OpenAI Test-Bed of Contact Based Grasping for Reinforcement Learning

October 2021

Words: 9999

Rob Lamprell

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

Abstract

In this project, we present a successfully implemented Mujoco-based [1] test-bed for Gomes' simulated GelSight sensor [2]. This forms our unique contribution to the research space, as it has not been done previously. The test-bed consists of a UR5 arm [3] and gripper [4] onto which two GelSight are appended. This GelSight-gripper is placed upon a table and presented with various spatially driven goals, defined within four unique environments which take the form of Reach, Slide, Push, and PickandPlace - mirroring OpenAI's Gym FetchEnvs [5]. Additionally, these environments can each be deployed in two forms. The first creates static objectives whereas the latter randomly places objectives within the limitations of the table's surface - constituting a more sparsely rewarded state space.

Though this paper's primary goal was around the implementation of the GelSight sensor, other observation types are freely available. These consist of vision, tactile, kinematic, or raw observations that can be utilized either singularly or in combination with one another. These observations are fed into our agents, who combine reinforcement learning algorithms with neural networks, to choose actions, in an attempt to solve their environment.

Two reinforcement learning algorithms have been appended to the test-bed. The first is Double Deep Q-Learning (DDQN) [6], an expansion of Deep Q-Learning [7] which utilizes two networks in order to avoid maximization bias [6, 8]. The second is Hierarchical Double Deep Q-Learning (HDDQN) [9, 10], an extension to DDQN which utilizes two agents - one extrinsic and one intrinsic. This extension aims to address the difficulties DDQN experiences in sparse reward environments - such as when randomizing objects and target locations across episodes.

Using a combination of the above, we offer a set of baseline results. Training both DDQN and HDDQN on static and random environments. We utilized a mixture of tactile, vision, and kinematic observations when training during development and training on only raw observations for the final model's implementation. The latter is due to the time-limited nature of the project, and as such it has not been possible to demonstrate convergence on sparse reward environments within complex observation spaces.

Student Declaration

I confirm that I have read and understood the University's Academic Integrity Policy.

I confirm that I have acted honestly, ethically and professionally in conduct leading to assessment for the programme of study.

I confirm that I have not copied material from another source nor committed plagiarism nor fabricated data when completing the attached piece of work. I confirm that I have not previously presented the work or part thereof for assessment for another University of Liverpool module. I confirm that I have not copied material from another source, nor colluded with any other student in the preparation and production of this work.

I confirm that I have not incorporated into this assignment material that has been submitted by me or any other person in support of a successful application for a degree of this or any other university or degree-awarding body.

SIGNATURE Rob Lamprell
DATE October 8, 2021

Acknowledgments

Daniel Gomes provided the original concept for this paper. Additionally, both he and Dr Shan Luo have dispensed invaluable input, guidance, and support throughout - both academic and emotional. Furthermore, alongside Dr Luo, Professor Yannis Goulermas gave critical feedback following the submission of both the Design Specification and Final Presentation documents.

It is with great pleasure that I express my thanks to each of them for their support throughout.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Approach . . . 2
1.3 Outcome . . . 3

2 Related Works **4**
2.1 Tactile Sensors . . . 4
2.2 Reinforcement Learning . . . 5
2.3 Transfer Learning . . . 7
2.4 Sim2Real . . . 7

3 Design **8**
3.1 Original and Updated Designs . . . 8
3.2 GelSight . . . 8
3.3 Deep Reinforcement Learning . . . 9
3.3.1 Preliminaries . . . 9
3.3.2 Double Deep Q-Learning . . . 11
3.3.3 Hierarchical Double Deep Q-Learning . . . 12
3.3.4 Potential Alternative Methodologies . . . 13

4 Implementation **16**
4.1 Acknowledgment of Key Software . . . 16
4.2 Environment . . . 16
4.2.1 Environment and Arm Evolution . . . 16
4.2.2 Base Class and Inheritance . . . 17
4.2.3 Reach, Slide, Push and PickandPlace . . . 17
4.2.4 Frame-Skipping . . . 20
4.3 Reinforcement Learning Classes . . . 20
4.3.1 ReplayBuffer . . . 20
4.3.2 Agents . . . 20
4.3.3 DDQN Model . . . 21
4.3.4 Neural Network . . . 21
4.3.5 Alternative Network Considerations . . . 22
4.3.6 Main . . . 23

5 Experiments **24**
5.1 Early Experiments . . . 24
5.2 Final Model Experiments . . . 25
5.2.1 Reach - Static TO . . . 25
5.2.2 Baseline - Raw Inputs . . . 26
5.2.3 Transfer Learning . . . 28

6 Conclusion **29**
6.1 Summary . . . 29
6.2 Potential Future Works . . . 29

A Code Samples **39**

B Project Timeline - Gantt Chart **41**

C Python Environment **42**

D Original design document **43**

E Final Presentation - Report **66**

Figures

1.1 A 2D Model of a GelSight Sensor (LHS) and its Mujoco implementation (RHS) [2]. The elastomer in the model is both translucent and collision-less to allow an object's depth to be detected by the inner camera. The final implementation appends an opaque cover which mimics the purpose of the reflective paint in the physical model [11]. 2

1.2 A sample render of the final model developed within Mujoco. The GelSight equipped robotic arm sits upon a table alongside a target object which must be moved onto a target position to successfully complete an episode. 3

2.1 A diagrammatic representation of reinforcement learning algorithms. This has been adapted from OpenAI's documentation - not inclusive of all reinforcement learning algorithms [12]. 6

3.1 Outputs of GelSight simulations within Mujoco. The LHS images display a sensor without a cover or tuned height-map, the object is far above the elastomer yet is rendered by the sensor. The middle and RHS images display a sensor in which the outer edge of the elastomer has been covered by a thin opaque layer - here the object must be in direct contact with the elastomer in order to produce any readings. 8

3.2 A depiction of a reinforcement learning model (adapted from Figure 3.1 of Sutton and Barto's An Introduction to Reinforcement Learning [13]). 9

3.3 A diagram describing the high-level interactions between agents for the HDDQN algorithm. The extrinsic agent's reward is cumulative and only delivered at the end of an episode [9, 14]. 12

3.4 HER Learning Map. This diagram illustrates an agent performing HER across a failed episode, here S' is denoted as S_k. The center nodes represent the choices the agent made on its initial run. Whereas, the top and bottom rows are those it chose while performing HER. 14

4.1 Evolution of the Mujoco-GelSight test-bed. Left-to-right, (1) our initial implementation of Gomes' GelSight simulation, (2) the installation of two GelSight onto a UR5 arm and gripper, (3) an initial lifting environment where a target object is placed upon a table, (4) a vision based model which uses a transparent tabletop so as not to occlude the arm and (5) the final model which places the arm on top of the table with the addition of a light source to generate shadows against the table's surface. 16

4.2 Describes a high-level overview as to how an environment is created. Reach, Slide, Push and PickandPlace all inherit directly from the BaseEnv (RobotArm class) which utilizes Mujoco XML models of the overall arena, the Robotic Arm, the gripper and the GelSight sensors. These four XMLs have their assets stored separately as STL files. 17

4.3 The four environments developed for our test-bed. Left-to-right they are Reach, Slide, Push and PickandPlace. The latter three contain a TO which must reach the TP to be successfully completed. Whereas Reach only contains a TP for the gripper to reach. The viewpoint shown is the same as the agent's - in the case of vision. 18

4.4 A Reach environment render (LHS) and a diagrammatic representation of the table within the Slide environment (RHS). The two-dimensional teal geom placed on the table displays the valid spawn locations for both target objects and target positions. This is only necessary when randomizing the placement of objects in each episode. This approach reduces the state space and helps avoid terminal states at episode initialization. Finally, the terminal boundary relates specifically to a terminal state calculation necessary for the Slide environment. 19

4.5 A figure describing the process of an Agent storing its observations into the ReplayBuffer which can then be sampled in batches later by the DDQN_Model. Observations are in the form, [State, Action, Reward, Next State, Done]. 20

4.6 A sample neural network, the structure utilizes all four input types, vision, tactile, kinematics and raw. Vision and tactile types are each passed into a pre-trained model of ResNet50v2 [14, 15, 16, 17]. 21

4.7 This Figure and the accompanying formulas describe the three loss functions Mean Square Error, Mean Absolute Error and Huber loss. Huber's illustration displays the effect three distinct δ's have on the function [18]. 22

4.8 A sample of activation functions commonly used within machine learning. Linear leaves the neuron unchanged whereas ReLU and Leaky ReLU transform the neuron's value when it falls below zero [19, 20]. 23

5.1 Early results from both DDQN (top) and HDDQN (bottom). Here a cylindrical object was statically placed in between the fingers of a blind gripper. The agent's observation space was limited to the two tactile sensors alongside the arm's joint positions and velocity. Results are averaged over 100-episodes, where red indicates rewards received and black indicates the number of steps taken. 24

5.2 Testing on the Reach environment with a non-randomly placed object in combination with real and kinematic based observations. The graphs, left-to-right, use learning rates and target update rates of [1, 50], [0.1, 500] and [0.01, 1000] respectively. The results here are averaged over 100-episodes, where red are the rewards and black is the number of steps taken. 25

5.3 Further testing within the static Reach environment. Left-to-right illustrate results for learning and update rates of [0.001, 1000] and [0.01, 10000]. 25

5.4 A comparison of seven independent runs of the Reach environment using [0.01, 1000]. Five of which converge to an optimal solution. 26

5.5 Baseline reward results of Reach, Slide, Push, and PickandPlace respectively (left to right). The neural network structure takes the form of three fully connected layers each with 256 neurons. The update rate mirrors the best results recorded in Figure-5.2, however, the learning rate was reduced to 0.0001. 26

5.6 Time results from the baseline tests of Reach, Slide, Push, and PickandPlace respectively (left to right). These results are taken from the same set of tests as in Figure-5.5, and therefore use the same hyperparameters. 27

5.7 Higher decaying epsilons for HDDQN. Reduced to 0.986 (LHS) and 0.976 (RHS). 27

5.8 Reward and time results comparing different kernel initializers within the (random target) Reach environment. Results are taken as an average over 100-episodes. . 28

5.9 Reach environment results trained on both vision and raw observation types. LHS uses ResNet50V2 for its pre-trained layers, whereas RHS uses ResNet18. 28

A.1 An image showing code directly from the Reach class's constructor. The four environments, including Reach, can take in a large amount of parameters in order to alter various aspects of its internal workings and outputs. These parameters are immediately passed onto the Base class, RobotArm, which contains the bulk of the environmental methodologies. 39

A.2 An image showing the main file's adjustable parameters. These parameters are separated into subsections of environment, agent, DDQN_model, neural network and misc. Where misc affects numerous sections. 40

B.1 A Gantt chart displaying the progress of the project over time. Policy Gradient-based methods, specifically DDPG was canceled in favor of experimenting with Hierarchical Double Deep Q-Learning. 41

Tables

4.1 Properties distinguishing the four environments from one another. 18

C.1 A listing of the Python Packages, alongside their version, utilized within this project. 42

Chapter 1
Introduction

1.1 Motivation

Robotics and machine learning have traditionally been synonymous with vision and kinematics. Allowing ideas to be pulled from science fiction, such as self-driving cars, into potential near-term reality. However, these observations may not be sufficient in all scenarios and environments. As humans, we use several senses intuitively to interpret and effectively interact with our surroundings. One of these senses is touch [21].

Touch sensors have rapidly developed over the last century, taking many different forms, allowing mimicry of human intuition and, enabling robots to more appropriately interact with their environment. One of these avenues of development has been the GelSight sensor, an optical-tactile-deformation based sensor. Developed in 2009, by Johnson and Adelson [11], the GelSight sensor has shown strong results in the prediction of both immediately apparent properties of an object, such as its shape, but also in more intuitive observations such as tension and slippage [22]. It achieves this through the combination of a high-resolution camera, LEDs, and a paint-coated elastomer. The depth recorded from objects pressed against the elastomer are what enable these intuitive observations. Physical models of these sensors are well documented, including the testing and training of them through the means of deep reinforcement learning, a subset of machine learning [23]. This paper implements a simulated GelSight which aims to alleviate some of the issues of physical models. Although GelSight is still somewhat new, use-cases have been sufficient enough for some iterations to be commercialized [22].

One of the largest challenges robots face is in their ability to adapt to an evolving environment. This evolution takes many forms, both controllable, such as an object's manipulation, and those uncontrollable such as the degradation of a robot's parts over time [24]. Reinforcement learning methodologies make it possible to overcome some of these issues. However, as an effective brute force statistical predictor, they can be extremely sample inefficient resulting in time-consuming modelling and training [25]. In addition to potentially damaging outcomes, be they financial or physical harm [2].

A physical robot's training speed would be limited to the movement of its parts. A means of alleviating this issue would be to initially mirror the model within a simulated environment followed by knowledge transference, of learnt behaviour, to the physical model. This could in turn both speed up training and avoid the previously mentioned damaging outcomes. This transference process has been coined as Sim2Real [26, 27, 28], and although the benefits are somewhat contentious [27], proper tuning could potentially lead to the aforementioned benefits.

As covered by Gomes [2], the main challenge of implementing virtual GelSight-based sensors is in the simulation of the internal view and aligning it to the real model. Though, we managed to avoid this issue, as we are utilizing both his approach and implementation in addition to the project not reaching the Sim2Real phase. Gomes'

approach involves ray casting [29, 30] and although there have since been papers which suggest physics-based rendering methods are superior [31], due to their ability to allow the physics engine to render the lighting - potentially making transference easier across different models. Segregation of lighting from the environment also brings its own benefits, which we cover in Section-3.2.

1.2 Approach

As mentioned above, the formation of our simulated GelSight sensor replicates Gomes' [2]. This utilizes a depth array, taken from a camera, to capture any object's contact with our simulated gel. Several formulas are then applied in order to better approximate the deformation and lighting model of a real sensor. These include Gaussian filtering and Phong's illumination model respectively [2]. Whereas Gomes' implementation is found within ROS's Gazebo [32], ours extends the functionality of Mujoco [1], a physics engine implemented ontop of OpenAI's Gym [5]. Specifically, we utilized the python wrapper *mujoco_py* [33].

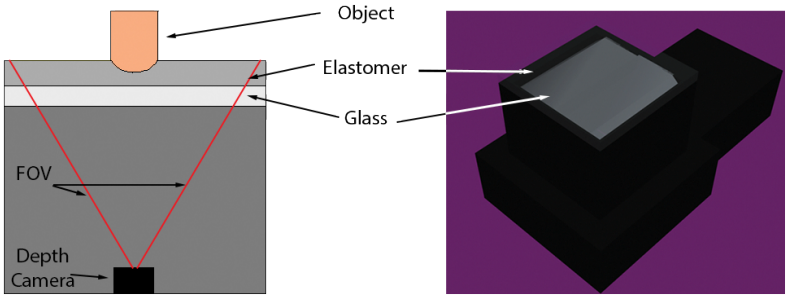


Figure 1.1: A 2D Model of a GelSight Sensor (LHS) and its Mujoco implementation (RHS) [2]. The elastomer in the model is both translucent and collision-less to allow an object's depth to be detected by the inner camera. The final implementation appends an opaque cover which mimics the purpose of the reflective paint in the physical model [11].

Following this, we appended two of our GelSight to a 2F-85 ROBOTIQ gripper [4] and UR5 arm [3] combination. After calibration, we then instantiated an agent to be trained through reinforcement learning algorithms. Our choice in these constituted two evolutions of Q-Learning - derivatives of Richard Bellman's famous equations [13]. Our initial algorithm took the form of Double Deep Q-Learning (DDQN) [6], which is a progression of Deep Q-Learning (DQN) [7]. DQN uses a neural network, as a function approximator, to choose actions - rather than an explicit table of Q-values. Allowing it to operate in environments with larger state spaces. Furthermore, DDQN uses two neural networks, separating target and evaluation, helping it to avoid the maximization bias DQN suffers from - avoiding falling into local optimal solutions [6, 8].

Our second approach is that of Hierarchical Double Deep Q-Learning (HDDQN) [9]. This algorithm is a further evolution of DDQN, employing two agents - nesting one inside the other. These agents take the form of extrinsic and intrinsic, where the former observes the environment and chooses goals and the latter has its own set of observations and uses these to attempt to complete the extrinsic agent's goal. This effectively offers further segmentation and the chaining of actions, similar to Markov Decision Chains (MDP) [34], which aims to enable better performance within sparse

reward environments [9, 14].

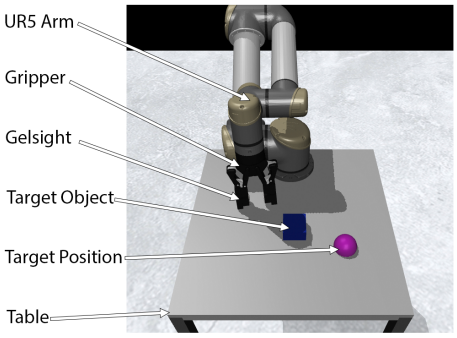


Figure 1.2: A sample render of the final model developed within Mujoco. The GelSight equipped robotic arm sits upon a table alongside a target object which must be moved onto a target position to successfully complete an episode.

One final note is around generalization. The effectiveness of which, on more complex modern neural networks, has recently come into question in Zhang's paper [35]. This is outside the scope of this project and therefore we will sidestep the issue and assume traditional understandings around generalization, inherited from the Perceptron algorithm, still hold [35, 36].

1.3 Outcome

To summarise, in this paper we discuss and demonstrate a successfully modeled Mujoco-based test-bed. In which four distinct environments, Reach, Push, Slide, and Pickand-Place can be trained upon using the reinforcement learning algorithms DDQN and HDDQN. To that end, we demonstrate baseline tests using only the end effector of the gripper as our input. Although the test-bed is capable of more complex observations, due to the limited time allotted for the project, experiments surrounding these have not been fully realized.

Chapter 2
Related Works

2.1 Tactile Sensors

As mentioned by Gomes [2], tactile sensors have seen many different implementations. These types include optical [11], piezo-electric [37], resistive [38], capacitive [39], magnetic [40, 41] and strain gauge [42]. Each type has different methods of producing a given sensor and each sensor has its own associated advantages and disadvantages. However, camera-based tactile sensors offer several distinct advantages over others. For example, capacitive and resistive sensors offer only low-resolution outputs in addition to requiring a great deal of wiring. This is in direct contrast to optical-tactile solutions, which offer not only a high-resolution output but also may only need a single wire [2, 43].

As mentioned previously, GelSight is one of these optical-tactile solutions. Implementations of this sensor can be found here [11, 29, 44, 45, 46], and have shown strong results in not only correctly approximating the shape of objects but also in identifying more intuitive properties, such as hardness and slippage, when combined with machine learning techniques. A natural progression within the research space, has been to integrate multiple GelSight sensors onto a single robotic gripper, such as in Wilson's paper [47]. The benefit of which is in the provision of additional environmental information, enabling sensing around the shape of an object, rather than on a singularly flat plane.

The most prominent weaknesses of GelSight are in its bulky design and small sensor area. This is something GelSlim, an iteration on the GelSight sensor, aimed to address. It offers a larger sensor area and is slimmer than its parent, attributed to the adoption of a wedge shape in lieu of a cube [48]. However, a flat sensor is still somewhat limited in its capabilities as its geometry makes it challenging to read some objects, such as large flat surfaces [43]. Furthermore, it would be useful when constructing robotic hands if sensors could surround a finger or even the entire hand as this would provide an even higher awareness of a given environment. Romero's paper [49] explores this topic through applying gel caps to the fingertips of a robotic hand. Adelson demonstrated some of these advantages in his keynote [43], where the aforementioned hand is able to gently unscrew a loose jam jar without disturbing the main body.

Gomes' GelTip is a further expansion on this idea [50]. GelTip differentiates itself by creating a sensor area about an entire finger. The revised shape and compactness enable higher levels of surface contact against objects, providing further environmental insight to previous methods. Additionally, this slim finger-like shape would also enable the sensor to reach and interact with spaces a traditional GelSight sensor could not.

GelSight influenced optical-tactile sensors have not been the only avenue of research. TacTip is a closely related implementation, again including both a camera and LED(s). However, TacTip uses a modular three-dimensionally printed tip for contact, rather than a gel [51, 52]. Though similar to GelSight, TacTip has seen its own evolution.

Culminating in the development of the TacTip-GR2, TacTip-M2, and TacCylinder sensors [53].

We limited the complexity of our paper by only implementing Gomes' GelSight. Although there are already several papers simulating tactile sensors in combination with grippers [54], none use Mujoco in combination with a fully-realized GelSight sensor. As such, this constitutes our unique contribution to the research space.

2.2 Reinforcement Learning

Reinforcement Learning (RL) encompasses a rich collection of algorithms which aim to train an agent on how to best interact with its environment. These techniques utilize a combination of observations obtained from an environment in order to establish a policy which can determine the best actions for the agent to take. This policy typically aims to maximise the rewards returned across a given episode [13].

RL types include both model-based and model-free learning. In the former, an agent is either provided a model, such as in Alphazero's case [55], or the agent attempts to learn the model - such as in Imagine-augmented agents (I2A) [56]. Similarly, model-free learning also splits into two distinct categories, Policy Optimization and Q-Learning. Policy Optimization (PO) methods include algorithms such as Actor-Critic (A2C and A3C) [57, 58]. Whereas, Q-Learning's scope encompasses algorithms such as Mnih's Deep Q-Learning (DQN) [7].

Q-Learning and PO based methods can be distinguished by their approaches to learning a policy. PO learns a policy directly, whereas Q-Learning will instead learn an action-value function - effectively inferring an optimal policy. This results in Q-Learning restricting the actions to only a discrete space, rather than the continuous action space of PO. This additional flexibility allows policy gradient methods to potentially apply to more problems than DQN-based methods [13].

As mentioned in Section-1.2, this paper focuses on the implementation of two DQN-based algorithms - DDQN and HDDQN. These deep RL models require neural networks to function. One popular network structure is the Multilayer Perceptron (MLP). MLP's have been adapted to form solutions for a myriad of problems, from Flood susceptibility mapping [59] to ammonium adsorption on zeolite (a mineral typically used as an adsorbent within the commercial space) [60].

A more closely related example can be found in Chiddarwar's paper [61], where an MLP is used to train a 6R serial robot. Here the network inputs are the arm's position, orientation and the end effector - the x, y and z coordinates of the gripper. Similarly, James' paper [62] covers training a robotic arm through DQN - again only using numeric inputs. Although these papers were a useful starting point, our observation spaces constitute not only numeric outputs, but also optical observations from both our real and GelSight sensors. Therefore, two-dimensional convolutional layers are necessary in order to extract features from the pixel arrays. An example can be found in Quillen's paper [63], which uses Deep Reinforcement Learning to train a vision-based grasping robot.

Further insight can be gained through the examination of agents training on games. In both Lample's and Hausknecht's papers [64, 65], convolutional layers of a neural network play a crucial part in processing game images. These layers operate toward feature extraction. These features allow an agent to understand how to distinguish

between state spaces and subsequently how to choose actions which will achieve the highest rewards.

As our project required two separate input types, both images and numerical, they needed to be processed separately. After which they are combined into a single fully connected layer. Inspiration for this approach was taken from papers by Gomes and Wang [29, 66].

DQN is powerful, however, it has been found to exhibit several weaknesses. These weaknesses have been overcome or mitigated through expansions. These include Double Deep Q-learning (DDQN) [6, 67], Prioritized Replay [68], Dueling Neural Networks [69], Multi-step Learning [70], Distributed RL [71] and Noisy Nets [72]. Each of these overcome some of the algorithm's flaws and do not need to be applied exclusively and can be implemented together. This was demonstrated in Hessel's paper [8], and has been coined as Rainbow DQN, where the combination is shown to perform far better than any individually applied expansion.

Finally, there are hybrid algorithms which inherit aspects from both PO and Q-learning, one of these is Lillicrap's Deep Deterministic Policy Gradient (DDPG) [73], an evolution of Silver's Deterministic Policy Gradient (DPG) [74]. DPG is an effective expansion to DQN and allows it to operate in continuous action spaces. DDPG further extends this by using two separate policies, for exploration and updates [73, 74]. However, DDPG suffers from function approximation errors which occur in Actor-Critic methods. Fujimoto addresses this, in his paper [75], taking further inspiration from Double Q-learning through the selection of the minimum calculated value from a pair of Critics rather than a single one. This is coined as Twin Delayed Deep Deterministic Policy (TD3).

