abbreviated to $r(s, a, s')$ or even r, unless it leads to ambiguity. The tuple (s, a, s', r) is called a transition.

A key observation here is the fact that the transition does not depend on previous states and actions. An environment abiding by this rule is called Markovian or is said to have the Markovian property. The tuple $(\mathcal{S}, \mathcal{A}, T, \mathcal{R}, \gamma)$ is called an MDP.

The cumulative reward from step n to infinity is defined as

$$R_n = \sum_{i=n}^{\infty} \gamma^i r_i(s_i, a_i, s_{i+1}), \tag{2.4}$$

where $0 < \gamma < 1$ (usually $\gamma \gg 0$) is commonly referred to as the discount factor.

The goal is then to find a policy π that maximizes the expected reward

$$J(\pi) = \mathop{\mathbb{E}}_{\substack{a \sim \pi \\ s' \sim T}} \left[R_0 \right]. \tag{2.5}$$

Partially observable Markov Decision Process

In some cases, it is impossible to observe the complete state of the system. To capture this, a partially observable Partially observable Markov decision process (POMDP) is introduced. Even if all states are observed, POMDP can serve to model measurement imprecision.

For example, in the case of MagMan (described in 1.1), each observation is an image from a camera. However, the state of the system also consists of velocities of all the objects, which cannot be extracted from that single photo. (In theory, we could set long exposition from which velocity could be approximated since the position would be blurred in its direction, but let's ignore that.) Note that this would be considered a fully observable system from the point of view of classical control theory, as the observability matrix is full-rank. The difference between the two stems from the fact that we essentially allow the use of past states to calculate the current only in the latter.

In addition to what was defined for a regular MDP, consider an observation space $\Omega \subseteq \mathbb{R}^O$ and a function of observation probabilites $\mathcal{O}$,

$$\mathcal{O}\colon \mathcal{S} \to P(\Omega), \tag{2.6}$$

describing the probability of observation $\omega \in \Omega$ given $s \in \mathcal{S}$.

2.1.2 Q-learning

One approach to finding an optimal policy is to find a measure of how "desirable" each of the states (and possibly actions) is. In other words, what is the expected discounted reward for a particular state given a policy π. This leads to a method (or rather a class of methods) called Q-learning. The following summarizes [5] and [6].

The value of state s in policy π is defined as

$$V^\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi \\ s'\sim T}} [R_0 \mid s_0 = s] \tag{2.7}$$

and the Q-value (Q stands for quality) of state s and action a is defined as

$$Q^\pi(s,a) = \int_{\mathcal{S}} T(s,a,s')(r + \gamma \mathop{\mathbb{E}}_{\substack{a'\sim\pi \\ s''\sim T}} [R_0 \mid s_0 = s'])\,\mathrm{d}s'\,. \tag{2.8}$$

By definition, it is also true that

$$Q^\pi(s,a) = \int_{\mathcal{S}} T(s,a,s')(r + \gamma V^\pi(s'))\,\mathrm{d}s'\,. \tag{2.9}$$

Because

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}[Q^\pi(s,a)]\,, \tag{2.10}$$

we can write equation 2.9 recursively as

$$Q^\pi(s,a) = \int_{\mathcal{S}} T(s,a,s')(r + \gamma\,\mathbb{E}_{a'\sim\pi}[Q^\pi(s,a')])\,\mathrm{d}s'\,. \tag{2.11}$$

This is known as the Bellman equation.

However, we are not as interested in these, as they are tied to a certain fixed policy, and it doesn't necessarily reflect the real values. What we really need to find to get the optimal policy are the "true" values of Q and V, denoted as Q^*, V^*.

The optimal values are given by

$$V^*(s) = \max_\pi V^\pi(s), \tag{2.12}$$

$$Q^*(s,a) = \max_\pi Q^\pi(s,a). \tag{2.13}$$

Because the following holds true:

$$V^*(s) = \max_{a\in\mathcal{A}} Q^*(s,a), \tag{2.14}$$

we get the Bellman equation for optimal value functions as

$$Q^*(s,a) = \int_{\mathcal{S}} T(s,a,s')(r + \gamma \max_{a'} Q^*(s',a'))\,\mathrm{d}s'\,. \tag{2.15}$$

In case the action and state spaces are discrete, the Bellman equation simplifies to

$$Q^{\pi}(s,a) = \sum_{s'} T(s,a,s')(r + \gamma \max_{a} Q^{\pi}(s',a)). \tag{2.16}$$

It has been shown, that by iterating 2.16, it is possible to obtain Q^*.

If the agent stores a table of all the Q-values, it can then choose an action according to policy μ

$$\mu(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a). \tag{2.17}$$

However, in the case of large action and observation spaces, this method suffers from the curse of dimensionality, even if they are discrete. This means that keeping such a table usually is intractable for real problems. In the case of MagMan, where both the observation space and the action space are continuous, even the $\arg\max$ operator is problematic as it would be necessary to solve the optimization problem at every iteration. This is where neural networks' role as function approximators comes in handy.

Many reinforcement learning algorithms (though certainly not all) are essentially a way to find an approximation of the Q-function. In the next section, I will go over the most common ways to classify RL algorithms.

2.1.3 Categories of RL algorithms

In this part, I go over different categories of RL algorithms. Unless stated otherwise, information comes from [6].

Deterministic & Stochastic

As mentioned in section 2.1.1, perhaps the most obvious way to differentiate the algorithms is to look at the resulting policy. If the output of the agent is an action that depends solely on the observation, the algorithm itself is called deterministic. If, on the other hand, the output is a probability distribution, the action is sampled from this distribution, and the agent is therefore stochastic.

Online & Offline

Another way to think about RL algorithms is from the point of view of which transitions are relevant to learning. If the agent can only learn from transitions obtained using current policy, it is referred to as an online learning algorithm. Being able to collect transitions relevant to the current policy has the benefit of letting the agent influence what type of experiences it collects. The agent can thus, for example, try actions where the outcome is not yet certain, which could indicate low exploration in that part of the state space or due to high stochasticity of the environment. A crucial benefit of online methods is better stability [7].

The opposite of this are offline learning algorithms. These can benefit from samples collected by any policy. This has been shown to lead to better sample-efficiency (i.e. how many samples are necessary to reach the same level of ability) [7]. Some version of Q-learning is usually used to make it possible.

Model-based & Model-free

We can also divide neural networks by whether they are using a mathematical model of the environment. The advantage of this is that model-based methods are usually more sample efficient [7].

In the case of discrete action spaces or low-dimensional continuous action spaces, it is possible to solve the problem using classical dynamic programming methods. This is done by simulating the decision tree (possibly discarding unpromising paths) and selecting the action with the highest expected reward.

For higher dimensions, where classical methods become intractable, neural networks can be used. An architecture that is often used for this is known as auto-encoder, which is essentially a network than learns a representation of the state space in a lower dimension (its so called latent space), similarly to principal component analysis. Either regular optimal control or control that utilizes neural networks can be then performed in its latent space.

In case no mathematical model is available (or is not sufficiently precise) a priori, it is possible to use a model-approximation algorithm to find the T function that describes the environment. However, according to [5], better results are usually achieved if the policy is trained directly on the data that would have had been used to teach the model.

Continous & Discrete

Another point of view from which we can categorize algorithms is by the nature of their action and observation spaces. There are inherent limitations to what kind of problems can be solved by an RL algorithm. First, there are algorithms that need both the observation as well as the action space to be discrete. However, this class is generally not interesting from a control engineering point of view.

More relevant are algorithms that have continuous observation spaces while still only handling discrete action spaces. A break-through algorithm in this regard was the Deep Q-learning (DQN) algorithm [8]. The DQN managed to learn to play several Atari games directly from pixels, while beating contemporary state-of-the-art (SOTA) algorithms (that were using augmented data as input) in all but one game. In some of the games, the agent beat an expert human player, which had not been done before.

This project aims to develop policies for MagMan, which has continuous both the action and observation spaces. Therefore, methods that can deal with this are necessary. One of the first "working" neural network (NN) algorithms was the Deep Deterministic Policy Gradient published in [9], described in detail in 2.2.1. Currently, one of the SOTA algorithms for this kind of problems is Soft Actor-Critic (see 2.2.2).

Different approximated functions

There are more approaches leading to the same goal. Some algorithms directly try to learn the policy π and some try to learn the value functions. The algorithms used in this thesis (SAC and DDPG) both belong to the group called Actor-Critic methods. This class of methods utilizes two networks simultaneously to train a policy (actor) and a value/Q-function (critic).

Others, as mentioned above, might try to learn the dynamics of the environment to enable easier control using different algorithms.

2.2 Implemented algorithms

Out of the plethora of published algorithms, I decided to implement two offline methods and compare them. The reason to choose two offline approaches is that should we later move to the real-world counterpart of the simulation, sample-efficiency is going to be a key advantage because there is only one "specimen" available to train on. In addition to that, it is also not possible to run it without breaks due to overheating.

The first of the selected algorithms is called Deep Deterministic Policy Gradient. It was one of the first methods for RL for continuous action and observation spaces. While comparably simple, it is still relevant and used to this day for comparison in newly published papers as a sort of a baseline in its category. [9]

The second algorithm is Soft Actor-Critic. As the name suggests, it belongs to the class of actor-critic algorithms like DDPG. However, unlike DDPG, the behavior is defined by a stochastic policy. In this case, the function being optimized is not only the sum of rewards but also depends on the policy entropy. Its second version, one of the SOTA methods nowadays, claims to have solved the brittleness problem of offline methods (i.e., their sensitivity to hyperparameters), while still keeping the advantage of high sample-efficiency [7]. The authors have shown a successfully trained quadruped robot that is able to walk even if obstacles are in the way[1].

2.2.1 Deep Deterministic Policy Gradient

The algorithm known as Deep Deterministic Policy Gradient was published in [9]. It is a model-free offline actor-critic learning algorithm that combined the advantages of the previously published DPG algorithm and a few insights

[1]`https://youtu.be/KOObeIjzXTY`

from the ground-breaking DQN algorithm (which only considered discrete action spaces).

A new idea introduced in DQN was the use of a replay buffer $\mathcal{D}$ [8], in which all past transitions are stored. Because DDPG is also an offline method, it can benefit from transitions sampled by a different policy, such as those saved in $\mathcal{D}$. The critic is represented using a set of parameters θ, denoted as Q_θ, the actor is deterministic and parametrized using ψ, written as μ_ψ.

Critic loss is computed as the mean squared error (MSE) over a mini-batch $B = (s, a, s', r) \sim \mathcal{D}$

$$L_Q = \frac{1}{|B|} \sum_B \frac{1}{2} \left(Q_\theta(s, a) - (r + \gamma V_{\theta'}(s')) \right)^2 , \tag{2.18}$$

where

$$V_{\theta'}(s') = Q_{\theta'}(s', \mu_{\psi'}(s')). \tag{2.19}$$

Notice that a different version of parameters, θ' and ψ', is used in the equation as well. The reason for this is that unlike vanilla actor-critic, DDPG uses four networks. This was also proposed (for the Q-network) in the DQN paper.

The reason for this is that it was found that vanilla Q-learning tends to be unstable due to the fact that the parameters' update directly depends on the parameters themselves. DDPG, therefore, uses a pair of both Q-networks and policy networks. One of the Q-networks is used to train the policy, the other (commonly referred to as the *target network*) to train the first one. Likewise, one policy is used to select actions, the other to train the first one.

Target networks' parameters lag behing their counterparts. They are updated at every step using Polyak averaging

$$\theta' \leftarrow (1 - \tau)\theta' + \tau\theta, \tag{2.20}$$

$$\psi' \leftarrow (1 - \tau)\psi' + \tau\psi, \tag{2.21}$$

where $\tau \in \mathbb{R}\colon\ 0 < \tau < 1$ is a constant usually close to zero.

Actor loss is computed as

$$L_\pi = -\frac{1}{|B|} \sum_B Q(s, \mu_\psi(s)). \tag{2.22}$$

The policy is deterministic, so noise is added to actions to stimulate exploration. The authors suggest using an Ornstein–Uhlenbeck process, so that the generated noise is temporarily correlated.

To get bounded action, the output of the neural network is passed through a tanh function (applied element-wise), limiting policy action to $a \in [-1, 1]^A$.

Algorithm 1: DDPG

```
initialize $\theta, \psi$ to random values
initialize replay buffer $\mathcal{D}$
copy parameters $\theta' \leftarrow \theta,\ \psi' \leftarrow \psi$
repeat
    initialize random process $\mathcal{N}$
    reset environment and sample new state $s$
    for $n = 1, \dots, n$ do
        select action $a = \text{clip}(\mu_\psi(s) + \mathcal{N}_n, a_{\min}, a_{\max})$
        perform action $a$, receive $s', r, done$
        store new transition $\mathcal{D} \leftarrow \mathcal{D} \cup (s, a, s', r)$
        sample a random batch $B$ of tuples from buffer, $B \sim \mathcal{D}$
        $\theta \leftarrow \theta - \lambda_\theta \nabla_\theta L_Q$
        $\psi \leftarrow \psi - \lambda_\psi \nabla_\psi L_\pi$
        update target networks using 2.20
        if $done$ then
            break
        end
    end
until satisfied
```

2.2.2 Soft Actor-Critic

There are two critical differences between DDPG and SAC, which was published in [10] and later improved upon in [7]. The latter learns a stochastic policy and, in addition to that, does not try to maximize only the total reward. Instead, the optimal policy is defined as

$$\pi^* = \arg\max_\pi \sum_t \mathbb{E}_{\substack{a\sim\pi \\ s'\sim T}} \left[r(s,a) + \alpha \mathcal{H}(\pi(\cdot \mid s)) \right], \tag{2.23}$$

where $\mathcal{H}$ is entropy and α is a parameter called temperature, which in practice controls the stochasticity of the policy. The authors argue that this incentivizes broader exploration and allows the policy to learn more modes of optimal behavior. According to their paper, it also achieves SOTA learning speed.

Another difference between SAC and DDPG is that in its second version, SAC makes use of two Q-functions (Q_{θ_1} and Q_{θ_2}) to train the policy. This change was inspired by [11], which was published concurrently with the first version. In that paper, the authors show that this approach limits the overestimation problem of value-based methods and significantly improves the performance of algorithms.

