Teaching a (soft) robot to walk

MEng Thesis

Marcin Morawski (s1512783)

THE UNIVERSITY *of* EDINBURGH

University of Edinburgh
April 2019

Contents

Personal statement iv

Executive summary v

Acknowledgements vi

Summary of Resources vii

Word count by chapters viii

Glossary ix

1 Introduction and literature review 1
- 1.1 Why build walking robots? . . . 1
- 1.2 Why use learning methods for control? . . . 1
  - 1.2.1 A short introduction to Reinforcement Learning . . . 2
  - 1.2.2 Using neural networks to represent the policy . . . 3
  - 1.2.3 Training the policy — choosing the right reinforcement learning algorithm . . . 4
  - 1.2.4 Merits of learning methods . . . 5
- 1.3 Why build soft robots? . . . 5
- 1.4 Previous work . . . 6

2 The test system 7
- 2.1 Overview . . . 7
- 2.2 Mechanical hardware (a) . . . 7
  - 2.2.1 Soft robot . . . 7
  - 2.2.2 Pneumatic control board . . . 10
- 2.3 Low-level control (a, o) . . . 13
  - 2.3.1 Valve actuation . . . 13
  - 2.3.2 Pressure sensing . . . 13
  - 2.3.3 Orientation measurement . . . 14
  - 2.3.4 Position tracking . . . 15
- 2.4 PC interface . . . 15
  - 2.4.1 Microcontroller . . . 16
  - 2.4.2 Firmware . . . 16
- 2.5 High-level control ($\pi(o)$) . . . 16
  - 2.5.1 Gym environment . . . 17
  - 2.5.2 Details of the control algorithm . . . 18
  - 2.5.3 Implementation of the control algorithm . . . 18

3 Results **21**
3.1 Tuning the algorithm 21
3.1.1 Control Frequency/Projection horizon 22
3.1.2 Neural network structure 23
3.1.3 Neural network training parameters 24
3.1.4 Number of random action sequences 24
3.1.5 Iterations and episode timing 24
3.2 Performance of modelling and control 25
3.2.1 Model 25
3.2.2 Model Predictive Control 27

4 Conclusions **30**

5 Future work **31**

List of Figures

1 Schematic diagram of a Markov Decision Process 2
2 Schematic diagram of a Multi Layer Perceptron 3
3 Overview of the test system 8
4 The constructed hardware 9
5 Robot actuation method 9
6 Pneumatic circuit used to actuate the joints 10
7 Valve driver circuit and PCB 14
8 Sensors attached directly to the robot 15
9 Live visualisations of instrumentation data 17
10 Performance of the model-based controller using default parameters. Iteration -1 used a purely random policy. Shaded area represents the standard deviation in results. 21
11 Determining the optimum size of the neural network. The algorithm was trained on the training data for 200 iterations using neural networks of varying sizes. 24
12 Mean square error of the network on the training dataset vs. number of training iterations. Evaluated on a 1×100 MLP network. 25
13 Open-loop predictions for each of the state parameters (blue) vs. real data (black) for data collected when running the benchmark policy. 27
14 Performance of the model-based controller using improved parameters. Iteration -1 used a purely random policy. Shaded area represents the standard deviation in results. Large gain between iterations 3 and 4 is due to increasing the number of neurons in the hidden layer. 28
15 Two-legged gait developed in the initial stages of training. This figure shows the motion of one leg, the gait alternates these movements between left and right legs. Shaded areas denote which part of the joint is actuated. Dashed line is for position reference. 29
16 Undulating gait developed in the final stages of training. Shaded areas denote which part of the joint is actuated. Dashed line is for position reference. 30

Personal statement

I got the rough idea of using machine learning to control mechanical systems a year before the start of this project. I knew that in order to build the project I wanted to build, I would need to understand mechanical, electrical and software engineering. Because in my degree I only learned about the first one, I had to learn the other two elsewhere. I learned about basic electronics from *The Art of Electronics* by Horowitz and Hill, and then I did a few courses on programming microcontrollers. I studied programming by reading Ritchie and Kernighan's *The C programming language* and others. Then, I proceeded to learn about machine learning from Massive Open Online Courses. My autodidactism culminated with CS294-Deep Reinforcement Learning, a postgraduate course from UC Berkeley, where I learned about the latest machine learning techniques. Most of the machine learning used in this work comes from that course. Most of this work was done before the start of this project.

Once I felt that I was ready to build a machine-learning-controlled robot, I pitched the idea to dr Adam Stokes, who kindly agreed to supervise me. He proposed to use inflatable soft robots as a testing platform. To do good quality research on those, I needed a pneumatic control board, which the Stokes' research group did not have, and which cost much more than the £150 given to us by the School. In order to finance the project, I applied for a Student Experience Grant (successfully). The conditions of the grant required me to build a slightly more complicated apparatus than I originally planned for, but receiving it allowed me to build the project I wanted to build.

I designed and constructed the electropneumatic soft robotic testing platform completely independently. I picked, sized, procured, and assembled the required pneumatic components. I designed, ordered, and hand-soldered the PCBs required for sensing and actuation. I reverse-engineered and cast one of dr. Stokes' old soft robot designs from a drawing in his paper. Finally, I programmed the microcontroller firmware and computer software which run the whole system. Because of the conditions of the grant, I had to make everything as user-friendly and robust as possible. In the end, this probably benefited the project as a whole.

I did all the machine learning work myself, although, as is explained in the report, the general structure of my code came from the aforementioned CS294 class. In short, all the work presented in this report is my own, except when other sources are referenced.

I would like to point the examiner's attention to the fact that this report is aimed at mechanical engineers. I assume some prior knowledge of pneumatics etc., but I spend a lot of time explaining basic machine learning concepts where they are necessary to understand the design choices I made. This naturally comes at the expense of depth and sophistication. For example, I explain what Stochastic Gradient Descent is, but I gloss over the details of the Adam optimiser that I actually used for training. To someone more versed in reinforcement learning, it may often seem like I am stating the obvious. This was not done out of ignorance, but to give the reader a good intuitive understanding of the parts of my project which I considered the most important.

Executive summary

Teaching a (soft) robot to walk

Marcin Morawski

April 2019

Walking robots and soft robots are governed by highly nonlinear dynamics, which presents a challenge to conventional, linear control theory. Linearised approaches to controlling them result in suboptimal policies which cannot exploit the systems' dynamics. This makes current walking robots less agile and energy-efficient than their biological counterparts. This work explores an alternative approach to controlling complex mechanical systems — one based on deep reinforcement learning algorithms. Instead of executing a predefined behaviour, the robot learns a control policy by interacting with its environment.

In order to test this approach, a quadrupedal walking soft robot is constructed, along with a dedicated suite of sensors, and an electro-pneumatic actuation system. The system is packaged as an OpenAI gym environment, which makes it easy to reuse for other machine learning projects. A model-based reinforcement learning algorithm combined with a model predictive controller is applied to the robot.

The main objective of the algorithm is to walk forward as fast as possible. After just 50 minutes of training, it achieves a speed of $1 \times 10^{-3}\ ms^{-1}$, 40% of the speed achieved by an open-loop benchmark policy. It learns two useful gait sequences — a slow bipedal walk and a slightly faster undulatory one.

There is a large standard deviation in the results achieved by the robot, which can be attributed to the chosen model predictive control strategy (random shooting). A number of possible solutions to this issue is presented in the discussion section.

Acknowledgements

I would like to thank dr Adam Stokes for agreeing to supervise this project, for his advice on soft robotics and sensing, and for letting me work on what I wanted. I would like to thank Paula, for being the only reason I'm still in Edinburgh, and Ernestas, for letting me use his ping-pong balls. Finally, I would like to thank dr Sergey Levine, for sharing materials from his amazing CS294 class on the Internet.

Summary of Resources

This project received additional funding in the form of a Student Experience Grant in order to finance hardware development. Two tables are provided — one for the resources purchased from the budget assigned by the School of Engineering (Table 1), one for those purchased using the grant money (Table 2). A more detailed bill of materials is included in the supplementary materials. No technician hours were used during the project.

Item	Cost (£)
ST-link microcontroller programmer	16.72
Laser-cut acrylic board	10.00
Crimp terminals	2.35
Honeywell ABP pressure sensors	78.50
Bosch BNO055 development board	26.67
Total	134.24

Table 1: Things purchased using the project budget assigned by the School of Engineering

Item	Cost (£)
Compressor + air filtering	351.48
Valves + manifold	305.87
Fittings + tubing	217.68
Casting silicone	130.34
PCB manufacture	52.53
PCB components	294.38
Microcontroller	25
Total	1377.28

Table 2: Things purchased using the Student Experience Grant budget

Word count by chapters

Chapter	Word count
Introduction and Literature Review	2614
The test system	4141
Results	2425
Conclusions	359
Future work	418
Total	**9957**

Table 3: Word counts of individual chapters

Glossary

A a sequence of actions.

C sonic conductance.

J reinforcement learning objective.

K number of episodes ran between training steps.

N number of model training iterations.

Q volumetric flowrate.

R universal gas constant.

R number of episodes ran using a random policy.

T temperature.

V volume.

Ω_w W component of the orientation quaternion.

Ω_x X component of the orientation quaternion.

Ω_y Y component of the orientation quaternion.

Ω_z Z component of the orientation quaternion.

χ projection horizon.

δn change in amount of gas molecules.

$\hat{o}_{t+1}$ next observation predicted by the model.

μC microcontroller.

ν frequency at which the model-based controller operates.

$\pi(o)$ policy — a mapping from observations to actions.

ρ density.

τ trajectory — the arrangement of states, actions and rewards in time.

θ model parameters.

a_t action taken in the current state.

b pressure ratio.

$f_\theta(s_t, a_t)$ model function mapping states and actions to next states.

m subsonic index.

o_{t+1} observation received in the next state.

o_t observation received in the current state.

p pressure.

r_t reward in the reinforcement learning task.

x position along x coordinate.

x' velocity along the x-coordinate.

API application programming interface.

COV coefficient of variation — ratio of standard deviation to mean of a dataset, gives a normalised measure of spread in the data.

GPGPU general purpose computing on graphics processing units — using graphics cards to perform computations which normally would be done on a CPU. The parallel nature of some machine learning operations (such as passing multiple actions sequences through the same model in this work), can be handled much more efficiently on GPUs than on CPUs..

IMU Inertial Measurement Unit — an instrument composed of an accelerometer, a gyroscope and, sometimes, a magnetometer, used to determine the orientation of an object in space.

MDP Markov Decision Process — a mathematical idealisation of a control process. It is composed of an agent which interacts with the environment by taking actions which influence the observed state of the environment.

MLP Multi-layer Perceptron — a type of neural network architecture.

MPC model predictive control — a control method where a model is used to predict a number of steps into the future, but the prediction is repeated at each step.

off-policy an off-policy algorithm can learn from historical data, not just from data it collected itself.

on-policy an on-policy algorithm can only learn from data collected during the current run of the policy, it cannot learn from previous experiments, or from data collected by another controller.

PCB printed circuit board.

PI proportional-integral (controller).

RL Reinforcement Learning — a branch of machine learning in which optimal behaviour is achieved by taking actions which maximise cumulative reward.

s.d. standard deviation.

SGD stochastic gradient descent — an optimisation method.

underactuated an underacutated system has fewer actuators than degrees of freedom, and cannot easily put itself in an arbitrary position in space. Consider a pendulum — a single pendulum with one motor is fully actuated, and easy to control. A double pendulum with one motor is much harder to work with..

1 Introduction and literature review

This work explores the use of reinforcement learning (RL) algorithms to control and model complex mechanical systems. An emphasis is placed on verifying the performance of algorithms on real hardware, not just in simulation. A walking soft robot is used as a testing platform.

Walking machines often require sophisticated linkages, and conventional, linear control methods are not sufficient for them to perform optimally. Soft robots often do not even have discrete linkages, and conventional, kinematics-based modelling methods are not sufficient to predict their behaviour.

Given all those difficulties, one must ask — why should we build walking or soft robots at all? Why not just make a metal chassis with four wheels? The next two sections explain why walking robots with soft actuators may be useful, and how one could use RL methods to control them. Then, a detailed design of the soft robotic testing platform is explained. Finally, the generated walking sequences are examined, and performance of the RL controller is compared with a preprogrammed benchmark gait.

1.1 Why build walking robots?

The most common argument and use case given for walking robots goes as follows — wheeled robots are easy to build and control, but they do not perform well in unstructured environments. Given that most of the world is unstructured, we may want to have robots which are able to access any place on the planet where we may not want to send a human. Disaster relief and military applications are often mentioned.

Even if they did not prove to be immediately useful, building walking robots is a good way to learn about walking itself. Given that humans are walking robots, it is natural that we would like to understand how walking really works. Research on walking robots has already inspired new directions in locomotion studies [1]. In a way, the best proof of whether one understands something is whether one can build it themselves, from scratch.