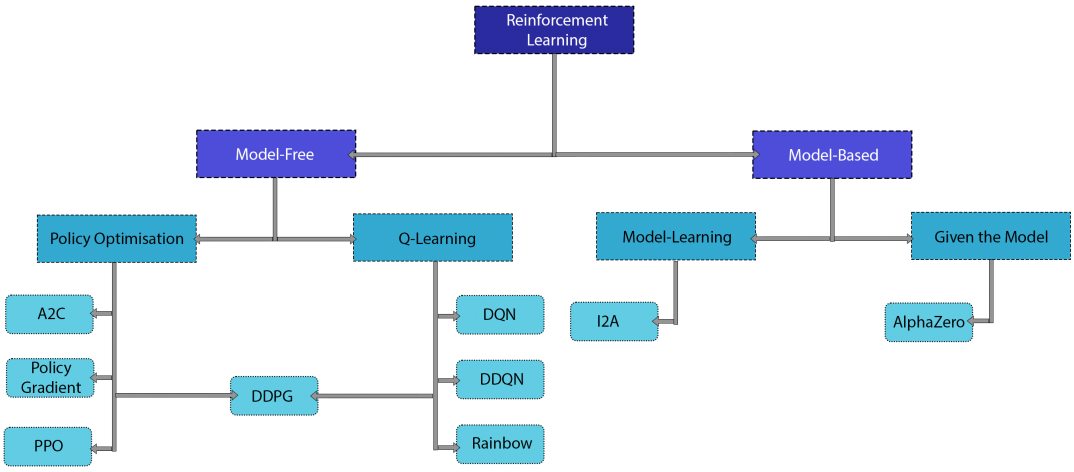


Figure 2.1: A diagrammatic representation of reinforcement learning algorithms. This has been adapted from OpenAI's documentation - not inclusive of all reinforcement learning algorithms [12].

DDPG was originally proposed for this project alongside DDQN, however, another consideration for our environment is in the sparseness of the rewards. If we set rewards in relation to the grasping, or displacement, of an object unto a specific point there are many actions a robot could take which result in no reward whatsoever. Hierarchical Deep Q-Learning (HDQN) overcomes some of these, as shown in Kulkarni's paper [9]. As a result of this, HDDQN [76] was implemented instead of DDPG.

2.3 Transfer Learning

As the problems we attempt to solve with Deep RL grow in complexity so must the complexity of the neural networks. Otherwise, their expressive power may not be sufficient [77, 78]. This complexity can result in lengthy training time, requiring days, weeks, or months. Transfer learning is one means of combating this. By loading a set of weights from a pre-trained neural network, trained on a similar problem, as a starting point [79].

First proposed in Bozinovski's and Fulgosi's paper in 1976 [80], this theory eventually evolved into a practical means of image recognition transfer learning. Krizhevsky's [81] describes using a five-layer-deep-convolutional neural network to classify items from ImageNet, a hierarchical image database [82]. Many different models have taken shape since, including Simonyan's and Zisserman's VGG [83] which demonstrated that adding further depth, to between 16 and 19 layers, could improve performance. Following this, Szegedy's inception [84] aimed to further optimise some of the computational inefficiencies within previous network structures. And Howard's MobileNets [85] again sought to further optimise networks through the introduction of a trade-off hyper-parameter - between the latency and accuracy of a network. Some of these pre-trained models are large, for instance ResNet can use up to 152 layers [14].

As our test-bed can contain up to three image-based observations, consisting of both a real observation and two GelSight sensors, transfer learning within the space of ImageNet is of particular interest to this project. Therefore, we utilize some of these pre-trained models in our neural network to assist in training speed. The specific models implemented are covered in Section-4.3.4.

2.4 Sim2Real

Transferring simulated learning to real models is a trend that is fast becoming a standard. Adopted by many researchers with the aim of reducing time and cost [86, 87, 88]. It not only provides a relatively inexpensive modeling environment but, also enables the rapid collection of training and testing data for a given agent [89].

One of the main challenges within this space is in generalization. The learning acquired in a virtual environment may allow an agent to perform well on synthetic data but that experience may not transfer well to a real agent [27, 86]. However, sometimes further optimization of stimulation parameters can have a dramatic impact, altering the transferable performance from low to high [27].

Although this paper remained focused solely on the creation and training of a model within a virtual space. It is difficult to ignore Sim2Real learning. As it provides insight around the motivations of the paper, in addition to theoretical next steps. As we are not performing sim2real we can ignore various other potential issues, such as accounting for wear, tear, and general degradation of the robot's components.

Chapter 3
Design

3.1 Original and Updated Designs

The originally submitted design document and report, which accompanied the final presentation, can be found in Appendix-D and Appendix-E respectively. The latter covers some of the changes in scope which the project took after several weeks, most importantly, it notes a change in the proposed algorithms implemented - swapping DDPG [73] to HDDQN [9].

3.2 GelSight

The implementation of a GelSight sensor within Mujoco was the main motivation for this project. As mentioned previously, our simulated approach mirrors that of Gomes'. In fact, the GelSight driver we utilized is directly from Gomes' ROS Gazebo implementation [2, 90]. This uses a depth camera, a blank image from a real GelSight sensor, and several formulas in order to approximate what should be output. These formulas are calculations for a height map, Gaussian filters, and Phong's illumination model [91]. This implementation can be seen in both Gomes' paper and Appendix-D [2].

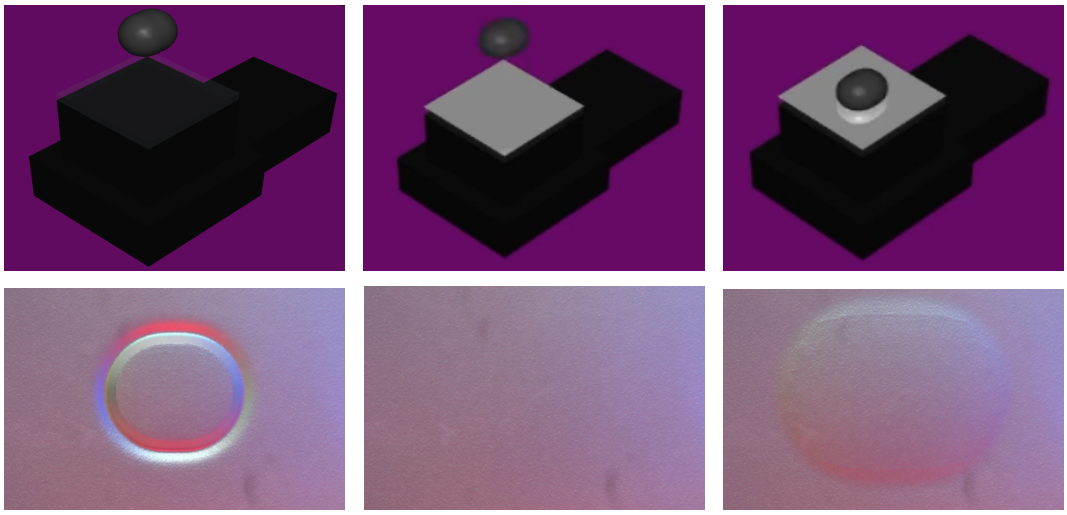


Figure 3.1: Outputs of GelSight simulations within Mujoco. The LHS images display a sensor without a cover or tuned height-map, the object is far above the elastomer yet is rendered by the sensor. The middle and RHS images display a sensor in which the outer edge of the elastomer has been covered by a thin opaque layer - here the object must be in direct contact with the elastomer in order to produce any readings.

The distinguishing feature between Gomes' implementation and the one found within

this paper, other than the physics engine utilized, is in the addition of a thin opaque layer to the elastomer's outer limit. This approach removes the need to manually calibrate a height map as this limits the internal camera's view. The approach takes inspiration directly from models of real GelSight sensors - mirroring the properties of a paint coating from a physical model [11].

3.3 Deep Reinforcement Learning

3.3.1 Preliminaries

Prior to describing our implementations of DDQN and HDDQN it makes sense to take a step back and provide a baseline understanding of the algorithms which led to their respective inceptions. A natural starting point would be Figure-3.2, which describes the recursive structure of modern RL algorithms. This is effectively learning through trial and error. Where an agent takes actions based on observations (states) and rewards provided from its environment. Over time, the agent aims to learn which actions in which states provide the best rewards [13].

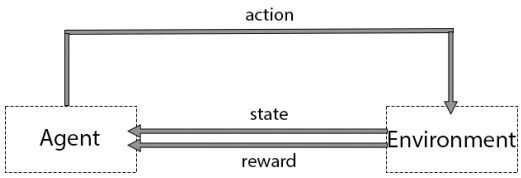


Figure 3.2: A depiction of a reinforcement learning model (adapted from Figure 3.1 of Sutton and Barto's An Introduction to Reinforcement Learning [13]).

Typically an agent can only observe part of an environment and therefore would not be able to construct a model which fully describes a solution without interaction. Hence, the need for reinforcement learning. A reasonably simple example of this would be Markov Decision Processes (MDPs), which combine states, actions, probabilities, rewards and a discount factor - (S, A, P, R, γ). From these variables, Bellman was able to construct a formula which defines a total discounted reward (G) at a particular time-step (t) [13, 34].

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \tag{3.1}$$

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \qquad [13] \tag{3.2}$$

Policies, Functions and Q-Learning

This trial and error structure is tantamount to learning, or approximating, a policy. Here state-action pairs are used to inform an agent's action in any given state. A formulaic description of this can be seen below in (3.3), here π, a and s denote the policy, action and state respectively at time-step, t [13].

$$\pi(a|s) = [A_t = a|S_t = s] \ [13] \tag{3.3}$$

Not all policies are equal, therefore in order to establish which policy is more optimal, it is necessary to evaluate both a state's value $v_\pi(s)$ and an action's value $q_\pi(s, a)$. The

two formulas below describe this, where $\mathbb{E}$ defines a random variable's value - given the agent utilizes the policy (π) on this particular time-step [13].

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{3.4}$$
$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \qquad [13] \tag{3.5}$$

In order to solve a particular task, the policy defined by the agent should aim to extract the highest possible reward. These optimal policies are defined by similar state-value and action-value functions. The formulas (3.6) and (3.7) are the famous Bellman optimality equations for v_* and q_*, which act as the foundation for modern RL [13].

$$v_*(s) = max_\pi v_\pi(s) \tag{3.6}$$
$$q_*(s,a) = max_\pi q_\pi(s,a) \qquad [13] \tag{3.7}$$

Effectively, the value and action states would return the best action if acting under the optimal policy [13].

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \tag{3.8}$$
$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \qquad [13] \tag{3.9}$$

From these formulae evolved Q-Learning, which is a subset of off-policy Temporal Difference (TD) learning algorithms. TD methods learn directly from experience and do not need a model of the environment. This experiential learning is directed through predictions, which are used as targets throughout learning. This is known as bootstrapping [13].

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(y_t - \hat{y}_t) \tag{3.10}$$
$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)) \qquad [13] \tag{3.11}$$

Here, Q-learning only bootstraps between two steps - the current and next states. As this is off-policy, q_* is directly approximated from the action-value function Q, which is independent of the policy followed throughout training. This simplifies the algorithm and allows early convergence [13]. Note, α, y_t and $\hat{y}_t$ represent an arbitrarily chosen learning rate between [0, 1], the actual label and the predicted label respectively.

Deep Q-Learning

The above methodologies work for simple state spaces. However, they fail in those which are larger and more complex. Especially those which are continuous, such as in our random object placement variation of the robotic arm environments. To overcome this DQN was developed, by Minh [7], as an evolution of Q-learning. This uses experience replay to store the agent's interactions with the environment and uses batches of these to update a neural network - aiming at convergence to a global optimal solution over time. The neural network acts as a function approximator, which in turn enables our agent to generalise across the state space [6, 7]. The weights of this network are typically instantiated randomly. Though, pre-trained models or kernel initializers could also be used.

$$Q(s_t, a_t; \theta) \approx Q^*(s,a) \ [13] \tag{3.12}$$

Training takes several key steps. First, initialize the ReplayBuffer (D), agent and environment. Second, for every episode, reset the environment and record the initial state. Use this state to predict an action based on epsilon greedy - covered in Algorithm-2. Use this action to perform an environment *step* and record the state return as the

next state (s'). Store the transition information as (s, a, r, s', d) to D - d is a boolean observation as to whether a terminal state has been reached. Take a batch sample of the previously stored observations from D and use this to calculate the target update in accordance with the formula below followed by gradient descent [6].

$$y_t = \begin{cases} R_{t+1}, & \text{if terminal} \\ R_{t+1} + \gamma maxQ(S_{t+1}, a; \theta_t^-), & \text{otherwise [6]} \end{cases} \tag{3.13}$$

The current state is then set to the value of the next state and this is repeated until the end of the episode is reached. This is then repeated across the number of desired episodes.

3.3.2 Double Deep Q-Learning

As mentioned in Section-2.2, DQN is not without its own limitations. As DQN is producing predictions on the same network it is updating at each time-step, it is susceptible to maximization bias. DDQN overcomes this issue by utilizing two networks. A target network and an evaluation network - inspired by Double Q-Learning [67]. Although separating the tasks across two networks means that most predictions are somewhat outdated, they will be more consistent and allow the agent to explore further before committing to any single path.

$$y_t \equiv R_{t+1} + \gamma Q(S_{t+1}, argmaxQ(S_{t+1}, a; \theta_t), \theta_t^-) \text{ [6]} \tag{3.14}$$

The above formula defines the target value update rule for DDQN. Here y, S, a, R, γ, θ, θ^- represent the update, state, action, reward, epsilon decay, online network weights, and target network weights at time t, where applicable. This formula enables an agent to learn at each time step, once there are enough samples within the ReplayBuffer.

Algorithm 1 – Double Deep Q-Learning [9]

```
Initialise Agent (k)
Initialise Target and Evaluation networks (θ)
Initialise ReplayBuffer (D)
Initialise Environment (env)
for each episode do
    s_t = env.Reset()
    while not done do
        a = EpsilonGreedy(s_t)
        s_{t+1}, a, r_t, d = env.step(a)
        D.storeTransition(s_t, a, r_t, s_{t+1})
        k.learn()
        s_t = s_{t+1}
    end while
end for
```

Algorithm 2 – **EpsilonGreedy(state)** [6]

```
if random() < ϵ then
    return random_action
else
    return max(network.predict(s_t))
end if
```

Algorithm 3 – learn() [6]

```
if ReplayBuffer's size > b then
    Take a batch sample from the ReplayBuffer
    Calculate target Q-value:
        y_t ≡ R_{t+1} + γQ(S_{t+1}, argmaxQ(S_{t+1}, a; θ_t), θ_t^-) [6]
    if target update then
        Update the target network:
            θ_{t+1} = θ_t + α(y_t^Q − Q(S_t, A_t; θ_t))∇_{θ_t} Q(S_t, A_T; θ_t) [6]
    end if
end if
```

3.3.3 Hierarchical Double Deep Q-Learning

HDDQN is an effective extension of DDQN and requires nesting a DDQN agent within another DDQN agent. This outer agent is referred to as extrinsic and the inner agent is referred to as intrinsic. Both agents act in a similar fashion to the standard DDQN model, taking observations from the environment and feeding them through a neural network to make predictions. However, there are several key distinctions. Firstly, the extrinsic agent now chooses goals rather than actions, these goals are passed to the intrinsic agent who combines them with its own observation state. The intrinsic agent's observations may differ from those of the extrinsic agent though, they do not necessarily need to. The intrinsic agent uses this combined state, to choose actions, in an effort to complete the goal set by the extrinsic agent [9].

Additionally, the overall reward structure is quite different. The extrinsic agent receives the typical reward values from the environment but, only cumulatively at the end of each episode or when the goal was completed successfully by the intrinsic agent. The intrinsic agent's rewards are slightly more abstract, typically boolean, receiving only one reward for fails and one reward for successes. These are typically not associated directly with the environment, but with the completion of the extrinsic agent's goal [9].

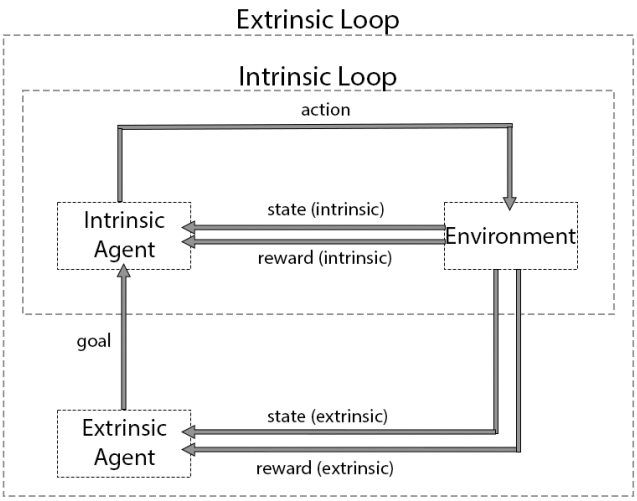


Figure 3.3: A diagram describing the high-level interactions between agents for the HDDQN algorithm. The extrinsic agent's reward is cumulative and only delivered at the end of an episode [9, 14].

The neural networks of both agents are separate and can vary in structure but do

not necessarily need to. One example might be for the extrinsic agent's network to be fed pixel data from a camera that can observe both the target object and the arm. This information can be used to predict which sub-goal would move the arm closer to the terminal state, or at least which would provide the best reward. By contrast, the intrinsic agent's neural network structure may contain only joint-related inputs, from the robotic arm, and use these to understand how to complete the sub-goal proposed by the extrinsic agent [92]. The fundamental idea is similar to that of a recipe. Where the intrinsic agent is performing all the low-level tasks and the extrinsic agent decides which low-level task the intrinsic agent should be attempting to solve [9, 93, 94].

One of the main influences for this meta-controller structure appears to have been taken from Markov Decision Processes (MDP). Whereby, the available action probabilities are reliant entirely on the current state space. Using these, chains of actions based on state spaces can be created. Using this structure, the model should ideally be able to determine its own goals, based on the rewards provided, and formulate both single and multi-action steps to achieve those goals [9, 94].

Algorithm 4 – Hierarchical Double Deep Q-Learning [9]

```
Initialise Agents, (k_e, k_i)
Initialise Target and Evaluation networks (θ_1, θ_2)
Initialise ReplayBuffers (D_e, D_i)
Initialise Environment (env)
for each episode do
    s_t = env.Reset()
    g = EpsilonGreedy(s_t)
    while not d do
        s_i = s_t
        r_e = 0
        while not d and g not reached do
            a = EpsilonGreedy(state, goal)
            s_{t+1}, a, r_t, d = env.step(a)
            r_e = r_e + r_t
            g, reached = intrinsicReward()
            D_i.storeTransition([s_t, g], a, r_t, [s_{t+1}, g])
            k_e.learn()
            k_i.learn()
            s_t = s_{t+1}
        end while
        D_e.storeTransition(s_i, g, r_e, s_t)
        if not done then
            k_i.decayEpsilon()
            g = EpsilonGreedy(s_t)
        end if
        k_e.decayEpsilon()
    end while
end for
```

3.3.4 Potential Alternative Methodologies

Prioritized Replay Experience

Prioritized Experience Replay (PER) is an extension of DQN. The idea of this extension is to recognize that not all memories hold equal significance. In context, significance

here denotes unknown knowledge. Thus the memory batches fed to an agent, when learning, should be geared toward higher significance rather being chosen completely randomly. An effective measure of this significance would be the Temporal Difference (TD) error - the amount of surprise experienced by the agent in any given transition. The original paper, by Schaul [68], proposed two forms. Either through rank-based prioritization or proportional prioritization. The probability sampling of these two methods is as follows:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_i^\alpha} \tag{3.15}$$

Here, $p_i = |TD| + \epsilon > 0$ denotes the priority transition for i and α defines what level of prioritisation is used [68].

$$P(i) = \frac{1}{rank(i)} \tag{3.16}$$

Within the rank-based solution, rank(i), instead determines the ordering of the replay memories. However, the formation of the experiences produced within the algorithm introduces certain biases which could affect overall convergence. Therefore, this bias is corrected through Importance Sampling [68].

$$w_i = (\frac{1}{N}\frac{1}{P(i)})^\beta \tag{3.17}$$

Here, N and β denote the number of states and a hyper-parameter respectively - it is expected that β is decayed during training. Schaul argues that unbiased updates become more important as training progresses [68].

Hindsight Experience Replay

Hindsight Experience Replay (HER) is another extension of DQN [95]. HER is a method which deals with sparse reward environments - similar to HDDQN. Assuming the agent fails to reach the terminal state and instead achieves some goal S', the implementation of this extension sees the agent replay this episode. In this replay, the agent samples different actions than the ones it took within each state. Comparing outcomes to S' with the assumption that S' was our agent's goal all along. This would effectively enable our agent to learn more from failures.

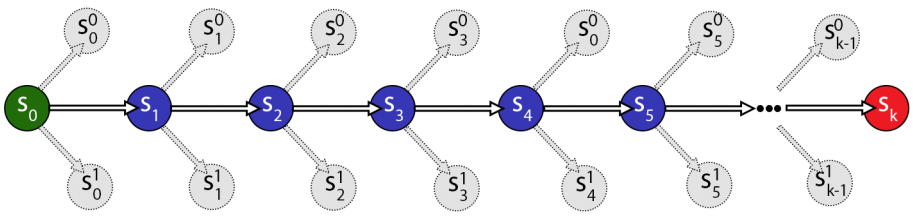


Figure 3.4: HER Learning Map. This diagram illustrates an agent performing HER across a failed episode, here S' is denoted as S_k. The center nodes represent the choices the agent made on its initial run. Whereas, the top and bottom rows are those it chose while performing HER.

Deep Deterministic Policy Gradient

DDPG was originally considered for this project before being exchanged for HDDQN. This algorithm would have dealt with one of the limitations of Q-learning, which

is its inability to deal with continuous action spaces. DDPG is a hybrid of both Policy Optimisation and Q-Learning methods. Similar to HDDQN, DPPG requires also requires four networks, however, this time for an actor and critic instead of nested agents. The update rule for the actor-network alongside the target update are as follows [73]:

$$\nabla_{\theta\mu} J \approx E_{s_t \sim \rho^\beta}[\nabla_{\theta\mu} Q(s, a|\theta^Q)_{s=s_t, a=\mu(s_t|\theta^\mu)}] \tag{3.18}$$

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \qquad [73] \tag{3.19}$$

$\nabla_{\theta\mu} J$ and θ^μ denote the gradient of the cost function J for the actor-network and critic-network's parameters respectively. DDPG randomly samples the agent's memory and uses the actor-network to determine which actions should be performed in those states. These actions are then passed to the critic in order to update based on the gradient with respect to the actor-network [73]. This is covered in more detail within both Lillicrap's paper [73] and Section-3.3 of Appendix-D.

Chapter 4
Implementation

This project comprises of five main classes, Environment, Agent, DDQN_Model, ReplayBuffer and Neural Network. Additionally, there exists a main file which holds a Handler class for user interaction as well as quick access to some headline hyperparameters and features.

4.1 Acknowledgment of Key Software

Prior to diving into the implementation, it makes sense to throw light on some of the software utilized. This project relies on numerous externally developed components, some of which have been mentioned previously. We are using a physics engine extension to OpenAI's Gym [5] named, Mujoco [1]. Specifically, the Python wrapper mujoco_py [33] controls most of the base environmental features and an extension dm_control, developed by DeepMind [96], has been used to construct dictionaries of joints to be referenced throughout our code. These libraries function through Cython [97] and OpenGL [98], the former is used to interface with the original C-based code and the latter enables 3d-rendering of an environment. Finally, for the construction of our neural networks, we have utilised Tensorflow [99] and Keras [100]. A full breakdown of the python environment can be found in Appendix-C.

4.2 Environment

4.2.1 Environment and Arm Evolution

As covered more explicitly in Appendix-E, the test-bed has seen numerous iterations throughout development. Figure-4.1 illustrates this development, from the initial implementation of a GelSight sensor through to the final environment where a fully realized gripper and arm are placed on a table and are given objectives.

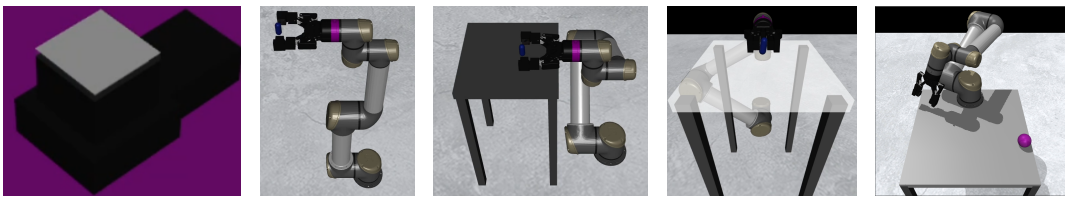


Figure 4.1: Evolution of the Mujoco-GelSight test-bed. Left-to-right, (1) our initial implementation of Gomes' GelSight simulation, (2) the installation of two GelSight onto a UR5 arm and gripper, (3) an initial lifting environment where a target object is placed upon a table, (4) a vision based model which uses a transparent tabletop so as not to occlude the arm and (5) the final model which places the arm on top of the table with the addition of a light source to generate shadows against the table's surface.

4.2.2 Base Class and Inheritance

Following the finalization of the initial environment model, a base environment class named RobotArm was created. This class holds the methods for processing state, action, reward, and terminal outputs. Though this is not intended to be called directly, the user should instead call one of the four distinct child environments - which mirror OpenAI's Gym FetchEnvs. These environments take the form of Reach, Slide, Push and PickandPlace [5]. Each environment's constructor harbors a large number of parameters which can be altered to tune both the inner workings of the environment and its outputs. A high-level overview of this structure as well as an example snippet of the code can be found in Figure-4.2 and Figure-A.1 respectively.