The Q-function loss is computed similarly as in the case of DDPG using MSE loss as

$$L_{Q_i} = \frac{1}{|B|} \sum_B \frac{1}{2} \left(Q_{\theta_i}(s,a) - (r + \gamma V_{\theta'}(s')) \right)^2. \tag{2.24}$$

The first version of SAC used a separate V-network but the newer version that I implemented computes the V-value as

$$V_{\theta'}(s') = \min\left(Q_{\theta'_1}(s', a'), Q_{\theta'_2}(s', a')\right) - \alpha \log \pi(a' \mid s'), \tag{2.25}$$

where $a' \sim \pi(s')$.

Policy loss is computed according to

$$L_\pi = \frac{1}{|B|} \sum_B \left(\alpha \log(\pi_\psi(a' \mid s)) - \min\left(Q_{\theta'_1}(s, a'), Q_{\theta'_2}(s, a')\right)\right), \tag{2.26}$$

where $a' \sim \pi$ is sampled from the current policy, rather than the buffer.

There is one last difference between the two versions of SAC. While the authors have demonstrated that even the first version is less sensitive to hyperparameters than DDPG, it is still brittle with respect to the temperature parameter. The proposed solution to this is to optimize α at each step by minimizing

$$L_\alpha = \frac{1}{|B|} \sum_B \alpha \left(-\log \pi(a \mid s) - \overline{\mathcal{H}}\right), \tag{2.27}$$

where $\overline{\mathcal{H}}$ is target entropy, in accordance with the paper set to $\overline{\mathcal{H}} = -\dim(\mathcal{A})$.

Caveats

The output of the policy is a mean vector μ and standard deviation vector σ, from which the action is sampled. A problem arises when the algorithm needs to backpropagate through sampling from a distribution $z \sim \mathcal{N}(\mu, \sigma)$.

To solve this, what is known as the *reparametrization trick*, is used. A noise vector is sampled from a gaussian distribution ($\epsilon \sim \mathcal{N}(0, 1)$) to preserve stochasticity. The action is then computed as $z = \mu + \sigma \odot \epsilon$ (where $\odot$ denotes an element-wise product). This way, differentiation with respect to network parameters is the same as for deterministic networks.

Another problem is that like in DDPG, we would like the action to be bounded. This is done the same way by passing the action through a tanh layer, in other words

$$a = \tanh(z). \tag{2.28}$$

However, in the case of SAC, we need to compute $\log \pi$ to find the loss.

Given that this is a random variable transformation, the relationship between the respective probability densities of π and ξ is

$$\pi(a \mid s) = \xi(z \mid s) \left|\det\left(\frac{\mathrm{d}a}{\mathrm{d}z}\right)\right|^{-1}. \tag{2.29}$$

Taking the logarithm and simplifying, we get

$$\log \pi(a \mid s) = \log \xi(z \mid s) - \sum_i \log\left(1 - \tanh^2(z_i)\right), \tag{2.30}$$

where z_i denotes the i-th element of the vector and ξ the unbounded probability distribution.

Algorithm 2: SAC

```
initialize $\alpha, \theta_1, \theta_2, \psi$ to random values
initialize replay buffer $\mathcal{D}$
copy parameters $\theta_i' \leftarrow \theta_i$,
repeat
    reset environment and sample new state $s$
    for $n = 1, \ldots, n$ do
        select action $a \sim \pi(s)$
        perform action $a$, receive $s', r, done$
        store new transition $\mathcal{D} \leftarrow \mathcal{D} \cup (s, a, s', r)$
        sample a random batch $B$ of tuples from buffer, $B \sim \mathcal{D}$
        $\theta_i \leftarrow \theta - \lambda_\theta \nabla_{\theta_i} L_{Q_i}$
        $\psi \leftarrow \psi - \lambda_\psi \nabla_\psi L_\pi$
        $\alpha \leftarrow \alpha - \lambda_\alpha \nabla_\alpha L_\alpha$
        update target networks using 2.20
        if $done$ then
            break
        end
    end
until satisfied
```

2.2.3 Implementation structure

I implemented both the DDPG and the SAC algorithm, along with several helper classes. Additional algorithms can easily be added to folder `algos/` as long as they provide the same interface. Each algorithm is implemented as two files: a “model”, where the model architecture is implemented, and a “trainer”, which implements the algorithm, along with the interface for the main training loop.

Apart from these, a logger is implemented, that can be thought of as the experiment manager. This object saves reward history, as well as both trainer and environment settings. This way, it is straightforward to continue with training after a halt seamlessly.

A scheme of the main training loop can be seen in figure 2.2.

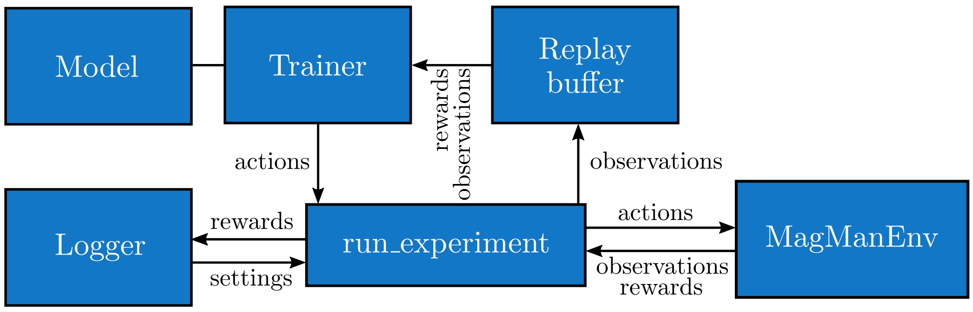


Figure 2.2: Main training loop architecture

Chapter 3

Simulation

To train a neural network, one ideally needs to have an environment where the agent can learn. While it is true that certain reinforcement learning algorithms, such as the offline methods presented in this thesis (2.2), can learn from transitions acquired beforehand by a different policy, it is often beneficial to sample transitions directly related to current policy. This fact is leveraged, for example, by the SAC algorithm (2.2.2), where the entropy term encourages exploration of states with high entropy (high uncertainty) [10]. Therefore, although it is theoretically possible to use a real-world environment to collect all the experience required (even policy-related), it is beneficial to train the agent using a computer simulation.

In this chapter, I present the physics simulator I created that can be used to simulate a real-world environment. For a concrete example, we decided to simulate the MagMan platform developed by the AA4CC team, described in 1.1. It should be noted, however, that the aim of the code is to be written in such a way as to allow for simulation of different actuator-array-centered systems with as few changes as possible.

There are two simulation modes available. The input to the environment can either be a list of currents to be passed to the coils (I will refer to this as "coil simulation") or a set of forces to be exerted on each of the balls in simulation ("force simulation"). Both of these modes can be simulated with dynamics or without dynamics (3.2.5).

3.1 Physics

I implemented the simulation according to [1]. However, there were a few problems concerning a lack of data. Given that measuring data is not ideal due to the COVID-19 pandemic, not to mention out of the scope of this thesis, I had to approximate in a few cases.

In addition to that, for the second simulation mode (force simulation), I had to devise a way to approximate the forces due to rolling resistance and eddy currents. While one solution would undoubtedly be solving the quadratic program at each step to find precisely how the coils would have to behave to exert the desired force, I decided not to take that path. The reason for this is that the simulation is not entirely accurate by definition (the exerted force will not be equal to the desired force in most cases), so it would make little sense to have hyper-realistic resistance.

3.1.1 Physical constants

Coefficients of rolling resistance and eddy currents

According to [12], equation 1.8 can be rewritten (in one dimension) as

$$F_{\mathrm{rr}} = C_{\mathrm{rr}} F_{\mathrm{N}} = \frac{b}{r} F_{\mathrm{N}}, \tag{3.1}$$

where b is a constant and r is ball's radius. This relationship was used to approximate C_{rr} for balls of different diameters, as it had only been measured for a ball of diameter $r = 30\,\mathrm{mm}$. Experimental verification should be carried out at a later time to confirm this.

As for the constant from 1.9, I was unable to find any sources on the matter. Therefore, it is kept the same for all ball sizes. As with C_{rr}, C_{eddy} should be measured in the future to improve simulation precision.

Polygon simulation

For polygon simulation, there is another friction force that dominates the other two. This force is due to the balls moving in the plastic casing. In the prediction currently used in MagMan, the friction is approximated as

$$\ddot{\mathbf{x}} = -k\dot{\mathbf{x}} \tag{3.2}$$

$$\ddot{\varphi} = -k\dot{\varphi}, \tag{3.3}$$

where $\mathbf{x}$ is the position of polygon's center of mass, φ its rotation and k is a constant ($k = 0.58$ for the triangle in 1.3). This value is stored in the corresponding shape instance and should be measured on a per body basis.

Collision ellasticity and friction

Another part of the simulation is the coefficient of restitution C_{R}, which determines the elasticity of the collisions. For a collision of two objects, Box2D uses the higher value of restitution of the two objects. Unlike the previous constants, it is not as necessary to have its exact value. This is due to the fact that collisions are generally not advantageous (at least for the tasks presented in 4) and should not happen for a reasonably trained policy. In theory, this value should thus not matter. I used a value of $C_{\mathrm{R}} = 0.7$ for the steel balls, indicating a relatively elastic collision, and $C_{\mathrm{R}} = 0.1$ for the polygons. These values were chosen so that the simulation behaves realistically, at least to the naked eye.

As for friction, I used a coefficient of friction of $C_f = 0.3$ for the polygons and and $C_f = 0.2$ for balls as suggested by[1]. The same reasoning as for the coefficient of restitution is valid, so this value is currently only set to "look reasonable".

3.1.2 Force simulation

Rolling resistance

Let $F_x = F_y = 0$. There is no way of knowing if there truly is no coil active nearby or two coils on opposite sides of the ball exerting the same force, yet the normal force will be drastically different in these two cases. However, it is reasonable to argue that if the second controller (in figure 1.4) works correctly and tries to minimize the action (currents) too, the coils "behind" the ball will mostly be turned off (at least when the ball is moving forward).

Because of this, I believe that it is reasonable to assume that the force is caused by a single actuator at a distance l, where $2l$ is the distance between actuators. Combining this assumption and equation 1.5 as

$$k = \frac{cf^2(i)}{(x^2 + y^2 + d^2)^3} \tag{3.4}$$

$$F_x^2 + F_y^2 = k^2(x^2 + y^2) = k^2 l^2, \tag{3.5}$$

it is possible to approximate F_z as

$$F_z \approx dk = d\frac{1}{l}\sqrt{(F_x^2 + F_y^2)}. \tag{3.6}$$

This gives a reasonable approximation for true distances not too close to zero as shown in 3.1.

Resistance due to eddy currents

An even greater challenge was to simulate resistance due to eddy currents. Equation 1.9 requires $\nabla|\mathbf{B}|$, while force is proportional to $\nabla|\mathbf{B}|^2$. Therefore, it is necessary to find $\nabla|\mathbf{B}|$ as a function of $\mathbf{F}$.

Using the equations 1.5 and 1.4 and

$$\frac{\partial|\mathbf{B}|}{\partial x} = -af(i)\frac{2x}{(x^2 + y^2 + d^2)^2} \tag{3.7}$$

[1]https://www.engineeringtoolbox.com/friction-coefficients-d_778.html

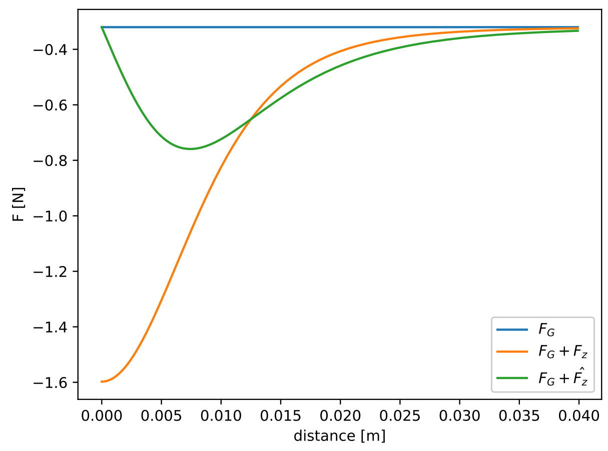


Figure 3.1: Approximation $\hat{F}_z$ compared with true values of F_z for one coil with $i = 420\,\text{mA}$ and a ball with radius $r = 15\,\text{mm}$

(which comes from taking a partial derivative of 1.2), we can write

$$\begin{aligned} F_x(i,x,y) &= f^2(i)\frac{cx}{(x^2+y^2+d^2)^3} && (3.8)\\ &= f^2(i)\frac{-4ka^2x}{(x^2+y^2+d^2)^3} && (3.9)\\ &= -\frac{k(x^2+y^2+d^2)}{x}\frac{4a^2x^2f^2(i)}{(x^2+y^2+d^2)^4} && (3.10)\\ &= -\frac{k(x^2+y^2+d^2)}{x}\left(\frac{\partial|\mathbf{B}|}{\partial x}\right)^2 && (3.11)\end{aligned}$$

Rearranging the equation gives us

$$\left|\frac{\partial|\mathbf{B}|}{\partial x}\right| = \sqrt{\frac{-xF_x}{k(x^2+y^2+d^2)}}. \tag{3.12}$$

Note that $xF_x < 0$ (force is always in the direction of the coil) and all other variables are greater than zero, so the square root is defined.

Using the same “guess” as in the case of normal force, we can find the $\hat{x}$ and $\hat{y}$ values for the equation as

$$\begin{aligned} \alpha &= \operatorname{atan2}(F_y, F_x), && (3.13)\\ \hat{x} &= l\cos\alpha, && (3.14)\\ \hat{y} &= l\sin\alpha. && (3.15)\end{aligned}$$

To get rid of the absolute value, we can use the insight that because squaring is a monotone function for positive inputs, the following must hold true:

$$\operatorname{sgn}\frac{\partial|\mathbf{B}|}{\partial x} = \operatorname{sgn}\frac{\partial|\mathbf{B}|^2}{\partial x} = \operatorname{sgn}F_x. \tag{3.16}$$

3.2 The MagMan environment

As mentioned above, a major (though implicit) part of my work was to create a flexible simulation environment that would serve as a practice range for the neural networks. In this section, I will describe the simulator I developed.