Knowledge on how walking works could be immediately fed into the design of active prosthetics, orthotics and exoskeletons. Devices currently available on the market tend to perform poorly compared to their biological counterparts. The fastest commercial exoskeleton (ReWalk) has a top speed of 2.6 kmh^{-1}, around half that of a strolling human, and requires the wearer to use crutches [2]. If we could design control algorithms and actuators that would make walking robots as agile as animals, we could maybe bring agility back to people who lost it.

1.2 Why use learning methods for control?

The walking robots that people can build today are inferior in most aspects to those known in nature. They are less agile, and an order of magnitude less energy efficient [3]. This is partially due to inferior hardware — biological systems tend to exploit the

dynamics of the environment to a much greater degree than the ones designed by humans [4].

Another part of the problem, however, stems from the fact that we are unable to develop efficient control methods for those systems. Walking is a dynamic, nonlinear, impact-driven, underactuated locomotion problem. This means that conventional, linear control algorithms will struggle to perform well across the entire domain of the problem. They can, however, achieve impressive results in narrow regions, where the system can be assumed to be linear [5]. In order to keep it linear, one often needs to use strong feedback to cancel the natural dynamics of the system, which decreases the robot's efficiency and agility. This is why robotic gaits often look inflexible and unnatural.

1.2.1 A short introduction to Reinforcement Learning

This work is about applying a different class of control algorithms to robots — ones based on reinforcement learning. This section presents a short introduction to RL. It is mostly based on work by Sutton [6]. The objective of RL is to find the most optimum controller for a given task. Behaviour is not programmed into the robot at the outset; it emerges organically as the control algorithm interacts with the environment.

The control problem is modelled as a Partially Observed Markov Decision Process (MDP). MDP is a flexible framework which allows one to mathematically formalise a control process. A diagrammatic view of a MDP is presented in Fig. 1. The formalism consists of an agent (the controller), and the environment (everything external to the controller). The agent interacts with and observes the environment (gathering observations o_t) and executes actions a_t according to a policy $\pi(o)$. After an action is taken, the environment changes, responds with a new observation o_{t+1} (which is determined according to the state distribution probability $p(o_{t+1}|o_t, a_t)$), and gives the agent a reward r_t. The whole cycle then repeats forever or until the task is terminated. One cycle is called a *run*, a *rollout*, or an *episode*. The sequence of all o, a and r in an episode is called a trajectory τ.

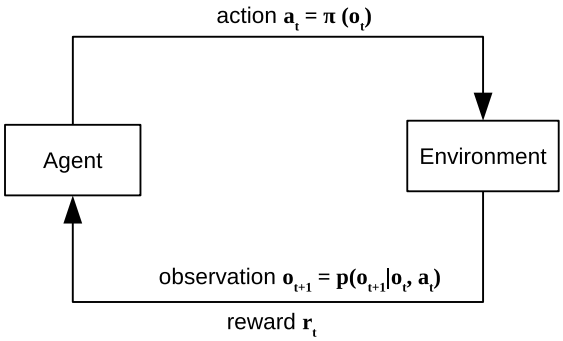


Figure 1: Schematic diagram of a Markov Decision Process

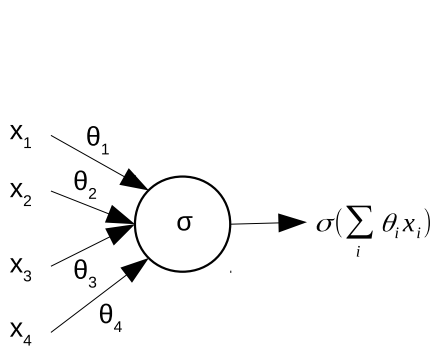
In this work, the reward is determined by the engineer (this is not always the case). For example, if one wanted a robot to walk, one could give it a positive reward proportional

to its forward velocity. The agent's aim is to adjust its control policy to maximise the total sum of rewards it will achieve over the duration of the task. This is called the *reinforcement learning objective*, J.

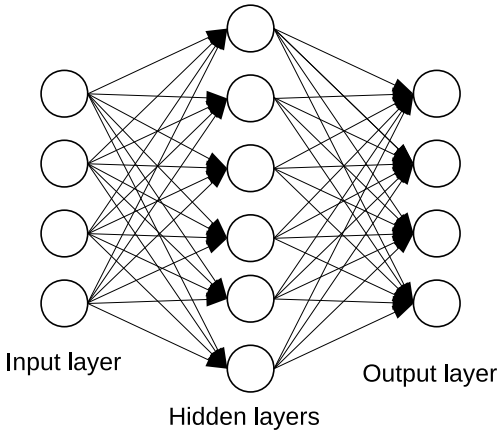
This may sound rather involved at first reading, but the essence of this subsection is that RL allows one to frame a problem in a way that allows them to specify *what* needs to be done, rather than *how* to do it. However, the RL framework does not specify how the policy should be represented, or how it should be optimised to maximise reward.

1.2.2 Using neural networks to represent the policy

A robot's controller is nothing more than a function mapping observations from the robot's sensors to actions of its mechanical hardware (motor torques, servovalve positions etc.). For simple cases, the space of observations can be discretised, and one can simply write a table (this is often called the *tabular method*), where for each state there is a specified action. For such states (and assuming that we know the environment's dynamics), one can use dynamic programming methods and prove that they will find the best possible control policy. However, for most real-life problems, this space is much too large to be represented directly. For example, the controller developed for this project has eight 14-bit pressure sensors, yielding $(2^{14})^8$ possible combinations of inputs. This is an example of *the curse of dimensionality*, which plagues most practical reinforcement learning systems. We need a way to approximate the mapping specified by π (or, in this work, the model of the environment) in order to reduce the computation required to obtain actions from observations to a manageable level.



(a) A single neuron



(b) Neurons connected to form a network

Figure 2: Schematic diagram of a Multi Layer Perceptron

A neural network does exactly that — it is a powerful universal function approximator. Neural networks are loosely inspired by biological brains. They are composed of individual neurons (Fig. 2a), which, by themselves, are very simple — they take a weighted sum of their inputs, pass that through some nonlinear function σ and output

a single value. In this work, the rectifier function is used:

$$\sigma(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{1}$$

The neurons are then connected into a net, with outputs of some neurons feeding into inputs of others. There are many possible ways to form a net. The architecture used in this report is called a Multilayer Perceptron (MLP). It is composed of an input layer (arguments of the function), an output layer (the value of the function), and a number of *hidden* layers between them. Those layers are fully connected — the output of each neuron feeds into inputs of all neurons in the next layer. Fig. 2b shows this schematically. Sections 2.5.3 and 3.1.2 explain the chosen network structure in more detail.

This combination of nonlinear functions can represent very complex mappings. In fact, it can be proven that, by adjusting the weights at the inputs of the neurons, one can approximate any continuous real function [7].

1.2.3 Training the policy — choosing the right reinforcement learning algorithm

Having decided on a way to represent the policy, we can now turn our attention to optimising it to achieve the highest possible reward. Optimisation will be done by changing the weights in the neural network. There are many classes of algorithms which do this (in fact the field of RL is mostly the study of those algorithms). Levine [8] provides a framework for choosing the appropriate class of learning methods based on their *sample efficiency*.

There is a tradeoff in learning methods between the speed at which the algorithm can learn, and the complexity of tasks it can handle (and the performance that it can achieve). The less the algorithm assumes about the environment it operates in, the better it will eventually perform. Any assumptions will have errors associated with them. In the extreme case, the errors may be too large for the algorithm to execute the task at all. However, the less one assumes about the world, the more learning resembles trial and error, which means that it could take prohibitively long to arrive at an optimum solution. Sample efficiency is defined as the amount of data which the algorithm needs to collect before it reaches an acceptable level of performance.

In the most extreme case, gradient-free methods such as genetic algorithms proceed entirely by trial and error. Any policy which did not receive a high ultimate reward is treated as a failure, even in a case where it may have worked well until the very last step of the task and then failed catastrophically.

Slightly more efficient are policy gradient methods, which try to optimise the policy directly, by attributing the achieved reward to actions taken at each step. Even if the algorithm ultimately does not perform satisfactorily, if it did well anywhere along the way, policy gradient methods can learn from it. However, those methods are usually on-policy, i.e. they can only learn from data collected during a single run of the algorithm. After being used to improve the algorithm, those data points are discarded.

More data-efficient are off-policy methods, such as Q-learning. They can learn from historical data connected during previous runs. This data is usually stored in a *replay buffer* and used to retrain the method every few iterations.

All the methods described above are model-free. They assume very little about the environment they operate in. The most efficient methods are model-based. They use data collected during a run of a policy to learn a model of the environment. This model can then be used to predict the results of actions suggested by the controller, and used to pick the most optimal trajectory. On a representative task, model-based methods were shown to achieve acceptable performance based on just 20 minutes of real-time data, compared to 15 days required by the gradient-free methods described above. In this project, a model-based method was used for two main reasons:

- The project had a limited timeframe. As will be shown in Section 3.1, RL algorithms require a lot of parameter tuning to perform satisfactorily. Parameters are tuned by iteratively testing them on the robot. If each experiment took a few hours or days, it would not be possible to finish the project on time.
- Model-based methods provide, in addition to the control algorithm, a model of the robot, which can be useful for other purposes than the original task. For example, one can learn a model by trying to walk forward, but the same model could be used for walking in other directions.

1.2.4 Merits of learning methods

The main benefit of using learning methods for control is that they can potentially find more optimal ways of behaving than a person could come up with. RL draws a lot on optimal control theory. Deep RL algorithms offer a structured way to search for optimum solutions to complex, nonlinear control problems, such as walking. Instead of deciding upfront on the gait patterns which a robot is supposed to execute and hoping that they are the best ones possible, we can try to let them develop organically as the robot learns from experience.

Additionally, because they can be model-free, or learn a model from experience, RL methods are well suited to the control of systems which are difficult to model, such as soft robots, which are the subject of the next section.

1.3 Why build soft robots?

Soft robots use compliant materials (textiles, rubbers etc.) instead of rigid linkages. This makes them more robust than their hard counterparts [9] and inherently safe when interacting with delicate objects (such as humans). They often use pneumatic and hydraulic actuators, which have power densities comparable to or exceeding that of muscle [10], and an order of magnitude better than electric motors which currently dominate in robotics. Power density is crucial for mobile, untethered robots, as it directly translates to lower weight and improved running time.

Soft robots have already demonstrated potential in devices such as grippers [11], orthoses [12] and exoskeletons [13]. A frequently cited advantage of soft robots is that, because of their compliance, they do not require complicated control and sensing methods to e.g. grip delicate objects.

On the other hand, their behaviour is hard to model and control precisely [14]. An expanding elastomer is a highly nonlinear system, which often requires finite element methods for accurate simulation [15].

Model-based or model-free RL methods could offer a viable solution to this problem. A soft robot was chosen as the testing platform for this project precisely for that reason — while there exist acceptable (if underperforming) solutions for walking hard robots, the soft robotics community seems to be in a search of novel control and modelling approaches.

1.4 Previous work

RL methods are routinely applied to simulated robots, because simulated robots are often used as a benchmark for comparing algorithms [16]. A soft robot — a simulated octopus arm — was also controlled in simulation [17]. One of the aims of this work, however, is to test the algorithms on real hardware.

RL algorithms were successfully used to learn control strategies for a variety of real situations, from helicopter acrobatics [18] to hanging clothes hangers on a rack [19]. However, a lot of new research is focused on relatively high-level tasks, which involve grasping and manipulating objects based only on visual feedback. The methods developed in those projects are of limited utility to the task of teaching a robot to walk.

Much more relevant are attempts to learn so called *motor primitives* — basic motion patterns, of which walking is just one example. Machine learning was used to produce controllers for real walking bipedal systems [20][21][3], and to learn a diverse set of simple patterns such as ball paddling or playing the game *Ball-in-A-Cup*[22]. However, those works either used a linear function to represent the control policy [20][3] (instead of a nonlinear neural network), or training algorithms which are now outdated [21]. Additionally, none of them dealt with soft robots and their unique nonlinear dynamics.

RL agents were used to control soft robots, but the examples are few and far between. Gupta et al. [23] used a novel form of imitation learning (where the robot learns by observing a human demonstrator) to teach a soft robotic hand to perform a variety of manipulation tasks, such as turning a valve and operating an abacus. Yang et al. [24] used Q-learning to optimise a soft actuator for a cuttlefish-inspired robot. Giorelli et al. [25] used a neural network to model and represent the inverse kinematics of a cable driven soft robotic actuator (i.e. to predict how much to pull each cable in order to place the end of the actuator in the desired position). The most similar piece of research to the present project that could be found in literature was done by Zhang et al. [26]. They used Q-learning to move a pneumatically actuated soft robotic arm to a specified position. However, they used motion tracking to directly determine joint positions instead of inferring them from pressure measurements, as will be proposed in this report.

Summing up, the literature presented above demonstrates that reinforcement learning is a viable method of controlling robots. It has been applied to locomotion, soft robots and nonlinear sensors, but never to a combination of the three.