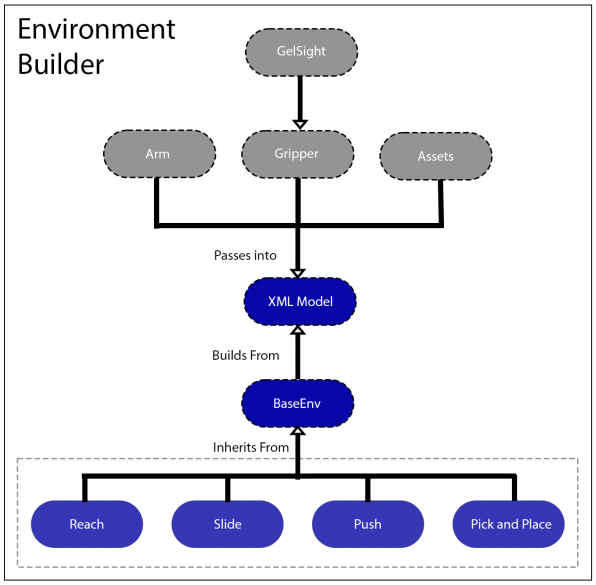


Figure 4.2: Describes a high-level overview as to how an environment is created. Reach, Slide, Push and PickandPlace all inherit directly from the BaseEnv (RobotArm class) which utilizes Mujoco XML models of the overall arena, the Robotic Arm, the gripper and the GelSight sensors. These four XMLs have their assets stored separately as STL files.

4.2.3 Reach, Slide, Push and PickandPlace

Each environment is distinct from one another, which constitutes varying levels of complexity. The terminal state, or goal, for each environment can be generalized as the Target Position (TP) being reached. In Reach's case, this simply requires the center of the gripper to reach the TP. However, in the case of Push, Slide and PickandPlace, the agent must manipulate a Target Object (TO) to the TP. These four environments can be seen in Figure-4.3.

The latter three environments further distinguish themselves through joint restrictions to the TO. Slide sees the TO's z-axis completely removed, this effectively disables contact between the object and table - enabling it to slide. Push sees the z-axis included but restricted, this ensures there is friction between the TO and table while making the TO unliftable. Finally, PickandPlace provides the TO with a free-joint, allowing a full six-degrees-of-freedom. A summary of these differences can be seen in Table-4.1

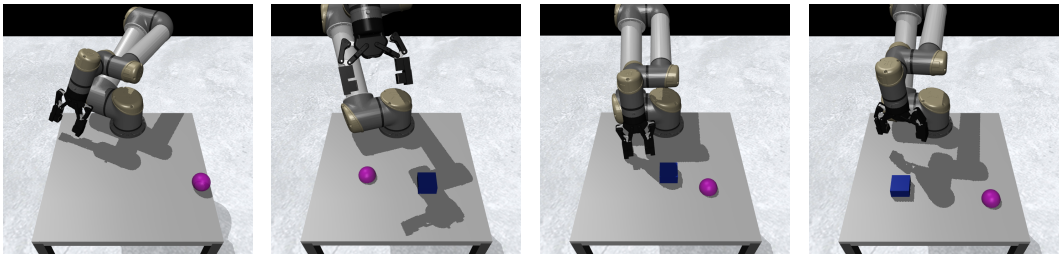


Figure 4.3: The four environments developed for our test-bed. Left-to-right they are Reach, Slide, Push and PickandPlace. The latter three contain a TO which must reach the TP to be successfully completed. Whereas Reach only contains a TP for the gripper to reach. The viewpoint shown is the same as the agent's - in the case of vision.

Name	Target Object (TO)	Target Position (TP)	Restrictions
Reach	No	Yes	None
Slide	Yes	Yes	TO z-axis removed
Push	Yes	Yes	TO z-axis restricted
PickandPlace	Yes	Yes	None

Table 4.1: Properties distinguishing the four environments from one another.

Returns

Similar to Gym, our environments return four items, state, reward, done and info. Effectively, these act as an agent's insight into their environment, enabling the agent to choose actions from an informed position.

States can take several forms, real, tactile, kinematics and raw. Real is a vision based observation, returning a NumPy array of pixel values from the same perspective we see in Figure-4.3. Tactile also returns arrays of pixel outputs but, instead, from the two GelSight sensors. Kinematics return numeric arm and joint-related values, including the position of the arm's end effector, this implementation leans heavily on Gym's [5]. Finally, raw returns only the x, y and z coordinate of the arm's end effector. The test-bed offers flexibility in which of these can be returned, allowing for both single-type and multi-type observations.

The reward function evolved substantially over the course of the project. Initially taking a static form, where a default value of -10 would be offset dependent on the completion of certain criteria. Checking whether the elastomers of the gripper were in contact with the TO and if the TO had been lifted above its initial z-axis. A further breakdown of this can be found in Section-2.4 of Appendix-E.

However, the reward model was found to be lacking and was refined to match Gym's implementation. Rewards now return the normalized Euclidean distance between either the TP and TO or TP and the gripper's end effector - dependent on the environment. As this value is now a magnitude of distance, our agent is able to better understand to what degree of correctness its action had.

$$distance(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \qquad [5,\ 101] \qquad (4.1)$$

Done, is a boolean representation of the overall environment's state, indicating if the episode has ended. The terminal state is triggered if one of three conditions are met, (1) if the goal of the episode is reached, (2) if the TO falls onto the floor, (3) if one-hundred actionable steps have been taken across the episode. The first entry ends an episode successfully, while the second and third entries result in failure. The time-step limit has been chosen somewhat arbitrarily, as it needs to allow enough time for the agent to reach its goal in any given randomized placement while also ensuring an agent is not indefinitely stuck in an unwinnable environment [102].

Info provides additional information as to what is occurring within the environment. Some notable inclusions are the actuation forces currently applied to the robotic arm's joints and the number of steps taken in the environment. Previously, this also counted the number of time-steps the gripper was in contact with the TO. Though, this was removed when the reward function was redefined.

Randomization and Terminal Boundaries

The complexity of each environment can be modified through the choice of either static or randomly placed TOs and TPs. It is important, when randomizing these placements, that they do not initialize in either a terminal state or one which is impossible to complete. To solve this, a thin layer name *spawn-surface* was added to the top of the table, whose dimensions are used to ensure random placements fall within that space. This can be seen as the teal layer in Figure-4.4.

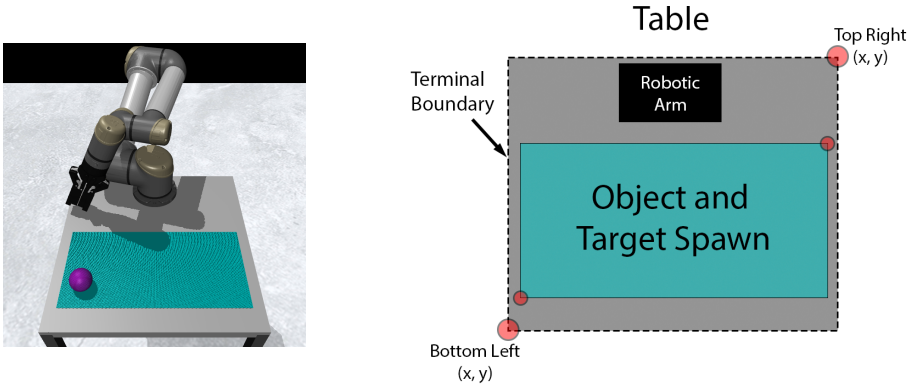


Figure 4.4: A Reach environment render (LHS) and a diagrammatic representation of the table within the Slide environment (RHS). The two-dimensional teal geom placed on the table displays the valid spawn locations for both target objects and target positions. This is only necessary when randomizing the placement of objects in each episode. This approach reduces the state space and helps avoid terminal states at episode initialization. Finally, the terminal boundary relates specifically to a terminal state calculation necessary for the Slide environment.

One additional consideration is in regards to Slide's terminal state. As covered above, one of the failure states occurs if the TO falls to the ground. However, as Slide's TO lacks a z-axis this will never occur - the TO will float if pushed over the table's edge. Therefore, an additional calculation in the same vein as the object's spawning was added. At each time-step, within Slide, the environment compares the TO's position to the table's bottom-left and top-right limits, ensuring the episode ends if these limits are exceeded.

4.2.4 Frame-Skipping

An agent needs to be given adequate time to both fully explore and complete its environment. This is especially crucial for this project, where multiple discrete actions, across joints, may be needed to simply move the agent in a given direction. However, there is pressure to limit this time, per episode, as RL algorithms typically need a great number of samples to converge. One method of alleviating this is in frame-skipping, increasing the number of frames between actionable steps. Effectively this could allow an agent's action to be fully realised before performing another. Bryalan's paper [103] demonstrates improved performance when utilizing frame-skipping in some Atari environments. However, one consideration is the reduction of control this brings which may be problematic if the solution to an environment falls between actionable states. Therefore, to reduce this risk, we only allow a small actuator force to be applied at each action-step.

4.3 Reinforcement Learning Classes

4.3.1 ReplayBuffer

The ReplayBuffer's task is simply to record all the observations made when an agent interacts with the environment and return batches of those observations on request. Typically this involves recording the state, action, reward, next state and terminal values at each step [7]. Our exception to this is within HDDQN, where the buffer must also be able to accept a state-goal pair.

Our ReplayBuffer is effectively the same naive approach, as the experience replay, utilized in the original DQN paper [7]. However, as covered in Section-3.3.4, more sophisticated solutions exist within the research space and would likely improve the performance of our results if implemented.

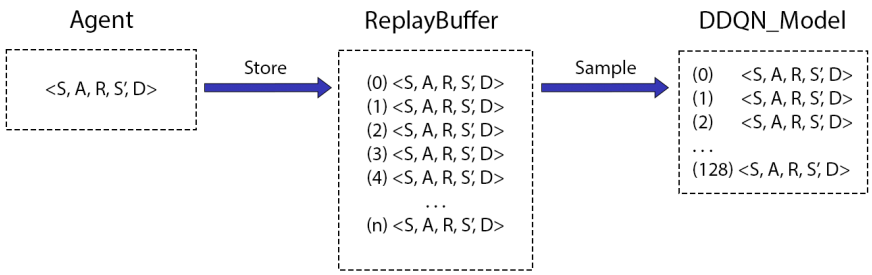


Figure 4.5: A figure describing the process of an Agent storing its observations into the ReplayBuffer which can then be sampled in batches later by the DDQN_Model. Observations are in the form, [State, Action, Reward, Next State, Done].

4.3.2 Agents

Two agents are available in the final solution, namely, bot_ddqn and bot_hddqn for DDQN and HDDQN respectively. These agents are effectively players within the environment and follow the algorithms laid out in Sections-3.3.2 and 3.3.3. They interact through choosing actions via the class DDQN_Model. These interactions are then stored in a ReplayBuffer, which the DDQN_Model later uses to learn from - updating its associated neural networks. This process is repeated until the agent reaches a terminal state, in our case, this is either when it completes the environment by reaching

the goal or after it has taken one hundred steps in the environment - chosen arbitrarily by the environment. This is repeated for as many episodes as decided by the user.

4.3.3 DDQN Model

The DDQN_Model acts as an intermediary between our Agent, ReplayBuffer and Neural Network. This class is responsible for choosing actions either randomly or through the prediction of the neural network(s) - Epsilon Greedy [13]. Additionally, this also contains both target and update formulas for the DDQN algorithm. Updating the target neural network(s) at each time step, once the ReplayBuffer holds enough memories to saturate the batch-size, and the evaluation network every thousand steps (by default). These formulas are defined in Van's paper [6], and are discussed in more detail within Section-3.3.2 of this paper.

4.3.4 Neural Network

The Neural Network class handles the construction of our networks. As mentioned previously, the observation types within this project are flexible and thus, so to are the networks. Any of the four types can be used independently or in combination with another. Furthermore, the number of layers and neurons across Dense layers pre and post concatenation can be adjusted, where any pre-Dense layers are appended to the numerical input layers. The activation function for each of these layers are Rectified Linear Units (ReLU) while the output layer is fitted with softmax for prediction.

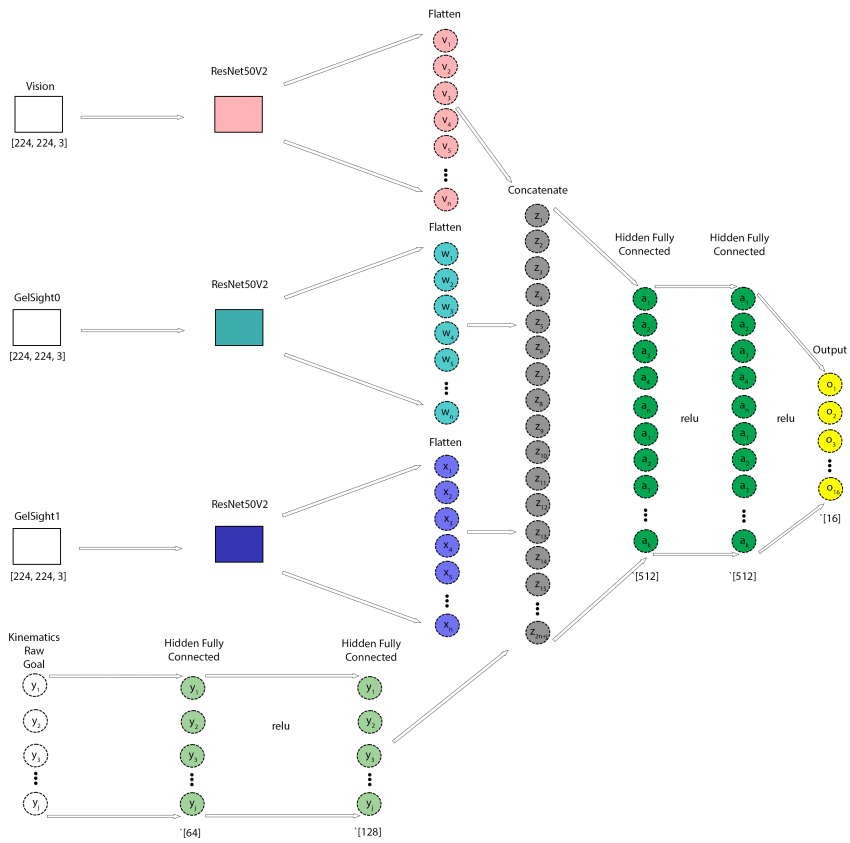


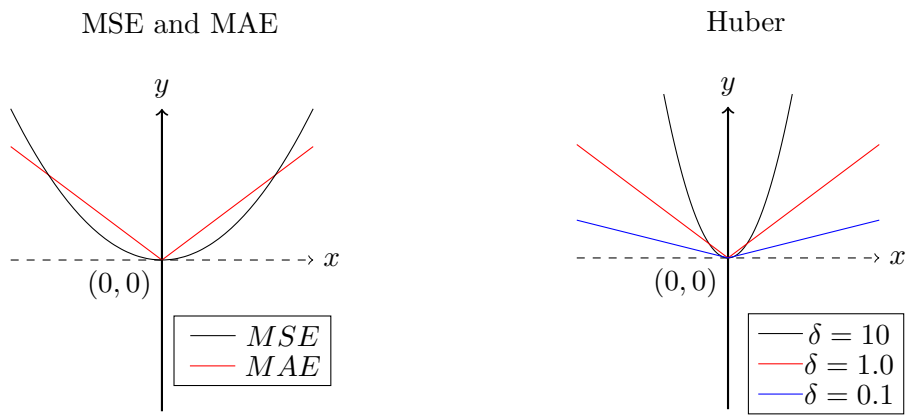
Figure 4.6: A sample neural network, the structure utilizes all four input types, vision, tactile, kinematics and raw. Vision and tactile types are each passed into a pre-trained model of ResNet50v2 [14, 15, 16, 17].

Following the submission of Appendix-E, additional models were appended to the code. This enables the use of not only ResNet50v2 but also, MobileNet [85], Inception [104], InceptionResNet [105] and ResNet18. All except ResNet18 use the inbuilt tensorflow methods, which instead uses Yakubovskiy's image-classifiers package [106]. The hope was to experiment with different variations of pre-trained image classification networks. However, due to time limitations, this has not been possible.

4.3.5 Alternative Network Considerations

Loss Functions and Optimizers

Mean Squared Error (MSE) is often chosen as a loss function when testing a reinforcement learning solution [107, 108, 109]. This measures averages across predicted and actual values. It has the advantage of forming a quadratic equation, thus with gradient descent there is only one global minimum, however, outliers can be particularly large - skewing the result. Mean Absolute Error (MAE) can compensate for these outliers, though there is now the potential for local minimums [110]. Huber is less sensitive than square error loss methods, handling outliers and local minimums far better, though the addition of another hyperparameter (δ) inherently increases training time [111].



$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

$$L_\delta = \begin{cases} \frac{1}{2}(y - \hat{y}_i)^2 & \text{if } |y_i - \hat{y}_i| < \delta, \\ \delta((y_i - \hat{y}_i) - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

Figure 4.7: This Figure and the accompanying formulas describe the three loss functions Mean Square Error, Mean Absolute Error and Huber loss. Huber's illustration displays the effect three distinct δ's have on the function [18].

Similarly, Adam [112] is often chosen as the default optimizer. This is because it has been proven to perform well in many problems through combining the properties of both AdaGrad [113] and RMSProp [114].

There are a great deal more functions than those discussed above. In order to find the best fit, more would need to be experimented upon, however, due to the limited time and scope available to this project this has not been possible. As such, only the loss function Huber [115] and the optimizer Adam [112] have been utilized.

Activation Functions

ReLU has fast become common practice within the Deep RL space [116, 117]. However, activation functions are not one-size-fits-all and are effectively hyperparameters - requiring tuning [19, 20].

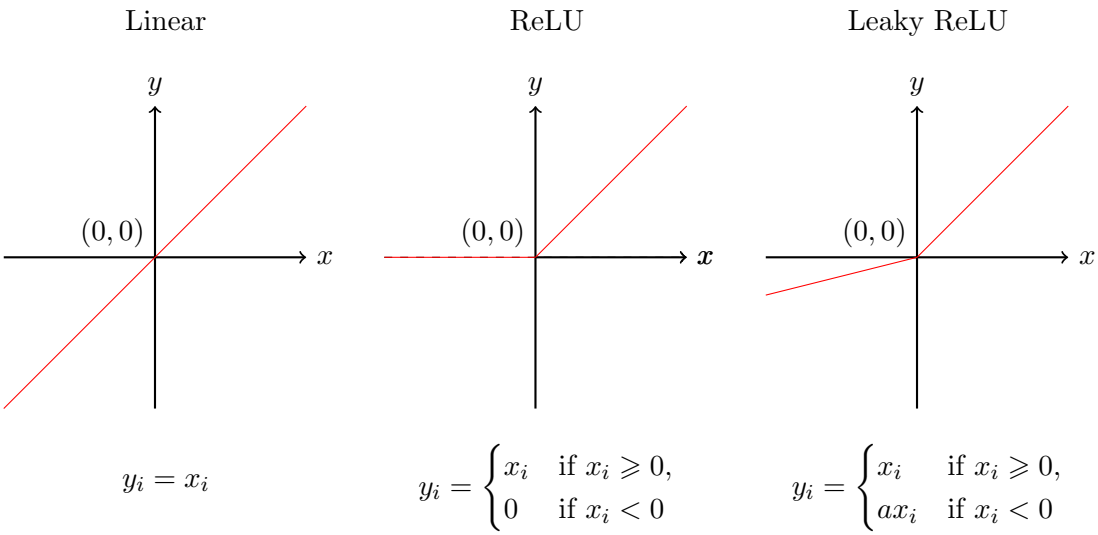


Figure 4.8: A sample of activation functions commonly used within machine learning. Linear leaves the neuron unchanged whereas ReLU and Leaky ReLU transform the neuron's value when it falls below zero [19, 20].

Observe, within Figure-4.8, that the linear function effectively passes its neuron's value without manipulation. This is potentially inefficient, as all neurons may not be required for solving a given problem. Thus ReLU was created, which crushes all negative values to zero, removing their information - killing the neuron. In some problems, this the total loss of this information might be an issue. Leaky ReLU is an attempt to solve this. While it still places less emphasis on negative values, in comparison to positive ones, it retains part of that information [19]. Potentially under ReLU, a neuron may be mistakenly killed early when in fact its inclusion is crucial to solving parts of an environment at later time-steps.

Figure-4.8 is not inclusive of all activation functions, in fact, there are a great many of them. Some examples include Gaussian Error Linear Unit (GELU) [118] and Scaled Exponential Linear Unit (SELU) [119]. The former combines properties of dropout, zoneout and ReLU while the latter constitutes a self-normalizing network - preserving means and variances of each layer [120]. Once again, time permitting it would be interesting to see how these could affect change within our results.

4.3.6 Main

Main offers quick access to some of the headline attributes and features of the project. It does this through a handler class which knits together the five classes, discussed above. A demo of an earlier version of this can be found within the Final Presentation Video [121] which accompanied Appendix-E. Since this video, several additions have been made. An updated listing of these parameters can be seen in Figure-A.2.

Chapter 5
Experiments

5.1 Early Experiments

This project grew organically as it progressed. As such, both DDQN and HDDQN agents saw training within environments other than the final iteration. One of these earlier instantiations was the *glass table* environment, observable in Figure-4.1. Here, the agent's goal was simply to lift the TO for five consecutive time-steps. The TO was placed between the two GelSight sensors at the beginning of each episode. This is obviously a fairly simple environment to solve, and so both DDQN and HDDQN did so in under one thousand episodes.

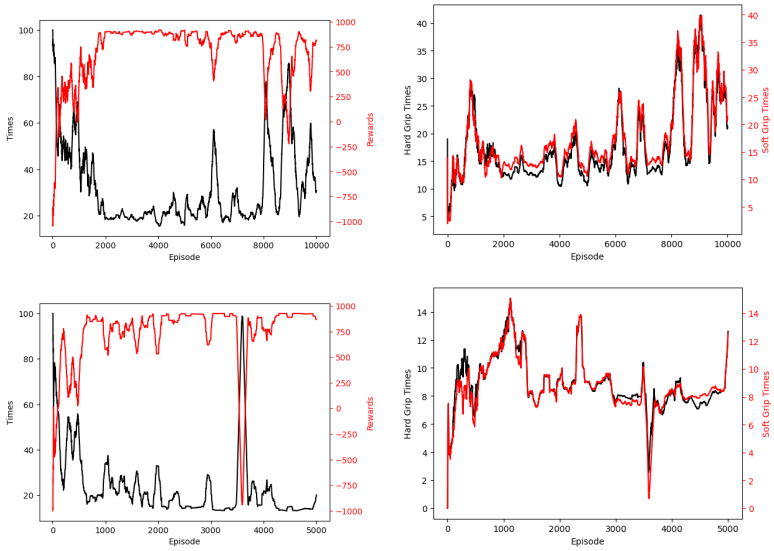


Figure 5.1: Early results from both DDQN (top) and HDDQN (bottom). Here a cylindrical object was statically placed in between the fingers of a blind gripper. The agent's observation space was limited to the two tactile sensors alongside the arm's joint positions and velocity. Results are averaged over 100-episodes, where red indicates rewards received and black indicates the number of steps taken.

Both algorithms show varying levels of destabilization, represented by the fluctuations observed on the LHS charts of Figure-5.1. This is likely caused by an unsuitable learning and target update rate combination, which we will address in the next Section. However, the results are still meaningful. We can see that DDQN converges to an optimal solution faster than HDDQN, yet HDDQN spends far less time in contact with the TO - observable in the grip time charts on the RHS of Figure-5.1. The latter is perhaps indicative of HDDQN's ability to better deal with sparse reward environments while the former is likely a result of the algorithm's additional complexity.

5.2 Final Model Experiments

5.2.1 Reach - Static TO

We have successfully trained and tuned a static Reach environment using DDQN, this can be seen in Figure-5.2. These tests took place prior to the implementation of transfer learning models, such as ResNet and so the neural network is based on a two-layer two-dimensional convolutional structure. It is clear that the tuning of hyper-parameters, specifically the learning rate and target update rate between the target and evaluation networks, can greatly affect the quality of the model an agent produces [122].

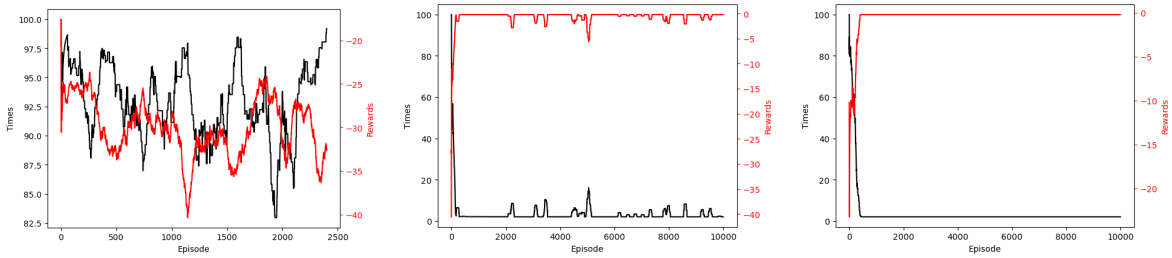


Figure 5.2: Testing on the Reach environment with a non-randomly placed object in combination with real and kinematic based observations. The graphs, left-to-right, use learning rates and target update rates of [1, 50], [0.1, 500] and [0.01, 1000] respectively. The results here are averaged over 100-episodes, where red are the rewards and black is the number of steps taken.