The goal of this text is only to provide a high-level understanding of what is possible and approximately how it is done, all the while delving into the implementation details as little as possible. The exceptions are section 3.2.7, where I explain the variables that are used to set the environment up, and sections 3.2.2 and 3.2.3, where I explain how to define the objects to be used for simulation. I made these exceptions because this information is vital for a potential end-user.

The code is understood to be a part of this thesis, and as such, is provided with ample comments, and it should (hopefully) be rather straight-forward to understand the particularities after reading this section.

Since Python is already used in MagMan (see 1.4) and also currently doubles as the go-to language for machine learning[2], I decided to implement all the code for my thesis in Python.

The architecture of the environment is illustrated in figure 3.2.

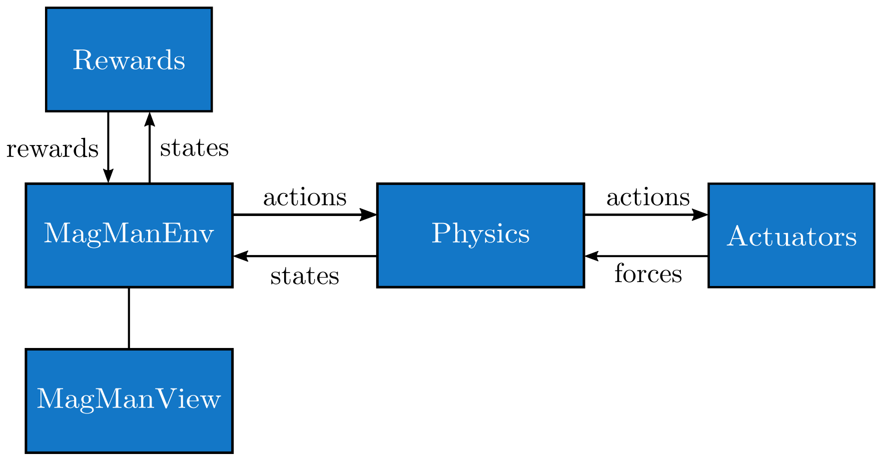


Figure 3.2: MagManEnv architecture

3.2.1 Used frameworks and libraries

PyTorch

PyTorch[3] is an open-source ML framework for Python. It provides all the necessary tools for general ML, such as gradient computation, optimizers,

[2]https://towardsdatascience.com/which-deep-learning-framework-is-growing-fastest-3f77f14aa318

[3]https://pytorch.org/

basic neural network building blocks (layers), common loss functions, graphics processing unit (GPU) acceleration, etc.

I decided to opt for this framework as opposed to its popular open-source alternative TensorFlow, as I was already familiar with it from previous university courses.

OpenAI Gym

The most common benchmark for new RL algorithms (used for example in many of the cited papers) is OpenAI's Gym[4], proposed in [13]. According to the authors, its primary goal is to provide a collection of benchmarks for RL with a common interface to make the comparison of different algorithms as easy as possible.

Since we want the MagMan platform to be as accessible as possible, I designed the environment to work in accordance with Gym's requirements. This should enable other potential researchers to incorporate this environment into their reinforcement learning workflow.

Physics library

In order not to reinvent the wheel, it makes sense only to deal with forces caused by the environment and agent and leave the heavy lifting to a specialized library. Therefore, it was necessary to choose the underlying physical simulation framework that takes care of collisions and acceleration caused by the forces applied. In order for the simulation to be reasonably fast, I was looking for a library written in a low-level language (such as C/C++) with bindings to Python.

One such popular framework, used in many of the AIGym benchmarks, is MuJoCo. Its main disadvantage is that it is necessary to obtain a license unlike the alternatives below. currently priced at 500 USD/year for personal non-commercial use and 3000 USD/year for an academic lab[5].

Other possible options include PyBullet[6] and Project Chrono[7]. These are both advanced free 3D simulators meeting the requirements. However, I found them to be a bit too complex for this project's needs.

Since the manipulation is strictly planar, I looked for a simpler, 2D simulator. I tested pybox2d[8] (based on chipmunk) and pymunk[9] (based on Box2D).

Most of the time, I was using pymunk because it was easier to install and easier to use, not to mention better maintained (last release of pybox2d had been in 2016). However, I found its simulation to be somewhat unreliable

[4]https://gym.openai.com/
[5]https://www.roboti.us/license.html (accessed April 20, 2020)
[6]https://github.com/bulletphysics/bullet3
[7]https://projectchrono.org/
[8]https://github.com/pybox2d/pybox2d
[9]http://www.pymunk.org/en/latest/

(in particular polygon simulation) and was unable to debug the problem. In April 2020, pybox2d got a new release, which finally gave me the impulse necessary to switch.

Currently both engines are included (`pymunkPhysics` and `Box2DPhysics` in `physics.py`). These classes share the same interface, so they can be used interchangeably. Nonetheless, the engine built on top of pymunk might not work as expected, so using Box2D is strongly encouraged. Therefore, I will only cover the latter in this text from now on.

Should it be necessary in the future, adding a different library should be rather straight-forward, given that there are already three engines (in addition to the two, there is also `SimplePhysics` as described in 3.2.5).

Other dependencies

Other external dependencies include the standard libraries NumPy and SciPy for fast computation and Matplotlib for visualization. In case video export is required, an installation of ffmpeg[10] is necessary as well.

In addition to this, Shapely[11] is used for its convenience functions for planar geometric objects.

3.2.2 Simulation objects

It was crucial to devise a user-friendly way to define objects to be used in simulation. To keep up with its real-life counterpart, at the very least ball and triangle (figure 1.3) simulation had to be implemented. As a generalization, the simulator is capable of simulating not only balls and triangles but any convex polygon containing steel balls.

All the necessary information for simulation and visualization is carried by special objects defined in `simulation_shapes.py`. A list of these objects is passed to the environment during its initialization (see 3.2.7).

An abstract class `SimulationShape` is implemented, from which all these objects are meant to inherit. Basic general classes are `Ball` and `Polygon`. Convenience classes that inherit from these are various ball sizes (see figure 3.3), as well as right isosceles triangle (3.4) and regular n-sided polygon (3.4). Otherwise, it is possible to inherit the Polygon class and define another polygon class or simply create an instance of the Polygon class with the appropriate parameters. The polygons have to be convex (this is a Box2D limitation), but they may contain different balls, as illustrated in 3.4.

3.2.3 Actuators

To make MagManEnv more versatile, the actuators are also passed as an array during environment initialization. The environment is flexible enough

[10] `https://www.ffmpeg.org/`

[11] `https://shapely.readthedocs.io/en/latest/`

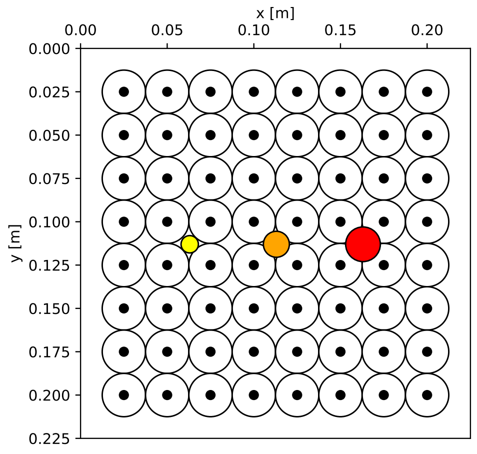


Figure 3.3: Visualization of classes `Ball10mm` (yellow), `Ball15mm` (orange) and `Ball20mm` (red)

to allow for any actuator configuration; every actuator can even be defined by a different force function.

The definitions can be found in file `actuators.py`. There is a class called `Actuator` that represents a single actuator. Since it accepts force function as a parameter, changing how the actuator behaves is a matter of changing one line of code (provided a different force function).

However, when the simulator was finished, I noticed during profiling that computing the forces was a serious bottleneck of the simulation. Therefore, I added an option to sacrifice some of flexibility for increased speed. The class is called `ActuatorArray` and as one might guess from its name, instead of having a separate object for every actuator, multiple actuators defined by the same force function are simulated using this one object.

Thanks to this approach, the simulation is able to leverage numpy's performance to cut simulation time by a factor of 11 for the full MagMan simulation with 64 coils. The only case where using the simple `Actuator` class is faster is when simulating a single coil. Speed comparison is shown in figure 3.5.

The only downside to the new class is that it is necessary to provide a fully numpy-vectorized force function, as the force is calculated during one function call per ball. Another slight downside is the already mentioned loss of flexibility, as all the actuators in the array have to be the same, though this should be acceptable in most cases. Furthermore, it is entirely possible to have multiple arrays in place.

There is another reason to use the vector version of actuators when simulating, that I only realized when deriving the force function for the array. According to [1], the force of all coils is approximately equal to the sum of

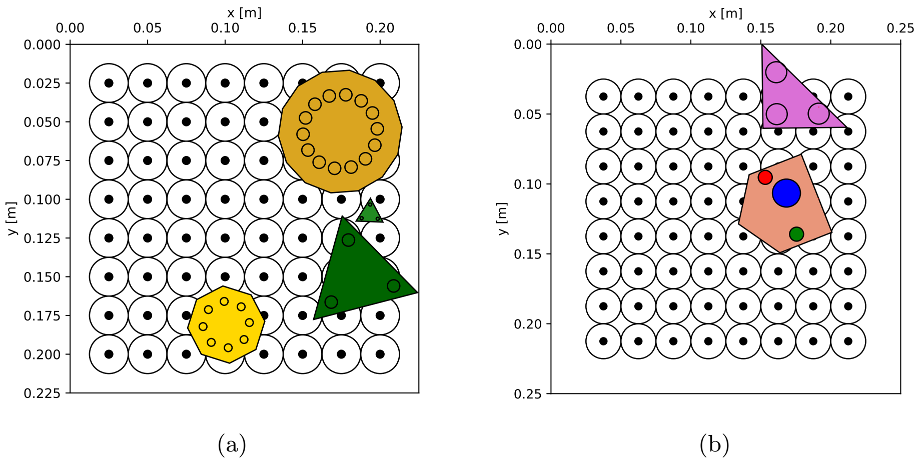


Figure 3.4: Examples of regular polygons with different radii and number of sides, as well as right isosceles triangle and a convex polygon with different ball sizes

forces of the individual coils, i.e.,

$$\nabla \left| \sum_i \mathbf{B}_i \right|^2 \approx \sum_i \nabla |\mathbf{B}_i|^2 \tag{3.17}$$

(recall force equation 1.5).

Let B_{ix} denote the magnetic flux density in the x-axis due to i-th coil. If we expand the left-hand side (only in one dimension for simplicity), we get

$$\frac{\partial |\sum_i \mathbf{B}_i|^2}{\partial x} = \sum_{d\in\{x,y,z\}} \sum_i \left(\frac{\partial B_{id}^2}{\partial x} + 2 \sum_{j>i} \frac{\partial B_{id} B_{jd}}{\partial x} \right). \tag{3.18}$$

Doing the same for the right-hand side, the resulting expression is

$$\sum_i \frac{\partial |\mathbf{B}_i|^2}{\partial x} = \sum_i \left(\frac{\partial B_{ix}^2}{\partial x} + \frac{\partial B_{iy}^2}{\partial x} + \frac{\partial B_{iz}^2}{\partial x} \right). \tag{3.19}$$

Comparing these two results, the following must hold true if they are to be approximately equal:

$$\sum_{d\in\{x,y,z\}} \sum_i \sum_{j>i} \frac{\partial B_{id} B_{jd}}{\partial x} \approx 0. \tag{3.20}$$

However, unfortunately, this does not mean that it is possible to compute the force due to eddy currents in a similar manner. This is because (using

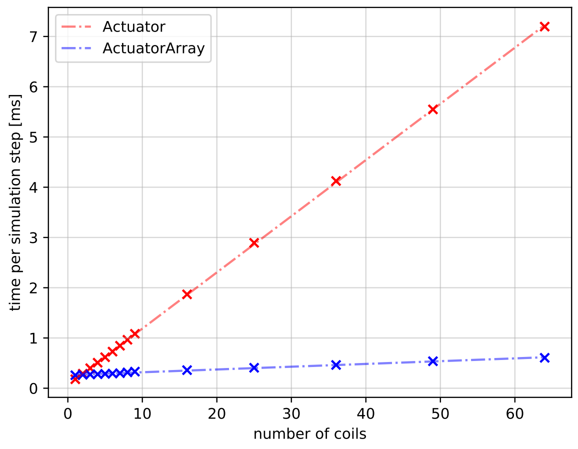


Figure 3.5: Speed comparison of the two classes available to simulate coils

the approximation above)

$$\frac{\partial|\sum_i \mathbf{B}_i|}{\partial x} = \frac{1}{2|\sum_i \mathbf{B}_i|} \sum_{d\in\{x,y,z\}} \sum_i \left(\frac{\partial B_{id}^2}{\partial x} + 2\sum_{j>i} \frac{\partial B_{id} B_{jd}}{\partial x} \right) \tag{3.21}$$

$$\approx \frac{1}{2|\sum_i \mathbf{B}_i|} \sum_{d\in\{x,y,z\}} \sum_i \frac{\partial B_{id}^2}{\partial x} \tag{3.22}$$

$$= \frac{1}{2|\sum_i \mathbf{B}_i|} \sum_i \left(\frac{\partial B_{ix}^2}{\partial x} + \frac{\partial B_{iy}^2}{\partial x} + \frac{\partial B_{iz}^2}{\partial x} \right) \tag{3.23}$$

$$= \frac{1}{2|\sum_i \mathbf{B}_i|} \sum_i \frac{\partial |\mathbf{B}_i|^2}{\partial x}, \tag{3.24}$$

while

$$\sum_i \frac{\partial|\mathbf{B}_i|}{\partial x} = \sum_i \frac{1}{2|\mathbf{B}_i|} \left(\frac{\partial B_{ix}^2}{\partial x} + \frac{\partial B_{iy}^2}{\partial x} + \frac{\partial B_{iz}^2}{\partial x} \right) \tag{3.25}$$

$$= \sum_i \frac{1}{2|\mathbf{B}_i|} \frac{\partial |\mathbf{B}_i|^2}{\partial x}. \tag{3.26}$$

Therefore, it is not possible to compute the friction force due to eddy currents by computing it one coil by one.

I kept the possibility of computing the forces one by one, because for some systems (i.e., by supplying a different force function), it might be possible.