2 The test system

2.1 Overview

In order to test the viability of RL methods for control of walking soft robots, a soft robotics testing platform was built and used to test a RL algorithm. As outlined above, a RL agent needs a way to execute actions a, a way to gather observations o, and a policy $\pi(o)$ which maps actions to observations. This naturally splits the platform into three layers (see Fig. 3).

Firstly, there is the mechanical hardware — the robot itself, and the pneumatic system used to actuate it. On top of it is the electronic layer used for low-level actuation and sensing (switching solenoids, interfacing with pressure sensors etc.). It communicates with the last layer, implemented purely in software, which carries out high-level planning and control. All elements of the system were developed from scratch, as part of this project. The development was funded by a University of Edinburgh Student Experience Grant. Funding was obtained on the condition that the system will be versatile, safe and easy to use, as it is meant to be used by students with limited engineering experience to experiment with soft robots. This influenced some of the design choices, as will be explained below.

The design is open source. Computer code is available from `https://gitlab.com/s1512783/tartw_control`, PCB design files from `https://gitlab.com/s1512783/tartw-pcbs`, and mechanical CAD files from `https://bit.ly/2WjP81q`. Fig. 4 shows the final embodiment of the design.

2.2 Mechanical hardware (a)

2.2.1 Soft robot

An eight-jointed pneumatic soft robot, first described by Stokes et al. [27] is used as the testing platform. The robot utilises Pneu-Net actuators [28]. Each joint is composed of 2 layers (Fig. 5). The bottom one has a larger elastic modulus than the top, so, when pressure is applied to the joint, the bottom deflects less. This results in the entire leg curving inwards. Because there are two channels per leg, it can also twist sideways when only one channel is activated. A combination of those two motions enables the robot to use a walking gait to move. Supplementary video 1 shows the robot executing a pre-programmed actuation sequence which causes it to walk.

The robot was manufactured using a method similar to the one reported by Stokes et al. [27] — it was cast from silicone. 00-50 Shore hardness material (Smooth-on EcoFlex 00-50) was used for the top, and 30A Shore hardness (Smooth-on DragonSkin 30) for the bottom. Molds for casting were reverse-engineered from drawings in the Stokes paper.

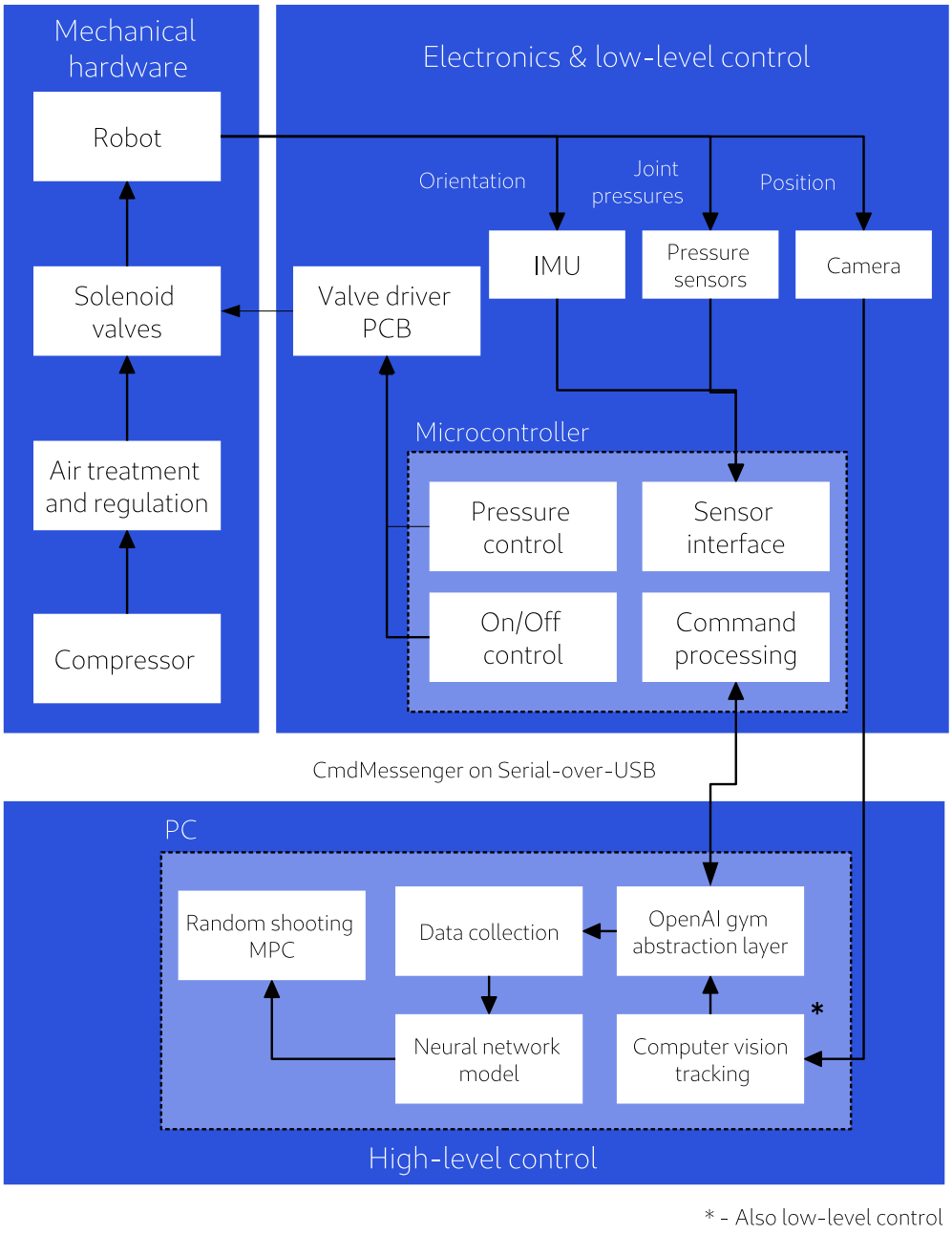


Figure 3: Overview of the test system

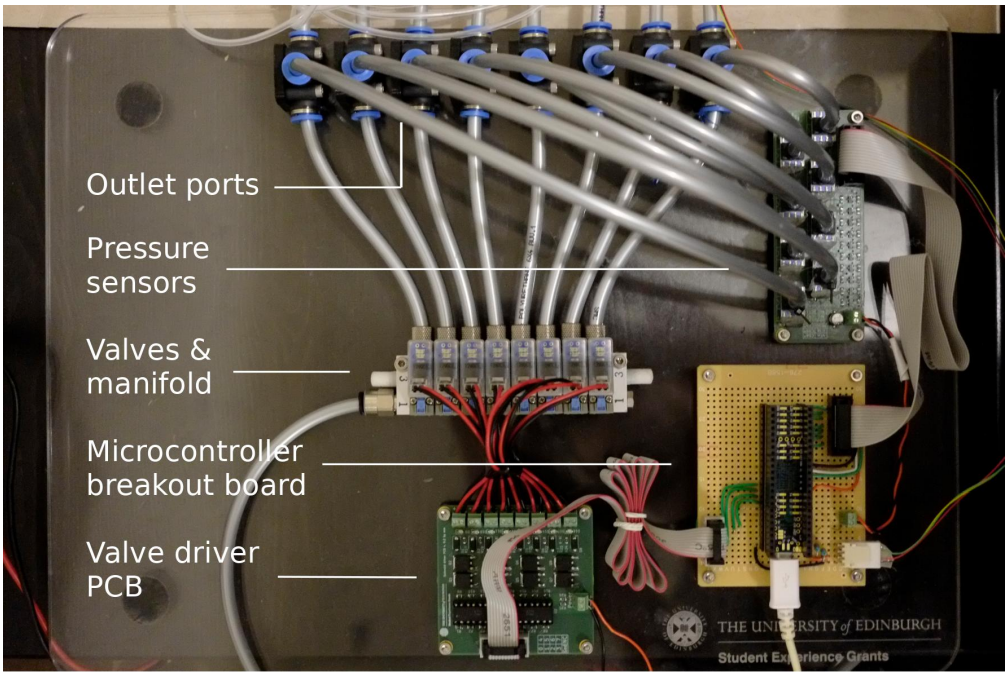
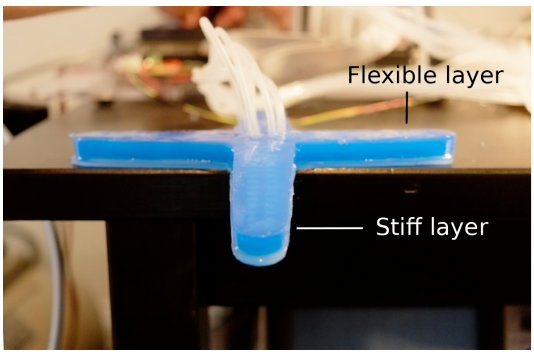
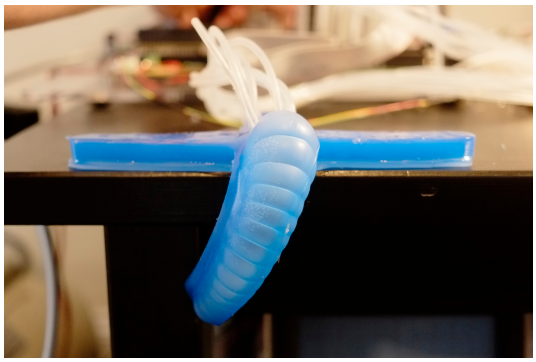


Figure 4: The constructed hardware



(a) Unactuated joint



(b) Actuated joint

Figure 5: Robot actuation method

A few improvements, such as draft angles, were made to the original design. The molds were 3D printed from PLA and PETG using fused-deposition-modelling technology, with a layer height of $150\mu m$. A vacuum chamber for degassing silicone was made from a food container to ensure a bubble-free cast.

Both parts were then glued together with the 30A hardness silicone to produce the final mechanism. 2.5 $mm \times 1.5$ mm silicone tubing was used to feed compressed air to the robot. 40 kPa (gauge) actuation pressure was used for most experiments.

In the interim report, another soft robotics platform — the Arthrobots [29], was proposed as a testbed for experimentation. A few Arthrobots were constructed early on in this project, but they were too unreliable for the extensive experimentation required to collect RL training data. As will be shown later, the learning process required the robot to perform consistently for hours, which the Arthrobots were not designed for. That was the reason for switching to a cast silicone robot.

2.2.2 Pneumatic control board

In order to produce and control compressed air for actuating the robot, an electropneumatic system was built (its circuit diagram is shown in Fig. 6). This section will outline how the components were picked and sized. Note that it was originally designed for Arthrobots, which operate at larger pressures and flowrates than the soft silicone robot, hence the slight overdesign of the pneumatics.

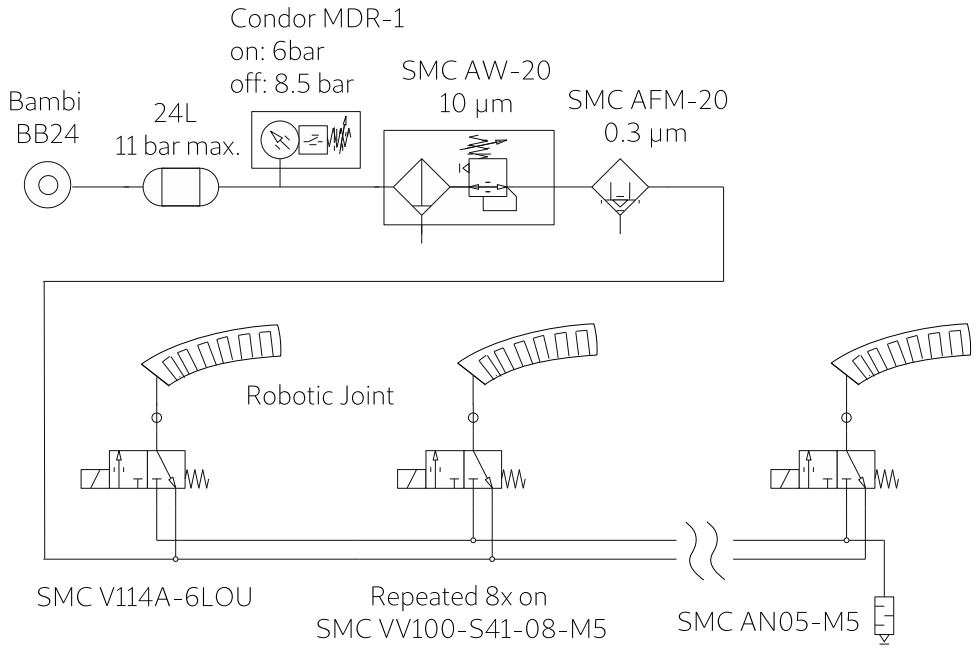


Figure 6: Pneumatic circuit used to actuate the joints

Compressor Most soft robots are inflated up to a pressure p of around 70 kPa. It was measured, with a syringe, that it took about 20 cm^3 of air at that pressure to inflate one joint of the Arthrobot. The robot system was designed for eight joints, so, if all eight were inflated at once, a volume V of 160 cm^3 would be required. At an arbitrary inflation time t of 0.3 s this would result in a volume flowrate Q of:

$$Q = \frac{V}{t} = 533.(3)\ cm^3 s^{-1} = 32\ lmin^{-1} \tag{2}$$

This is the flowrate at 70 kPa, but compressor capacity is usually defined at atmospheric pressure. The inlet flowrate can be calculated from the ideal gas law, assuming that the process is isothermal:

$$Q_{in} = \frac{p_{out}}{p_{in}} Q_{out} = \frac{170kPa}{100kPa}\ 32\ lmin^{-1} = 54.4\ lmin^{-1} \tag{3}$$

This is the peak consumption. In reality, one would rarely open all joints at the same time, and most compressors have a storage tank to keep up with demand peaks. Nevertheless, an oversized compressor is better than an undersized one. A Bambi BB24 oiled, sealed, direct drive compressor with a capacity of 50 $lmin^{-1}$ was picked for this application, mostly because of its low noise level (under 40 dBA, regular compressors are around 70 dBA). The system is meant to be operated in a confined space for long periods of time, and using a silent compressor eliminates the need for ear protection.