Although lowering the learning rate and increasing the time between network updates correlates to much better solutions. These cannot be increased indefinitely. An additional increase of magnitude can be observed in Figure-5.3. The lower learning rate performs better than the worst of Figure-5.2, yet it appears to be getting worse as it trains. Similarly, increasing the update rate seems to only have a slight impact on the stability of convergence pre-episode one-thousand. However, there is a sharp decline almost immediately.

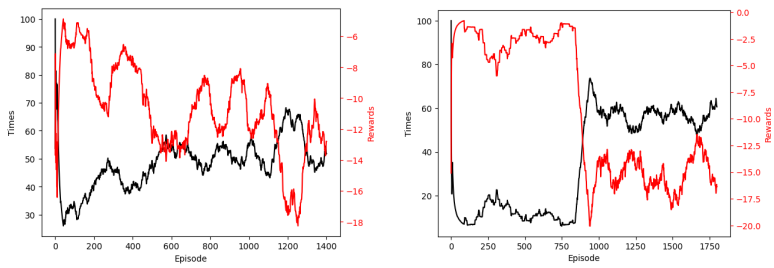


Figure 5.3: Further testing within the static Reach environment. Left-to-right illustrate results for learning and update rates of [0.001, 1000] and [0.01, 10000].

Finally, Figure-5.4 illustrates seven independent runs of our best hyperparameter combination [0.01, 1000]. From these results it is clear that both the learning and target update rates require balancing. Additionally, it demonstrates that even post tuning, convergence is not necessarily guaranteed on every run.

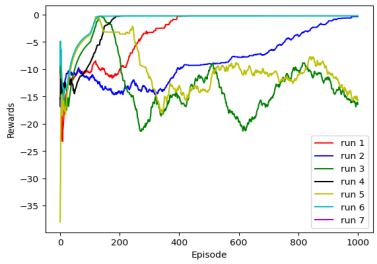


Figure 5.4: A comparison of seven independent runs of the Reach environment using [0.01, 1000]. Five of which converge to an optimal solution.

5.2.2 Baseline - Raw Inputs

Testing within environments using random object placements has also been completed. These tests use only raw observations and act as a set of baseline results. The reward and time results are averaged over 100 episodes and can be seen in Figure-5.5 and Figure-5.6 respectively.

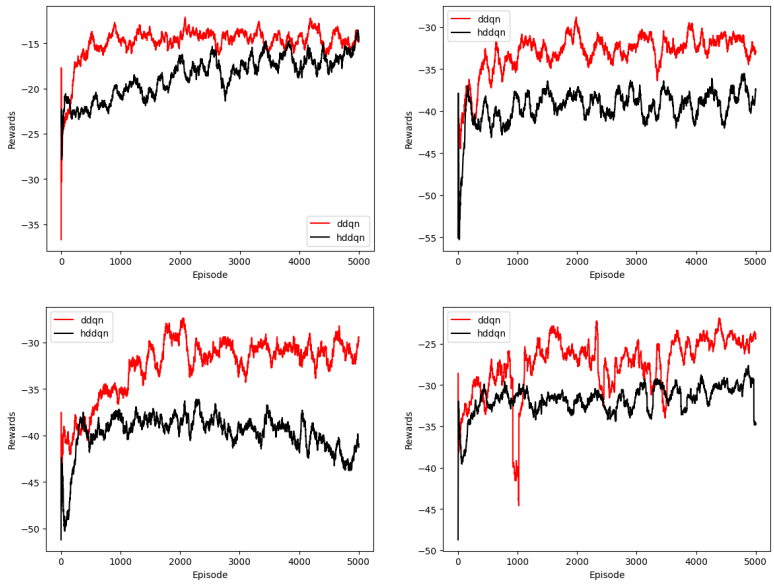


Figure 5.5: Baseline reward results of Reach, Slide, Push, and Pickand-Place respectively (left to right). The neural network structure takes the form of three fully connected layers each with 256 neurons. The update rate mirrors the best results recorded in Figure-5.2, however, the learning rate was reduced to 0.0001.

When comparing DDQN and HDDQN, we can see that HDDQN is unable to create stronger models than DDQN within 5000 episodes. This is likely due to HDDQN's additional complexity, originating from the additional agent that manages two further neural networks. Whereas DDQN only utilizes a single epsilon value, which it decays at each time-step, it is necessary to decay several for HDDQN. Here, the intrinsic agent has one epsilon per goal and can only decay the active goal's epsilon if said goal is completed. Furthermore, although the extrinsic agent only uses a single epsilon, this is only decayed at the end of each episode.

To compensate, the algorithm could be tuned to decay more quickly to match the

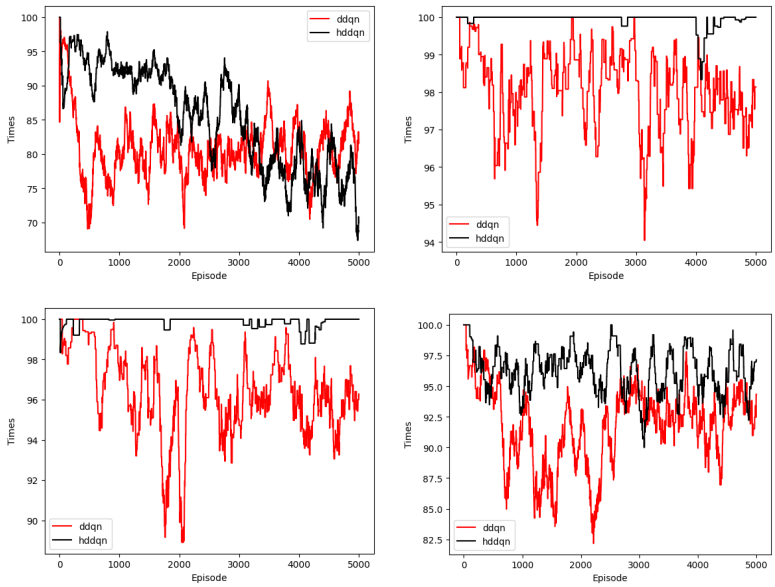


Figure 5.6: Time results from the baseline tests of Reach, Slide, Push, and PickandPlace respectively (left to right). These results are taken from the same set of tests as in Figure-5.5, and therefore use the same hyperparameters.

overall decay seen in DDQN. However, in testing, this negatively affected performance and can be seen in Figure-5.7.

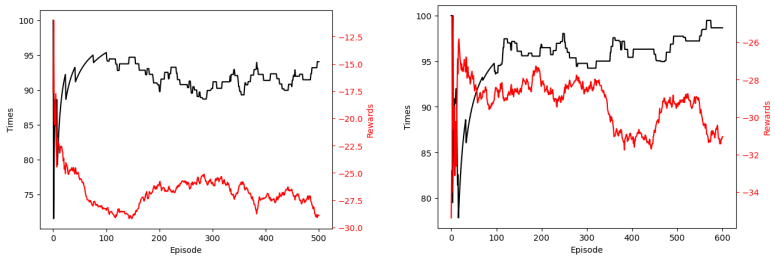


Figure 5.7: Higher decaying epsilons for HDDQN. Reduced to 0.986 (LHS) and 0.976 (RHS).

Although Figure-5.5's results illustration both DDQN and HDDQN converge positively, in all except HDDQN's Push, we can see that neither are able to fully solve any of the environments. This is to be expected, as the raw observation state provides incomplete information, giving the agents a way to identify the position of their target(s). Therefore, it is reasonable to assume the step would increase the observation space fed to the agents, this is covered in Section-5.2.3.

Finally, we close this Section with experiments concerning different kernel initializers. By default, Tensorflow initializes networks using Glorot (Xavier) uniform [123]. However, Lippe demonstrates in this article [124], that HeNorm or HeUniform (Kaiming) [125] are a better fit for ReLU activation layers.

When observing the reward results, in Figure-5.8, it can be seen that all four initializers closely approximate one another across the training set. However, between episodes 200 and 1200, HeNormal and HeUniform both outperform their Glorot counterparts. Although, Glorot uniform is the best performing by the end of training. A

larger test set would be required in order to make a firm distinction as to which initilizer is the best fit.

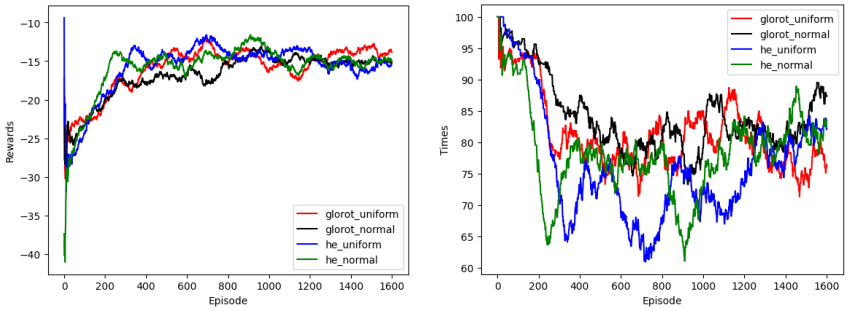


Figure 5.8: Reward and time results comparing different kernel initializers within the (random target) Reach environment. Results are taken as an average over 100-episodes.

5.2.3 Transfer Learning

The choice of algorithm, observation and environment all clearly have an effect on performance. Time limitations were the primary consideration when choosing to use raw observations as the baseline. As this simplified the network, allowing for both faster convergence and less time spent on each episode. When attempting to expand training to more complex observation states, training time barriers made themselves very apparent. As such, progress was stifled and we were unable to categorically prove convergence within these more complex environments.

However, Figure-5.9 demonstrates some of that testing. And while worse than the baseline results, they do appear to be trending upwards.

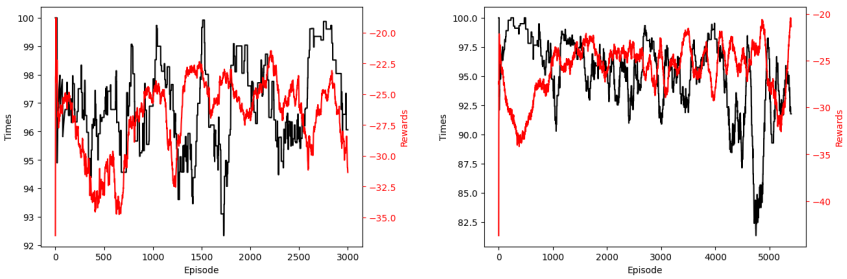


Figure 5.9: Reach environment results trained on both vision and raw observation types. LHS uses ResNet50V2 for its pre-trained layers, whereas RHS uses ResNet18.

Chapter 6
Conclusion

6.1 Summary

This project has seen the successful implementation of Gomes' simulated GelSight sensor within Mujoco, alongside four distinct environments. Agents consisting of both DDQN and HDDQN have been instantiated and trained within these, producing a set of baseline results across the complex iterations (random object placements), using only raw observations.

However, it has proven difficult to produce models which converge on more complex observation spaces due to time restraints. Therefore, it is unclear as to whether the models require more training time, if a suitable combination of hyperparameters has yet to be found, or if there is a more fundamental problem - such as the need to revise the reward function or HDDQN's goals.

In fact, one of the largest challenges within HDDQN's implementation has been in defining appropriate goals. Whether those goals should be representations of the state space or if they should constitute abstract actions. State space representations could take the form of whether the TO has been gripped or not. However, the complexity of how to best represent this between different observation types, in each environment, makes this time-consuming to produce. Therefore, abstract actions were chosen. These take the form of moving up, down, left or right within the three-dimensional space, with PickandPlace having the additional goals of grasping and lifting.

6.2 Potential Future Works

If the time limitations were removed as a factor, the next logical step might be to compare different combinations of observation types, pre-trained models, reward functions and goals. There are also a vast number of hyperparameters to explore and the potential to even extend, or add entirely new algorithms to the test-bed.

Following this, it would be interesting to attempt a Sim2Real transference. Shifting the simulated agent to a physical one.

Bibliography

[1] Mujoco - advanced physics simulation. *mujoco200 win64*, 2018. URL `http://www.mujoco.org/`.

[2] Daniel Fernandes Gomes, Paolo Paoletti, and Shan Luo. Generation of gelsight tactile images for sim2real learning. *IEEE Robotics and Automation Letters*, 6 (2):4177–4184, 2021.

[3] Travis DeWolf. Ur5 6dof robotic arm. 2020. URL `http://www.mujoco.org/forum/index.php?resources/ur5-6dof-robotic-arm.27/`.

[4] ROBOTIQ. 2f-85 adaptive gripper. 2019. URL `https://robotiq.com/`.

[5] Open AI. Openai gym. 2016. URL `https://gym.openai.com/`.

[6] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[8] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[9] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29: 3675–3683, 2016.

[10] Kaleb Ben Naveed, Zhiqian Qiao, and John M Dolan. Trajectory planning for autonomous vehicles using hierarchical reinforcement learning. *arXiv preprint arXiv:2011.04752*, 2020.

[11] Micah K Johnson and Edward H Adelson. Retrographic sensing for the measurement of surface texture and shape. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1070–1077. IEEE, 2009.

[12] OpenAI. Part 2: Kinds of rl algorithms. 2018. URL `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`.

[13] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[15] Cheng-Chi Wang and Yong-Quan Zhu. Identification and machine learning prediction of nonlinear behavior in a robotic arm system. *Symmetry*, 13(8):1445, 2021.

[16] Kateryna Zinchenko and Kai-Tai Song. Autonomous endoscope robot positioning using instrument segmentation with virtual reality visualization. *IEEE Access*, 9:72614–72623, 2021.

[17] Gunjan Chourasia, Akarsh Shrivastava, Piyush Jaipuriyar, Zeel Shailesh kumar Bhatt, Ali Shahbaaz Khan, and Amrit Das. 7-dof robotic manipulator for autonomous segregation using transfer learning. In *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 698–701. IEEE, 2019.

[18] Thibaut Théate and Damien Ernst. An application of deep reinforcement learning to algorithmic trading. *Expert Systems with Applications*, 173:114632, 2021.

[19] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[20] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[21] Shan Luo, Joao Bimbo, Ravinder Dahiya, and Hongbin Liu. Robotic tactile perception of object properties: A review. *Mechatronics*, 48:54–67, 2017.

[22] Wenzhen Yuan, Siyuan Dong, and Edward H Adelson. Gelsight: High-resolution robot tactile sensors for estimating geometry and force. *Sensors*, 17(12):2762, 2017.

[23] Alexander C Abad and Anuradha Ranasinghe. Visuotactile sensors with emphasis on gelsight sensor: A review. *IEEE Sensors Journal*, 20(14):7628–7638, 2020.

[24] Jinda Cui and Jeff Trinkle. Toward next-generation learned robot manipulation. *Science Robotics*, 6(54), 2021.

[25] Ilya Kostrikov, Denis Yarats, and Rob Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv preprint arXiv:2004.13649*, 2020.

[26] Fereshteh Sadeghi, Alexander Toshev, Eric Jang, and Sergey Levine. Sim2real viewpoint invariant visual servoing by recurrent control. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4691–4699, 2018.

[27] Abhishek Kadian, Joanne Truong, Aaron Gokaslan, Alexander Clegg, Erik Wijmans, Stefan Lee, Manolis Savva, Sonia Chernova, and Dhruv Batra. Are we making real progress in simulated environments? measuring the sim2real gap in embodied visual navigation. 2019.

[28] Fereshteh Sadeghi, Alexander Toshev, Eric Jang, and Sergey Levine. Sim2real view invariant visual servoing by recurrent control. *arXiv preprint arXiv:1712.07642*, 2017.

[29] Daniel Fernandes Gomes, Achu Wilson, and Shan Luo. Gelsight simulation for sim2real learning. In *ICRA ViTac Workshop*, 2019.

[30] Edward J Smith, Roberto Calandra, Adriana Romero, Georgia Gkioxari, David Meger, Jitendra Malik, and Michal Drozdzal. 3d shape reconstruction from vision and touch. *arXiv preprint arXiv:2007.03778*, 2020.

[31] Arpit Agarwal, Tim Man, and Wenzhen Yuan. Simulation of vision-based tactile sensors using physics based rendering. *arXiv preprint arXiv:2012.13184*, 2020.

[32] Gazebo. 2021. URL `http://gazebosim.org/`.

[33] Jonas Schneider Jonathan Ho Peter Welinder Wojciech Zaremba Jerry Tworek Alex Ray, Bob McGrew. mujoco-py. 2018. URL `https://github.com/openai/mujoco-py`.

[34] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, 6(5):679–684, 1957.

[35] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.

[36] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[37] J Dargahi. A piezoelectric tactile sensor with three sensing elements for robotic, endoscopic and prosthetic applications. *Sensors and Actuators A: Physical*, 80 (1):23–30, 2000.

[38] Karsten Weiß and Heinz Worn. The working principle of resistive tactile sensor cells. In *IEEE International Conference Mechatronics and Automation, 2005*, volume 1, pages 471–476. IEEE, 2005.

[39] Hong-Ki Kim, Seunggun Lee, and Kwang-Seok Yun. Capacitive tactile sensor array for touch screen application. *Sensors and Actuators A: Physical*, 165(1): 2–7, 2011.

[40] Ren-Chyuan Luo, Fuling Wang, and You-xing Liu. An imaging tactile sensor with magnetostrictive transduction. In *Intelligent Robots and Computer Vision*, volume 521, pages 264–270. International Society for Optics and Photonics, 1985.

[41] Wendong Zheng, Bowen Wang, Huaping Liu, Yunkai Li, Ran Zhao, Ling Weng, and Changgeng Zhang. Structural design and output characteristic analysis of magnetostrictive tactile sensor for robotic applications. *AIP Advances*, 8(5): 056622, 2018.

[42] Josivaldo Godoy da Silva, Aparecido Augusto de Carvalho, and Doriedson Dutra da Silva. A strain gauge tactile sensor for finger-mounted applications. *IEEE Transactions on Instrumentation and measurement*, 51(1):18–22, 2002.

[43] Edward Adelson. Session 3 - (keynote) - the future of camera-based tactile sensing. *ViTax WorkShop ICRA*, 2021. URL `https://www.youtube.com/watch?v=IuCDAjhDnak`.

[44] Wenzhen Yuan, Rui Li, Mandayam A Srinivasan, and Edward H Adelson. Measurement of shear and slip with a gelsight tactile sensor. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 304–311. IEEE, 2015.

[45] Rui Li, Robert Platt, Wenzhen Yuan, Andreas ten Pas, Nathan Roscup, Mandayam A Srinivasan, and Edward Adelson. Localization and manipulation of small parts using gelsight tactile sensing. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3988–3993. IEEE, 2014.

[46] Wenzhen Yuan, Mandayam A Srinivasan, and Edward H Adelson. Estimating object hardness with a gelsight touch sensor. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 208–215. IEEE, 2016.

[47] Achu Wilson, Shaoxiong Wang, Branden Romero, and Edward Adelson. Design of a fully actuated robotic hand with multiple gelsight tactile sensors. *arXiv preprint arXiv:2002.02474*, 2020.

[48] Elliott Donlon, Siyuan Dong, Melody Liu, Jianhua Li, Edward Adelson, and Alberto Rodriguez. Gelslim: A high-resolution, compact, robust, and calibrated tactile-sensing finger. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1927–1934. IEEE, 2018.

[49] Branden Romero, Filipe Veiga, and Edward Adelson. Soft, round, high resolution tactile fingertip sensors for dexterous robotic manipulation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4796–4802. IEEE, 2020.

[50] Daniel Fernandes Gomes, Zhonglin Lin, and Shan Luo. Geltip: A finger-shaped optical tactile sensor for robotic manipulation. *arXiv preprint arXiv:2008.05404*, 2020.

[51] Zihan Ding, Nathan F Lepora, and Edward Johns. Sim-to-real transfer for optical tactile sensing. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1639–1645. IEEE, 2020.

[52] René Traoré, Hugo Caselles-Dupré, Timothée Lesort, Te Sun, Natalia Díaz-Rodríguez, and David Filliat. Continual reinforcement learning deployed in real-life using policy distillation and sim2real transfer. *arXiv preprint arXiv:1906.04452*, 2019.

[53] Benjamin Ward-Cherrier, Nicholas Pestell, Luke Cramphorn, Benjamin Winstone, Maria Elena Giannaccini, Jonathan Rossiter, and Nathan F Lepora. The tactip family: Soft optical tactile sensors with 3d-printed biomimetic morphologies. *Soft robotics*, 5(2):216–227, 2018.

[54] Tarık Keleştemur, Colin Keil, John P Whitney, Robert Platt, and Taşkın Padır. Learning bayes filter models for tactile localization. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9253–9258. IEEE, 2020.

[55] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[56] Sébastien Racanière, Théophane Weber, David P Reichert, Lars Buesing, Arthur Guez, Danilo Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 5694–5705, 2017.

[57] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[58] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous

methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[59] Yi Wang, Zhice Fang, Haoyuan Hong, Romulus Costache, and Xianzhe Tang. Flood susceptibility mapping by integrating frequency ratio and index of entropy with multilayer perceptron and classification and regression tree. *Journal of Environmental Management*, 289:112449, 2021.

[60] Fulya Aydın Temel, Özge Cağcağ Yolcu, and Ayşe Kuleyin. A multilayer perceptron-based prediction of ammonium adsorption on zeolite from landfill leachate: Batch and column studies. *Journal of Hazardous Materials*, 410: 124670, 2021.

[61] Shital S Chiddarwar and N Ramesh Babu. Comparison of rbf and mlp neural networks to solve inverse kinematic problem for 6r serial robot by a fusion approach. *Engineering applications of artificial intelligence*, 23(7):1083–1092, 2010.

[62] Stephen James and Edward Johns. 3d simulation for robot arm control with deep q-learning. *arXiv preprint arXiv:1609.03759*, 2016.

[63] Deirdre Quillen, Eric Jang, Ofir Nachum, Chelsea Finn, Julian Ibarz, and Sergey Levine. Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6284–6291. IEEE, 2018.

[64] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[65] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*, 2015.

[66] Chen Wang, Shaoxiong Wang, Branden Romero, Filipe Veiga, and Edward Adelson. Swingbot: Learning physical features from in-hand tactile exploration for dynamic swing-up manipulation. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5633–5640. IEEE, 2020.

[67] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23:2613–2621, 2010.

[68] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[69] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[70] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[71] Martin Lauer and Martin Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*. Citeseer, 2000.

[72] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[73] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[74] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[75] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

[76] Ramy E Ali, Bilgehan Erman, Ejder Baştuğ, and Bruce Cilli. Hierarchical deep double q-routing. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2020.

[77] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6232–6240, 2017.

[78] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the expressive power of deep neural networks. In *international conference on machine learning*, pages 2847–2854. PMLR, 2017.

[79] Zhuangdi Zhu, Kaixiang Lin, and Jiayu Zhou. Transfer learning in deep reinforcement learning: A survey. *arXiv preprint arXiv:2009.07888*, 2020.

[80] S Bozinovski and A Fulgosi. The influence of pattern similarity and transfer learning upon training of a base perceptron b2. In *Proceedings of Symposium Informatica*, pages 3–121, 1976.

[81] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[82] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[83] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[84] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[85] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[86] Carl Doersch and Andrew Zisserman. Sim2real transfer learning for 3d human pose estimation: motion to the rescue. *Advances in Neural Information Processing Systems*, 32:12949–12961, 2019.

[87] Alexander Pashevich, Robin Strudel, Igor Kalevatykh, Ivan Laptev, and Cordelia Schmid. Learning to augment synthetic images for sim2real policy transfer. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2651–2657. IEEE, 2019.

[88] Ofir Nachum, Michael Ahn, Hugo Ponte, Shixiang Gu, and Vikash Kumar. Multi-agent manipulation via locomotion using hierarchical sim2real. *arXiv preprint arXiv:1908.05224*, 2019.

[89] Olivier Michel. Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004.

[90] Daniel Gomes. Gelsight simulation. 2021. URL `https://github.com/danfergo/gelsight`$_{s}imulation$.

[91] Frederico Francisco, Orfeu Bertolami, Paulo JS Gil, and Jorge Páramos. Modelling the reflective thermal contribution to the acceleration of the pioneer spacecraft. *Physics Letters B*, 711(5):337–346, 2012.

[92] Filipe Veiga, Riad Akrour, and Jan Peters. Hierarchical tactile-based control decomposition of dexterous in-hand manipulation tasks. *Frontiers in Robotics and AI*, 7, 2020.

[93] Yannis Flet-Berliac. The promise of hierarchical reinforcement learning. *The Gradient*, 2019. URL `https://thegradient.pub/the-promise-of-hierarchical-reinforcement-learning/`.

[94] Ari Azarafrooz and John Brock. Hierarchical soft actor-critic: Adversarial exploration via mutual information optimization. *arXiv preprint arXiv:1906.07122*, 2019.

[95] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.