3.2.4 Box2D physics

Simulation is performed in 2D using pybox2d. Ball masses are set according to 1.7, where m is supplied via the shape object. Polygons are simulated as a single body with uneven mass distribution. In Box2D language, this translates to a single body and multiple fixtures. One fixture represents the plastic triangle, and then there is a fixture for each of the balls (defined in the shape object). Moment of inertia is computed by the library based on these fixtures and their positions.

As per the documentation, Box2D is best used for objects between 0.1 and 10 meters. In practice, this meant that objects would collide "too early". I found an approximately 0.01 m gap that is likely used internally as a safety margin. This isn't noticeable for objects in the recommended range but was very noticeable in MagMan simulation. I therefore resorted to internally simulate everything 100x bigger than the real-life and then appropriately rescale. This is only an implementation detail that is not visible outside of the class.

Object masses are defined using density in Box2D, so that was not an issue. Likewise, moments of inertia are calculated automatically. What had to be changed, however, were the forces. For a k-fold increase in length, areas increase by a factor of k^2 and so do masses. All the forces thus had to be scaled by k^2 as well.

Setting the timestep was also important. For force simulation, it was set to $1/50\,\mathrm{s}$ because Box2D is able to deal with collisions without artifacts even at this frequency and the forces acting on the bodies are considered the same during the whole timestep.

For coil simulation, the timestep had to be chosen more carefully. This stems from the fact that in coils simulation, which strives to stay as close to real-world behavior as possible, the force can be rather sensitive to small changes in position. A reasonable compromise between simulation speed is $1/200\,\mathrm{s}$, at which I found the simulation to behave the same as for smaller timesteps.

According to [1], top velocity for oscillation above a coil only reaches about $v_{\max} = 0.1\,\mathrm{ms}^{-1}$. Even at this velocity, a ball will only travel a distance of

$$\Delta x = v_{\max} t = 0.5\,\mathrm{mm}, \tag{3.27}$$

which gives a reasonable approximation.

3.2.5 Simple physics

A lot of the times, it might be useful to have a simpler yet similar enough environment to test out new ideas. This is not necessary for new (or newly implemented) algorithms, as those can be tested on simple environments provided by the Gym library. However, an essential part of creating a performant neural network is designing an appropriate reward function. To this end, I also included a simpler "physics engine" that does not take dynamics into account.

As with regular physics simulation, two simulation modes are possible. When the input is an array of forces, instead of actually applying the forces and thereby causing acceleration, a simple change in position proportional to the requested force is applied. The new position is clipped to stay in the work arena.

On the other hand, instead of the inputs being understood as currents flowing through coils, a change of position is computed according to

$$\Delta\mathbf{x}(\mathbf{i}, \mathbf{x}) = \sum_{m=1}^{M} i_m \frac{\mathbf{x} - \mathbf{x}_m}{\|\mathbf{x} - \mathbf{x}_m\|^3}, \tag{3.28}$$

where $\mathbf{x}_m$ are the actuator positions, $\mathbf{x}$ is ball's positions and i_m are the requested currents. These solutions are conceptually similar yet significantly faster to learn on (sometimes the algorithms only learn on an environment without dynamics, such as in 4.4), not to mention that simulation time is negligible.

There are two key limitations of this simpler "engine". First of all, it only makes sure that the balls do not "escape" the area specified. It does not, however, resolve collisions. Another limitation to keep in mind is the fact that it only supports ball simulation. This is only natural, given the approach this mode is taking - it is simply not possible to simulate rigid body dynamics and their interaction without dynamics.

These inherent disadvantages are not too critical, though, as the goal is not to simulate the environment in lower fidelity but rather to make quick sanity checks and proofs of concept realizable.

Another potential advantage is that should the model not perform well on physics with dynamics, while performing very well on simple physics, the control system could be split into three parts. The first part would be trained on simple physics and would request positions, while other controllers decide what forces to use and how actually to exert them.

This, while certainly not as impressive as having just one neural net, is still useful, if the goal is more abstract. For example, when dealing with distribution shaping as in 4.4, it is easier to decide if the distribution looks uniform, as opposed to directly deciding which ball to move where, which might be necessary with traditional approaches.

3.2.6 MagManView

To visualize what the algorithms are doing, I created a visualization using Matplotlib. Defined shapes, simulation time, and exerted forces are rendered, as well as actuator positions. When simulating coils, the action is also displayed by changing the color of the coil. The MagManView module is designed to work even with actuators with negative action values, in which case, the color is blue instead of red (as illustrated by 3.6).

Visualization of task-specific features was also implemented. For position control (described in 4.2), visualization of goals is implemented as illustrated by 3.7. For coil simulation, target force is shown as in figure 3.8.

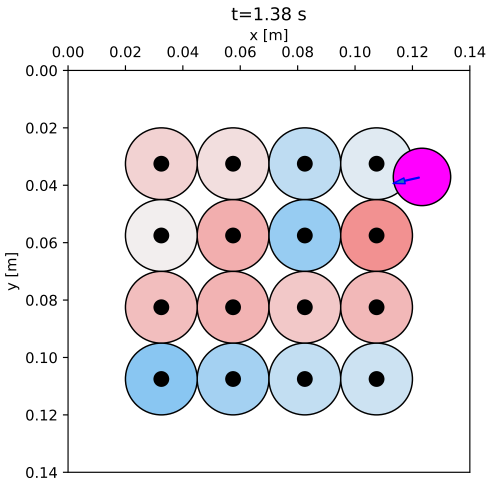


Figure 3.6: Visualization of actuator actions

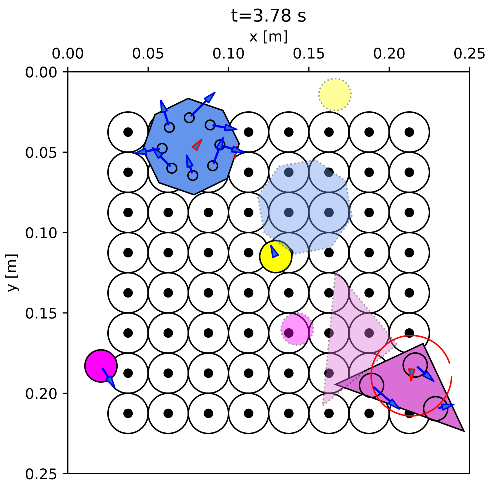


Figure 3.7: Visualization of target positions (partially transparent)

It is also possible to easily export simulation as an mp4 video (ffmpeg installation needed).

3.2.7 Usage

The goal of this section is to provide a simple user manual. I go over the parameters used to initialize the environment, as well as a way to change the number of objects (randomly) when resetting the environment.

To set environment dimensions, set the `width` and `height` parameters. A border is created around the area $[0, \texttt{width}] \times [0, \texttt{height}]$ past which objects cannot get.

Using `simulation_mode`, one may pick either `'forces'` or `'coils'`. If picking coils, the actuators (as described in 3.2.3) to be used in simula-

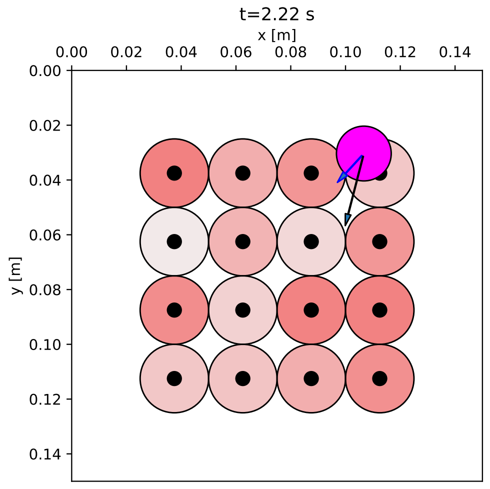


Figure 3.8: Visualization of target force (in black) and actual force (in blue)

tion, are specified using parameter `actuators` (passed as a list). To choose `SimplePhysics` (in which case only actuator positions will be taken into consideration), set the `simulate_dynamics` switch to `False`. Otherwise, simulation will use "realistic" physics, possibly using the actuators defined (if simulating coils).

Choosing the objects for simulation is similar to defining the actuators. A list of shapes (3.2.2) is passed as a variable called `objects`.

MagManEnv also allows for measurement inaccuracy simulation. Setting `position_error_mean` and `position_error_std` will configure the normal distribution from which artificial measurement error will be drawn. Please note that errors are not added to velocity, which although part of the observation if `share_velocity` is true, is not be measured by real MagMan. Delay can be specified using `delay`.

Last but not least regarding initialization, setting the `render_mode` to `'live'` will allow for calling `render()` at later time to show current simulation state, while setting it to `'video'` will take care of rendering a snapshot at 50 FPS and saving it as an mp4 file with name specified in `video_name`.

In order to change the number of objects, set parameters `min_objects` and `max_objects` when calling the `reset()` function. A random subset of objects within the specified bounds will be used for simulations. Observation space stays the same; its parts related to unused objects will be filled with dummy values (-1).

In case repeatable experiments are necessary, such as when different algorithms are being compared, there are additional parameters to the `reset()` function. Setting `goals`, `starting_positions` and `starting_velocities` allows the user to configure the experiment as needed.

Chapter 4
Tasks

In this chapter, several tasks' formulations and results are presented. As in the current solution of the control problem (see 1.4), I split the problem into two (three) parts.

A significant advantage of this approach is, that should the neural networks outperform the current algorithm in one part of the problem and fall behind in the other, we can combine the two methods to get the best of both worlds. Even if the neural networks outperformed the traditional approach in both of these problems, it is possible to only train one coil-controlling NN for a great variety of tasks, saving time that would have to be spent on training.

For all training, I used the PyTorch implementation of Adam Optimizer[14] and three training runs. The results are then presented as the running average (of the last 10 rewards) with a range of between the lowest and highest rewards (also smoothed) shown as a partially transparent band of the same color.

For many of the problems, I tested different hidden layer combinations and algorithms. For example, a curve labelled as *SAC (3x256)* means that the particular run was a NN trained with the SAC algorithm with three hidden layers of size 256, while *DDPG (32-64-32)* would mean training with DDPG and three hidden layers, first and third of size 32, the middle of 64.

All the training curves are attached in Appendix B.

4.1 Challenges

4.1.1 Variable number of objects

A problem with neural networks is their fixed input size. Therefore, changing the number of objects mid-run poses a great problem. Several solutions are possible, as described below.

Switching networks ad-hoc

Probably the simplest way to tackle this problem is to simply train a handful of neural networks. After object detection, the network with the appropriate input dimension, based on the number of objects detected, could be picked.

This is the way to go if other approaches fail. The obvious downside is the need for having many networks at hand.

Multiple forward passes through the neural network

For some problems, such as position control, it is possible to simply have one NN that takes observation of only one object as an input. In case more objects are to be controlled, the neural network would run multiple times. Better yet, if GPU is available, the computation can be run in parallel as a batch (such as in training).

The obvious problem of this approach is the fact that the neural network does not have any information about other objects. Therefore, it cannot be trained, for example, to avoid collisions between objects when dealing with position control, which may be desirable. In addition to that, some problems, such as distribution shaping (4.4) or controlled mixing (4.5) cannot be performed without knowledge about the other objects.

Padding

Another rather straightforward approach would be to decide the maximum number of objects to control. In case an object is missing, simply fill the observation with dummy values that do not occur naturally. Because positions are non-negative values (both in the simulator and in real MagMan), such value can be -1.

This is the approach I went with, due to limited time. However, I also suggest possible further approaches below.

Recurrent neural network

A more advanced method for dealing with variable input size, are RNNs. This is an architecture in which some sort of hidden state is kept; the output then depends not only on the input but also on the inner state. A popular example of recurrent NN architecture is the Long Short-Term Memory (LSTM) network (in figure 4.1). Their main advantage, compared to more simple RNNs, is their capacity to learn dependencies over long sequences. [2]

Nowadays, a lot of the research is in the area of natural language processing, where sequences of words of different lengths are abundant. Notably, Google with their transformer network with multiheaded attention in [15] has reached significant improvements in sentence translations.

I believe that it would be worth it to try tasks 4.4 and 4.5 with a variable number of balls using this method. The reason for this is that much like in sentence translation, great importance lies in the relationship between the objects, while not much can be said about any individual one of them.

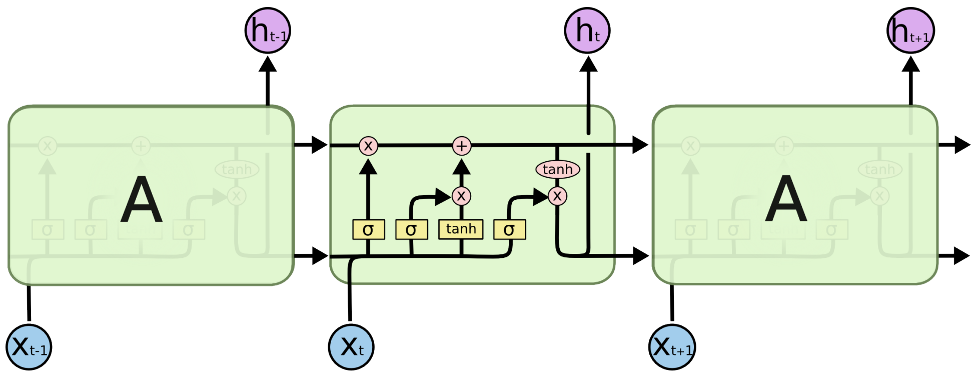


Figure 4.1: Scheme of an LSTM network (reprint from [2])

Convolutional neural network

Another possible approach is using a convolutional neural network (CNN). This is, however, only applicable if the observation is formulated differently. The idea here is that the observation would not be on a per-object basis but rather dependent on the size of the work arena, which stays fixed. For example, the area could be split into 512x512 rectangles, and the observation would effectively be an image (in black and white). Its values would be 0 in "pixels" there is no ball and 1 where there is. Another option would be to use the representation derived in 4.4 for the reward function and normalize the function values to $[0, 1]$.

The input to the network would then consist of two parts. One, representing the high-level view of the whole arena, is fed into the CNN, and the output would then be concatenated with ball position. This vector would then serve as input for the second part of the NN.