A large (24 l) storage tank is used, so that the compressor can operate at an acceptable duty cycle. The BB24 is rated for 50% duty cycle. On average, it can only be on half of the time, for up to 30 minutes on, 30 minutes off. The compressor features a Condor MDR-1 pressure switch, which will automatically switch the compressor on when pressure in the tank falls to $p_{low} = 6\ bar$, and off at $p_{high} = 9\ bar$. The amount of gas molecules δn which will leave the tank as it discharges is:

$$\delta n = (p_{high} - p_{low}) \frac{V_{tank}}{RT} \tag{4}$$

Where R is the universal gas constant and T is the temperature. If the robot draws a steady flowrate Q, those molecules will deliver Q for a time:

$$t = \frac{\delta nRT}{p_{out}Q} = \frac{(p_{high} - p_{low})V_{tank}}{p_{out}Q} \tag{5}$$

Assuming that the mean flowrate will be one-third of the maximum calculated above, and that the compressor should only switch on every 5 min, the volume of the tank should be 19.4 l. A 24 l tank is the closest standard size.

Valves A bank of eight two-position three-way SMC V114A-6LOU valves, mounted on a SMC VV100-S41-08-M5 manifold is used to control the flow of air to the robot's joints. Those valves are the high-capacity version of V100 — the smallest valve in the SMC range. They are designed for use as pilot valves, and offer fast switching times (under

5 ms) and long service life (200 million cycles) in a small package [30]. To estimate if a valve is the right size, one first needs to determine the pressure ratio b between the inlet and outlet:

$$b = \frac{p_{out}}{p_{in}} = \frac{100\ kPa}{170\ kPa} = 0.59 \tag{6}$$

In a pneumatic component, if the pressure ratio is less than a certain value called $b_{transition}$, the flow through it will be sonic. It is also called *choked* flow, as the velocity of the gas cannot increase over the speed of sound. This means that below $b_{transition}$, the valve reaches its maximum volumetric flowrate. V114A datasheet [30] specifies that this occurs at $b_{transition} = 0.07$, so the valves operate in the subsonic condition, and they will not achieve the maximum possible flowrate. The amount of flow they will pass through is determined by the sonic conductance C of the valve, which the valves' datasheet specifies at $7.6 \times 10^{-10} \frac{m^3}{Pa\ s}$. According to the ISO 6258-2 standard [31], the nominal subsonic flowrate through a valve is:

$$Q = C\rho p_{out}\left(1 - \left(\frac{b - b_{transition}}{1 - b_{transition}}\right)^2\right)^m \tag{7}$$

Where ρ is the density of air (1.2041 kgm^{-3} at 20 °C) and m is the subsonic index. m was not given by the datasheet, but it is commonly assumed to be 0.5. Those numbers give $Q = 7.74\ l\ min^{-1}$, which is close to the original peak flowrate estimation of 8 $l\ min^{-1}$, so the valves are large enough. This calculation is the reason why high flow capacity version of the V100 valve is used — the regular one can only deliver 50% of the required flowrate.

To decrease the noise emitted when the valves discharge to the atmosphere, SMC AN05-M5 silencers are installed on the outlet ports of the manifold. This, combined with a silent compressor, makes it possible to comfortably operate the system for long periods of time.

Filtering, Regulation, Lubrication The V100 valves are lubricated for life during manufacturing, but they require air filtered down to 5 μm to operate correctly. In addition, the BB24 oiled compressor generates a lot of airborne oil droplets, which could displace the lubricant in the valves and decrease their lifetime.

To deal with those two issues, air from the compressor is fed through a SMC AW-20 filter-regulator (with a 10 μm filter) to bring the pressure down to the required 40 kPa and to remove coarse debris, and then through a SMC AFM-20 mist remover (with a 0.3 μm filter) to remove any residual oil and condensation from the airflow. Both filters are rated for a flowrate of 150 $lmin^{-1}$; much larger than required, but they are, again, the smallest available in the SMC range.

Layout Most parts of the system are mounted on a piece of laser-cut acrylic sheet. M3 bolts, inserted into holes tapped directly in the acrylic, are used to securely hold them in place. $6 \times 4mm$ polyurethane tubing connects all the pneumatic components. Tees

mounted to the sheet using 3D-printed brackets direct air to pressure sensors and serve as outlet ports.

2.3 Low-level control (a, o)

The low-level control system acts as an interface layer between the high-level control and planning system and the physical hardware. As such, it provides both a and o capabilities. It is managed by a microcontroller (μC). The actuation capability is simply an extension of the mechanical system; it consists of one board which interfaces the microcontroller to the solenoid valves.

The sensing capability is more sophisticated. As a general rule, a control algorithm will perform better if it has better knowledge of the state of the world that it operates in, i.e. if it has more sensors. However, it is difficult to obtain accurate sensing capability in a small, soft robot. It is not possible to e.g. use optical encoders to obtain accurate joint angles, or torque sensors to measure joint forces. In an effort to provide as much sensing capability as possible, the following sensors are used:

- pressure sensors connected to each of the joints. Joint pressure corresponds to the position of the leg, but the relationship is highly nonlinear, with a large hysteresis (8 inputs).
- inertial motion unit (IMU) mounted on top of the robot measures orientation of the robot in space in WXYZ quaternion notation (4 inputs).
- computer vision tracking system gives the position and velocity of the robot on a 2D plane (4 inputs).

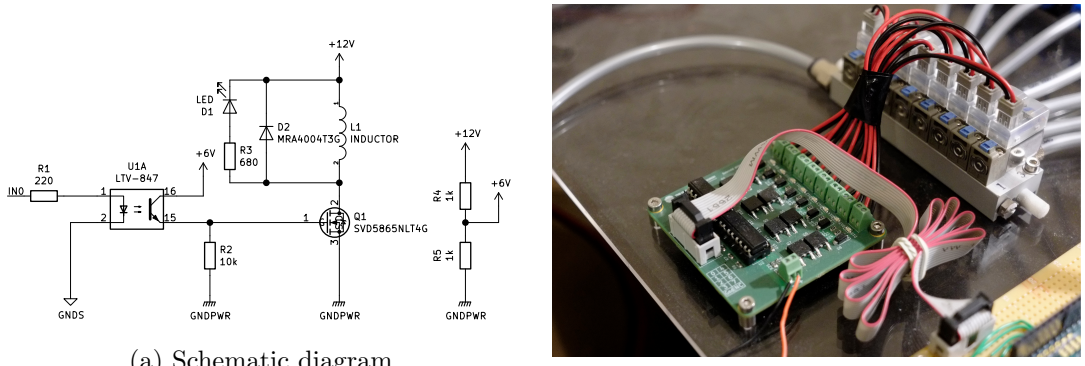
The following paragraphs outline how those capabilities were implemented in more detail.

2.3.1 Valve actuation

The valves require 1 W of power at 12 V in order to remain open, and the μC used in the system operates at 3.3 V. An interface circuit (Fig. 7) was designed to allow the microcontroller to switch the valves. It is composed of a 12V power supply to provide the required current, power MOSFETs (On-Semi SVD5865NLT4G) to switch the valves, flyback diodes (On-Semi MRA4004T3G) to protect the MOSFETs from flyback voltages, and optocouplers (Lite-On LTV-847) to optically isolate the microcontroller from the 12 V rail. LEDs were also added at the output to provide a visual indication that a channel is actuated. A PCB with the circuit was designed, procured and assembled by the author. It features eight driver circuits in a compact (60 $mm \times 65$ mm) package.

2.3.2 Pressure sensing

Honeywell Tru-Stability HSCDAND030PGSA3 pressure sensors are used to measure the instantaneous pressure in each channel. The sensors digitise the pressure signal and apply temperature compensation internally. They interface with the μC through the



(a) Schematic diagram

(b) Assembled PCB

Figure 7: Valve driver circuit and PCB

SPI bus. Integrated pressure sensors like the ones used here are more expensive than the average models on the market (which are usually just a strain gauge), but they do not require an external instrumentation amplifier and other complicated analog circuitry. The manufacturer claims that their accuracy is within ±0.25%.

A separate PCB was designed for mounting the pressure sensors. The PCB also features bidirectional voltage level shifters on each data channel. Circuit for the shifters was derived from Phillips application note AN97055 [32]. They were included, because the pressure sensors come in both 5V and 3.3V versions. Depending on market conditions, one of those can be noticeably cheaper than the other. A shifter allows the user of the board to easily swap the sensors without worrying about voltage level compatibility. It also means that 3.3 V sensors can be used with 5 V microcontrollers, such as the Arduino Uno, which is very popular with hobbyists.

Sensors were mounted in sockets on the board and secured with thermoplastic adhesive. They can be easily removed in case of failure, or if they would be needed for another project.

2.3.3 Orientation measurement

A Bosch BNO055 nine degree of freedom IMU measures the orientation of the robot in space. It contains an accelerometer, a gyroscope, and a magnetometer. Just like for the pressure sensors, this particular chip was used because it is easier to work with than most solutions on the market. Cheaper IMUs simply output raw sensor data, and require the user to use sensor fusion algorithms to convert this data into orientation in space. BNO055, on the other hand, contains an on-board μC with dedicated sensor fusion firmware. The orientation quaternion can simply be read from it via the I^2C bus. After calibration, Euler orientation accuracy is within ±3°.

An off-the-shelf breakout board containing the IMU (Adafruit #2472) was purchased. It is mounted directly on top of the robot (Fig. 8), and connected to the μC using ultrathin (0.25 mm) insulated wire, to ensure that there is no drag force on the robot from the cable.

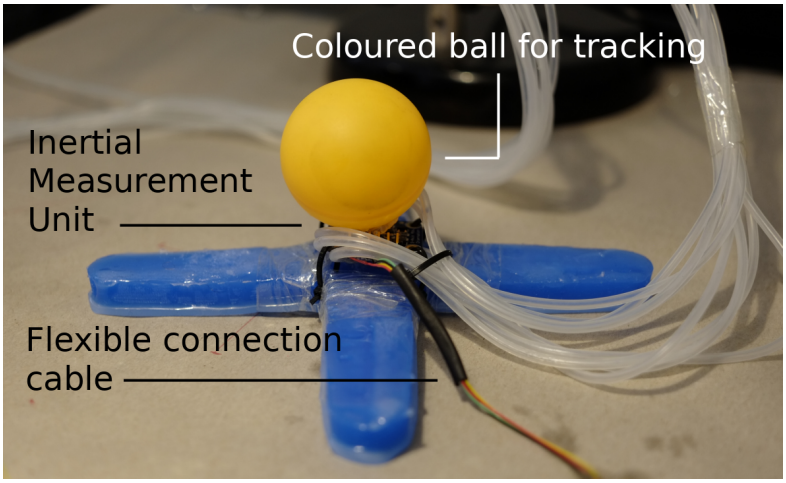


Figure 8: Sensors attached directly to the robot

2.3.4 Position tracking

Position of the robot is measured using a computer vision system (see Fig. 9b and 8). A camera is mounted on top of the operating area, and connected directly to the computer which is running the high-level control algorithm. A ball of contrasting color is glued to the top of the robot. An algorithm, implemented in OpenCV [33], is used to track the position of the ball. It works as follows:

Algorithm 1: Position tracking algorithm

```
initialise the camera;
while not done do
    fetch an image frame from the camera;
    downsample the image to 600 × 480 pixels;
    convert the image to HSV color space;
    apply a color mask to isolate the ball;
    find the outside contour of the ball;
    calculate the image moments of that contour to find the centre of the ball;
end
```

The system can operate at 120 frames per second, and has a positional accuracy of around ± 5 *mm* across the entire operating range of the robot.

2.4 PC interface

The μC interfaces with a computer using a serial-over-USB connection. The binary CmdMessenger [34] protocol is used to ensure robust communication.