[96] Yuval Tassa, Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, and Nicolas Heess. dm_control: Software and tasks for continuous control, 2020.

[97] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

[98] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[99] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[100] Francois Chollet et al. Keras, 2015. URL `https://github.com/fchollet/keras`.

[101] Jim Lambers. Mat 169, lecture 17 notes. 2009. URL `https://www.math.usm.edu/lambers/mat169/fall09/lecture17.pdf`.

[102] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning. In *International Conference on Machine Learning*, pages 4045–4054. PMLR, 2018.

[103] Alex Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. Frame skip is a powerful parameter for learning to play atari. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[104] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[105] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[106] Pavel Yakubovskiy, Ganesh Anandm, and Fernando Camargo. Classification models zoo - keras (and tensorflow keras), 2019. URL `https://github.com/qubvel/classification`$_m odels$.

[107] Ming Yin and Yu-Xiang Wang. Asymptotically efficient off-policy evaluation for tabular reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, pages 3948–3958. PMLR, 2020.

[108] Matthew E Taylor, Gregory Kuhlmann, and Peter Stone. Autonomous transfer for reinforcement learning. In *AAMAS (1)*, pages 283–290, 2008.

[109] Philip Thomas and Emma Brunskill. Data-efficient off-policy policy evaluation for reinforcement learning. In *International Conference on Machine Learning*, pages 2139–2148. PMLR, 2016.

[110] Cort J Willmott and Kenji Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.

[111] Zhou Wang and Alan C Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *IEEE signal processing magazine*, 26(1):98–117, 2009.

[112] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[113] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[114] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Lecture 6a overview of mini–batch gradient descent. *Coursera Lecture slides https://class. coursera. org/neuralnets-2012-001/lecture,[Online*, 2012.

[115] Peter J Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics*, pages 492–518. Springer, 1992.

[116] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[117] Yuanzhi Li and Yang Yuan. Convergence analysis of two-layer neural networks with relu activation. *arXiv preprint arXiv:1705.09886*, 2017.

[118] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[119] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Proceedings of the 31st international conference on neural information processing systems*, pages 972–981, 2017.

[120] Pragati Baheti. 12 types of neural network activation functions: How to choose. *Microsoft*, 2021 (accessed). URL `https://www.v7labs.com/blog/neural-networks-activation-functions`.

[121] Rob Lamprell. Openai test-bed of contact based grasping for reinforcement learning – final presentation video. 2021. URL `https://stream.liv.ac.uk/nsvmaujh`.

[122] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[123] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[124] Phillip Lippe. Tutorial 4: Optimization and initialization. *UvA DL Notebooks*, 2021. URL `https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial4/Optimization_and_Initialization.html`.

[125] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Appendix A
Code Samples

```python
# Reach class, generates the reach environment
class RobotArm__Reach(RobotArm):
    # user input
    def __init__(self,
                 render              = True,
                 return_real         = False,
                 return_tact         = False,
                 return_kine         = False,
                 return_raw          = False,
                 gel_width           = 224,
                 gel_height          = 224,
                 image_grayscale     = False,
                 rescale             = 128,
                 image_normalise     = False,
                 envcam_width        = 720,
                 envcam_height       = 720,
                 frame_stack         = False,
                 k_frames            = 3,
                 random_target       = False,
                 dist_threshold      = 0.07,
                 frame_skip          = 600,
                 seed                = 42
                 ):

        # load the base class's init to setup the environment
        super().__init__(
            render          = render,
            return_real     = return_real,
            return_tact     = return_tact,
            return_kine     = return_kine,
            return_raw      = return_raw,
            gel_width       = gel_width,
            gel_height      = gel_height,
            envcam_width    = envcam_width,
            envcam_height   = envcam_height,
            image_grayscale = image_grayscale,
            rescale         = rescale,
            image_normalise = image_normalise,
            frame_stack     = frame_stack,
            k_frames        = k_frames,
            has_object      = False,
            random_object   = False,
            has_target      = True,
            random_target   = random_target,
            t_dist_threshold= dist_threshold,
            model_name      = 'gripper_testbed_reach.xml',
            frame_skip      = frame_skip,
            seed            = seed
        )
```

Figure A.1: An image showing code directly from the Reach class's constructor. The four environments, including Reach, can take in a large amount of parameters in order to alter various aspects of its internal workings and outputs. These parameters are immediately passed onto the Base class, RobotArm, which contains the bulk of the environmental methodologies.

```python
# handler changing some of the features of the environment and neural network settings
class handler:
    def __init__(self,
                # env related params
                env                 = "reach",
                render              = True,
                random_spawns       = True,
                gelsight_dims       = [128, 128],
                envcam_dims         = [32, 32],
                image_rescale       = 32,
                image_grayscale     = False,
                image_normalise     = True,
                framestack          = False,
                k_frames            = 1,

                # agent related params
                agent               = "ddqn",
                mode                = "train",
                number_of_episodes  = 1000000,
                model_weights0      = None,
                model_weights1      = None,

                # DDQN_model params
                replay_buffer_size  = 10000,
                batch_size          = 128,

                # Neural Network params
                alpha_lr            = 0.00001,
                beta_lr             = None,
                nn_dense            = [1024, 512, 256],
                nn_num_dense        = [256, 256, 256],
                ext_nn_dense        = None,
                ext_nn_num_dense    = None,
                transfer_model      = 'resnet50',

                # misc params
                type_               = "vision_raw",
                extrinsic_type2_    = None,
                seed                = 42
                ):
```

Figure A.2: An image showing the main file's adjustable parameters. These parameters are separated into subsections of environment, agent, DDQN_model, neural network and misc. Where misc affects numerous sections.

Appendix B
Project Timeline - Gantt Chart

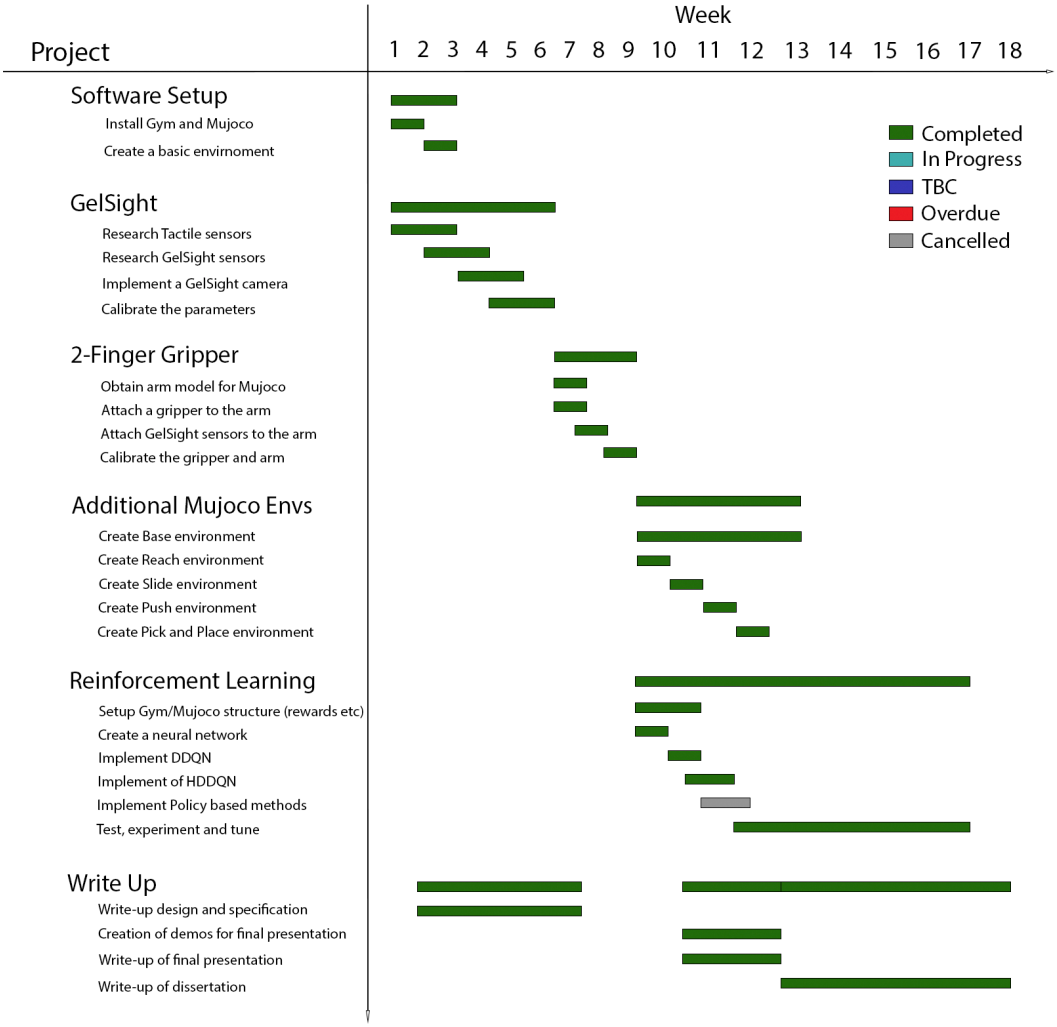


Figure B.1: A Gantt chart displaying the progress of the project over time. Policy Gradient-based methods, specifically DDPG was canceled in favor of experimenting with Hierarchical Double Deep Q-Learning.

Appendix C
Python Environment

Name	Version
anaconda-client	1.7.2
anaconda-navigator	2.0.3
conda	4.10.1
conda-build	3.21.4
conda-verify	3.4.2
Box2D	2.3.10
glfw	2.1.0
dm-control	0.0.364896371
Name	Version
dm-env	1.5
dm-tree	0.1.6
gym	0.18.3
gym-retro	0.8.0
h5py	3.1.0
HeapDict	1.0.1
ikfast-pybind	0.1.1
keras-nightly	2.5.0.2021032900
Keras-Preprocessing	1.1.2
keras-tuner	1.0.4
mujoco_py	2.0.2.5
numpy	1.19.5
tensorflow	2.5.0

Table C.1: A listing of the Python Packages, alongside their version, utilized within this project.

Appendix D
Original design document

This Appendix item is a complete copy of the project's the originally submitted design document - starting on the next page.

OpenAI Test-Bed of Contact Based Grasping for Reinforcement Learning

Design and Specification

July 2020

Rob Lamprell

Abstract

Robotics and reinforcement learning have traditionally been synonymous with vision. Allowing ideas to be pulled from science fiction, such as self-driving cars, into potential near-term reality. However, sight alone may not be sufficient in all scenarios and environments. As humans, we use several senses intuitively to interpret and effectively interact with our surroundings. One of these senses is touch [1].

The implementation and utilisation of touch-based sensors have been shown to greatly improve robotic interactions with objects. This is demonstrated by the great strides made through various innovations discussed below. One of these innovations is the GelSight sensor, which utilises a camera and gel-based elastomer to produce high-resolution tactile feedback. The use of this type of sensor, in tandem with reinforcement learning algorithms, has been shown to produce agents which are extremely competent in interacting with unseen objects [2].

In this paper, we aim to outline and implement a simulated environment, to be used as a test-bed to train a two-finger GelSight gripper. We will build on what has already been established, by first mirroring a ROS-Gazebo-based implementation of a Gelsight sensor [3] within the OpenAI gym extension, Mujoco. After which we will append the sensor to a UR5 arm and attempt to teach the gripper how to pick up novel objects using various model-free reinforcement learning algorithms, such as Double Deep Q-Learning (DDQN) [4] and Deep Deterministic Policy Gradient (DDPG) [5]. With this we aim to demonstrate the limitations of discrete action space learning methods, such as Q-Learning, versus their policy optimisation counterparts when training tactile sensor-based arms.

1 Introduction

Robotics is becoming an ever-growing normality within the roles we typically think of being held by humans. Industries such as health care, agriculture, food preparation, manufacturing, and the military have all seen vast benefits from traditional robotic automation. Generally increasing productivity while lowering the associated costs [6, 7].

One of the largest challenges robots face is in their ability to adapt to an ever-changing environment - something humans take for granted. This challenge not only extends to exterior environmental changes they can have some effect toward, but also those they cannot [8]. One potential solution in overcoming these issues is the expansion of robotic senses in combination with reinforcement learning algorithms.

Traditional robotic manipulation methods rely solely on vision. However, as humans, we intuitively use a multitude of senses in order to process our environment and any objects we

may interact with. Tactile sensory information is one of these means of evaluation we take for granted. Picking up objects implicitly conveys a significant amount of information around an object's mass, structure, and material. From these inputs, we are then able to intuitively amend our approach when manipulating any given object [1, 9, 10].

In addition to the benefits mentioned above, touch-based sensors may also be able to operate more optimally in environments in which purely vision-based implementations struggle. These situations include but are not limited to, environments where lighting may be inadequate or where objects are constructed using material compositions that are difficult to distinguish between through sight alone. This lack of information could in turn greatly affect the approach chosen to manipulate the object, potentially being the difference between success and failure [11].

Tactile sensors have evolved substantially over the last century, resulting in numerous different implementations. However, optical-based deformation sensors, such as GelSight offer several advantages over its peers [3]. First developed in 2009 by Johnson and Adelson [2, 12], GelSight sensors utilize a high-resolution camera placed inside an enclosure. This camera is pointed towards a paint-coated gel which captures the elastomer's deformation when objects are pressed against it. The recorded lateral and vertical measurements implicitly communicate other information such as tension and slippage [10]. Although GelSight is still somewhat new, use-cases have been sufficient enough for some implementations to be commercialized [10].

Figure 1: Physical GelSight Camera [3]

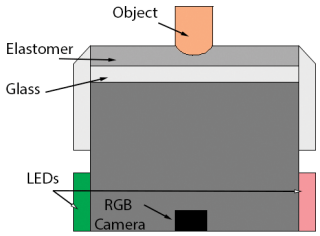


Figure 2: 2D Representation of a Real GelSight Camera [3]

As covered by Gomes [3], the main challenge of implementing virtual GelSight-based sensors is in the simulation of the internal view and aligning it to the real model. This approach, and the one we will be implementing, involves ray casting [13, 14]. However, there have since been papers that instead utilize physics-based rendering methods [15]. Allowing the physics engine to render the lighting could potentially make any built models easier to transfer across environments. Although, as covered below in Section-3.1, segregation of lighting from the environment also has its own benefits.

The formation of our simulated GelSight sensor will replicate Gomes' [3]. It will utilize a depth array to capture any object's contact with our simulated gel, and apply several formulas in order to better approximate the deformation and lighting model of a real sensor through Gaussian filtering and Phong's illumination model respectively [3]. Although the implementation of the GelSight here is within Gazebo [16], our implementation will be within the Mujoco extension to OpenAI's Gym - specifically utilising the python wrapper *mujoco_py* [17].

Following this, our second goal will be to implement a two-fingered-Gelsight gripper within the same environment. For this, we will be utilising a UR5 arm and 2F-85 ROBOTIQ gripper.

Our final steps will be to feed this into reinforcement learning algorithms. As a baseline, we will be using Double Deep Q-Learning (DDQN), which is an evolution of Deep Q-Learning [4]. Both of these are derivatives of Richard Bellman's famous equations [18], but they instead use a neural network as a function approximator to choose actions, rather than an explicit table of Q-values. Something similar can be seen in this paper by Gomes [13]. The second approach we will implement is Deep Deterministic Policy Gradient (DDPG). We wish to evaluate these two algorithms against one another. As our environment is effectively a continuous action space, rather than a discrete one, and Q-based learning algorithms perform poorly in continuous action spaces. It is expected DDPG will provide a better model, as policy-based methods are more flexible and are not limited to discrete spaces like Q-Learning [19].

The motivation for implementing all this within a simulated environment should not be overlooked. Training robots can be time-consuming, expensive, and potentially dangerous [3]. A physical robot's appendage's training speed would be limited to how quickly its components could move without breaking. Additionally, any unexpected behavior could result in potentially damaging repercussions. A solution to both these problems may be to develop the agent firstly within a simulated environment. Considering typical reinforcement learning algorithms can take thousands, hundreds-of-thousands, or more episodes in order to provide an agent with a sufficient level of skill.

Agents within virtual environments are able to cycle through potentially many episodes a second, as they are not held to the same limitations as physical models. Therefore, it makes sense to develop, tweak and train a model virtually so that this knowledge can be passed to a real agent in the future. This trend, of simulating before developing the real model is typically referred to as Sim2Real [20, 21, 22], and although the benefits are somewhat contentious [21], with proper tuning, they can potentially lead to saving great deals of time.

One final note is around generalisation. The effectiveness of which, on more complex modern neural networks, has recently come into question [23]. However, this topic is outside the scope of this paper. Therefore, we will sidestep this issue and assume traditional understandings around generalisation still hold.

In summary, our goal is to implement a testbed within the Mujoco environment [24, 25]. Where a two-finger gripper, equipped with dual GelSight sensors, can be trained, using both DDQN and DDPG, to grasp and manipulate objects. Before finally evaluating the algorithms against one another.

2 Related Works

2.1 Tactile Sensors

As mentioned by Gomes [3], tactile sensors have seen many different implementations. These types include optical [2], piezo-electric [26], resistive [27], capacitive [28], magnetic [29, 30] and strain gauge [31]. Each type has different methods in producing a given sensor. For example, Magnetic tactile sensors have two main methods, one is in measuring the change of flux density of using magneto resistant devices and the other is the change of magnetic coupling the deformation and a magneto-elastic core [29, 30]. As you might expect, each sensor and its associated implementation come with its own advantages and disadvantages. Though, camera-based sensors offer several distinct advantages over the others. For example, capacitive and resistive sensors offer only low-resolution solutions and require a lot of wiring, whereas an op-

tical solution offers not only a high-resolution output but also may only need a single wire [3, 9].

GelSight sensors can be found here [2, 13, 32, 33, 34], and have shown strong results in not only correctly approximating the shape of objects but, also their properties such as hardness and slippage when combined with machine learning techniques such as deep neural networks. Additionally, multiple GelSight sensors have been integrated into a single robotic gripper, such as in this paper by Wilson [35]. This has the benefit of providing tactile information around the shape of an object, rather than just on a flat plane.

Some other weaknesses of GelSight are its bulky design and small sensor area. This is something GelSlim, an iteration on the GelSight sensor, aimed to address. It has a larger sensory area and is much slimmer than its counterpart - attributed to a wedge shape instead of a box shape [36]. However, a flat sensor, is still somewhat limited in its capabilities, as its geometry makes it difficult to read some objects [9]. Furthermore, it would be much more useful when constructing hands if the sensors could surround the finger or even the entire hand. As this would provide a higher awareness of the environment. This paper, by Romero [37], explores applying gel caps to the fingertips of a robotic hand. The advantages of equipping a robotic hand with these can be seen in one of the demos in Adelson's keynote [9]. Here, the aforementioned hand is able to gently unscrew a loose jam jar without disturbing the main body.

Another skew on this idea is the GelTip, defined in this paper by Gomes' [38]. The GelTip differentiates itself by creating a sensor with a surface area similar to the end of a finger. The shape and compactness would potentially allow it both to gain more insight as to its environment and operate in spaces a bulky GelSight sensor could not.

GelSight is not the only variation of optical tactile sensors. TacTip is a closely related implementation, again including both a camera and LED(s). The TacTip uses a modular three-dimensionally printed tip for contact, rather than a gel [39, 40]. Though similar to GelSight, TacTip has seen its own iterations culminating in the development of the TacTip-GR2, TacTip-M2, and TacCylinder sensors [41].

We will be limiting the complexity of our paper by only implementing GelSight. As although there are already several papers simulating tactile sensors in combination with grippers [42], none appear to use Mujoco in combination with a fully realised GelSight sensor.

2.2 Reinforcement Learning

Reinforcement learning is a subset of machine learning which encompasses a rich collection of learning techniques, that utilise various levels of observed data in order to learn in combination with a reward system, with the goal of establishing a model to yield the highest rewards.

Their types include model-based learning, where an agent is either given the model such as in Alphazero's case [43], or where they learn the model such as in Imagine-augmented agents (I2A) [44]. The other skew of Reinforcement learning is model-free learning. These again split into two distinct categories, Policy Optimization and Q-Learning. The former includes algorithms such as Actor Critic (A2C and A3C) [45, 46] and the latter, includes models such as Deep Q-Learning [47]. Notably, there are some algorithms which incorporate elements of both Policy Optimisation and Q-Learning. DDPG is one of these hybrids. A further breakdown of these algorithms can be found in Figure-13 [48].

As mentioned above, we will be using DDQN as a baseline for our training. This class of algorithms requires a neural network to function. One of the most commonly used today is the multilayer perceptron (MLP) - a subset of Deep Neural Networks. MLP's have been adapted to form solutions for a myriad of problems, from Flood susceptibility mapping [49] to ammonium adsorption on zeolite (a mineral typically used as an adsorbent within the commercial space) [50].

A more closely related example is here in Chiddarwar's paper [51], where an MLP is used to train a 6R serial robot. Observing the network structure we can see that the network inputs are, the position and orientation of only the end "effector", its three-dimensional positional space alongside its roll pitch and yaw. The outputs, or actions, the MLP can provide are mapped to the actuation forces the agent can apply to the six-arm joints. A similar methodology is followed in this paper, by James [52], but this time using Deep Q-Learning (DQN) to approximate Q-values for the given inputs. Both of these examples only take in numeric inputs. However, convolution and max-pooling layers will be necessary to feed and evaluate our GelSight images. An example of this is here, by Quillen [53], who uses Deep Reinforcement Learning to train a vision-based grasping robot to pick up objects. A key distinction is that Quillen is modeling based on vision but, we can use this methodology for our GelSight sensors.

Although not related specifically to the elements of our paper, reinforcement learning algorithms surrounding games can also provide some insight. This can be seen in these papers here, by Lample and Hausknecht [54, 55]. Where the aforementioned convolutional layers of a neural network play a crucial part in processing game images, operating towards feature extraction, so that an agent may understand how to choose actions to achieve the best rewards in a complex system. As GelSight is specifically an optical-based tactile sensor, understanding the handling of images is crucial.

As our project requires two separate input types, images and numerical, we will need to intake the inputs and process them separately. After which they can be combined into a single fully connected layer. An example of this can be found in these papers by Gomes and Wang [13, 56]. Both examples are close physical representations of what we wish to create in a simulated environment, utilising arm-held GelSight based grippers to interact with their respective environments.

Deep Q-Learning Evolutions

Deep Q-Learning describes a class of algorithms, which utilise Deep Q-Networks. Although DQN is powerful, it contains a number of weaknesses, [57]. Double Q-Learning [58], prioritized replay [59], dueling neural networks [60], multi-step learning [61], distributed RL [62] and noisy nets [63] each overcome some of the algorithm's flaws. These do not need to be applied exclusively and can be implemented together. This was demonstrated in this paper, by Hessel [57], and has been coined as Rainbow DQN. This algorithm performs far better than any of the others individually. However, DQN houses a major weakness that may affect our agent's ability to learn.

Policy Gradient

With DQN-based methods, the agent is essentially learning the action-value function, or Q-function, which only works for discrete actions. This is because the Q-function is effectively inferring an optimal policy. Policy gradient methods instead learn the action probabilities di-

rectly, in other words, they learn a policy directly. This means when an agent interacts with the environment, the parameters of the associated neural network are nudged so that *better*, more rewarding, actions have a higher chance of being chosen in future iterations. This continues until the network converges to an optimal policy [18]. Essentially this means policy gradient methods are better suited to continuous action spaces - such as the degrees of freedom in our robotic arm.

Because of this, policy gradient methods are generally thought to apply to more problems than DQN. If the Q-function is too complex, DQN will fail but policy gradient may still be able to find a solution. Additionally, policy gradient tends to learn faster than DQN. Though this comes with a trade-off, they are more likely to converge to local optimal solutions rather than a global optimal [18].

Another issue with policy gradient methods is in the gradient update, which uses an estimation of the gradient generated by a series of data points gathered across an episode. This makes estimation very noisy when compared to DQN. There are methods to reduce this weakness, such as offsetting the estimation by a baseline function. This is called Advantage Actor-Critic (A2C) - mentioned above. Where an advantage term is used as a critic to evaluate how good an action is [18, 45].

An evolution of actor-critic incorporates some of the lessons learned from DQN. This is visible in Lillicrap's paper on DDPG [5]. Similar to DDQN, DDPG also uses multiple neural networks, however, DDPG requires four networks instead of two. This is because actor and critic both require two networks each. An example of DDPG in combination with tactile sensing can be found in Melnik's paper [64]. Here Melnik combines DDPG with Hindsight Experience Replay, resulting in increased sample efficiency.

It would be interesting to see what level of DQN is required in order to both provide a competent grasping agent or to outperform DDPG.

2.3 The Purpose of Simulation Learning - Sim2Real

Transferring simulated learning to real models is a trend that is fast becoming a standard. Adopted by many researchers with the aims of reducing time and cost [65, 66, 67]. It not only provides a relatively inexpensive modeling environment but, also enables the rapid collection of training and testing data for a given agent [68].

One of the main challenges within this space is generalisation. The learning acquired in a virtual environment may allow an agent to perform well on synthetic data but that experience may not transfer well to a real agent [21, 65]. However, sometimes further optimisation of simulation parameters can have a dramatic impact, altering the transferable performance from low to high [21].