4.1.2 Partial observability

Another significant problem for real-time control is the fact that the system is a POMDP (2.1.1). Three factors come into play:

- time delay (40 ms)
- velocities are not part of the observation
- noisy measurement ($\sigma = 0.7\,\text{mm}$)

Again, there are different methods to deal with this. For example, if the delay is not significant, it is possible to just augment the observation space by all the relevant observations and actions.

A more advanced and promising approach is the Stochastic Latent Actor-Critic [16] (SLAC) that uses the SAC algorithm (described in detail in 2.2.2) together with a variational auto-encoder to predict the true states based on observations. This is done by teaching it to correctly predict future observations. In the original paper, the authors successfully learn SOTA policies directly from images. Further relevant work similar to SLAC was described in [17], where the authors solved POMDP control tasks, again reaching SOTA performance.

4.1.3 Scalability

As the number of objects increases, the networks seem to perform worse and worse (or at least take longer and longer to reach the same level), even if from the point of view of a human, the complexity does not increase so drastically. This is nicely illustrated in 4.2.2.

The problem lies in the fact that the networks seem to have trouble learning the abstract pattern. An approach to solve this is called curriculum learning. In [18], the researchers have proposed a curriculum learning framework that they used to teach a robotic arm to stack cubes on top of each other. Conventional architectures have not succeeded in teaching the policy to stack more than one block, while their solution managed to stack 5.

4.2 Position control

The goal of this thesis was to explore what opportunities the RL framework offers. Therefore, even though it was not technically required, I decided to first focus on position control, which has a more clearly defined goal than the following tasks.

Another advantage of starting with position control is that it is solvable using a simple PID controller. As a result, there is some sort of a reference to compare the neural network's performance to, whereas in tasks 4.4 and 4.5, there is a lack of such baseline.

In this case, the input to the simulation are directly the forces and it is assumed that a different controller solves the problem of finding the currents, as illustrated in figure 1.4.

4.2.1 Reward function

The loss function for N balls was calculated as

$$L = \frac{1}{N} \sum_{i=1}^{N} \left(\|\mathbf{x}_i - \mathbf{x}_{i,\text{target}}\| + \|\mathbf{a}_i\|^2 \right) \tag{4.1}$$

and the condition for determining if the task has finished was

$$\|\mathbf{x}_i - \mathbf{x}\| < 5\,\text{mm} \quad \wedge \quad \|\mathbf{v}_i\| < 1 \times 10^{-2}\,\text{ms}^{-1} \quad \forall i \in \{1, \ldots, N\}. \tag{4.2}$$

An extra reward of 1 (so $L = -1$) was given for each ball passing this criterion, and an additional reward of 10 ($L = -10$) was given for finishing the task completely. The episode was stopped after 500 steps $\sim$ 10 s if the task had not been finished before that.

4.2.2 Results

Single object

A few different experiment settings were tested. For runs with shared velocity, the observation was

$$\omega_n = (x_n, y_n, \dot{x}_n, \dot{x}_y, \text{goal}_x, \text{goal}_y), \tag{4.3}$$

while runs without it had observations consist of

$$\omega_n = (x_n, y_n, \text{goal}_x, \text{goal}_y). \tag{4.4}$$

In case there was delay of k frames, the observation was created as a concatenation

$$\omega'_n = (\omega_{n-k}, a_{n-k}, \omega_{n-k+1}, a_{n-k+1}, \dots, \omega_n). \tag{4.5}$$

The same applied to the experiment, in which no velocities were shared and the networks were given the past observations and actions ($k = 2$) to be able to approximate the velocities themselves. The actions were bounded to $\mathbf{a}_i \in [-0.2, 0.2]^2$.

The results are shown in B.1. As we can see in the figure, even simple networks reach comparable results as the complex ones in this simplest case, regardless of the method. An interesting observation we can make is that a delay of 40 ms does not hinder the performance. On the other hand, if velocities are not available, the networks fail to learn a reasonable policy even after 400 episodes with no signs of improvement.

An interesting observation I made that cannot be seen from the rewards directly is that the polices (in general) seem to get stuck in what I suppose are local minima. In these, the final position (when the the NN does not exert any force anymore) of the ball, is very close to the goal but does not quite reach it (see figure 4.2). I was unable to find an issue in the code, nor am I able to find a logical explanation for this problem. A clue could, perhaps, be that this issue got even more pronounced when controlling multiple balls.

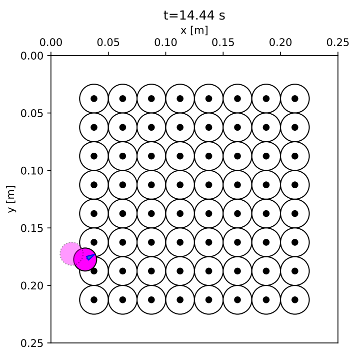
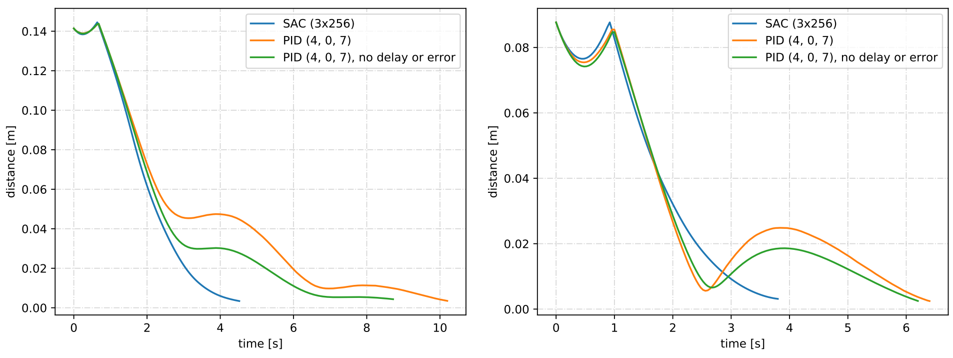


Figure 4.2: Example of a non-zero final error

I tried comparing the resulting policy with a PID controller; results are shown in figure 4.3. We can see that the resulting policy has comparable performance. A few things should be noted, however.

First of all, the PID controller is likely sub-optimal. Nonetheless, it works reasonably well (as shown by the run without delay and measurement noise) and the goal of the test was not to establish that NN-based control is superior but rather to show that these methods are roughly in the same ball-park. Second, the neural network had an unfair advantage by having access to velocities, which the PID did not have. On the other hand, the neural network was trained without noise, so it is possible that it could potentially improve after a few episodes with noise.



(a) (b)

Figure 4.3: Example of two runs with a random initial velocity, position and goal, with a delay of 2 frames and random measurement error ($\sigma = 0.7\,\text{mm}$)

Multiple objects

To test the dependence on the number of objects being manipulated, I also ran the experiment without delay and with 2 and 5 balls with velocities. In the case of multiple balls, the observation sent to the NN was a concatenation of the observations of each of the balls. The results are shown in B.2.

We can observe that controlling two balls does not seem to be that much harder of a task for most of the networks, while five balls pose a significantly greater challenge.

4.3 Coil control

In coil control, the goal was to find currents to pass through the coils such that the force exerted on the ball(s) matches the desired force. This is an essential part of the whole control loop, as seen in figure 1.4.

4.3.1 Reward

I proposed two rewards. First one is a simple euclidian distance, i.e.,

$$L_1 = \sum_i \|\mathbf{F}_i - \mathbf{F}_{i,\text{target}}\|, \tag{4.6}$$

where $\mathbf{F}_i$ is the force exerted and $\mathbf{F}_{i,\text{target}}$ is the desired force.

In the second reward, I tried to penalize the angle more strictly:

$$L_2 = \sum_i \cos(\varphi_i) + (\cos(\varphi_i) - 1.5)\|\mathbf{F}_i - \mathbf{F}_{i,\text{target}}\|, \tag{4.7}$$

where

$$\varphi_i = \frac{\mathbf{F}_i \cdot \mathbf{F}_{i,\text{target}}}{\|\mathbf{F}_i\|\|\mathbf{F}_{i,\text{target}}\|} \tag{4.8}$$

is the angle between the desired and the exerted force.

For this task, I decided to limit the number of steps per episode to 100, because if the target force was met, the ball would likely be out of the work arena even before that.

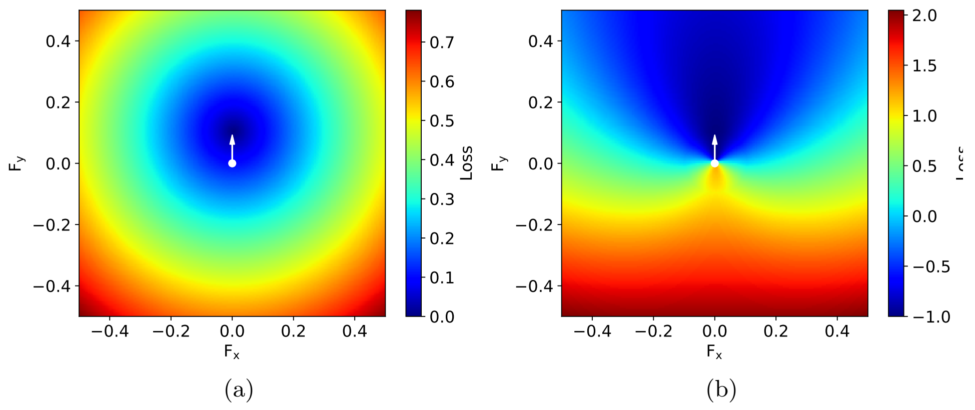


Figure 4.4: Visualization of the the loss functions for coil control (requested force as a white arrow), (a) L_1 (4.6), (b) L_2 (4.7)

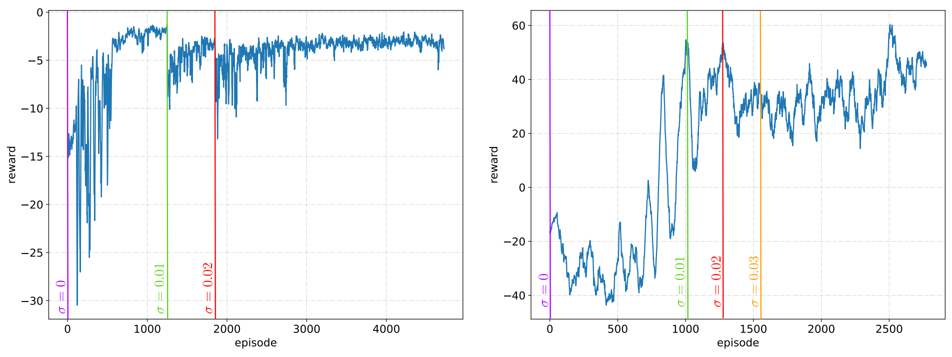
4.3.2 Results

The neural networks were unable to learn anything using either of the loss functions, even on a 4x4 coils arrangement with a single ball, no delay, and known velocities. They were, however, able to learn to exert a correct force if the starting position was always the same (in the middle).

I then tried sampling the initial position from a normal distribution with mean in the middle and non zero variance. By gradually increasing the variance, I managed to reach a level where, while far from perfect, the exerted force is mostly in the direction of the desired force for positions near the center.

Therefore, I believe it might eventually be possible to teach the network to control the coils by gradually increasing the variance, as illustrated in 4.5.

Nonetheless, due to a lack of time, I decided not to pursue this task any further after consulting it with my supervisor. The issue is that doing it this way is time-consuming, but most importantly, we already have a solution to this problem, and we are currently satisfied with its performance.



(a) (b)

Figure 4.5: Example of how gradually increasing σ seems to aid the learning process of coil control, (a) L_1 (4.6), (b) L_2 (4.7)

4.4 Distribution shaping

Another desirable ability is to be able to shape a collection of objects in a certain way. The difference between this task and position control, is that instead of focusing on each of the objects' position separately, the positions are viewed in the context of all the others. Unlike in classical position control, switching two balls does not change the perceived state.

For smaller ensembles, such as those presented here, it may be useful to spread objects initially concentrated at one part of the work arena uniformly all over the surface. Once separated, certain tasks, such as picking up a single object, might result to be easier.

On MagMan's scale, this may allow for, say, a robotic hand with a magnet pick to one object at a time for further processing. On a smaller scale, also studied by AA4CC, where manipulation might be done, for example, using electrophoresis ([1]), picking a single object is even harder, and the benefits could potentially be even more significant.

The exact problem formulation presented here does not extend to collections of thousands or even more objects. This is because setting forces to each object independently becomes practically impossible, as the number of objects increases. However, the reward function presented below should, in fact, still be able to capture the problem. As a far-fetched goal, such technology could possibly aid in producing metamaterials, where it might be necessary to distribute some compound in a liquid evenly.

As in 4.2, the inputs to the simulation are directly the forces and the problem of finding the currents to pass through the coils is left to a different controller, as shown in 1.4.

4.4.1 Reward function

A chief challenge in this task was to define a criterion (reward function) that would be general enough to be applicable to different distributions. If this condition were relaxed, the task would be much simpler. Say only uniform distribution were to be solved. In that case, it is possible to just penalize how close the ball is to others, but alas, that does not generalize well to other distributions.

Below, I briefly go over some of the blind alleys I took, as well as my proposed solution, which I then use in the next section.

Kolmogorov-Smirnov test

First, I turned to statistics, in particular to the Kolmogorov-Smirnov test, which is a test to compare a sample with a reference distribution, defined as

$$D_n = \sup_x |F_n(x) - F(x)|, \tag{4.9}$$

where $F(x)$ is the reference distribution's cumulative distribution function (CDF) and $F_n(x)$ is the portion of samples that satisfy $x_n < x$ defined as

$$F_n(x) = \frac{1}{N} |\{x_n \,|\, x_n < x\}|. \tag{4.10}$$

However,this definition is only valid for one-dimensional distributions. Therefore, I decided to take one for each dimension (as they are independent for uniform distribution) and sum the results.

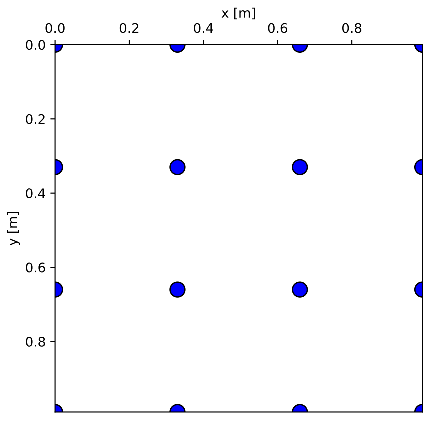
The reason why this approach does not work even for uniform distribution is illustrated in 4.6.