2.4.1 Microcontroller

The speed at which the control loop can run is fundamentally limited by how fast the microcontroller can transfer sensor readings to the computer. To ensure that this is not an issue, a Teensy 3.5 development board (featuring a NXP MK64FX512VMD12 ARM Cortex-M4 chip) is used as the microcontroller platform. The Teensy can carry out serial communication at USB speeds (12 $Mbits^{-1}$), which ensures that data throughput is not a limiting factor in the system (the time required to send data from the μC to the computer is $32\mu s$). Teensy was also used because it can easily be programmed with Arduino. This made the platform more accessible to hobbyists.

A breakout board for the Teensy was made from a strip of perfboard to reliably connect the other PCBs to it. IDC connectors were used for data connections, and screw terminals for power.

2.4.2 Firmware

Two control methods were implemented in the μC firmware — on/off mode and pressure control mode. In on/off mode, the controller simply switches the valves on and off. In pressure control mode, it is possible to directly set the pressure in each channel.

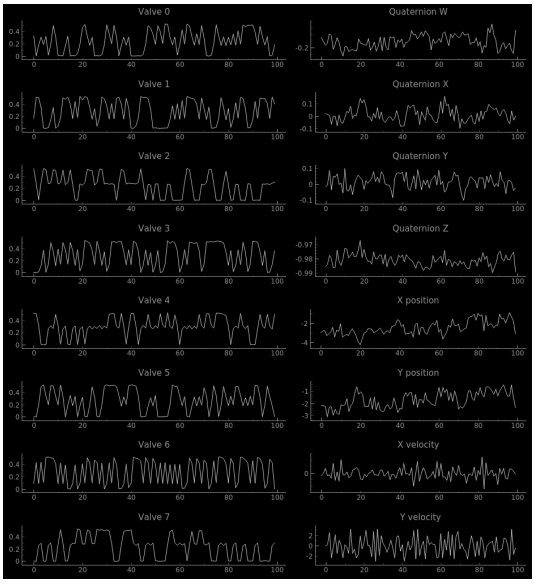
The pressure is controlled by pulse-width-modulating the valves at 15 Hz. Pressure in each channel is measured at 500 Hz and a low-pass, first order exponential-moving-average filter described by Arzi [35] is used to filter out the switching frequency and obtain a measurement of mean channel pressure. Pulse width is regulated by a proportional-integral (PI) controller. Desired pressure in the channel is used as a set-point for the controller. After tuning, it is possible to set mean pressure within 0.05 bar, although, of course, the actual pressure usually fluctuates by more than that due to the relatively slow switching frequency.

To make the system easy to use, all sensing and actuation capabilities are abstracted from the end user. Two simple application programming interfaces (APIs) can be used to access all the functions. For example, PI-controlled pressure in a channel can be set by using a `setPressure(int channel_no, float pressure)` method. One API is written in simple C++ and can be used directly on the Arduino board, the other is written in Python and can be used to control the Arduino from a PC. The APIs are portable — they should work on any Arduino board, and on Linux, Windows and OS X.

Finally, to make it easier to monitor and troubleshoot the system, a live data plotting system was implemented on the PC side using pyqtgraph (Fig. 9a), which enables real-time monitoring of data from the sensors.

2.5 High-level control ($\pi(o)$)

All the high-level control functionality was implemented in Python. All experiments were ran on an Intel Core i5-5300U computer with 8 GB of RAM. No GPGPU was used for computation.



(a) Live plot of data from the robot's sensors

(b) Visualisation of position tracking

Figure 9: Live visualisations of instrumentation data

2.5.1 Gym environment

In order to abstract the hardware described above from the control algorithm, it was packaged as an OpenAI gym environment [16]. Gym environments are routinely used in the RL community as a standardised way of testing the performance of control algorithms. Their structure closely follows the MDP formalism. Episodes are divided into steps. The `step(action)` function takes an action a_t as an input, applies a_t in the environment, and outputs an observation o_{t+1} and reward r_t, which are then fed back into the control algorithm, which uses them to provide a new action etc.

In the constructed environment, this loop is run at 3Hz (even though the hardware permitted much faster iterations, reasons for this are explained in the next section). A delay of $50ms$ is placed between taking an action and reading sensors, to ensure that the observed state is the result of the taken action. o is a 16-dimensional float vector containing all the sensor data, and a is an eight-dimensional binary vector, where 1 corresponds to an active valve, and 0 to an inactive one. In other words, the controller only works by switching valves on or off, the pressure control scheme is not used.

Abstracting the hardware in a convenient form will allow any continuators of this work to easily pick it up. It also means that many off-the-shelf, open-source RL algorithms which have already been adapted to operate on OpenAI gym environments can be used to control the system with minimal modifications.

2.5.2 Details of the control algorithm

The particular model-based RL method chosen for this project was first introduced by Nagabandi et al. [36]. It is described in Algorithm 2.

Algorithm 2: Model-based reinforcement learning with data aggregation

```
/* collect random data                                              */
run the environment R times using a purely random policy to collect a random
 dataset D_r;
initialise new dataset D ⟵ D_r;
for i = 0 to N do
    use D to learn a model of the environment;
    Initialise empty dataset D_iteration;
    for i = 0 to K do
        /* run the robot using model-predictive control             */
        while not done do
            gather observations o_t;
            use the model-based controller to obtain an action a_t from o_t;
            execute a_t;
            observe next state o_{t+1}, and get reward r_t;
            append (o_t, a_t, o_{t+1}, r_t) to D_iteration;
        end
    end
    Append D_iteration to D
end
```

First, the robot is run using a completely random policy, i.e. by drawing actions either from a $Uniform[-1, 1]$ distribution, if the actions are continuous, or from a binomial distribution with 0.5 probability of success, if they are binary. Data collected during those random runs is used to train a model. The robot is then run with a model predictive controller (MPC), which tries to maximise its cumulative reward (implementation details of the controller are described in the next section). Data collected during those runs is added to the dataset, which is used to periodically retrain the model.

This scheme is called *data aggregation*. The initial model is not very accurate, as it does not correspond to the actual running conditions of the machine. Observations created when operating in a purely random manner are different from those collected when the machine is fulfilling its intended function. By iteratively running the algorithm, collecting data and improving the model, it will get closer and closer to the true dynamics of the system.

2.5.3 Implementation of the control algorithm

The algorithm was implemented in Python, using Tensorflow (a numerical computing and automatic differentiation package) for most of the computational work. It is based

on an architecture provided by Kahn et al. [37]. This section will describe the most salient features of the implementation.

Reward function The robot's task is to simply walk forward in a straight line as fast as possible. The reward function $r(o_t, o_{t+1}, a_t)$ must incentivise it to do that, so it was set equal to the forward velocity of the robot:

$$r(o_t, o_{t+1}, a_t) = x' \tag{8}$$

Where x is the position of the robot along the x-coordinate and x' is its forward velocity. A reward-maximising controller will try to walk forward as fast as possible. The same reward function will also be used to quantify the algorithm's performance in Section 3.

Kahn et. al [37] used a slightly different reward, which incentivised their agent to keep its body level. This was implemented initially, with the reward:

$$r(o_t, o_{t+1}, a_t) = x' - 5\Omega_x - 5\Omega_y \tag{9}$$

Where Ω_x and Ω_y are the x and y components of the orientation quaternion. However, on the present system, this reward function resulted in worse performance than the simpler one in Equation 8.

Model The model is parameterised by a neural network $f_\theta(s_t, a_t)$, where θ represents the parameters of the network. The inputs to the network are the state of the robot and the planned action — the input layer is a vector in $\mathbb{R}^{24}$, a combination of data from 16 sensors and 8 actions. The output is the difference δo between the next state o_{t+1} and the current state o_t, which is a vector in $\mathbb{R}^{16}$. The next state can be simply calculated from:

$$o_{t+1} = o_t + \delta o \tag{10}$$

Optimum structure of the network was determined experimentally, as will be shown in the next section. The network is trained on a dataset of transitions $(o_t, o_{t+1}, a_t)_i$. o_t is subtracted from o_{t+1} to obtain δo. Then δo is normalised according to:

$$\delta o_{norm} = \frac{\delta o}{\sigma_o} - \mu_o \tag{11}$$

Where σ_o is the standard deviation and μ_o is the mean of the dataset. The training objective is to minimise the mean squared error on the dataset:

$$\epsilon(\theta) = \frac{1}{|D|} \sum_{(o_t, o_{t+1}, a_t) \in D} \frac{1}{2} ||\delta o_{norm} - f_\theta(o_t, a_t)||^2 \tag{12}$$

In other words, the aim of the training is to make the state transitions predicted by the neural network match the real ones as closely as possible. Training was done using the Adam optimiser, which is a variation of stochastic gradient descent (SGD). SGD is an optimisation method (minimising an error function is simply an optimisation problem).

A single sample $(o_t, o_{t+1}, a_t)_i$ is drawn at random (hence Stochastic) from the dataset. o_t and a_t are set as the inputs of the neural network, and a forward pass is run through it to see what its prediction is for that sample. Error between the predicted δo and the real one is calculated. By using the *backpropagation algorithm* (the details of which will not be explained here), this error is attributed proportionally to the weights in the network, i.e. the gradient of the error with respect to each weight in the network is calculated (hence Gradient). The weights are then adjusted proportionally to the magnitude of their gradients — the algorithm takes a *step* in the direction of the gradient. For certain (convex) optimisation problems, this method is guaranteed to find the global minimum of the function. The process can be visualised as taking small steps down a hill in the direction of steepest gradient, hence Descent. The Adam optimiser differs from SGD in the way step size is calculated.

Controller Action selection in the inner while loop of Algorithm 2 is done by random shooting, described in Algorithm 3.

Algorithm 3: Model Predictive Control

```
draw S random action sequences A of length χ from a binomial distribution with
 0.5 success probability;
for each A do
    use the learned model to simulate the trajectory τ that the robot will enter if
     it executes each action sequence, and the rewards it will get along the way;
end
pick the action sequence with the largest expected cumulative reward;
execute the first action from the sequence;
```

The algorithm picks a number of random action sequences A from a binomial distribution with 0.5 success probability. The length of those sequences is called the projection horizon χ. Then, it uses the learned model to simulate what states it will enter if this random sequence of actions will be executed, and the rewards it will receive along the way. It then estimates the best action sequence by picking the one with the highest cumulative reward:

$$A_{best} = \underset{A}{\operatorname{argmax}} \sum_{t_{start}}^{t_{start}+\chi} r(a_t, s_t) \tag{13}$$

Even though the model is used to predict a number of steps forward, random shooting and prediction is repeated at every step, and only the first action of the sequence is taken. This scheme is called model predictive control (MPC), and it helps to stabilise the controller in spite of model errors. Model errors tend to accumulate, which makes the accuracy of predicted state transitions worse as projection horizon is increased. If one was to calculate the best action sequence only once, at the start, and then execute it without any remodelling (i.e. if one were to use an open-loop controller), the performance would be suboptimal.

Testing the algorithm The algorithm was tested on a simulated OpenAI gym environment (HalfCheetah-v2), using the default parameters specified by Kahn et al. [37]. It achieved the same reward (300 after 9 iterations) as that described in their paper. It was used, with the same default parameters, to control the soft robot. The next section explains the results of testing and how the algorithm was tuned to improve its performance.

3 Results

As can be inferred from the previous section, RL algorithms require the user to set an extensive range of parameters — the exact structure of the neural network, the MPC projection horizon, the amount of rollouts before retraining the model etc. When run with the default parameters suggested by Kahn et al. [37], the algorithm does work (see Fig. 10). After training, the reward increases substantially, and the robot does move forward, but its performance plateaus at a mean reward of 8.6 (standard deviation 4.5). Data collected during this experiment can be used to tune the algorithm parameters.

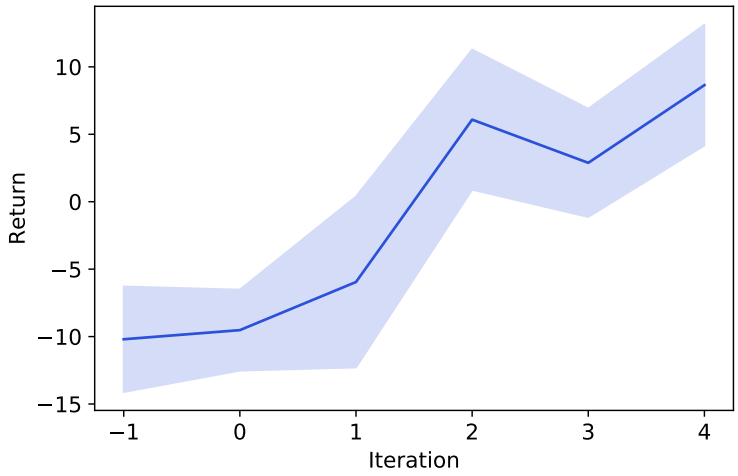


Figure 10: Performance of the model-based controller using default parameters. Iteration -1 used a purely random policy. Shaded area represents the standard deviation in results.