Although this paper will remain focused solely on the creation and training of a model within a virtual space. It is difficult to ignore sim2real learning. As it provides insight around the motivations of the paper, in addition to theoretical next steps. Because we are not performing sim2real we can ignore various other potential issues, such as accounting for wear, tear, and general degradation of the robot's components.

3 Proposed Design, Specification & Methodology

3.1 Simulated GelSight Implementation

As discussed above, we will be following the methodology laid out in Gomes' paper [3]. Here we can see that in order to implement the sensor, we need to first calculate the elastomer Heightmap, H. This can be obtained by using the depth, D, captured by a depth camera placed at the bottom center of the GelSight sensor. This relationship is defined by the formula below, where x and y relate to coordinates within the captured image [3].

$$H_0(x,y) = \begin{cases} D(x,y) & \text{if } D(x,y) \leq D_{max} \\ \\ d_{max} & \text{otherwise} \end{cases} \quad [3] \tag{1}$$

This step requires tuning of the camera's distance from the lower bound of the GelSight (or the upper bound of the glass) as well as the exact depth of the GelSight. Otherwise, the camera would capture everything within its field of view and not just the objects in contact with the elastomer. A simple solution to this would be to add a thin opaque cover to the exterior of the elastomer. Effectively replicating the role of the reflective paint used in the physical models [2]. As we will cover below, the material and color should have no impact on the effectiveness of our sensor, as our lighting model is simulated separately.

Running the model without any further processing would result in a fairly coarse image. Therefore, further filtering is implemented in order to better approximate the smoother deformations afforded by the physical gel. This takes the form of first applying Gaussian filters, G, across the Heightmap [3].

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}} \quad [3] \tag{2}$$

$$H_t^{'} = H_{t-1} * G \quad [3] \tag{3}$$

Here t represents the current time step, $*$ represents a convolution on the Heightmap, and σ is the standard deviation about G.

$$H_t(x,y) = \begin{cases} H_t'(x,y) & \text{if } H_t'(x,y) \geq H_0(x,y) \\ \\ H_0(x,y) & \text{otherwise} \end{cases} \quad [3] \tag{4}$$

An additional calculation is needed to represent the displacement of gel when an object is pressed firmly against it. In the formula below *narrow* and *wide* use different standard deviations to calculate their heightmaps [3].

$$H_{DoG} = 2H_{narrow} - H_{wide} \ [3] \tag{5}$$

The final step is the creation of the LEDs. These are simulated through Phong's shading model [3, 69].

$$I(x,y) = k_a i_a + \sum_{m \in lights} (k_d(\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s(\hat{R}_m \cdot \hat{V})^\alpha i_{m,s}) \quad [3,\ 69] \tag{6}$$

This formula is made up of three primary components, ambient, diffuse, and specular lighting. The constants k_a, k_d and k_s represent a material's ratio of reflection for each lighting component, respectively. Similarly, i_a, i_d and i_s are respectively the intensity of each light source and α governs overall shininess. Furthermore, we can see that these components are used in conjunction with several directional vectors for each light source. $\hat{L}_m$ is the vector from the point on the surface to each light source, $\hat{N}$ is the normal at this point on the surface, $\hat{V}$ is the direction toward the camera and $\hat{R}_m$ is the reflected ray of light from this surface coordinate [69].

$$\hat{R}_m = 2(\hat{L}_m * \hat{N})\hat{N} - \hat{L}_m \qquad [3] \qquad (7)$$

Figure-3 and Figure-4 depict a GelSight model implementation within Mujoco, here we utilized the 2014 GelSight model STL files [3, 33]. In combination with the formulas described above, we make the glass of the GelSight transparent but with collision enforced so that it will not interfere with the depth camera while allowing the gel to sit upon it. Note that when compared to the real model in Figure-2, the lights have been excluded - as mentioned above they are handled by Phong's illumination model.

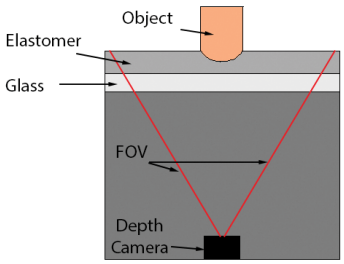


Figure 3: 2D Representation of a Virtual GelSight Camera within the Mujoco environment [3]

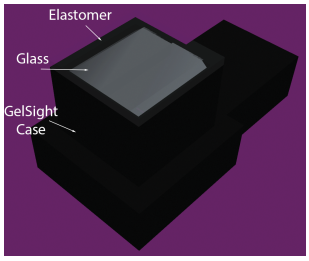


Figure 4: Virtual GelSight Camera - Rendered in Mujoco

The elastomer can then be created using a simple box, which is both transparent and collisionless, allowing objects to occupy the same space as it. Above we have defined a quantitative and simplified approach to limiting this information within the simulated environment - the Heightmap calculation [3] and a thin opaque cover respectively. As the latter enables the physics engine to effectively calculate the Heightmap, it is simplified for a user and potentially more flexible. Therefore, that is the approach we will take. The initial results of this approach can be observed in Figure-6.

Returning to an earlier point, because the illumination model is tuned outside of the Mujoco environment, any reflective qualities assigned to an object will not be transferred and so any material type can be used as a GelSight paint stand-in.

One final consideration is whether Mujoco can handle resistive deformation on collision of an object, rather than simply letting the object clip through the gel stand-in. As mentioned previously, we are not performing sim2real, so this is not a show-stopper for us. However, the lack of this feature would likely represent a significant gap when transferring experience from virtual to physical models.

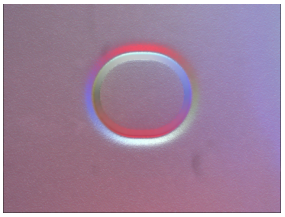


Figure 5: An uncalibrated image taken from the GelSight model within Mujoco. The object is far outside the Heightmap of the elastomer, yet the depth camera still captures the object.

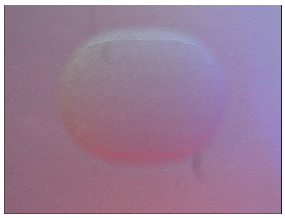


Figure 6: Virtual GelSight Output with an additional Opaque Cover - Mirroring The Gel Paint Layer of a Physical Sensor - as mentioned in [2]

3.2 UR5 Robotic Arm and Gripper Within Mujoco

Within the Mujoco environment, we will set the arm up with a combination of spring-loaded joints, actuators, and friction. This setup will mostly fall into trial and error, unfortunately. Resulting in time-consuming manual calibration.

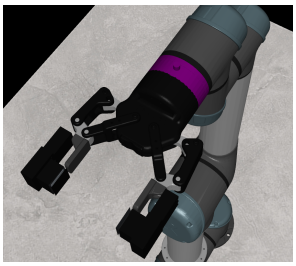


Figure 7: Prototype of a 2-finger GelSight Gripper [70], using a UR5 robotic arm [71] - simulated within Mujoco

The spring-loaded joints will ensure the arm is sturdy when the actuators are both passive and active. This should allow the arm to associate a particular actuation force with a particular position. The added friction will provide a smooth transition between states, eliminating any potential shakiness.

When uninhibitedly paired with a reinforcement learning algorithm, the robot would likely be able to issue several actions prior to the first action reaching a point of equilibrium. Therefore, we will either need to add the current velocities of each joint (to the neural network) or, only allow the robot to make another action after it becomes motionless. The latter simplifies the network, however, it also adds time to each of the agent's steps. It should be reasonably simple to alter the inputs and experiment with both settings.

There are other considerations to make, such as the balancing of inertia, damping, and center mass. However, Mujoco is able to infer the majority of these properties merely from the mass

and arrangement of geometric objects within the kinematic tree. This is something we will be relying on as much as possible.

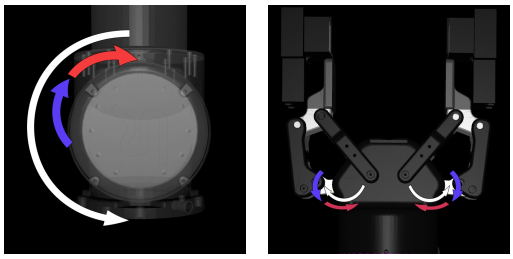


Figure 8: An example of rotational forces which could be applied to an arm joint (LHS) and the gripper fingers (RHS). The colours white, red and blue represent the actuation, spring and friction forces respectively. Note, both gravity and subsequently the weight of additionally attached joints would also affect the position but are excluded from the diagram.

A final concern is that there may be too many joints on the robot arm which will create unnecessary computational complexity, increasing the performance overhead to the point where the system hardware cannot train the model in a time-effective manner or, in the worst case, may not even converge to a solution regardless of time. Therefore, if the need arises, we will reduce the number of joints. This is something we will come back to in the *State Space* section below.

3.3 Reinforcement Learning Model

Double Deep Q-Learning

The initial reinforcement learning algorithm we will be implementing with our model is DDQN [4]. This is an iteration of the standard Deep Q-Learning (DQN) algorithm. DDQN utilises two separate networks for Q-value estimation, where one is online and constantly interacting with the environment and the other is used to update the network weights. This way we separate the workhorse network, allowing for delayed feedback which should reduce the model's sensitivity and overestimationf bias - lowering the chance of it falling into a local optimal [57]. Below are the DQN and DDQN formulas for calculating the updated target values, Y:

$$Y_t^{DQN} \equiv R_{t+1} + \gamma maxQ(S_{t+1}, a; \theta_t^-) \tag{8}$$

$$Y_t^{DDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, argmaxQ(S_{t+1}, a; \theta_t), \theta_t^-) \text{ [4]} \tag{9}$$

Here s_t, a_t and r_t represent the state, action and reward at time t respectively. θ are the weights of online neural network, θ_t^- are the weights of the target network and γ is an arbitrarily selected decay rate. Below is the DDQN algorithm we will use for the training of our agent. It includes the DDQN update calculation mentioned above as well as a formula for updating the target network θ'. Here, α represents the learning rate.

Algorithm 1 – Double Deep Q-Learning [4]

```
Initialise Agent
Initialise Target and Evaluation networks
Initialise ReplayBuffer
Initialise Environment

for each episode do
    for each step do
        Get the observation state
        Pass this state to the neural network so that it can choose an action
        Record the resulting state and reward
        Store these results into the ReplayBuffer
    end for

    for each update do
        if ReplayBuffer's size > Batch size then
            Take a batch sample from the ReplayBuffer
            Calculate target Q-value:
                Y_t^{DDQN} ≡ R_{t+1} + γQ(S_{t+1}, argmaxQ(S_{t+1}, a; θ_t), θ_t^-) [4]
            Update the target network:
                θ_{t+1} = θ_t + α(Y_t^Q − Q(S_t, A_t; θ_t))∇_{θ_t} Q(S_t, A_T; θ_t) [4]
        end if
    end for
end for
```

Deep Deterministic Policy Gradient

As mentioned previously, DDPG requires four networks. Two for the critic and two for the actor. The update rule for the actor-network is [5]:

$$\nabla_{\theta\mu} J \approx E_{s_t \sim \rho^\beta}[\nabla_{\theta\mu} Q(s, a|\theta^Q)_{s=s_t, a=\mu(s_t|\theta^\mu)}] \qquad [5] \qquad (10)$$

Here $\nabla_{\theta\mu} J$ is the gradient of the cost function J, with respect to network parameters of the actor-network, μ. Similarly, θ^μ denotes the critic's network parameters. Effectively, we randomly sample states from the agent's memory and use the actor to determine actions for those states. We then plug those actions into the critic to get a value and take the gradient with respect to the actor-network parameters [5].

The critic network is more closely aligned to DQN, the loss function defined is a mean squared error (MSE) between the target value of y_i and Q for the current state-action pair [5]:

$$L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \qquad [5] \qquad (11)$$

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \qquad [5] \qquad (12)$$

Effectively, we randomly sample states, next states, actions, and rewards and use the target actor-network to determine actions for the new states. We then insert those actions into the target critic to get the target value, y. We then finally insert the state-action pairs into the critic network to calculate the delta [5].

This leaves us with the update rules for the target networks. The initial values will be randomised and every other step will be updated using the below formulas, where τ is some arbitrary

small value [5]:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \qquad [5] \qquad (13)$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \qquad [5] \qquad (14)$$

Neural Network Structure

The neural networks will need to facilitate multiple input types. As we will be taking in numerical data, from joint positions, and also the pixel matrices of images, captured by the Gelsight sensors. The latter will consist of, *(height, width, channel)* and will adopt the common practice of stacking frames on top of one another. This will allow the network to interpret further things from the environment, giving it context as to the movement of any objects the GelSight is pressing against [57, 72]. Another way around this might be to incorporate some form of recurrent network layer (RNN) [73]. Variations of RNNs include, but at not limited to, Long-Short-Term-Memory (LSTM) or Gated Recurrent Units (GRU), which effectively use activation functions (or gates) to keep or forget information. These have previously been shown to have great success in learning to play games, such as Doom and Pong [54, 55]. It's currently unclear as to whether the images of both GelSight sensors could be combined into a single array using separate channels, or if it would be more beneficial to have two separate convolution inputs for each GelSight.

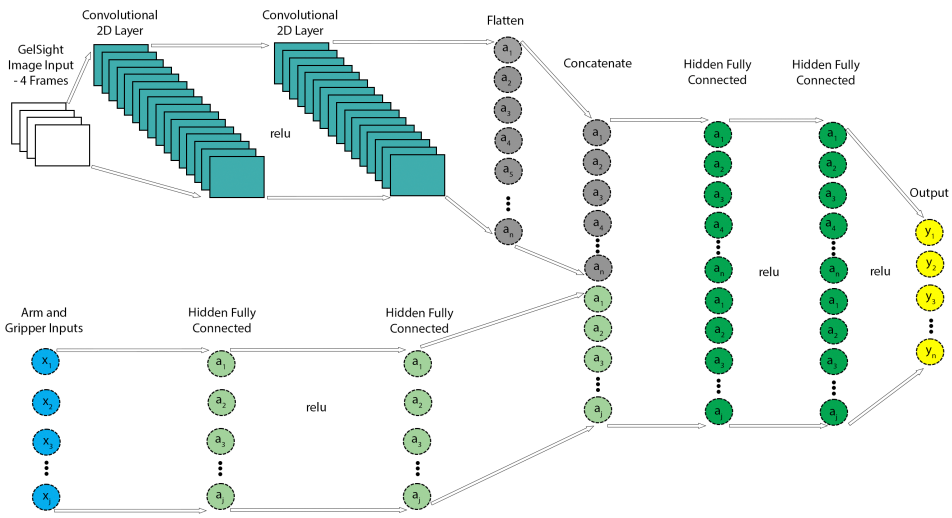


Figure 9: Sample Proposed Neural Network Model [9, 13, 56]

Image inputs within neural networks are typically both down-scaled and gray-scaled. This is done to significantly reduce the performance requirements without impacting the prediction performance of the model produced. The latter operation is curious, as this GelSight implementation specifically relies on different coloured LEDs, it would be interesting to see if a non-gray-scaled version performs any differently.

As for the core structure of the neural network, the robotic arm and gripper are numerical and can be forwarded immediately into fully connected layers. Unlike the image inputs which

must have feature extraction performed on them first. Feature extraction takes the form of convolutional layers, and possibly max-pooling to reduce the complexity, before being flattened and combined with the other fully connected layers. This can be seen in Figure-10 [13, 56].

The structure is effectively a hyper-parameter that needs to be tuned through experimentation. Hyper-parameters to consider are the number of neurons or convolutional filters in each layer, the choice of optimiser, the intensity of the learning rate, the decay of that learning rate, the batch size, the number of epochs, the number of layers, and the type of activation functions. This offers a great deal of customisability, which should be exploited during the experimental phase.

State Space

The state space is a crucial component of Gym. It is effectively an agent's window into the current state of the environment. For our model, it will provide feedback as to the current joint positions and actuation forces applied to them. As defined above, these partially make up the network inputs.

Traditionally, within Gym environments, there is also a *done* attribute. This can signify general states of the environment as a whole, for a game this signifies completion or when reaching a new level. In our model, it makes sense to change the state to done after we have picked up the target object - moving it above the z-axis.

Action Space

For DDQN to function, the action space must constitute only discrete actions an agent may take within the environment. These actions are not necessarily limited to moving a single joint at a time but, can also consist of combinations of actions. Using the latter across all actions would increase the space considerably. This is not ideal as smaller action spaces typically result in, not only faster but also, increased likelihood of convergence [74].

The arm and gripper have several joints each. However, the only actionable joints are those with actuators attached to them. This leaves us with two and six joints for the gripper and arm respectively. At least as a baseline, our action space will be reduced to the smallest form - allowing adjustment of a single actuator only for each step. This gives us an action space of sixteen (eight joints in two directions), however, we can reduce this by linking the gripper joints - moving the finger synchronously. Therefore, our action space is fourteen.

With regards to how the model moves from state to state using actions, this should be done by offsetting the previous state by whatever action is selected. Therefore, if joint x_1 is selected to be shifted in the first step and then joint x_2 is selected in the second step, the first joint should remain unchanged. This will allow the arm to perform slight adjustments in each state. In other words, the action a at step n is the accumulation of all actions taken so far.

$$a_n = a_n + a_{n-1} + a_{n-2} + ... + a_0 \tag{15}$$

As DDPG is more flexible and is able to handle continuous action spaces, the above may need to change to accommodate it. However, I am currently uncertain as to how this will look.

Reward Structure

The initial structure we will use will attempt to encourage the robot to grasp and pick up an object, and do so quickly. To that end, every time step will be recorded as a negative value. This will hopefully instantiate a sense of urgency as the neural network evolves. A reward value will be offered if either the GelSight sensors come into contact with the target object, this reward will stack if both sensors are in contact simultaneously. And the final reward should be if the target object moves positively in the z-axis, while in contact with the GelSight.

It is important that these rewards do not make the overall step reward positive, as this could allow the agent to exploit the interaction - possibly learning to only grasp the object without moving it upwards, or something else unexpected. This paper, by Wu [75], suggests ending the episode as soon as the object is picked up. However, I want to be certain the robot understands that the goal is to actually hold the object. Therefore, the simulation will end if the object is held above the planar origin for a specified amount of time. This time will initially be set to five seconds.

Finally, there will be an end state following an arbitrary number of steps passed. This is to ensure, especially early on, that an episode that will not find the success condition does not carry on indefinitely.

3.4 Evaluation Approach

Training and Testing the Model

In order to train and test our model effectively, we will need a substantial amount of varied environments. Ideally, the agent would be able to search the environment around itself, detect the presence of a novel object through tactile feedback from the GelSight sensors, grasp it, and finally raise it above the ground. However, this may prove too time-consuming for the project. Therefore, to begin with, the training environments will limit the objects to a single location and cycle through various objects. This type of generation should be fairly easy as it requires simply modifying XML files, exchanging the names of STL files or predefined box objects.

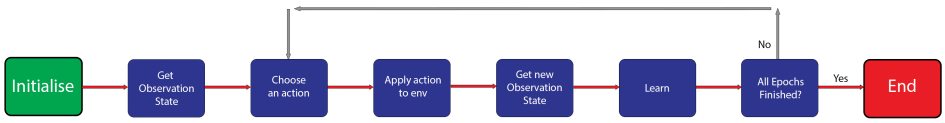


Figure 10: Training Process Flow

The test data can be set up in the same manner. Though, care needs to be taken to keep these objects exclusive and unseen. As this will give us a better appreciation for how well the model generalises.

Our metrics to measure the evolution of the agent should take the form of; the number of episodes that ended early due to a successful pickup; the average, minimum and maximum times for these episodes; the amount of time the target object spent above the floor; and the amount of time the GelSight spent in contact with the target object. These should be taken at both consistent intervals during training and once for the test data-set.

Based on the above discussion, it is assumed DDPG will outperform DDQN and any other Q-Learning algorithms we have time to implement.

4 Plan

Our group has been meeting weekly on Fridays and intends to continue doing so throughout the project. This allows for general discussion and guidance around the topic as well as an opportunity to raise any roadblocks and amend the project's course if necessary.

The project contains five main streams of work, which can be broken down into four practical tasks and one written. The ordering of the practical tasks is fairly linear, going from an initial setup of the software required for the paper, to the implementation of a GelSight sensor with Mujoco, followed by the implementation of a two-finger gripper, before finally applying DDQN and DDPG to the model. The sub-tasks of each workstream vary in size and complexity and this has been taken into account during the planning stages.

Also as part of the project, I have constructed a Gantt chart to framework progress over the next few weeks, which can be found below in the Appendix. The chart is colour coded in order to establish which tasks are completed, in progress, yet to be started, or overdue. This will be updated as more sub-tasks naturally become apparent as the project continues. Also worth noting is that several stages have already been completed, including the research and implementation of a basic GelSight Sensor within Mujoco, as well as the attachment of the sensors to the robotic arm and gripper. Various elements of this work can be seen in the above sections.

Figure-11 defines an adapted take on mathematical modeling from this book [76]. This recurrent improvement model has already been utilised to complete the aforementioned steps. The intention is to continue this approach throughout the development of the test-bed.

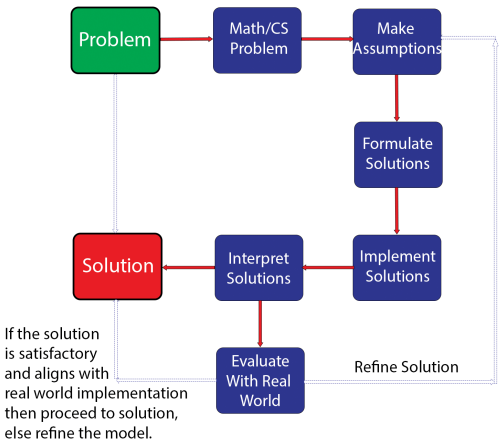


Figure 11: Proposed Mathematical Modelling Process To Be Utilised When Defining and Iterating On The Proposed Implementations of The Project - Adapted From Figure 1-1 of [76]

References

[1] Shan Luo, Joao Bimbo, Ravinder Dahiya, and Hongbin Liu. Robotic tactile perception of object properties: A review. *Mechatronics*, 48:54–67, 2017.

[2] Micah K Johnson and Edward H Adelson. Retrographic sensing for the measurement of surface texture and shape. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1070–1077. IEEE, 2009.

[3] Daniel Fernandes Gomes, Paolo Paoletti, and Shan Luo. Generation of gelsight tactile images for sim2real learning. *IEEE Robotics and Automation Letters*, 6(2):4177–4184, 2021.

[4] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[5] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[6] A Hussain, A Malik, MU Halim, and AM Ali. The use of robotics in surgery: a review. *International journal of clinical practice*, 68(11):1376–1382.

[7] Patrick Lin, George Bekey, and Keith Abney. Autonomous military robotics: Risk, ethics, and design. Technical report, California Polytechnic State Univ San Luis Obispo, 2008.

[8] Jinda Cui and Jeff Trinkle. Toward next-generation learned robot manipulation. *Science Robotics*, 6(54), 2021.

[9] Edward Adelson. Session 3 - (keynote) - the future of camera-based tactile sensing. *ViTax WorkShop ICRA*, 2021. URL `https://www.youtube.com/watch?v=IuCDAjhDnak`.

[10] Wenzhen Yuan, Siyuan Dong, and Edward H Adelson. Gelsight: High-resolution robot tactile sensors for estimating geometry and force. *Sensors*, 17(12):2762, 2017.

[11] Gerd Häusler and Svenja Ettl. Limitations of optical 3d sensors. In *Optical measurement of surface topography*, pages 23–48. Springer, 2011.

[12] Alexander C Abad and Anuradha Ranasinghe. Visuotactile sensors with emphasis on gelsight sensor: A review. *IEEE Sensors Journal*, 20(14):7628–7638, 2020.

[13] Daniel Fernandes Gomes, Achu Wilson, and Shan Luo. Gelsight simulation for sim2real learning. In *ICRA ViTac Workshop*, 2019.

[14] Edward J Smith, Roberto Calandra, Adriana Romero, Georgia Gkioxari, David Meger, Jitendra Malik, and Michal Drozdzal. 3d shape reconstruction from vision and touch. *arXiv preprint arXiv:2007.03778*, 2020.

[15] Arpit Agarwal, Tim Man, and Wenzhen Yuan. Simulation of vision-based tactile sensors using physics based rendering. *arXiv preprint arXiv:2012.13184*, 2020.

[16] Gazebo. 2021. URL `http://gazebosim.org/`.

[17] Jonas Schneider Jonathan Ho Peter Welinder Wojciech Zaremba Jerry Tworek Alex Ray, Bob McGrew. mujoco-py. 2018. URL `https://github.com/openai/mujoco-py`.

[18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[19] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

[20] Fereshteh Sadeghi, Alexander Toshev, Eric Jang, and Sergey Levine. Sim2real viewpoint invariant visual servoing by recurrent control. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4691–4699, 2018.

[21] Abhishek Kadian, Joanne Truong, Aaron Gokaslan, Alexander Clegg, Erik Wijmans, Stefan Lee, Manolis Savva, Sonia Chernova, and Dhruv Batra. Are we making real progress in simulated environments? measuring the sim2real gap in embodied visual navigation. 2019.