Nearest neighbor

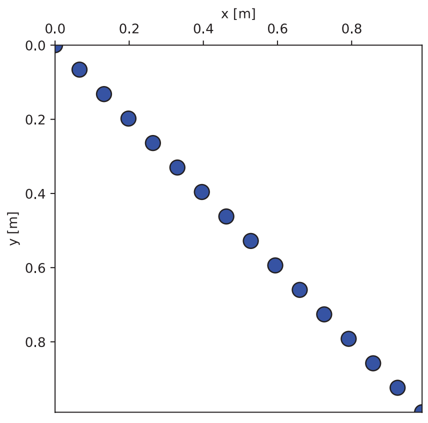
Having failed at the previous attempt, I tried to come up with a criterion based on the density function.

One approach I tried was splitting the area into regions based on the ball that was nearest (making a Voronoi diagram). Integrating the target density function over each of these areas should be equal to $1/N$ in the case of ideal ball distribution.

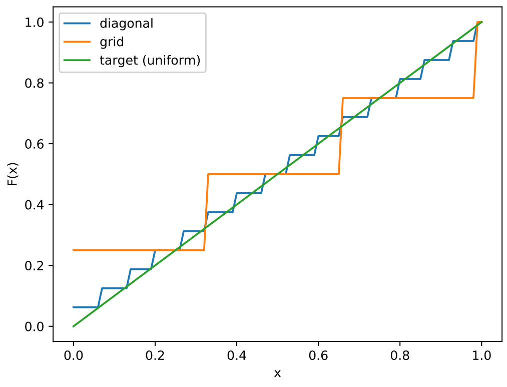
The problem is that this does not happen only if the above holds true. The policy that the neural network learned, though minimizing the criterion, was not exactly what was desired, as illustrated by 4.7.



(a)



(b)



(c)

Figure 4.6: Illustration of why the 1D K-S test fails at capturing the uniformity distribution shaping task, (a) Grid "distribution" - what would one expect to learn, (b) Diagonal "distribution", (c) Comparison of the CDFs in the x-axis

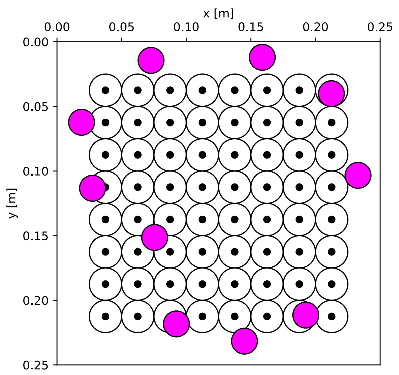


Figure 4.7: Final state caused by a policy learned with the nearest neighbor reward function

Density function norm

The final version that seems to be most versatile, while being conceptually simpler than criteria presented above, is based on the idea of function norm. The p-norm of a function u (u is μ-measurable on X) is defined as

$$\|u\|_p = \left(\int_X |u|^p \,\mathrm{d}\mu\right)^{1/p}. \tag{4.11}$$

In particular, we desire the density g induced by the current ball position to be as close to target distributions' density function f as possible. Therefore the goal is to minimize

$$\|f-g\|_p = \left(\int_X |f-g|^p \,\mathrm{d}\mu\right)^{1/p}. \tag{4.12}$$

The question now is how to get the density function. The first idea would be to define it as a sum of Dirac functions at each of the ball's locations. However, since the integrand would be equal to f almost everywhere, the resulting integral would just equal to $\|f\|$.

Instead, I represented the position of the ball itself by a function similar to the probablity density function (PDF) of normal distribution as

$$g_i(x,y) = k\exp\left(-\frac{(x_i-x_0)^2+(y_i-y_0)^2}{\sigma^2}\right), \tag{4.13}$$

where $k, \sigma \in \mathbb{R}$ are constants, x_0, y_0 is the position reported by the environment.

There are a few caveats, however. First of all, unless the condition that the area (volume) under the curve (surface) should equal to one is relaxed, the value of the integral would be

$$\iint_{\mathbb{R}^2} \sum_i g_i = N, \tag{4.14}$$

where N is the number of objects. Since it is not desirable for the loss of the optimal solution to go to infinity as $N \to \infty$, I decided to have 4.14 equal to 1, which is ensured by picking correct values of both σ and k. Next, I will make a few observations for uniformity distribution to find reasonable values for these variables.

As σ decreases, the actual ball positions matter less and less because the "area of effect" of each ball goes to zero. As a result, for a small enough σ, the reward would essentially be the same regardless of the actual positions, as in the first row of figure 4.8, not to mention the difficulty of numerically computing the integral because of the resulting high derivative of the function.

Conversely, if the σ is too high, the position does not matter either as the function value would be almost uniform over the whole area of integration. We need to find such σ that steers clear of both of these extremes.

Two variables come into play - the number of objects N and the area S. Because g_i, like normal distribution density, decreases fast as the distance from the actual position increases, it is possible only to consider a finite area A_i in the shape of a circle, where the ball is still "relevant". As a rule of thumb, let it be equal to the area, integration over which gives us 90 % of the total value of the integral.

To strike a balance between the two extremes, let's put a condition in place that

$$S = \sum_i A_i = N\pi R^2, \tag{4.15}$$

where R is the radius that defines A_i. This condition should allow for coverage of the whole area by the balls, if necessary, while keeping σ high enough.

To find σ, we can compute the volume under the surface using transformation to polar coordinates as

$$\iint_{\mathbb{R}^2} g_i = k \lim_{\rho\to\infty} \int_0^\rho \int_0^{2\pi} re^{-r^2\sigma^{-2}} \,\mathrm{d}\varphi\,\mathrm{d}r = k\pi\sigma^2 \lim_{\rho\to\infty} \left(1 - e^{-\rho^2\sigma^{-2}}\right), \tag{4.16}$$

which describes how the percentage of the value depends on the radius of the integration area (fixed by 4.15) and σ. To find *sigma*, for which 90 % of the total "volume" lies above the integration area, we can do

$$0.9 = 1 - e^{-R^2\sigma^{-2}}, \tag{4.17}$$

$$\sigma = \sqrt{-\frac{R^2}{\ln 0.1}} = R\sqrt{\frac{1}{\ln 10}} \tag{4.18}$$

(where R can be computed using 4.15).

To find the corresponding value of k, we notice that if the integral is to be equal to $1/N$, then

$$k\pi\sigma^2 = \frac{1}{N} \to k = \frac{1}{N\pi\sigma^2}. \tag{4.19}$$

The final loss function was computed as

$$\int_{x_{\min}}^{x_{\max}} \int_{y_{\min}}^{y_{\max}} \left(f(x,y) - \sum_i g_i(x,y)\right)^2 \mathrm{d}x\,\mathrm{d}y + \|\mathbf{a}\|^2, \tag{4.20}$$

where the work arena is the area $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$.

This approach has several benefits. It can very naturally be extended to mixing (4.5) and it generalizes well for different continuous probability distributions and even to 3D probability distributions.

For each of the balls, the observation was

$$\omega_n = (x_n, y_n, \dot{x}_n, \dot{y}_n) \tag{4.21}$$

or just the first two terms, if velocities were not shared. The resulting observation was again a concatenation of these partial observations. This means that in case we wanted to train one network for parametrized distributions (for example variable variance of the normal distribution), changes would have to be made.

4.4.2 Results

When dynamics are not considered, the NNs learn a very reasonable policy even for 20 balls. An example of final positions for uniformity and normal distribution is shown in 4.9. Unfortunately, for a system with dynamics, even without delay and with shared velocities, no reasonable policy was found even with a more complex network and 800 episodes (only 400 shown in the training curve). Results can be seen in B.3 for uniformity distribution and B.5 for a normal distribution.

This problem should theoretically be solvable by employing three separate controllers — one without dynamics to request a position for each of the balls, another one to request a force and a third one to find the currents to pass through the coils.

I also tested how do the NNs cope with a variable number of objects. The results shown in figure B.4 might not look too different from B.3a). However, if you look closely, you will notice that the partially transparent bands are much wider. This is because while not too much worse at shaping the distribution of 10 objects than networks only trained for 10 objects (though still noticeably inferior), the NNs failed to learn much for a different number of objects.

4.5 Feedback-controlled mixing

In this task, the goal was to mix two groups of bodies, initially separated, in a controlled manner. While, perhaps, not evident at first sight, this is rather similar to the previous task of distribution shaping.

For large amounts of manipulated objects, feedback-controlled mixing could theoretically aid self-assembly. In this process, there are objects that "know" how to assemble themselves once they are close to each other. On a microscopic scale, the forces guiding the final assembly could be of chemical nature, while on a macroscopic scale, this could be done by the objects fitting into each other due to their shapes. However, in order for this to come into play, the objects have to be close to each other, which could be done using this approach. Like in the case of distribution shaping, the reward is what should still be relevant, not the exact problem formulation.

4.5.1 Reward

The reward function for feedback-controlled mixing is a very natural extension of the reward for 4.4. For two groups of bodies labeled r, b (for red and blue respectively), we can define

$$g_r = g_i \tag{4.22}$$

$$g_b = -g_i. \tag{4.23}$$

The goal of the NN is then to minimize the loss

$$L = \left\| \sum_r g_r - S \right\| + \left\| \sum_b g_b + S \right\| + \left\| \sum_b g_b + \sum_r g_r \right\|, \tag{4.24}$$

where S is the area of work space.

The last term ensures that the objects are mixed, while the first two ensure that the balls are spread across the whole surface evenly. The reason I decided to include the first two terms was to avoid a local minimum, in which the policy would simply learn to send all the balls into one corner (which is very easy), without actually putting effort into mixing the balls.

4.5.2 Results

The results were similar to the previous task, which was to be expected, given how similar the tasks are. The algorithms were able to find a reasonable policy even for 20 objects without dynamics (example of a final state is shown in figure 4.10) and failed at the system with dynamics. The training curves are shown in figure B.6.

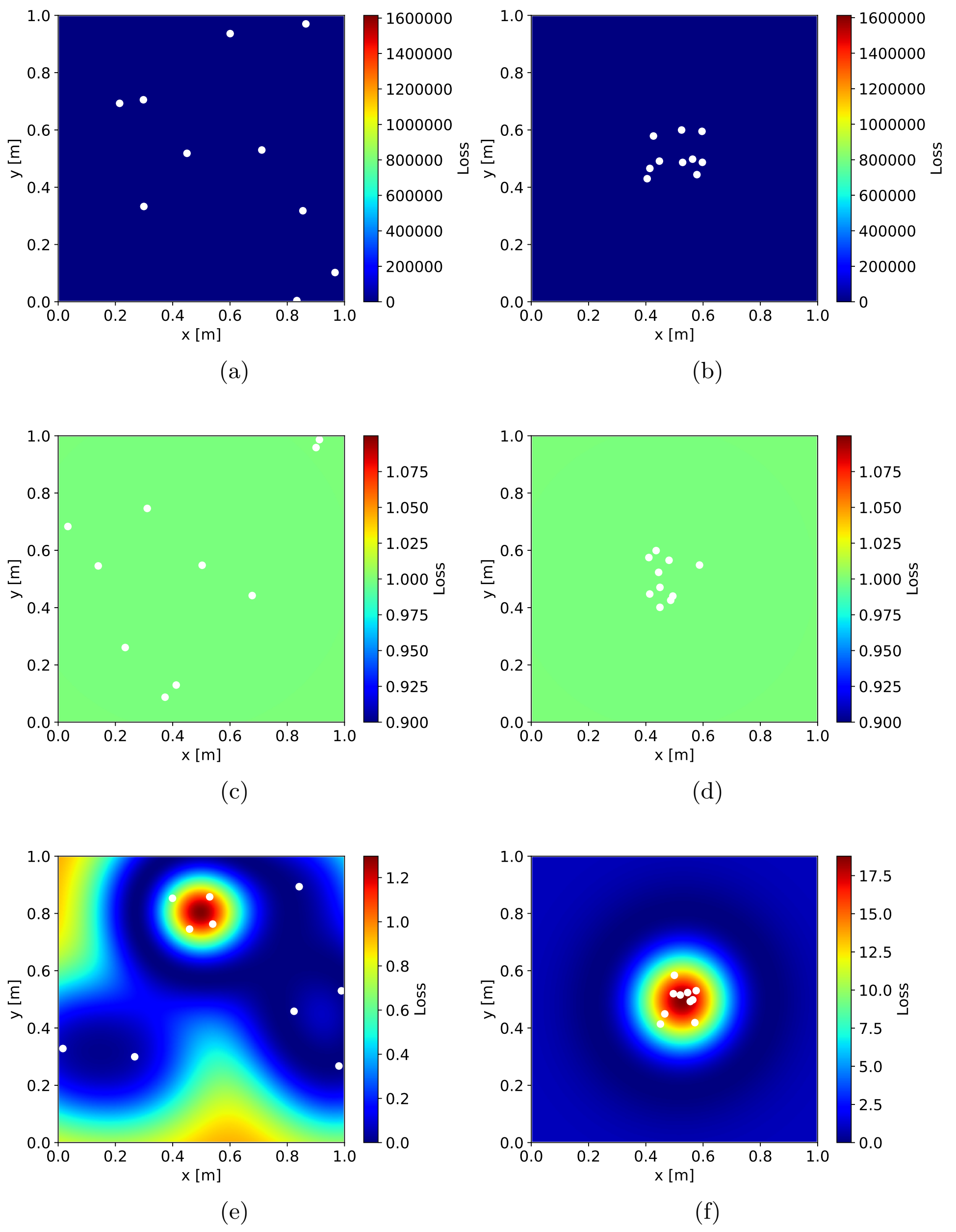
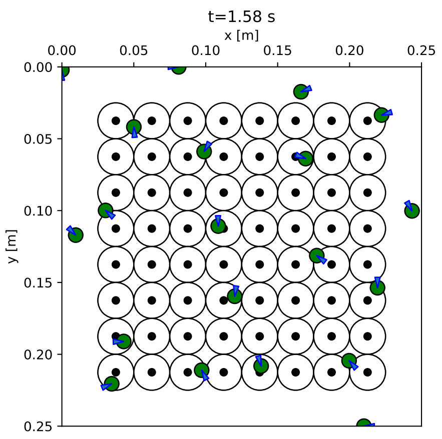
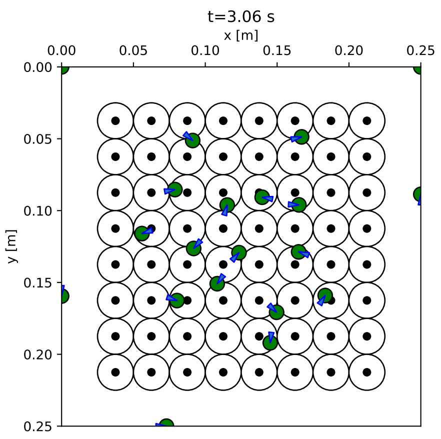


Figure 4.8: Illustration of states with the same uniformity reward for different values of σ. Only in the last row is there a difference between the integral values. Left: random positions from $[0, 1] \times [0, 1]$, right: random positions from $[0.4, 0.6] \times [0.4, 0.6]$. (a), (b): $\sigma = 5 \times 10^{-3}$ (too low), (c), (d): $\sigma = 5 \times 10^{3}$ (too high), (e), (f): $\sigma = 0.24$ computed using 4.18.