3.1 Tuning the algorithm

The modelling process is an instance of supervised learning — a branch of machine learning where the true values of datapoints are known, and the model's task is to approximate them as closely as possible. In order to improve the model, one needs a *training* dataset, and an *evaluation* dataset. Two separate datasets are needed, because powerful function approximators such as neural networks are prone to *overfitting* — if

the number of model parameters is too large, the network will simply learn to replicate each data point. While this would achieve a perfect fit, the model would then fail to generalise to data it has not seen before. A good analogy is to think about fitting best fit line in a two-dimensional experiment. One could fit a straight line to the data, or a complex polynomial. With a sufficient number of terms, the polynomial will pass through all the points, but it may be worse at predicting the true trend in the data than a straight line.

In this case, the training dataset was simply the dataset obtained while running the model with default parameters. In order to collect the evaluation dataset, a walking algorithm first presented by Stokes at al. [27] was reimplemented. It is an open-loop, timing-based method. It simply specifies which valves need to be switched at what time in order for the robot to walk forward. This implementation is also used later on as a benchmark to evaluate the performance of the model-based controller. It achieved an average reward of 74.5 (s.d. 22.2) across five runs.

The prediction is also open-loop — the algorithm is given an initial state and the sequence of actions taken by the benchmark policy in a given episode. Its task is to predict the entire episode based only on this data. The mean square error between the prediction and real measurements serves as a measure of the model's performance:

$$\epsilon(\theta) = \frac{1}{|D|} \sum_{\tau \in D} \sum_{(o_t, o_{t+1}, a_t) \in \tau} \frac{1}{2} ||o_{t+1} - \hat{o}_{t+1}||^2 \tag{14}$$

Where $\hat{o}_{t+1}$ is the model's prediction of the next state.

The most robust way of tuning an algorithm's parameters is to change them one by one, run an experiment after each change, and test how quickly it learns and how well it performs. Ideally, one would also test each combination of parameters. It was not possible to do in this project, because of time constraints and because testing was done on real hardware (each experiment took at least two hours to run). Instead, a combination of testing a posteriori on already collected data and engineering intuition was used. The sections below explain how the most crucial model parameters were determined. Table 4 summarises the results.

3.1.1 Control Frequency/Projection horizon

Frequency at which the controller runs (ν) is related to the number of steps which the algorithm simulates forward. As was explained in Section 2.5.3, at each step the controller simulates the results of taking a number of random action sequences. The length of those action sequences has two bounds.

If it was too short, the controller would be short-sighted. In some cases, in order to get a larger reward, one needs to make a small sacrifice. In this work, the reward is proportional to the forward velocity of the robot. Fig. 13 shows the behaviour of the benchmark controller. Note that it swings the robot's body slightly backwards (the component of velocity is negative, which incurs a small negative reward), in order to make a step forward (which gives a large positive reward). A short-sighted controller

Parameter	Symbol	Tuned value	Default value
Control frequency	ν	3 Hz	-
Projection horizon	χ	25 actions	15 actions
Random action sequences tested	S	15000→5000	4096
Hidden layers in neural network	-	1	1
Neurons in hidden layer	-	100→300	500
Training iterations	-	500	200
Optimiser parameters	-	default	default
Episode length	-	300 steps (100 s)	-
Episodes between training	K	5	10
Training iterations	N	5	10
Random collection episodes	R	10	10

Table 4: Tuned parameters of the control algorithm

would not be able to notice this possibility, and could behave suboptimally. However, as the projection horizon gets longer, model errors start to accumulate, and the projection gets worse, which could also lead to suboptimal behaviour.

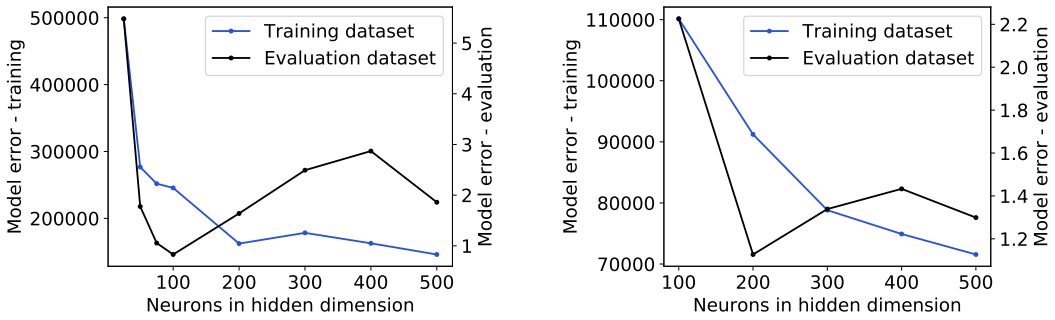
This is where the interplay between control frequency and projection horizon comes into play. The model error is dependent on the number of control steps that are being projected forward, not the real (wall clock) time of the projection. If the model can project 10 steps forward with an acceptable error level, running the control algorithm at 1 Hz would enable one to project 10 s into the future, while running it at 100 Hz — only 0.1 s. However, if one sets the control frequency too low, the robot could move too slowly, or become unstable. In general, one would like to use the lowest frequency possible, as it also increases the time available for computation.

The benchmark algorithm used a 0.35 s timestep (2.86 Hz). The control algorithm was run at similar frequency of 3 Hz. The period of the robot's *swing* was around 6 s, so a horizon of 25 steps (50% larger than the swing period) was used.

3.1.2 Neural network structure

The deeper and wider the neural network, the more likely it is to overfit the dataset. However, if the structure is not large enough, it may not be able to accurately approximate data in the model. In order to determine the optimal architecture, model error on the evaluation dataset was calculated, using Equation 14, for a number of architectures of varying layer sizes and numbers of layers. Results are presented in Fig. 11. Overfitting is clearly visible — after the network becomes more complicated than a single hidden layer with 100 units, the model's error on the training dataset keeps decreasing, but error on the evaluation dataset increases. A 1×100 network was therefore used for subsequent experiments. Note that the experiment was only run once for each net size, and then repeated a few times for the chosen network size to ensure that the coefficient of variation (COV) was not too large (it was under 0.05 for three repetitions). There

is a natural variation between runs of the training algorithm as weights in the net are initialised to pseudorandom values at the start of training. A better experiment would test each network architecture multiple times and then report mean and variance in the results. However, on the available hardware, even this experiment took over three hours; repeating it multiple times would be too costly.



(a) One hidden layer

(b) 2 hidden layers

Figure 11: Determining the optimum size of the neural network. The algorithm was trained on the training data for 200 iterations using neural networks of varying sizes.

3.1.3 Neural network training parameters

The optimal number of gradient descent steps for the Adam optimiser was determined in a similar way to the neural network architecture. Training losses on the training dataset were evaluated for different numbers of steps. The results are shown in Fig. 12. One can see that there is no improvement after 500 iterations, so 500 iterations were used for the evaluation runs. TensorFlow's default parameters for Adam (e.g. learning rate of 0.001) were used.

3.1.4 Number of random action sequences

As was explained above, at each step the algorithm attempts to simulate a number of 25-step random action sequences, and executes the first step of the one that promises the highest reward. The higher this number, the more likely it is to find the optimal A. The number of simulations that one can carry out in a single step is limited by the available computational resources. On the available hardware, it was possible to simulate 15000 sequences in each step.

3.1.5 Iterations and episode timing

The last parameters of the training algorithm treated here are episode lengths and iteration numbers, as described in Algorithm 2. They influence how often the model is

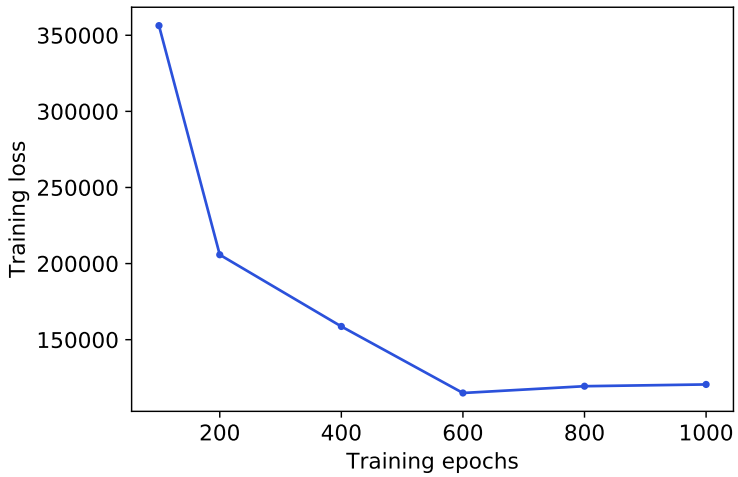


Figure 12: Mean square error of the network on the training dataset vs. number of training iterations. Evaluated on a 1×100 MLP network.

retrained (the amount of data collected between training steps). In theory, one could retrain the model after each step or episode, which would mean that it would be always up to date, and maximum performance will be reached in the smallest number of steps possible. However, on the available computer hardware, training took too long to make this possible.

Instead, episode length was set to 300 steps (each episode took 100 s). The algorithm was retrained every five episodes. Initial random data collection was done for 10 episodes (16.7 min). Training was run until the policy stopped improving, which was usually after 4-5 iterations (50 min). This means, that, at the end of the training, the dataset contained about 29% of random data, and 71% of data collected while training. This result is consistent with those reported by Nagabandi et al. [36], who found that there is no improvement in model performance after 50% of the dataset is composed of training data.

This concludes the parameters tuning section. The next one examines the performance of modelling and control which used the improved parameters.

3.2 Performance of modelling and control

3.2.1 Model

Example open-loop predictions of a fully trained controller for an example episode from the evaluation dataset are presented in Fig. 13. This model was trained on 50 min of data, as explained in the previous section. Only the first 100 steps are included for clarity.

It is difficult to quantify the performance of a model, as any error estimations one could come up with would be relative, depending on the thing being modelled. For

certain large, open source datasets used in e.g. image recognition, researchers publish the metrics of their models, and compare their results to evaluate the relative merits of their approaches. Naturally, there is no such data for this work. It was shown in Figures 11 and 12 in the previous section that model error on the evaluation dataset does improve with training and with better network structure. It goes down from 5.2 to 0.8, but, without a comparison, those numbers are meaningless. Only a qualitative assessment of model performance is possible. Fig. 13 shows that the model can predict most state variables reasonably well, except for the x-position, and W and Z components of the orientation quaternion (Ω_w, Ω_z).

The error prediction of Ω_z is most likely because the real value hardly changes at all (it varies by less than 1%). This makes it hard to compute its gradients and, hence, train the network. A possible reason for the errors in prediction of position is that position does not satisfy the Markov state property. The future state of a system which posseses the Markov property depends only on its current state, and the action taken by the agent. It does not depend on the previous states that the algorithm entered. Distance from a starting point is not Markovian. The chosen neural network structure does not have any memory of past states, so it should have issues with predicting the numerical value of position, which depends on all past states.

However, the model is accurate at predicting the x velocity component, which is the most important parameter in the model, as it is being used to calculate rewards for the controller. This means that the model can be used to control the robot.

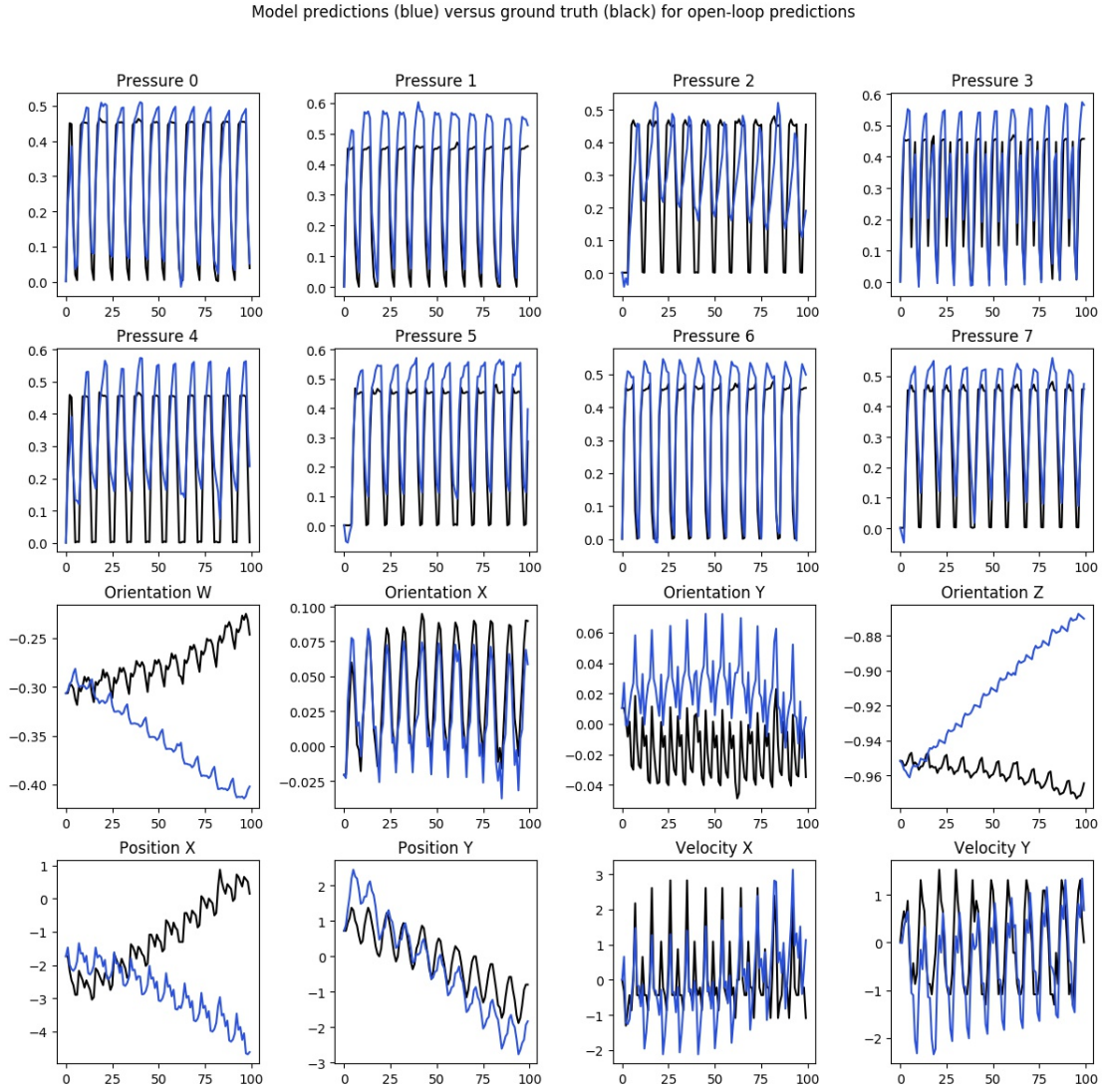


Figure 13: Open-loop predictions for each of the state parameters (blue) vs. real data (black) for data collected when running the benchmark policy.