[22] Fereshteh Sadeghi, Alexander Toshev, Eric Jang, and Sergey Levine. Sim2real view invariant visual servoing by recurrent control. *arXiv preprint arXiv:1712.07642*, 2017.

[23] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.

[24] Open AI. Openai gym. 2016. URL `https://gym.openai.com/`.

[25] Mujoco - advanced physics simulation. *mujoco200 win64*, 2018. URL `http://www.mujoco.org/`.

[26] J Dargahi. A piezoelectric tactile sensor with three sensing elements for robotic, endoscopic and prosthetic applications. *Sensors and Actuators A: Physical*, 80(1):23–30, 2000.

[27] Karsten Weiß and Heinz Worn. The working principle of resistive tactile sensor cells. In *IEEE International Conference Mechatronics and Automation, 2005*, volume 1, pages 471–476. IEEE, 2005.

[28] Hong-Ki Kim, Seunggun Lee, and Kwang-Seok Yun. Capacitive tactile sensor array for touch screen application. *Sensors and Actuators A: Physical*, 165(1):2–7, 2011.

[29] Ren-Chyuan Luo, Fuling Wang, and You-xing Liu. An imaging tactile sensor with magnetostrictive transduction. In *Intelligent Robots and Computer Vision*, volume 521, pages 264–270. International Society for Optics and Photonics, 1985.

[30] Wendong Zheng, Bowen Wang, Huaping Liu, Yunkai Li, Ran Zhao, Ling Weng, and Changgeng Zhang. Structural design and output characteristic analysis of magnetostrictive tactile sensor for robotic applications. *AIP Advances*, 8(5):056622, 2018.

[31] Josivaldo Godoy da Silva, Aparecido Augusto de Carvalho, and Doriedson Dutra da Silva. A strain gauge tactile sensor for finger-mounted applications. *IEEE Transactions on Instrumentation and measurement*, 51(1):18–22, 2002.

[32] Wenzhen Yuan, Rui Li, Mandayam A Srinivasan, and Edward H Adelson. Measurement of shear and slip with a gelsight tactile sensor. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 304–311. IEEE, 2015.

[33] Rui Li, Robert Platt, Wenzhen Yuan, Andreas ten Pas, Nathan Roscup, Mandayam A Srinivasan, and Edward Adelson. Localization and manipulation of small parts using gelsight tactile sensing. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3988–3993. IEEE, 2014.

[34] Wenzhen Yuan, Mandayam A Srinivasan, and Edward H Adelson. Estimating object hardness with a gelsight touch sensor. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 208–215. IEEE, 2016.

[35] Achu Wilson, Shaoxiong Wang, Branden Romero, and Edward Adelson. Design of a fully actuated robotic hand with multiple gelsight tactile sensors. *arXiv preprint arXiv:2002.02474*, 2020.

[36] Elliott Donlon, Siyuan Dong, Melody Liu, Jianhua Li, Edward Adelson, and Alberto Rodriguez. Gelslim: A high-resolution, compact, robust, and calibrated tactile-sensing finger. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1927–1934. IEEE, 2018.

[37] Branden Romero, Filipe Veiga, and Edward Adelson. Soft, round, high resolution tactile fingertip sensors for dexterous robotic manipulation. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4796–4802. IEEE, 2020.

[38] Daniel Fernandes Gomes, Zhonglin Lin, and Shan Luo. Geltip: A finger-shaped optical tactile sensor for robotic manipulation. *arXiv preprint arXiv:2008.05404*, 2020.

[39] Zihan Ding, Nathan F Lepora, and Edward Johns. Sim-to-real transfer for optical tactile sensing. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1639–1645. IEEE, 2020.

[40] René Traoré, Hugo Caselles-Dupré, Timothée Lesort, Te Sun, Natalia Díaz-Rodríguez, and David Filliat. Continual reinforcement learning deployed in real-life using policy distillation and sim2real transfer. *arXiv preprint arXiv:1906.04452*, 2019.

[41] Benjamin Ward-Cherrier, Nicholas Pestell, Luke Cramphorn, Benjamin Winstone, Maria Elena Giannaccini, Jonathan Rossiter, and Nathan F Lepora. The tactip family: Soft optical tactile sensors with 3d-printed biomimetic morphologies. *Soft robotics*, 5 (2):216–227, 2018.

[42] Tarık Keleştemur, Colin Keil, John P Whitney, Robert Platt, and Taşkın Padır. Learning bayes filter models for tactile localization. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9253–9258. IEEE, 2020.

[43] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[44] Sébastien Racanière, Théophane Weber, David P Reichert, Lars Buesing, Arthur Guez, Danilo Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 5694–5705, 2017.

[45] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

[46] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[47] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[48] OpenAI. Part 2: Kinds of rl algorithms. 2018. URL `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html`.

[49] Yi Wang, Zhice Fang, Haoyuan Hong, Romulus Costache, and Xianzhe Tang. Flood susceptibility mapping by integrating frequency ratio and index of entropy with multilayer perceptron and classification and regression tree. *Journal of Environmental Management*, 289:112449, 2021.

[50] Fulya Aydın Temel, Özge Cağcağ Yolcu, and Ayşe Kuleyin. A multilayer perceptron-based prediction of ammonium adsorption on zeolite from landfill leachate: Batch and column studies. *Journal of Hazardous Materials*, 410:124670, 2021.

[51] Shital S Chiddarwar and N Ramesh Babu. Comparison of rbf and mlp neural networks to solve inverse kinematic problem for 6r serial robot by a fusion approach. *Engineering applications of artificial intelligence*, 23(7):1083–1092, 2010.

[52] Stephen James and Edward Johns. 3d simulation for robot arm control with deep q-learning. *arXiv preprint arXiv:1609.03759*, 2016.

[53] Deirdre Quillen, Eric Jang, Ofir Nachum, Chelsea Finn, Julian Ibarz, and Sergey Levine. Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6284–6291. IEEE, 2018.

[54] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[55] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 aaai fall symposium series*, 2015.

[56] Chen Wang, Shaoxiong Wang, Branden Romero, Filipe Veiga, and Edward Adelson. Swingbot: Learning physical features from in-hand tactile exploration for dynamic swing-up manipulation. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5633–5640. IEEE, 2020.

[57] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[58] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23: 2613–2621, 2010.

[59] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[60] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[61] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[62] Martin Lauer and Martin Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *In Proceedings of the Seventeenth International Conference on Machine Learning*. Citeseer, 2000.

[63] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[64] Andrew Melnik, Luca Lach, Matthias Plappert, Timo Korthals, Robert Haschke, and Helge Ritter. Using tactile sensing to improve the sample efficiency and performance of deep deterministic policy gradients for simulated in-hand manipulation tasks. *Frontiers in Robotics and AI*, page 57, 2021.

[65] Carl Doersch and Andrew Zisserman. Sim2real transfer learning for 3d human pose estimation: motion to the rescue. *Advances in Neural Information Processing Systems*, 32: 12949–12961, 2019.

[66] Alexander Pashevich, Robin Strudel, Igor Kalevatykh, Ivan Laptev, and Cordelia Schmid. Learning to augment synthetic images for sim2real policy transfer. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2651–2657. IEEE, 2019.

[67] Ofir Nachum, Michael Ahn, Hugo Ponte, Shixiang Gu, and Vikash Kumar. Multi-agent manipulation via locomotion using hierarchical sim2real. *arXiv preprint arXiv:1908.05224*, 2019.

[68] Olivier Michel. Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):5, 2004.

[69] Frederico Francisco, Orfeu Bertolami, Paulo JS Gil, and Jorge Páramos. Modelling the reflective thermal contribution to the acceleration of the pioneer spacecraft. *Physics Letters B*, 711(5):337–346, 2012.

[70] ROBOTIQ. 2f-85 adaptive gripper. 2019. URL `https://robotiq.com/`.

[71] Travis DeWolf. Ur5 6dof robotic arm. 2020. URL `http://www.mujoco.org/forum/index.php?resources/ur5-6dof-robotic-arm.27/`.

[72] Aravind Srinivas, Michael Laskin, and Pieter Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning. *arXiv preprint arXiv:2004.04136*, 2020.

[73] Hasim Sak, Andrew W Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. 2014.

[74] Gellért Weisz, Paweł Budzianowski, Pei-Hao Su, and Milica Gašić. Sample efficient deep reinforcement learning for dialogue systems with large action spaces. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26(11):2083–2097, 2018.

[75] Bohan Wu, Iretiayo Akinola, Jacob Varley, and Peter Allen. Mat: Multi-fingered adaptive tactile grasping via deep reinforcement learning. *arXiv preprint arXiv:1909.04787*, 2019.

[76] K. Ang. Teaching and learning mathematical modelling with technology. 2010.

5 Appendix

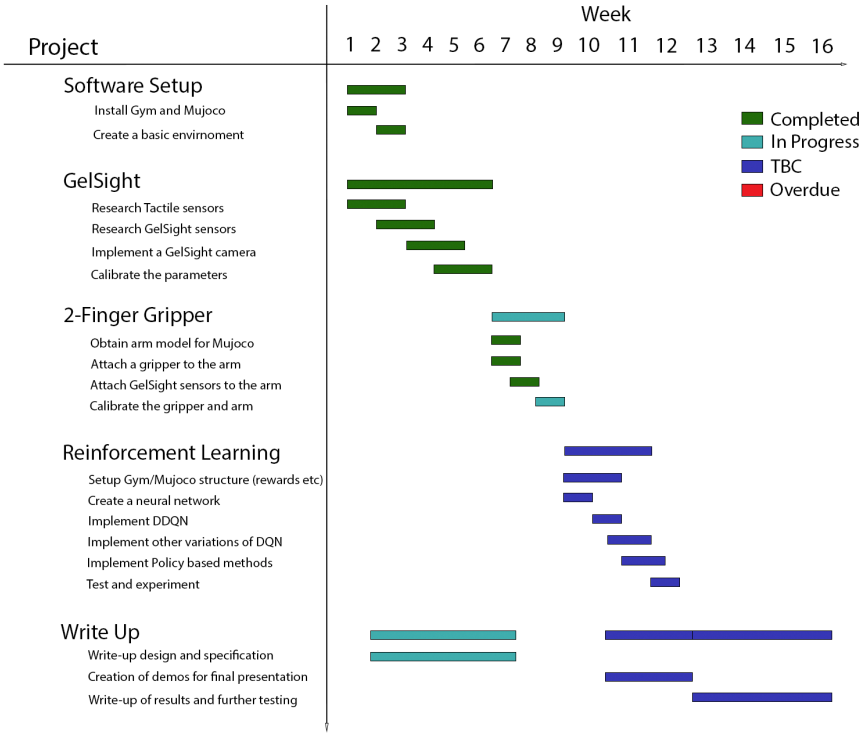


Figure 12: A Gantt Chart Displaying Progress Throughout the Project

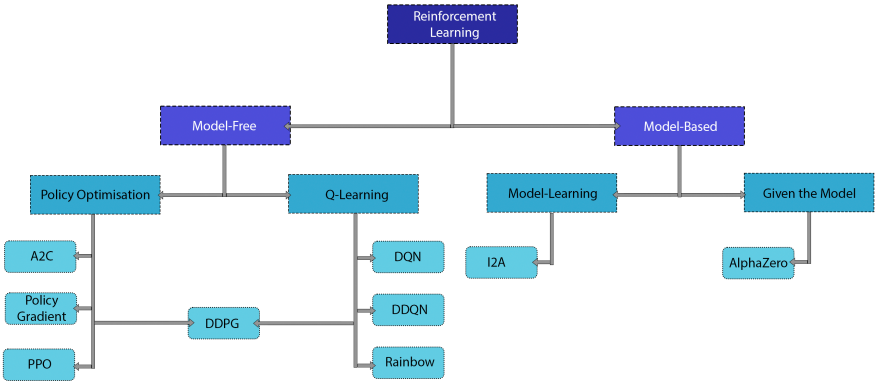


Figure 13: A breakdown of reinforcement based learning models adapted from OpenAI's documentation [48]. This is not inclusive of all reinforement learning algorithms.

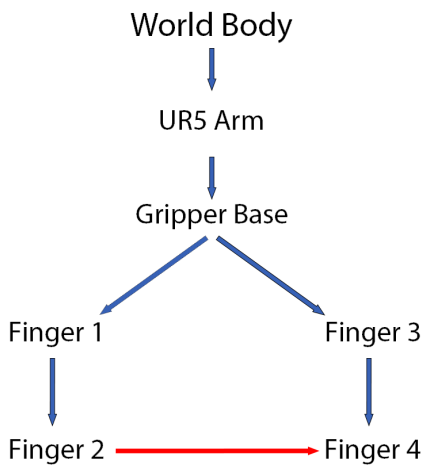


Figure 14: Mujoco Joint Body Model. The Blue Lines represent valid body nesting(s) whereas the red line represents an illegal join. This is due to the requirement of a loop join, where a child body requires two parents. This is solved through use of an equality, outside the kinematic tree.

Appendix E
Final Presentation - Report

This Appendix item is a complete copy of the project's the originally submitted report alongside the final presentation - starting on the next page.

OpenAI Test-Bed of Contact Based Grasping for Reinforcement Learning

Final Presentation - Report

September 2021

Rob Lamprell

Abstract

This project has successfully replicated Gomes' Gelsight sensor within Mujoco [1, 2]. This is a unique contribution and has not been done previously. Following this, we produced a test-bed for a two-fingered-gelsight tactile sensor attached to a UR5 arm [3, 4]. This testbed consists of four Mujoco-based environments, similar to those found within OpenAI's gym [5]. The final code structure is reasonably modular. Separating the elements into five distinct classes, *environment, neural network, replaybuffer, ddqn model and agent.* This modular design should allow for additional elements to be incorporated in the future, such as new environments or reinforcement learning algorithms.

In contrast to the original design proposal, Hierarchical Double-Deep Q-Learning [6, 7] was implemented instead of Deep Deterministic Policy Gradient [8]. As it was thought that this algorithm would better handle the sparse rewards in environments with random object placement. Since Q-learning-based methods cannot handle continuous action spaces, like Policy Gradient methods can, the action space has remained discrete [9].

Additionally, a revised neural network model has been implemented. The original model consisted of only two 2D-convolutional layers for feature extraction. Comparing to other similar projects, it could be seen that our model was quite shallow in comparison. Therefore, a pre-trained model, specifically ResNet50v2, was employed to better mirror previous works [10, 11]. We exclusively use the pre-trained weights from ResNet's 50 layer solution for feature extraction on both our vision and tactile observations [12].

After successfully implementing the above, baseline tests using both reinforcement learning algorithms were completed - constituting a proof of concept. However, due to both the time required to train and the time-limited nature of this project, it has been difficult to demonstrate convergence on sparse reward environments with complex observation spaces.

1 Summary of Updated Proposal

Our original plan was to construct a Mujoco [2] simulation of a two-fingered-gelsight-gripper [1] and train it through reinforcement learning algorithms, specifically Double Deep Q-Learning (DDQN) [6] and Deep Deterministic Policy Gradient (DDPG) [8]. The DDQN implementation came to fruition. However, in place of DDPG we instead opted to further augment DDQN by incorporating ideas observed within Hierarchical Deep Q-Learning (HDQN) [7]. Effectively becoming Hierarchical Double Deep Q-Learning (HDDQN) [13].

HDDQN requires nesting a DDQN agent within another DDQN agent, where the outer agent is referred to as extrinsic and the internal is referred to as intrinsic. The extrinsic agent operates similarly to a typical DDQN model, taking observations and rewards directly from the environment. However, it chooses abstract goals instead of direct actions for the robotic arm to take [7].

These goals are then passed to the intrinsic agent, which combines them with its own set of environmental observations. It uses this combined state to choose actions for the robotic arm to perform, with the aim of completing the extrinsic agent's goal [7].

The neural networks of both agents are separate and can vary in structure but do not necessarily need to. One example might be for the extrinsic agent's network to be fed pixel data from a camera that can observe both the target object and the arm, it can then use this information to predict which sub-goal would move the arm closer to the terminal state, or at least which would provide the best reward. By contrast, the intrinsic agent's neural network structure may contain only joint-related inputs, from the robotic arm, and use these to understand how to complete the sub-goal proposed by the extrinsic agent [14]. The fundamental idea is similar to that of a recipe. Where the intrinsic agent is performing all the low-level tasks and the extrinsic agent decides which low-level task the intrinsic agent should be attempting to solve [7, 15, 16].

One of the main influences for this meta-controller structure appears to have been taken from Markov Decision Processes (MDP). Whereby, the available action probabilities are reliant entirely on the current state space. Using these, chains of actions based on state spaces can be created. Using this structure, the model should ideally be able to determine its own goals, based on the rewards provided, and formulate both single and multi-action steps to achieve those goals [7, 16].

Finally, with regards to Mujoco. We have developed multiple environments for our reinforcement learning algorithms to be tested upon. The structure of each element is relatively modular and would ideally allow for further environments, agents or reinforcement learning algorithms to be implemented in the future.

2 Outputs

2.1 Robotic Arm Evolution

Over the course of the project, our Mujoco environment has seen a great number of iterations. Initially, as demonstrated previously in the Design and Specification documentation, we implemented a gelsight sensor within Mujoco and attached it to a UR5 arm. Following on from this DDQN was implemented, and tested, in a simple grasping environment as a proof of concept. This demo space included only the gripper and a floating object, which would fall to the ground as soon as the environment was initialized. In turn, the agent learned to grab the object before it could fall out of reach.

Following this, a table was added for the object to sit upon. A DDQN agent was then trained to grasp and lift the object, using tactile and kinematic observations. Vision (or real) based observations were implemented next - altering the perspective and making the table transparent so as not to occlude the robotic arm [17, 18]. Before finally placing the arm on top of the table. In this final iteration, the perspective was altered so that it could include less 'empty' space and a light source was also incorporated so that both the gripper and target object would cast shadows onto the table. These shadows make it easier to understand the arm's position in the space, relative to the table and should also help close the gap from sim2Real [19].

These evolutionary stages can be seen in Figure-1.

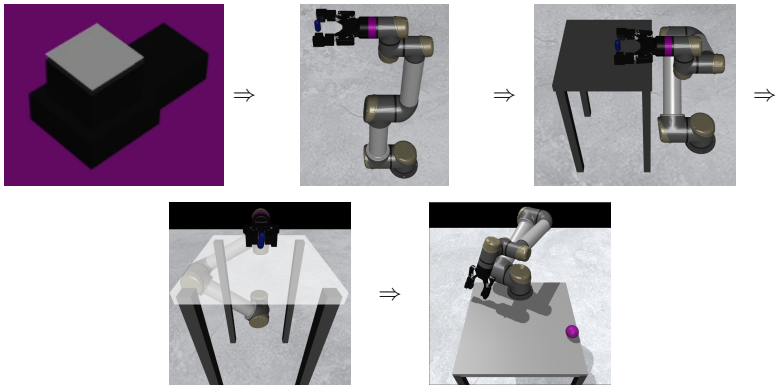


Figure 1: Evolution of the Mujoco environment utilized for our test-bed. Starting with the implementation of Gomes' Gelsight sensor within Mujoco. Followed by the installation and appending of those sensors to a UR5 arm and gripper. The target object was then placed on top of a table, which was changed into a glass table to avoid occlusion in vision observations. Before finally placing the arm on top of the table alongside the object.

2.2 Base Environment

This project sees the development of four environments, each of which all inherit from the base environment class. This class acts as the main workhorse, containing all but some of the goal-specific methods necessary for HDDQN. The constructor of the base environment is able to adapt its methods dependent on the inputs provided, such as whether the environment contains an object to manipulate. This approach is similar to those found within OpenAI's gym fetch environments [5].

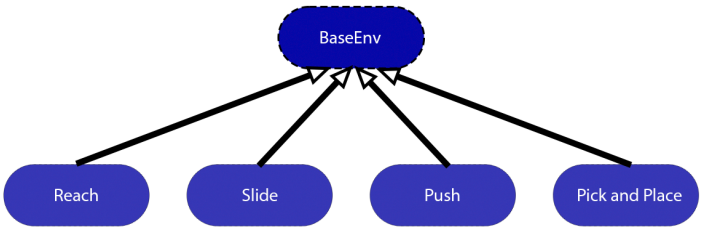


Figure 2: Environments inheriting from the base class.

2.3 The four environments

As mentioned previously, four environments have been developed for the tactile grasping test-bed. These include Pick-and-place, Slide, Push and Reach. Each environment contains a two-fingered-robotic-arm and either a target object and target position or, in Reach's case, simply a target position to maneuver towards. Our Mujoco solution inherits heavily from both OpenAI's Gym and the *mujoco_py* wrapper. These solutions offer basic functionalities such as rendering the environment and providing a physics platform to run our simulations upon [2, 5, 20].

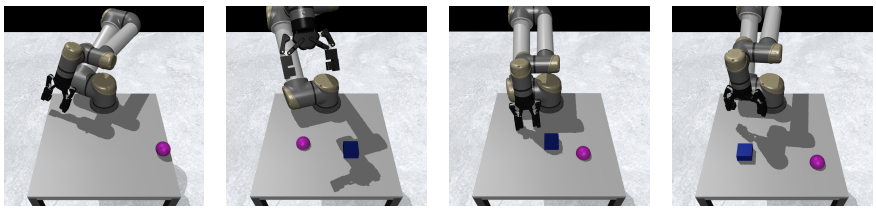


Figure 3: The Four environments reach, pick and place, slide, and push developed for our test-bed (left-to-right). The latter three contain a target object which must reach the target position to be successfully completed. Whereas reach only contains a target position for the gripper to reach. The viewpoint shown is the same as the agent's - in the case of vision.

Reach

Reach, is our simplest environment, only requiring the gelsight fingers to come into contact with the target position. Because of this, reach was used as a development platform when creating and tuning the various methodologies associated with the test-bed - covered later in the paper as well as in the associated video.

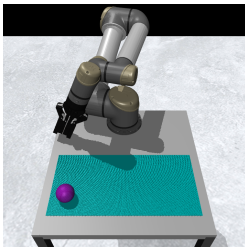


Figure 4: A Reach environment render. The two-dimensional teal geom placed on the table displays the valid spawn locations for both target objects and target positions. This is only necessary when randomizing the placement of objects in each episode. This approach reduces the state space and helps avoid terminal states at episode initialization.

Slide

Slide requires moving the target object to the target position's space. The gripper is not able to grasp the object, instead, it is only able to slide it across the table with a low coefficient of friction. To realize this, only x and y-axis joints have been enabled within the model. As there is no z-axis the table and the target object do not actually contact one another. This has the benefit of no friction between the two geoms, however, this also means there are no inbuilt means to calculate if the object is slid off the table's surface. Therefore, an additional method was created which takes the bottom-left and top-right surfaces of the table as planar and, at each time step, calculates whether the object has moved outside of these boundaries.

Push

Push is very similar to Slide, however, it requires a coefficient of friction between the table and the object. This can be produced through the physics environment by including a z-axis for the target object but, ensuring to restrict its movements so that the object cannot be lifted. Given

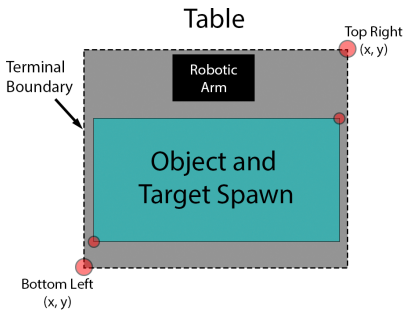


Figure 5: A diagrammatic representation of the table within the Slide environment. The slide environment is unique in that the object is unable to fall due to the lack of a z-axis. Therefore, a terminal boundary is checked for at each time step. This treats the table's surface as planar and compares the object's x and y coordinates to the bottom-left and top-right corners of the table.

the same start and end positions of the object, it is likely that the addition of friction would result in the Push environment needing more interactions with the target object per episode (than Slide).

Pick and Place

PickAndPlace is the most complex environment, combining elements from each of the above. Similar to Slide and Push, the agent needs to move the target object to the target position. However, the target object now has a free joint. This means that the object can move freely with six degrees of freedom.

2.4 Environment Returns

Observations

Observations, or states, act as our agent's insight as to what is currently happening within the environment. Our test-bed has four observation types available to each of the four environments, which take the form of real, tactile, kinematics or raw. Our test-bed solution is flexible and can output and utilize combinations of these four observations.