(a)



(b)

Figure 4.9: Examples of final states of the systems (no dynamics), (a) uniform distribution, (b) normal distirbution

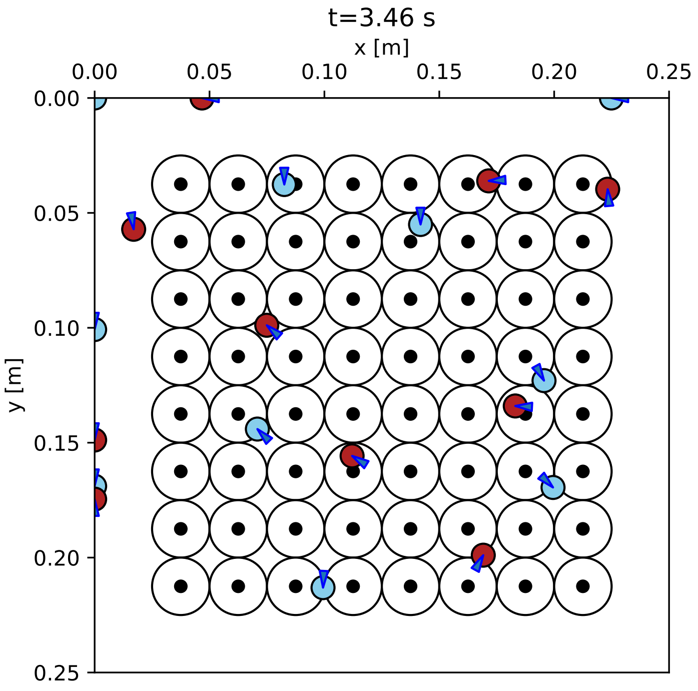


Figure 4.10: Example of a final state for mixing (no dynamics)

Chapter 5
Conclusion

The goal of this thesis was to explore the opportunities the RL framework offers to the area of feedback control of dynamic systems. For a concrete physical system, we decided to use the experimental magnetic manipulation platform called MagMan, developed by AA4CC at the Faculty of Electrical Engineering, Czech Technical University in Prague.

In this thesis, I developed a simulator of said system in Python, capable of simulating the movement not only of steel balls but also of a special type of plastic polygons that contain steel balls. Two simulation modes are available. It is either possible to simulate the whole system, where the inputs to the simulation are currents flowing through coils, or to skip coil simulation and have forces to be the inputs directly. While the first mode was implemented directly using previous research done in [1], for the latter, it was necessary to derive some approximations for the resistive forces.

Two actor-critic RL methods were presented in this text: Deep Deterministic Policy Gradient and Soft Actor-Critic. The latter algorithm was published last year and currently achieves SOTA performance, while the former was one of the first "functional" algorithms ever to be published. In the experiments, I found out that for low-dimensional state and action spaces ($\dim \mathcal{S} \leq 10, \dim \mathcal{A} \leq 4$), the two methods achieve very similar results across different hidden layer configurations. However, SAC learns a significantly better-performing policy for high-dimensional $\mathcal{S}$ and $\mathcal{A}$, despite only having been published three years later. Based on these results, I believe that there is no reason to keep using DDPG for further experiments.

I then used these methods to solve several tasks with varying levels of success. The neural networks managed to learn position control of steel balls successfully, regardless if a delay of 40 ms was present or not. However, this was only achieved if true velocities were known. The networks were unable to learn any reasonable policy if only past actions and positions were known, from which current velocity could be approximated using standard methods. I believe that it would be beneficial to employ the architecture presented in [17], where the authors used a variational auto-encoder to learn a latent space representation of the actual states and control was then performed using the latent vector as input.

Afterward, I tried to solve the other part of the problem, where the networks were to learn a policy that would control the coils in such a way that the exerted force on the ball would match the desired one. The results were not too impressive, as the only policy that did what was desired was only able to deal with positions close to the center of the work arena.

The last task I tackled was controlling ball positions so that they would look as if they were sampled from a specified distribution. I devised a reward based on the idea of function norm to train the networks. A similar reward was used to condition the policies for feedback-controlled mixing. This approach led to reasonable policies for a system with no dynamics with 20 balls, represented by points. However, results obtained on systems for dynamics did not turn out to be performant in either of the problems.

Appendix A

Used acronyms

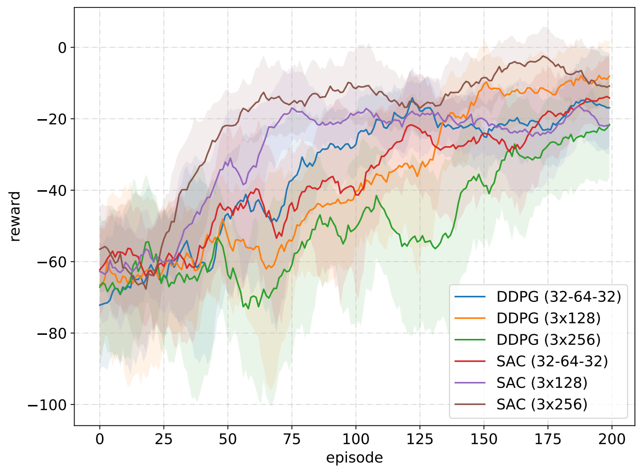
AA4CC	Advanced Algorithms for Control and Communications
CNN	convolutional neural network
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q-learning
FPS	frames per second
GPU	graphics processing unit
LSTM	Long Short-Term Memory
MDP	Markov decision process
ML	machine learning
MSE	mean squared error
NN	neural network
POMDP	Partially observable Markov decision process
RL	reinforcement learning
RNN	recurrent neural network
SAC	Soft Actor-Critic
SOTA	state-of-the-art

Appendix B

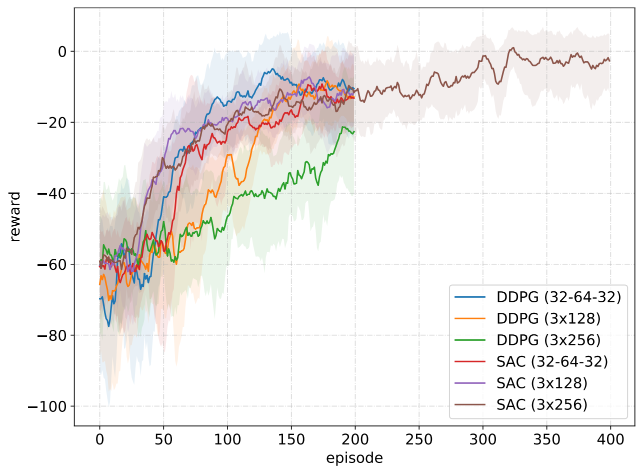
Training curves

Training curves of the several tasks solved by this thesis are presented below. The tasks were the following:

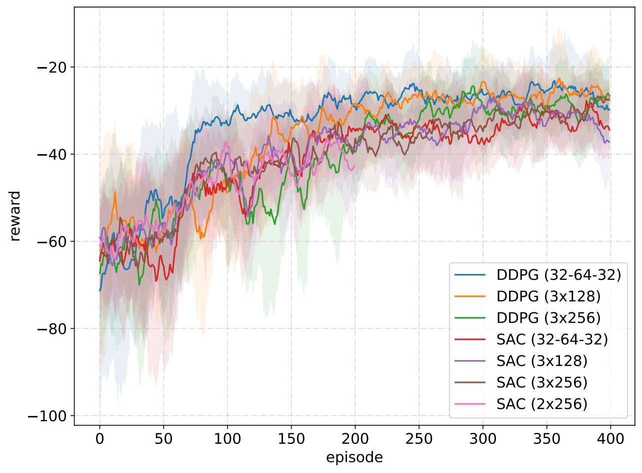
- position control (4.2): figures B.1 and B.2
- distribution shaping (4.4): figures B.3, B.4 and B.5
- mixing (4.5): figure B.6



(a)

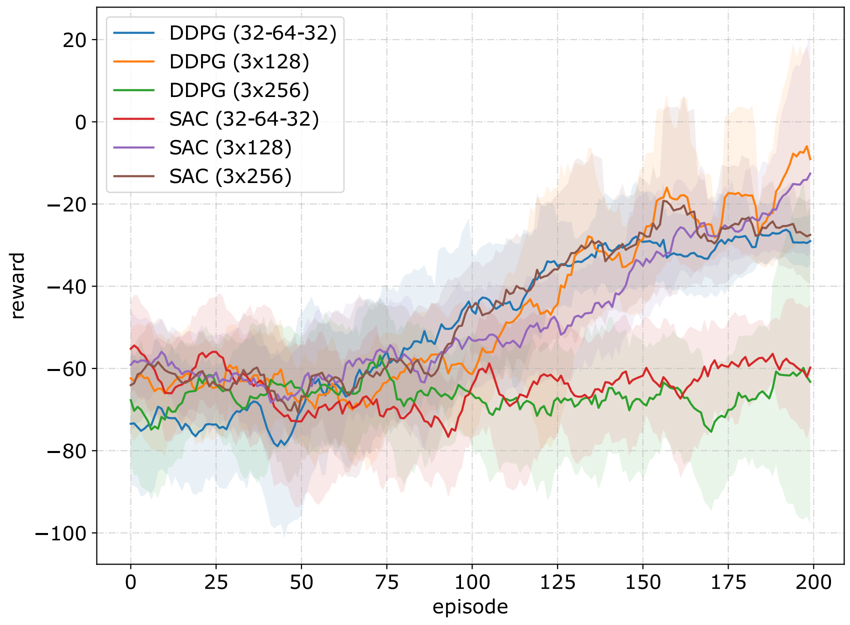


(b)

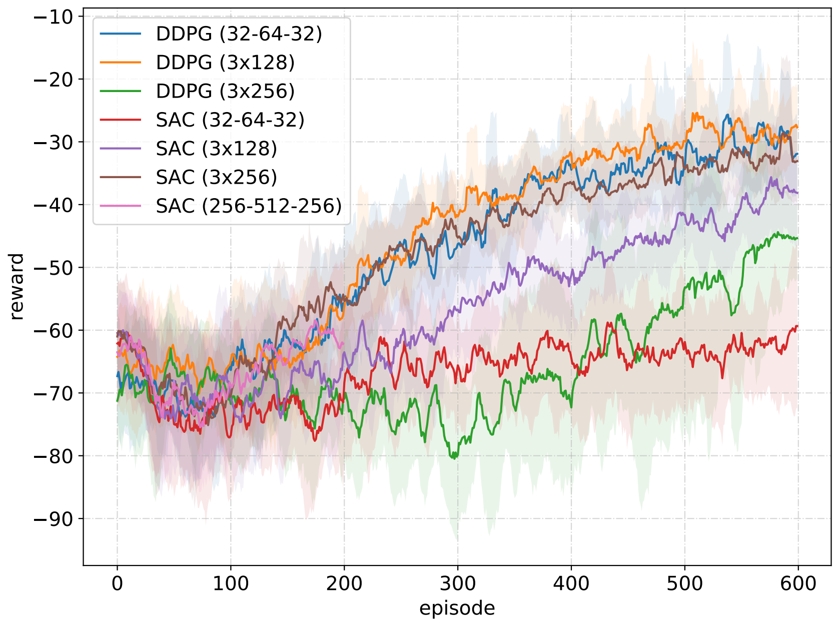


(c)

Figure B.1: Position control of one steel ball, (a) no delay, with velocities, (b) with a 40 ms delay, with velocities, (c) no delay, no velocities, past three observations and actions

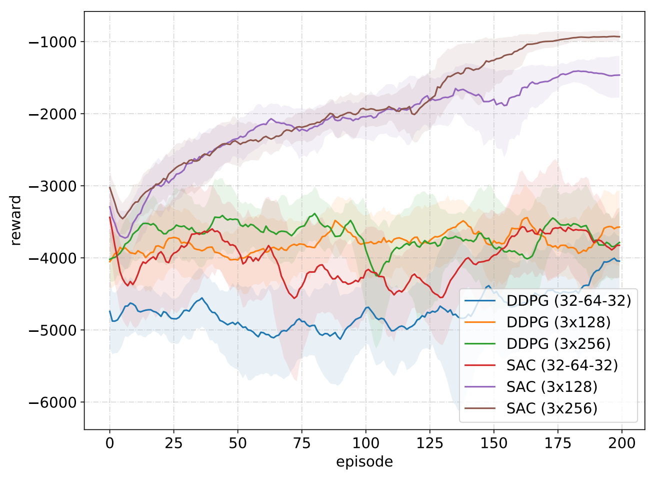


(a)