3.2.2 Model Predictive Control

Quantitative analysis Fig. 14 presents the results of running the MPC described above using the tuned parameters. Initially, the controller behaved marginally better than a random policy, with a mean reward of 2.8 (s.d. 8.8), and then got much worse. It consistently received a reward of -12.3 (s.d. 1.2).

This was caused by using the relatively small, 1×100, neural network. Even though in the investigation carried out in the previous section this network size seemed to perform best, it was clearly too small for this application. After shifting to a network with 300

hidden units, the performance improved drastically, reaching a mean reward of 16.9 (s.d. 16.5). The number of simulated action sequences had to be proportionally decreased to 5000 in order to maintain the 3 Hz control frequency. Maximum reward reached in one episode was 28.6, 40% of that achieved by the benchmark policy. Mean walking speed was 7×10^{-4} ms^{-1}, compared with 2×10^{-3} ms^{-1} for the benchmark policy.

The network size may have been inadequate in the new setting because, when more optimal parameters were used, the dynamics of the robot got more complicated, and the 1×100 network was not able to model them accurately. Alternatively, inadequate sizing could be attributed to random variations in the experiment explained in Section 3.1.2.

The optimised policy achieves 1.94 times the reward of the one with default parameters. However, it has a large COV of 0.97. Reasons for this variability are presented in the next section.

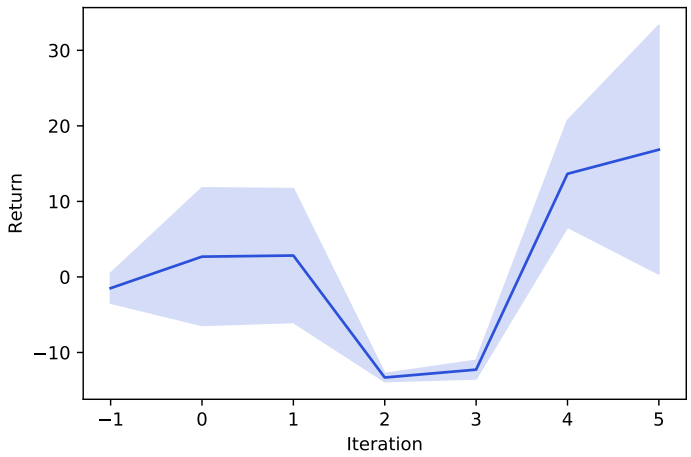


Figure 14: Performance of the model-based controller using improved parameters. Iteration -1 used a purely random policy. Shaded area represents the standard deviation in results. Large gain between iterations 3 and 4 is due to increasing the number of neurons in the hidden layer.

Qualitative analysis — observed gaits Performance of the control algorithm is heavily affected by the action selection method. Because actions are drawn randomly from a binomial distribution, a lot of noise is introduced into the controller. Even though on average it may behave optimally, it will inevitably take suboptimal actions from time to time, when no optimal ones are drawn. This issue could probably be alleviated by using more powerful computer hardware, which could carry out more simulations between steps, which would increase the probability of drawing optimal action sequences. However, it is inherent in the algorithm, and cannot be completely eliminated. There are $(2^8)^{25}$ possible action sequences, one could never investigate all of them.

For this reason, Nagabandi et al. [36] suggest using their RL algorithm to extract gait

sequences, which can then be used in a simpler, open-loop controller, or as a starting point for a more optimal model-free controller, which would not be subject to the noise issue.

Two distinct gaits could be observed in the experiments. The first one (see Fig. 15 and supplementary video 2) emerges in the first two iterations of training (after 15–20 minutes of experience). The robot does not actuate its hind legs at all, and moves forward by alternatively using the two front ones. It uses the sideways twisting of a leg when only one channel in it is actuated to first put it in front of itself (phase III), and then to push the entire body forwards (phase IV).

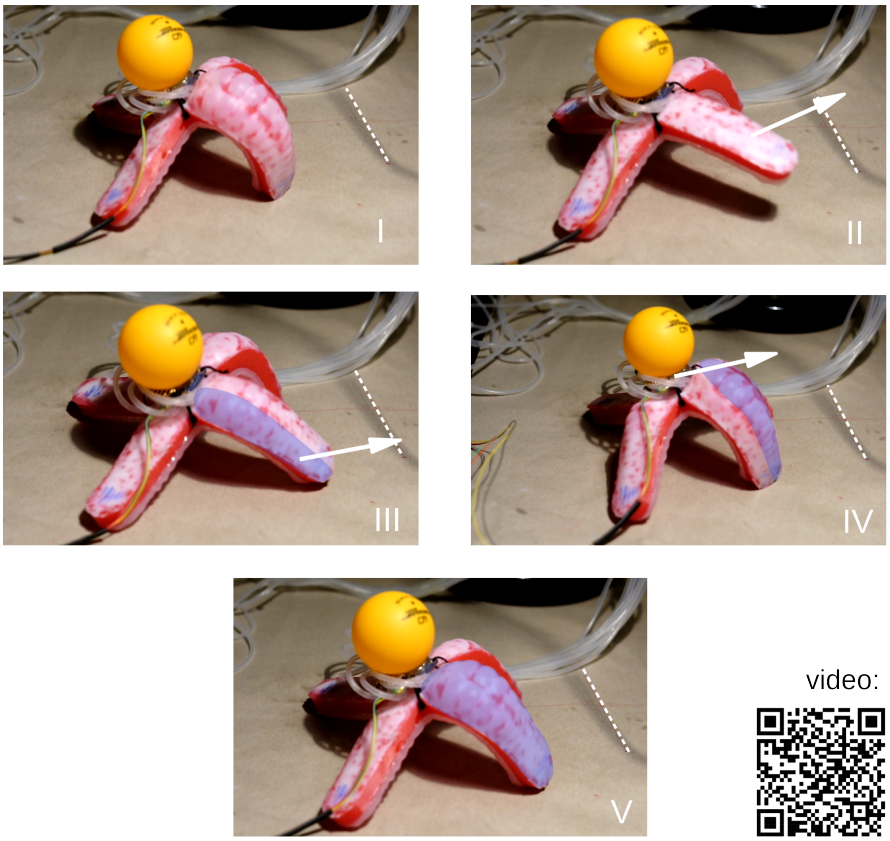


Figure 15: Two-legged gait developed in the initial stages of training. This figure shows the motion of one leg, the gait alternates these movements between left and right legs. Shaded areas denote which part of the joint is actuated. Dashed line is for position reference.

The second gait (Fig. 16 and supplementary video 3) is faster and emerges in the final stages of training (after around 45 minutes). The robot first rotates itself so that only one leg faces the front, and then creates a wave-like motion with front- and back- facing legs, while using the sideways-facing ones to push itself forward. It first actuates the entire front leg to stabilise itself (phase I). Then, like in the first gait, it first twists the

sideways-facing legs to move them forward (phase III), and then uses them to push the entire body (phase IV). As it moves forward, it deactivates the front limb, which would prevent the movement, and actuates the back, which stops it from slipping backwards. This front-to-back transition creates an undulation, much like in a crawling snake.

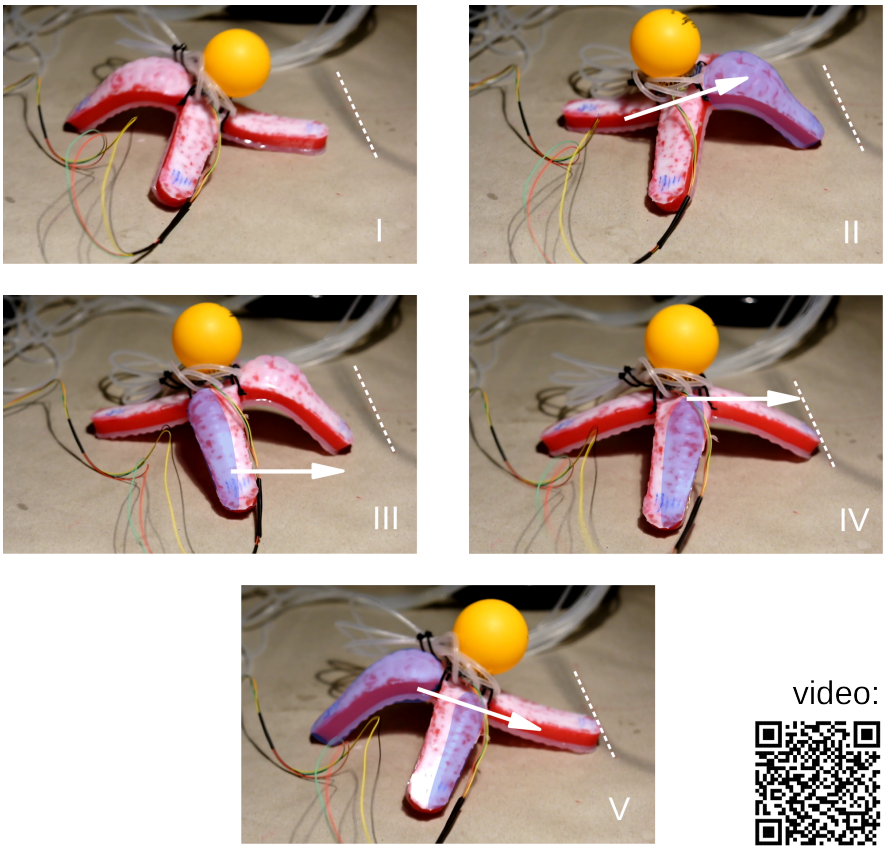


Figure 16: Undulating gait developed in the final stages of training. Shaded areas denote which part of the joint is actuated. Dashed line is for position reference.

The behaviour was consistent across different experiments. Retraining the model using the same dataset, and with the same network topology produced the same gaits. When the robot is initiated with one leg facing forwards, it starts using the undulating gait without an initial rotation phase.

4 Conclusions

This work demonstrated a practical application of model-based reinforcement learning methods to the control and modelling of soft robots. An eight-jointed soft walking robot was used as a testbed. An electro-pneumatic fluidic control board was designed and built to actuate it. Design of the board emphasises versatility and ease of use, as

its target audience are hobbyists and enthusiasts who would like to learn about soft robotics. A suite of pressure, orientation and position sensors along with a the required microcontroller firmware and software abstraction layers was developed to provide inputs for the control algorithm.

A state-of-the-art reinforcement learning algorithm was implemented and adapted for use on the pneumatic hardware. Its main objective was to learn to walk forward as fast as possible. Algorithm parameters were tuned to achieve optimal performance. The resulting policy produced two plausible walking gaits, a bipedal one developed early in training, and an undulating one in the later stages. After 50 *min* of training, the policy achieved an average reward equal to 23% of that received by a benchmark policy. The bechmark was an open-loop control sequence developed by the inventors of the soft robot used for experiments. The best reward achieved was 40%.

Nagabandi et al. [36], who first introduced the reinforcement learning algorithm used in this work, highlighted that they managed to achieve good performance on simulated systems with large numbers of degrees of freedom and complex frictional contacts. This work pushes the boundary further by showing that it can perform satisfactorily on a continuous system (the soft robot used did not have any discrete degrees of freedom to speak of) with real-world dynamics.

The method described in this work is versatile — it could be used to produce useful gait patterns and motor primitives in a variety of real systems. However, because actions are selected from a pool of randomly drawn action sequences, the controller is very noisy, and the preformance varies substantially between episodes (coefficient of variation is 0.97). This means that this particular model-based algorithm should be used as a source of plausible gaits for open-loop control, or as an initialisation for another, perhaps model-free method. The next section describes this in more detail.

5 Future work

Because of time constraints imposed upon this project, a number of obviously interesting experiments which could further test the capabilities of the system could not be carried out. Firstly, the observed gait sequences could be implemented in an open-loop timing-based controller, and their performance compared with the benchmark policy.

Nagabandi et al. [36] showed that they could use the same model to walk in different directions, and even to follow a prescribed trajectory. Fig. 13 shows that the produced model was equally competent at predicting x and y velocity components, so it should be possible to make the robot follow a trajectory as well.

Nagabandi et al. [36] also used the model-based network as an initialisation for a model-free method, which achieved much higher performance, but required substantially more training data (still, however, less than a model-free method initialised randomly). Because the produced hardware was abstracted as an OpenAI gym environment, it should be relatively easy to apply an off-the-shelf model-free algorithm, such as a Deep-Q Network, to the robot. However, because such test could require the robot to operate for an even longer period of time than in the present experiments, a few modifications

would have to be made to the hardware.

The soft robots used here lasted for around four hours of testing. This was probably caused by fluctuations in input pressure. The SMC AW20 regulator which came with the compressor is not accurate at the low 40 kPa pressures used in the experiments. In fact, its minimum rated operational pressure is 50 kPa. Because the pneumatic system was originally designed to be used at the higher pressures required by Arthrobots, this was not deemed a problem. However, in the present setup. as the pressure in the tank decreases, the output pressure increases by too much, which results in excessive inflation and premature failure of the robots. A percision regulator designed to operate at lower pressures (such as SMC ARP20) would be a much better fit for this application.

For longer tests, it would also be useful to design an automated method of resetting the robot to the starting position (a piece of string pulled by an electric motor would be sufficient). The current method requires the researcher to do this, which could be time consuming for longer tests.

Finally, if the algorithms were run on better computer hardware (with a modern GPGPU), it would be possible to simulate more random action sequences (which should result in better algorithm performance), and to carry out more comprehensive tests to determine optimum algorithm parameters.

References

[1] M. Raibert, *Legged robots that balance*, ser. MIT Press series in artificial intelligence. Cambridge, Mass. ; London: MIT, 233 pp.

[2] J. Choi, B. Na, P.-G. Jung, D.-w. Rha, and K. Kong, "WalkON suit: A medalist in the powered exoskeleton race of cybathlon 2016," *IEEE Robotics & Automation Magazine*, vol. 24, no. 4, pp. 75–86, Dec. 2017, ISSN: 1070-9932. DOI: 10.1109/MRA.2017.2752285. [Online]. Available: http://ieeexplore.ieee.org/document/8097012/ (visited on 10/23/2018).

[3] S. Collins, "Efficient bipedal robots based on passive-dynamic walkers," *Science*, vol. 307, no. 5712, pp. 1082–1085, Feb. 18, 2005, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.1107799. [Online]. Available: http://www.sciencemag.org/cgi/doi/10.1126/science.1107799 (visited on 10/13/2018).

[4] D. N. Beal, F. S. Hover, M. S. Triantafyllou, J. C. Liao, and G. V. Lauder, "Passive propulsion in vortex wakes," *Journal of Fluid Mechanics*, vol. 549, no. -1, p. 385, Feb. 8, 2006, ISSN: 0022-1120, 1469-7645. DOI: 10.1017/S0022112005007925.

[5] R. Tedrake, *Underactuated robotics: Algorithms for walking, running, swimming, flying, and manipulation (course notes for MIT 6.832)*, Oct. 23, 2018. [Online]. Available: http://underactuated.mit.edu/ (visited on 02/23/2019).

[6] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*, ser. Adaptive computation and machine learning. Cambridge, Mass: MIT Press, 1998, 322 pp., ISBN: 978-0-262-19398-6.

[7] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, 1991, ISSN: 08936080. DOI: 10.1016/0893-6080(91)90009-T. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/089360809190009T (visited on 10/24/2018).

[8] UC Berkeley. (2018). CS294-112 deep reinforcement learning lecture notes, [Online]. Available: http://rail.eecs.berkeley.edu/deeprlcourse/ (visited on 10/20/2018).

[9] M. T. Tolley, R. F. Shepherd, B. Mosadegh, K. C. Galloway, M. Wehner, M. Karpelson, R. J. Wood, and G. M. Whitesides, "A resilient, untethered soft robot," *Soft Robotics*, vol. 1, no. 3, pp. 213–223, Sep. 2014, ISSN: 2169-5172, 2169-5180. DOI: 10.1089/soro.2014.0008. [Online]. Available: https://www.liebertpub.com/doi/10.1089/soro.2014.0008 (visited on 10/23/2018).

[10] J. E. Huber, N. A. Fleck, and M. F. Ashby, "The selection of mechanical actuators based on performance indices," *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 453, no. 1965, pp. 2185–2205, Oct. 8, 1997, ISSN: 1471-2946. DOI: 10.1098/rspa.1997.0117. [Online]. Available: http://www.royalsocietypublishing.org/doi/10.1098/rspa.1997.0117 (visited on 03/23/2019).

[11] R. Deimel and O. Brock, "Soft hands for reliable grasping strategies," in *Soft Robotics*, A. Verl, A. Albu-Schäffer, O. Brock, and A. Raatz, Eds., Springer Berlin Heidelberg, 2015, pp. 211–221, ISBN: 978-3-662-44506-8.

[12] A. C. McConnell, M. Vallejo, R. C. Moioli, F. L. Brasil, N. Secciani, M. P. Nemitz, C. P. Riquart, D. W. Corne, P. A. Vargas, and A. A. Stokes, "SOPHIA: Soft orthotic physiotherapy hand interactive aid," *Frontiers in Mechanical Engineering*, vol. 3, Jun. 2, 2017, ISSN: 2297-3079. DOI: 10.3389/fmech.2017.00003. [Online]. Available: http://journal.frontiersin.org/article/10.3389/fmech.2017.00003/full (visited on 03/23/2019).

[13] C. Lamb, G. Nucci, K. Lamson, B. Piercy, and P. Lynn, "Leg exoskeleton system and method," U.S. Patent 2018/0296425.

[14] C. Armbrust, L. Kiekbusch, T. Ropertz, and K. Berns, "Soft robot control with a behaviour-based architecture," in *Soft Robotics*, A. Verl, A. Albu-Schäffer, O. Brock, and A. Raatz, Eds., Springer Berlin Heidelberg, 2015, pp. 81–91, ISBN: 978-3-662-44506-8.

[15] J. Roßmann, M. Schluse, M. Rast, E. G. Kaigom, T. Cichon, and M. Schluse, "Simulation technology for soft robotics applications," in *Soft Robotics*, A. Verl, A. Albu-Schäffer, O. Brock, and A. Raatz, Eds., Springer Berlin Heidelberg, 2015, pp. 100–119, ISBN: 978-3-662-44506-8.

[16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," *arXiv:1606.01540 [cs]*, Jun. 5, 2016. arXiv: 1606.01540. [Online]. Available: http://arxiv.org/abs/1606.01540 (visited on 10/23/2018).

[17] Y. Engel, P. Szabo, and D. Volkinshtein, "Learning to control an octopus arm with gaussian process temporal difference methods," 2005.

[18] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An application of reinforcement learning to aerobatic helicopter flight," 2006.

[19] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *arXiv:1504.00702 [cs]*, Apr. 2, 2015. arXiv: 1504.00702. [Online]. Available: http://arxiv.org/abs/1504.00702 (visited on 09/28/2018).

[20] J. Peters, S. Vijayakumar, and S. Schaal, "Reinforcement learning for humanoid robotics," p. 20,

[21] H. Benbrahim and J. A. Franklin, "Biped dynamic walking using reinforcement learning," *Robotics and Autonomous Systems*, vol. 22, no. 3, pp. 283–302, Dec. 1997, ISSN: 09218890. DOI: 10.1016/S0921-8890(97)00043-2. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0921889097000432 (visited on 10/23/2018).

[22] J. Kober and J. Peters, "Learning motor primitives for robotics," in *2009 IEEE International Conference on Robotics and Automation*, Kobe: IEEE, May 2009, pp. 2112–2118, ISBN: 978-1-4244-2788-8. DOI: 10.1109/ROBOT.2009.5152577. [Online]. Available: http://ieeexplore.ieee.org/document/5152577/ (visited on 09/28/2018).

[23] A. Gupta, C. Eppner, S. Levine, and P. Abbeel, "Learning dexterous manipulation for a soft robotic hand from human demonstration," *arXiv:1603.06348 [cs]*, Mar. 21, 2016. arXiv: 1603.06348. [Online]. Available: http://arxiv.org/abs/1603.06348 (visited on 09/27/2018).

[24] T. Yang, Y. Xiao, Z. Zhang, Y. Liang, G. Li, M. Zhang, S. Li, T.-W. Wong, Y. Wang, T. Li, and Z. Huang, "A soft artificial muscle driven robot with reinforcement learning," *Scientific Reports*, vol. 8, no. 1, Dec. 2018, ISSN: 2045-2322. DOI: 10.1038/s41598-018-32757-9. [Online]. Available: http://www.nature.com/articles/s41598-018-32757-9 (visited on 10/23/2018).

[25] M. Giorelli, F. Renda, G. Ferri, and C. Laschi, "A feed-forward neural network learning the inverse kinetics of a soft cable-driven manipulator moving in three-dimensional space," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo: IEEE, Nov. 2013, pp. 5033–5039, ISBN: 978-1-4673-6358-7 978-1-4673-6357-0. DOI: 10.1109/IROS.2013.6697084. [Online]. Available: http://ieeexplore.ieee.org/document/6697084/ (visited on 10/23/2018).

[26] H. Zhang, R. Cao, S. Zilberstein, F. Wu, and X. Chen, "Toward effective soft robot control via reinforcement learning," in *Intelligent Robotics and Applications*, Y. Huang, H. Wu, H. Liu, and Z. Yin, Eds., Springer International Publishing, 2017, pp. 173–184, ISBN: 978-3-319-65289-4.

[27] A. A. Stokes, R. F. Shepherd, S. A. Morin, F. Ilievski, and G. M. Whitesides, "A hybrid combining hard and soft robots," *Soft Robotics*, vol. 1, no. 1, pp. 70–74, Mar. 2014, ISSN: 2169-5172, 2169-5180. DOI: 10.1089/soro.2013.0002. [Online]. Available: https://www.liebertpub.com/doi/10.1089/soro.2013.0002 (visited on 02/23/2019).

[28] F. Ilievski, A. D. Mazzeo, R. F. Shepherd, X. Chen, and G. M. Whitesides, "Soft robotics for chemists," *Angewandte Chemie International Edition*, vol. 50, no. 8, pp. 1890–1895, Feb. 18, 2011, ISSN: 14337851. DOI: 10.1002/anie.201006464. [Online]. Available: http://doi.wiley.com/10.1002/anie.201006464 (visited on 03/23/2019).

[29] Nemiroski, Alex, Y. Shevchenko, A. A. Stokes, B. Unal, A. Ainla, A. Sahradha, G. Compton, E. MacDonald, Y. Schwab, C. Zellhofer, and G. M. Whitesides, "Arthrobots," *Soft Robotics*, vol. 4, no. 3, pp. 183–190, Sep. 2017, ISSN: 2169-5172, 2169-5180. DOI: 10.1089/soro.2016.0043. [Online]. Available: https://www.liebertpub.com/doi/10.1089/soro.2016.0043 (visited on 09/27/2018).

[30] SMC Corp., "Rubber seal, 3 port solenoid valve/direct operated: Series v100," 2019. [Online]. Available: `http://content2.smcetech.com/pdf/V100_EU.pdf` (visited on 04/06/2019).

[31] British Standard Institution, "BS ISO 6358-2:2013 pneumatic fluid power — determination of flow-rate characteristics of components using compressible fluids part 2: Alternative test methods," in, 1st ed., May 1, 2013.

[32] Philips Semiconductors, "Application note AN97055: Bi-directional level shifter for i2c-bus and other systems.," p. 16, 1997.

[33] G. Bradski, "The OpenCV library," *Dr. Dobb's Journal of Software Tools*, 2000.

[34] T. Elenbaas. (2017). CmdMessenger, [Online]. Available: `https://github.com/thijse/Arduino-CmdMessenger` (visited on 04/01/2019).

[35] J. Arzi, "Tutorial on a very simple yet useful filter : The first order IIR filter," p. 8, [Online]. Available: `http://www.tsdconseil.fr/tutos/tuto-iir1-en.pdf`.

[36] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine, "Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning," *arXiv:1708.02596 [cs]*, Aug. 8, 2017. arXiv: `1708.02596`. [Online]. Available: `http://arxiv.org/abs/1708.02596` (visited on 02/23/2019).

[37] G. Kahn and S. Levine, "CS294-112 deep reinforcement learning HW4: Model-based RL," Oct. 10, 2018. [Online]. Available: `https://github.com/berkeleydeeprlcourse/homework/tree/master/hw4`.