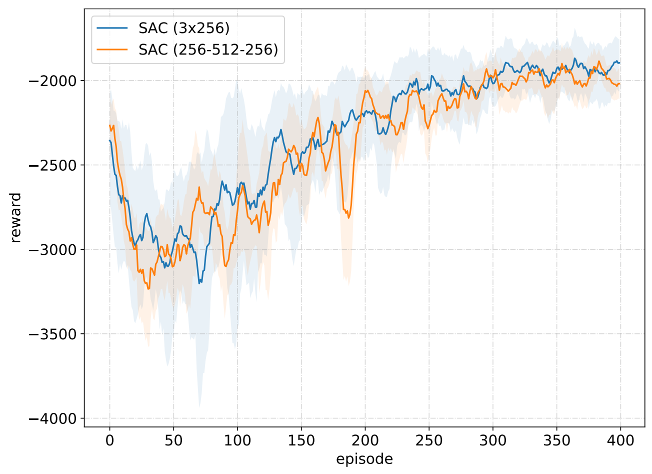


(b)

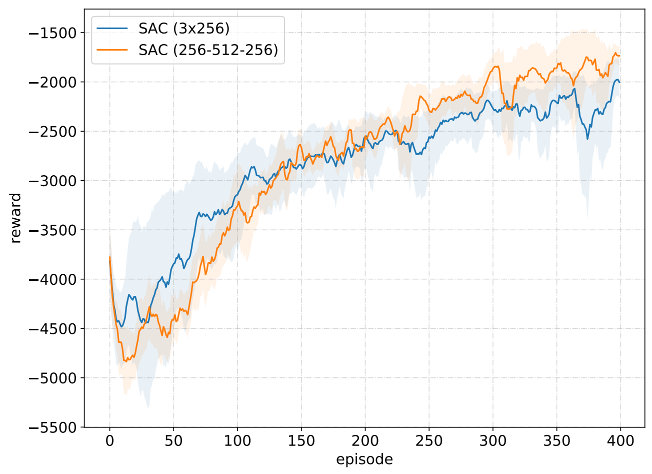
Figure B.2: Position control of multiple steel balls, no delay, with velocities, (a) 2 balls, (b) 5 balls



(a)



(b)



(c)

Figure B.3: Uniformity distribution shaping with no delay, (a) 10 objects, no dynamics, (b) 10 objects, dynamics, velocities, (c) 20 objects, no dynamics

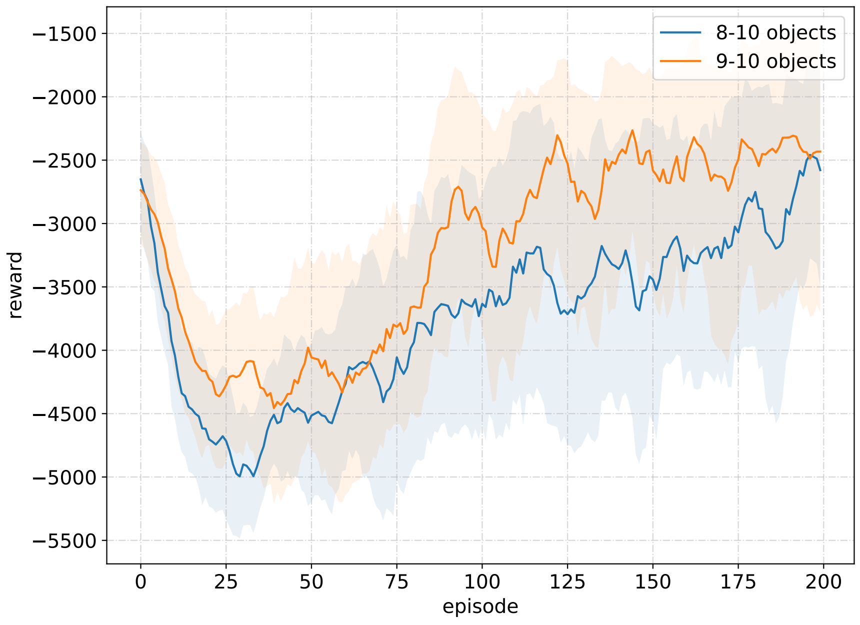
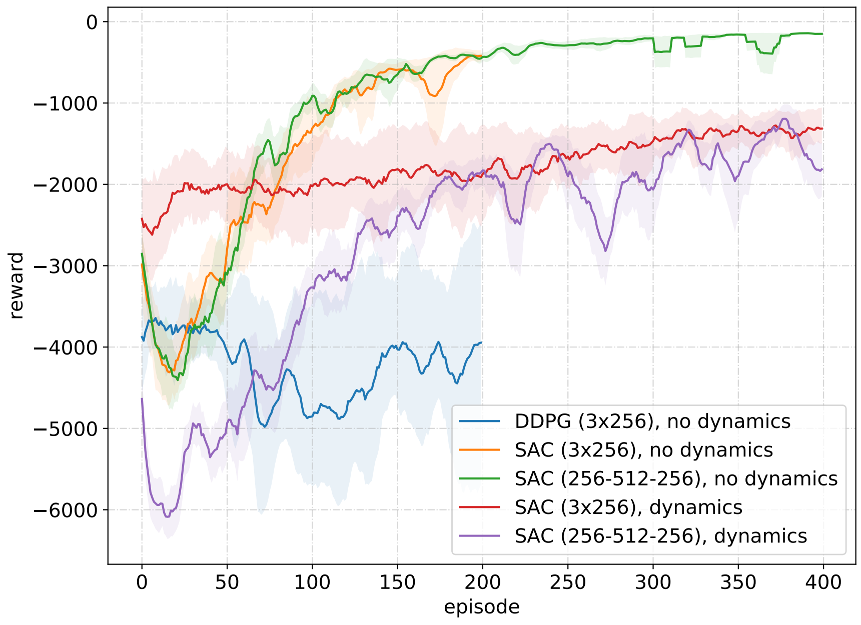
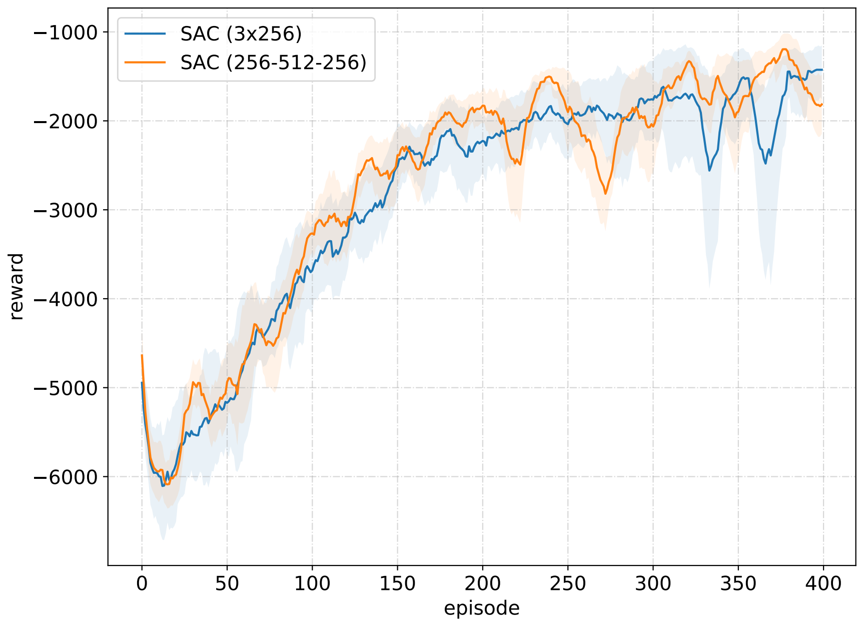


Figure B.4: Uniformity distribution shaping with no dynamics or delay and variable number of objects

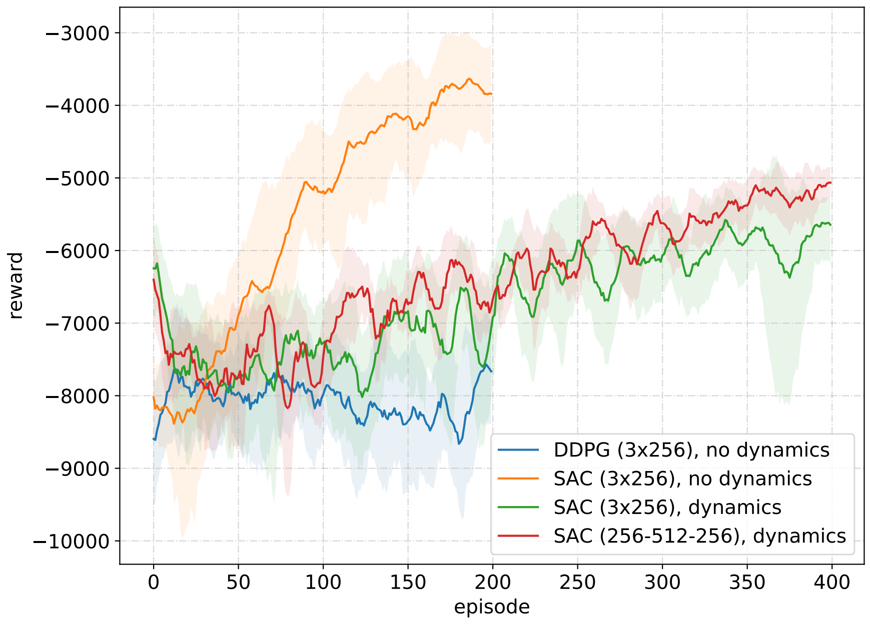


(a)

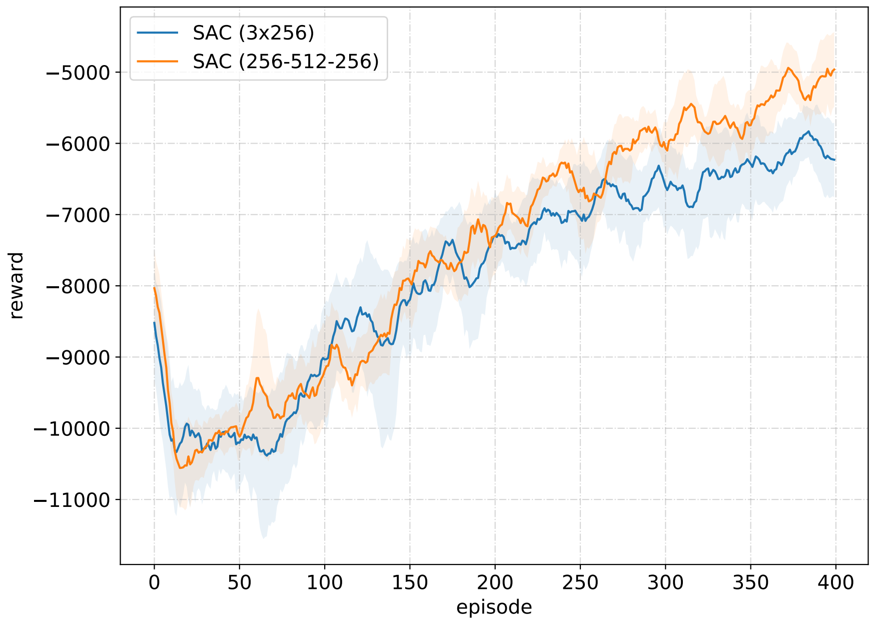


(b)

Figure B.5: Normal distribution shaping with no delay, (a) 10 objects, (b) 20 objects, no dynamics



(a)



(b)

Figure B.6: Controlled mixing with no delay, (a) 10 objects, (b) 20 objects, no dynamics

Appendix C

Contents of the attachment

`text/`	this thesis in PDF format
`code/`	all the Python code presented in the thesis
`code/algos`	reinforcement learning algorithms
`code/environments`	MagMan simulator
`weights/`	weights of some of the well-performing neural networks

Appendix D

Bibliography

[1] J. Zemánek, *Distributed manipulation by controlling force fields through arrays of actuators.* PhD thesis, Czech Technical University in Prague, 2018.

[2] C. Olah, “Understanding lstm networks.” `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`, accesed May 10, 2020.

[3] K. He, X. Zhang, S. Ren, and J. Sun, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” *arXiv:1502.01852 [cs]*, Feb. 2015. arXiv: 1502.01852.

[4] F. Richter, “Extension of the platform for magnetic manipulation,” diploma thesis, Czech Technical University in Prague, 2017.

[5] H. van Hasselt, “Reinforcement Learning in Continuous State and Action Spaces,” in *Reinforcement Learning* (M. Wiering and M. van Otterlo, eds.), vol. 12, pp. 207–251, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Series Title: Adaptation, Learning, and Optimization.

[6] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, “An Introduction to Deep Reinforcement Learning,” *Foundations and Trends® in Machine Learning*, vol. 11, no. 3-4, pp. 219–354, 2018. arXiv: 1811.12560.

[7] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, “Soft Actor-Critic Algorithms and Applications,” *arXiv:1812.05905 [cs, stat]*, Jan. 2019. arXiv: 1812.05905.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” *arXiv:1312.5602 [cs]*, Dec. 2013. arXiv: 1312.5602.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” *arXiv:1509.02971 [cs, stat]*, July 2019. arXiv: 1509.02971, original paper was published in 2016.

[10] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor,” *arXiv:1801.01290 [cs, stat]*, Aug. 2018. arXiv: 1801.01290.

[11] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing Function Approximation Error in Actor-Critic Methods,” *arXiv:1802.09477 [cs, stat]*, Oct. 2018. arXiv: 1802.09477.

[12] R. C. Hibbeler, *Engineering mechanics.* Hoboken, New Jersey: Pearson, 2016.

[13] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, “OpenAI Gym,” *arXiv:1606.01540 [cs]*, June 2016. arXiv: 1606.01540.

[14] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *arXiv:1412.6980 [cs]*, Jan. 2017. arXiv: 1412.6980.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention Is All You Need,” *arXiv:1706.03762 [cs]*, Dec. 2017. arXiv: 1706.03762.

[16] A. X. Lee, A. Nagabandi, P. Abbeel, and S. Levine, “Stochastic Latent Actor-Critic: Deep Reinforcement Learning with a Latent Variable Model,” *arXiv:1907.00953 [cs, stat]*, Feb. 2020. arXiv: 1907.00953.

[17] D. Han, K. Doya, and J. Tani, “Variational Recurrent Models for Solving Partially Observable Control Tasks,” *arXiv:1912.10703 [cs, eess, stat]*, Dec. 2019. arXiv: 1912.10703.

[18] R. Li, A. Jabri, T. Darrell, and Pulkit Agrawal, “Towards Practical Multi-Object Manipulation using Relational Reinforcement Learning,” *arXiv:1912.11032 [cs]*, Dec. 2019. arXiv: 1912.11032.