Rewards

Rewards are our agent's only means of assessing the quality of actions they take within the environment. In the initial stages of development, a complex reward structure was developed. The agent would receive rewards if the gelsight sensors came into loose contact with the target object (c_t^s) and would receive a further reward if it grasped the object more firmly (c_t^h). It also would receive a reward if the object was held above its original z-coordinate (l_t). Finally, if the object was held above its origin for more than five consecutive environment steps, the episode would be successful and the agent was heavily rewarded (s_t). Mathematically, the reward function could be defined as:

$$r_t = c_t^s + c_t^h + l_t + s_t + f_t + o \tag{1}$$

Where r_t is the overall reward for the step at time t. Each attribute and its values are defined in the formulas below:

$$c_t^s = \begin{cases} +1, & \text{if one sensor is in soft contact} \\ +2, & \text{if both sensors are in soft contact} \\ 0, & otherwise \end{cases} \tag{2}$$

$$c_t^h = \begin{cases} +2, & \text{if one sensor is in hard contact} \\ +4, & \text{if both sensors are in hard contact} \\ 0, & otherwise \end{cases} \tag{3}$$

$$l_t = \begin{cases} +2, & \text{if } c_t^h = 2 \textbf{ and } obj_z > \text{start_position}_z \\ 0, & otherwise \end{cases} \tag{4}$$

$$s_t = \begin{cases} +1000, & \text{if } (l_{t-1}, l_{t-2}, l_{t-3}, l_{t-4}, l_{t-5}) = true \\ 0, & otherwise \end{cases} \tag{5}$$

$$f_t = \begin{cases} -1000, & \text{if } obj_z <= 0 \\ 0, & otherwise \end{cases} \tag{6}$$

$$o = \begin{cases} -10 \end{cases} \tag{7}$$

The final attribute is the offset, o. The selection of this value was not arbitrary, as it would offset the total reward so that the agent is unable to receive a positive reward unless the episode is terminated early from reaching the success criteria. Similarly, the failed terminal state (f_t), which is triggered if the target object contacts the floor, needed to be lower than if the agent only received the offset value for the entire episode. Otherwise, an association may be made that knocking the object off the table is better than doing nothing.

Although this reward model was successful in producing reasonable results in static environments, it struggled to converge when object positions were randomized. Presumably, this was due to the rewards being static rather than an effective magnitude of success. Additionally, the model is bespoke to a single environment and although this is a known issue in reinforcement learning [21, 22], it was decided to utilize gym's normalized Euclidean distance solution. As this not only has the benefit of being more readable and offering a lower time complexity across the duration of the training, it is also more general and can be more easily applied to the four environments without the need for bespoke reward modeling.

$$distance(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \qquad [5,\ 23] \tag{8}$$

Done

Done, or the terminal state, defines when our simulation should end - either by success or failure. There is a one-hundred-step limit imposed on all environments. This allows the agent to make one hundred actions before the environment ends. Having an arbitrary limit ensures the agent is not indefinitely stuck in an unwinnable environment [24]. The terminal state criteria followed a similar development cycle to rewards, initially taking a bespoke form before settling on the gym-inspired solution of measuring the euclidean distance between the sites - ending the environment once they are within a certain threshold of one another. These sites are placed within the target object, the target position, and at the gripper's end effector [5].

Info

Info provides additional information as to what is occurring within the environment. Some notable inclusions are the actuation forces currently applied to the robotic arm's joints, the number of steps taken in the environment, the number of steps the gripper was in contact with the target object and the number of steps the gripper has lifted the object.

2.5 Neural Networks

The initial neural network structure we proposed in the Design and Specification document can be seen in Figure-6. Here we process the two Gelsight inputs, separately utilizing two 2D-convolutional layers for feature extraction before combining them with joint-related features taken directly from the robotic arm. However, this implementation was found to be lacking as the structure is possibly too shallow. Which resulted in an inability to adequately identify features within the observation spaces. Additionally, training the model weights from scratch each time meant a potentially unrealistic time required for training. As an example, Kostrikov's paper trains Soft Actor-Critic on gym's Cartpole-Swingup environment utilizing vision observations. This required two million steps to solve. [25].

Therefore, in our final solution, we have opted to utilize transfer learning, replacing our convolutional layers with the pre-trained model ResNet50V2 [10, 12, 26, 27]. This updated model can be found in Figure-7.

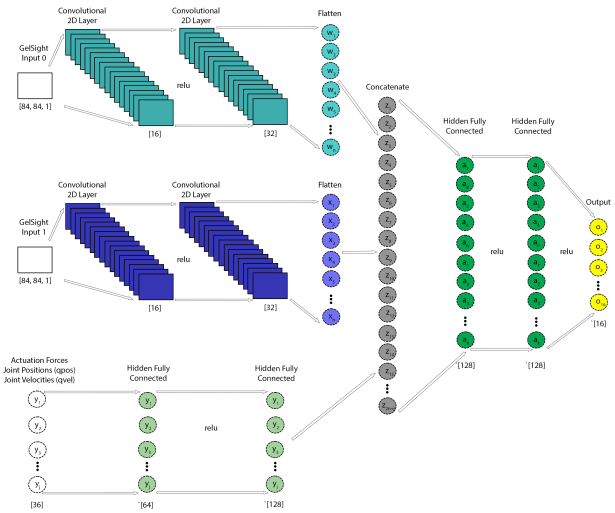


Figure 6: Original Tactile Network. Input layers consist of the two gelsight sensors and joint-related information, including the current actuation forces, the positions of the joints, and finally their velocities. The gelsight sensors are fed separately and each uses convolutional layers for feature extraction, before being flattened and concatenated [28, 29, 30].

Within the software's design, these neural network models are somewhat flexible. Any camera-based inputs, from vision and tactile, are limited to being processed through a ResNet50v2

model. However, the number of fully connected layers following these inputs is customizable, allowing for alterations to both the number of layers as well as the number of nodes. This is also the case for any joint or arm-related input. Finally, throughout testing Adam [31] and Huber [32] were utilised as the optimizer and loss functions respectively. Given more time, it would be interesting to experiment with other popular functions, such as Stochastic Gradient Descent (SGD) [33], Root Mean Square Propagation (RMSprop) [34] and Mean-Squared-Error (MSE) [35].

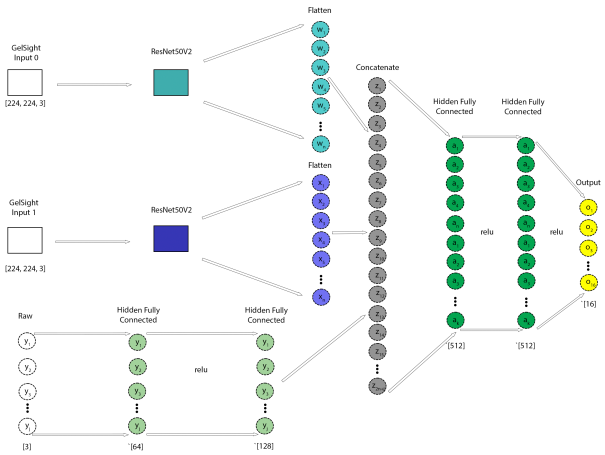


Figure 7: An updated neural network structure, utilizing the pre-trained model ResNet50V2. ResNet contains 50 layers, far more than our original network [10, 11, 12, 27].

2.6 Reinforcement Learning Models

Double Deep Q-Learning

As covered in the Design and Specification document, DDQN uses two neural networks in order to avoid the maximization bias found in DQN. It achieves this by separating the tasks into a target and evaluation network, only updating the target network after a selected number of steps. This enables DDQN to avoid falling into the same local optimal(s) as DQN. The algorithm for DDQN can be seen below, alongside the method for performing learning on the networks [6, 36].

Algorithm 1 – Double Deep Q-Learning [7]

```
Initialise Agent
Initialise Target and Evaluation networks (θ)
Initialise ReplayBuffer (D)
Initialise Environment
for each episode do
    state ← env.Reset()
    state ← env.Reset()
    while not done do
        action ← EpsilonGreedy(state, goal)
```

```
        state_, action, reward, done ← env.step(action)
        D.storeTransition(state, action, reward, state_)
        agent.learn()
        state = state_
    end while
end for
```

Algorithm 2 – **learn()** [6]

```
for each update do
    if ReplayBuffer's size > Batch size then
        Take a batch sample from the ReplayBuffer
        Calculate target Q-value:
            Y_t^{DDQN} ≡ R_{t+1} + γQ(S_{t+1}, argmaxQ(S_{t+1}, a; θ_t), θ_t^-) [6]
        Update the target network:
            θ_{t+1} = θ_t + α(Y_t^Q − Q(S_t, A_t; θ_t))∇_{θ_t} Q(S_t, A_T; θ_t) [6]
    end if
end for
```

Algorithm 3 – **EpsilonGreedy(state)** [6]

```
if random() < epsilon then
    return random action
else
    return max(network.predict(state))
end if
```

Hierarchical Double Deep Q-Learning

As mentioned previously, HDDQN is effectively an evolution of DDQN. Nesting one DDQN model inside of another. This creates a controller and meta-controller scenario, where the meta-controller (extrinsic agent) chooses goals and the inner controller (intrinsic agent) tries its best to complete that goal. The extrinsic agent receives the cumulative rewards across the span of an entire episode. Whereas the intrinsic agent only receives rewards when it completes the goal set by the extrinsic agent. In contrast, the intrinsic agent typically follows a separate reward model which follows a boolean structure - one reward for failing and another for succeeding [7].

Algorithm 4 – Hierarchical Double Deep Q-Learning [7]

```
Initialise Agent
Initialise Target and Evaluation networks (θ_1, θ_2)
Initialise ReplayBuffers (D_1, D_2)
Initialise Environment
for each episode do
    state ← env.Reset()
    Have the extrinsic agent choose a goal
        goal ← EpsilonGreedy(state)
    while not done do
        initialState = state
        extrinsic_reward ← 0
```

```
        while not done and intrinsicGoal not reached do
            action ← EpsilonGreedy(state, goal)
            state_, action, reward, done ← env.step(action)
            extrinsic_reward ← extrinsic_reward + reward
            intrinsicGoal, intrinsic_reward ← intrinsicReward()
            D1.storeTransition([state, goal], action, reward, [state_, goal])
            extrinsicAgent.learn()
            intrinsicAgent.learn()
            state = state_
        end while
        D2.storeTransition(initialState, goal, extrinsic_reward, state)
        if not done then
            intrinsicAgent.decayEpsilon()
            goal ← EpsilonGreedy(state)
        end if
        extrinsicAgent.decayEpsilon()
    end while
end for
```

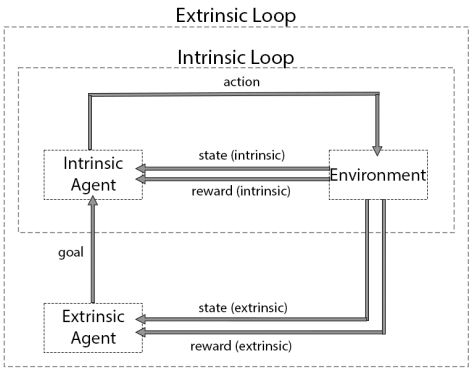


Figure 8: A diagram describing the high-level interactions between agents for the HDDQN algorithm. The extrinsic agent's reward is cumulative and only delivered at the end of an episode [7, 12].

There are a few differences in our implementation of HDDQN compared to Kulkarni's paper [7]. The most obvious is that we are using Double Deep Q-Learning for our learning updates, rather than DQN. The second is in the decaying, or annealing, of the epsilon values. The original algorithm calls for annealing to only take place, for both extrinsic and intrinsic agents, at the end of an episode. However, this would result in an extremely long training period for both agents. Therefore, for our HDDQN solution, the intrinsic agent will decay the epsilon value whenever it successfully completes its goal. The extrinsic agent's decay remains in the same place as the original paper. Though this does prove to take a great deal of time to reduce.

Finally, the goals utilized requires the intrinsic agent to move either up, down, left or right in three-dimensional space. Additionally, PickandPlace has two further goals, requiring the gripper to either grasp or release the target object.

3 Evaluation

3.1 Results So Far

Training has progressed slowly. The complexity of both the environment and neural networks makes it time-consuming to perform high step-count episodes. This introduces a challenge, as an agent needs to be given sufficient time to explore an environment - especially when it may take numerous discrete actions to manipulate the arm into performing something as simple as a forward motion. A partial solution to this has been in the utilization of frame skipping - increasing the number of frames between actionable steps. Braylan even demonstrates frame skipping improving performance in some Atari environments [37].

Some of the first relevant results come from testing a blind gripper on the glass table - displayed in Figure-1. The goal in this environment was simply to grasp the object and lift it above the table's surface for five consecutive environment steps. The results of this environment can be seen in Figure-9. Where both DDQN and HDDQN can be seen to converge to a solution in under one thousand episodes. What is interesting is the difference in results between the two algorithms. DDQN was able to solve the environment at a much earlier episode count. However, observing the grip times, it can be seen that HDDQN was able to complete the episodes while touching the target object far less. Perhaps revealing the algorithm's ability to better deal with sparse environments. For reference, Figure-9 also displays the best results received for DDQN over the course of the experiments, though this result is not representative of typical results.

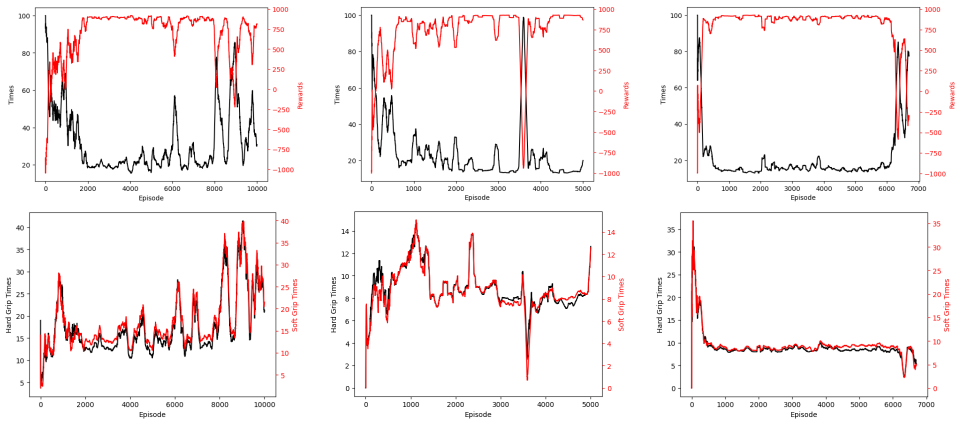


Figure 9: Early results from both DDQN (left), HDDQN (middle) and our best result from DDQN (right). These are the results from the deprecated glass table environment, shown in Figure-1. Here a cylindrical object was statically placed in between the fingers of a blind gripper. The agent's observation space was limited to the two tactile sensors alongside the arm's joint positions and velocity. The results here are averaged over 100-episodes, where red are the rewards and black is the number of steps taken.

In regards to testing the four finalized environments, we have successfully trained and tuned a static reach environment using DDQN, this can be seen in Figure-10. It is clear that the tuning of hyperparameters, specifically the learning rate and target update rate between the target and evaluation networks, can greatly affect the quality of model an agent produces [38].

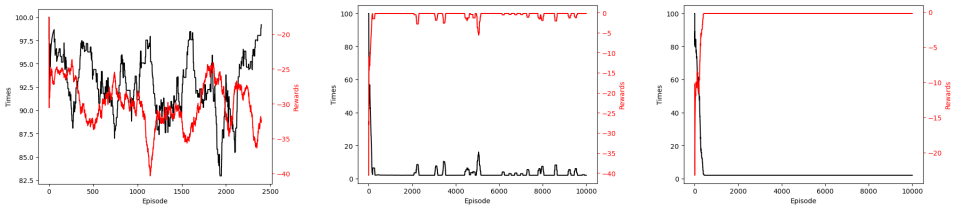


Figure 10: Testing on the Reach environment with a non-randomly placed object in combination with real and kinematic based observations. These tests were part of the pre-ResNet implementation, thus the neural network is similar to that of Figure-6. The graphs, left to right, demonstrate the dramatic affect tuning hyper-parameters can have, using learning rates and DDQN network update rates of [1, 50], [0.1, 500] and [0.01, 1000] respectively. The results here are averaged over 100-episodes, where red are the rewards and black is the number of steps taken.

Additionally, we have also completed validation of the baseline reinforcement learning algorithms using random object placement for both DDQN and HDDQN. In these baselines, we utilize only the *raw* state space for our calculations, which contains only the x, y, and z coordinates of our gripper's end effector. The reward and times results, averaged over 100 episodes, can be seen in Figure-11 and Figure-12 respectively.

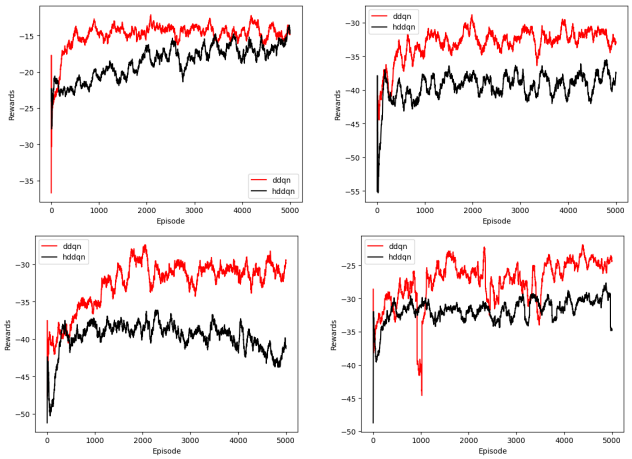


Figure 11: Reward results from the baseline tests of Reach, Slide, Push, and PickandPlace respectively (left to right). The neural network structure takes the form of three fully connected 256 layers. The update rate mirrors the best results recorded in Figure-10 at 1000 and the learning rate was reduced to 0.0001.

When comparing the raw results of DDQN and HDDQN, we can observe that HDDQN is unable to create a stronger model than DDQN, within 5000 episodes. HDDQN is far more sample inefficient than DDQN. This is likely a combination of the additional complexity offered by HDDQN. The addition of a second agent and two more neural networks introduces a great deal more randomness.

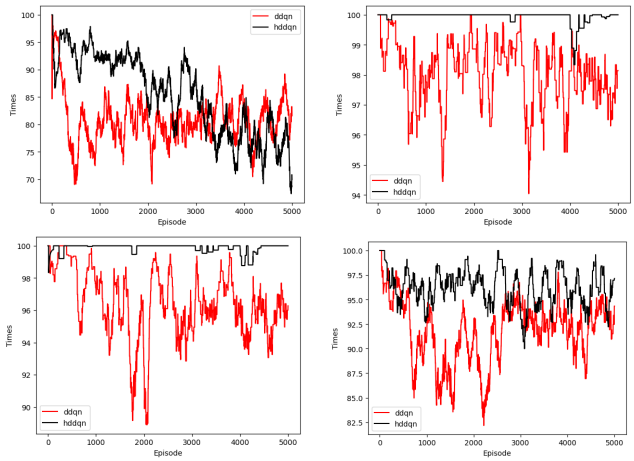


Figure 12: Time results from the baseline tests of Reach, Slide, Push, and PickandPlace respectively (left to right). These results are taken from the same set of tests as in Figure-11, and therefore use the same hyperparameters.

Additionally, the epsilon decay functions differently between DDQN and HDDQN. The former decays a single epsilon value at every time step. Whereas, the latter decays its extrinsic epsilon every episode, and the intrinsic agent has several epsilons to decay (one per goal) and can only decay a single epsilon if the agent completes its goal. To compensate, epsilon could be tuned to decay more quickly to match DDQN's, however, in my limited testing this negatively affected performance.

We can see that neither DDQN nor HDDQN are able to solve any of the environments with this observation state. This is to be expected, as they have so little information and no direct way to identify the position of the target. Regardless, all except HDDQN's 'Push' result converge positively. With our simplest environment reach demonstrating the most improvement.

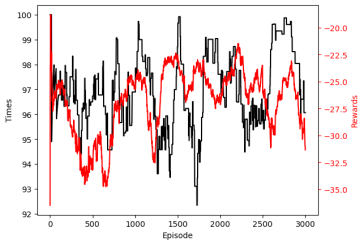


Figure 13: A Reach environment trained using a DDQN ResNet50V2 model, with only vision observations. Though it does appear to be converging, the increased network complexity results in a slower episode iteration rate than raw. The results here are averaged over 100-episodes, where red are the rewards and black is the number of steps taken.

There is obviously a clear division in performance dependent on the algorithm chosen and the environment itself. One of the reasons for choosing *raw* as the baseline is because the fewer inputs provided, the simpler the neural network produced and thus the faster it would converge.

When training on any of the other combinations of inputs the algorithms struggle to converge. An example of this can be seen in Figure-13, which is the result of five days of training and only completed three thousand episodes. Although it does appear to be climbing over time, it's still worse than the baseline.

Finally, different kernel initializers were experimented with. Tensorflow, by default, utilizes Glorot (Xavier) uniform [39]. However, Lippe demonstrates in this article [40], that HeNorm or HeUniform (Kaiming) [41] may be a better fit for the ReLu activation layers. Figure-14 describes a 1600 episode test-set comparing these methods. When observing the reward results it can be seen that all four initializers closely approximate one another across the training set. HeNormal and HeUniform both outperform their Glorot counterparts when considering the time spent early on in training, between episodes 200 and 1200. However, Glorot uniform is the best performing by the end of training. It is likely that more episodes averaged across several runs would be needed in order to establish which is truly the best performing.

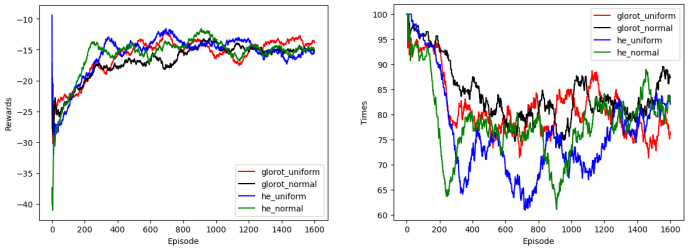


Figure 14: Reward and time results comparing different kernel initializers within the (random target) Reach environment. Results are taken as an average over 100-episodes.

4 Conclusion

The implementation of the four Mujoco environments and the two reinforcement learning algorithms has been successful. Both DDQN and HDDQN have managed to provide strong models for a statically placed object. Although both reinforcement learning algorithms have produced models that positively converge when training with raw observations, in the more sparse environments, it has proven difficult to produce models which converge on more complex observation spaces.

This presents a challenge, as it is currently unclear as to whether the models simply require more training time, if a suitable combination of hyperparameters has yet to be found, or if there is a more fundamental problem - such as the need to revise the reward function. All of these require a substantial amount of time to test, making it a potentially unrealistic goal to train multiple environments (or possibly even a single environment) successfully within the remaining time span of this project.

If the time limitations were removed as a factor, some possible next steps might take the form of comparing various different observation types against one another. For example, exploring whether vision and tactile are better at grasping certain shapes than vision and kinematics. Another path might be to explore different goal models and record how they affect HDDQN's results. Finally, perhaps experimenting with a Resnet model shallower than 50 layers, or even alternative pre-trained models such as MobileNet [42], could lead to both a reduced training time while still allowing for convergence to an adequate solution.

References

[1] Daniel Fernandes Gomes, Paolo Paoletti, and Shan Luo. Generation of gelsight tactile images for sim2real learning. *IEEE Robotics and Automation Letters*, 6(2):4177–4184, 2021.

[2] Mujoco - advanced physics simulation. *mujoco200 win64*, 2018. URL `http://www.mujoco.org/`.

[3] ROBOTIQ. 2f-85 adaptive gripper. 2019. URL `https://robotiq.com/`.

[4] Travis DeWolf. Ur5 6dof robotic arm. 2020. URL `http://www.mujoco.org/forum/index.php?resources/ur5-6dof-robotic-arm.27/`.

[5] Open AI. Openai gym. 2016. URL `https://gym.openai.com/`.

[6] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[7] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29:3675–3683, 2016.

[8] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[9] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

[10] Cheng-Chi Wang and Yong-Quan Zhu. Identification and machine learning prediction of nonlinear behavior in a robotic arm system. *Symmetry*, 13(8):1445, 2021.

[11] Kateryna Zinchenko and Kai-Tai Song. Autonomous endoscope robot positioning using instrument segmentation with virtual reality visualization. *IEEE Access*, 9:72614–72623, 2021.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[13] Kaleb Ben Naveed, Zhiqian Qiao, and John M Dolan. Trajectory planning for autonomous vehicles using hierarchical reinforcement learning. *arXiv preprint arXiv:2011.04752*, 2020.

[14] Filipe Veiga, Riad Akrour, and Jan Peters. Hierarchical tactile-based control decomposition of dexterous in-hand manipulation tasks. *Frontiers in Robotics and AI*, 7, 2020.

[15] Yannis Flet-Berliac. The promise of hierarchical reinforcement learning. *The Gradient*, 2019. URL `https://thegradient.pub/the-promise-of-hierarchical-reinforcement-learning/`.

[16] Ari Azarafrooz and John Brock. Hierarchical soft actor-critic: Adversarial exploration via mutual information optimization. *arXiv preprint arXiv:1906.07122*, 2019.

[17] Fangyi Zhang, Jürgen Leitner, Michael Milford, Ben Upcroft, and Peter Corke. Towards vision-based deep reinforcement learning for robotic motion control. *arXiv preprint arXiv:1511.03791*, 2015.

[18] Ricson Cheng, Arpit Agarwal, and Katerina Fragkiadaki. Reinforcement learning of active vision for manipulating objects under occlusions. In *Conference on Robot Learning*, pages 422–431. PMLR, 2018.

[19] Abhishek Kadian, Joanne Truong, Aaron Gokaslan, Alexander Clegg, Erik Wijmans, Stefan Lee, Manolis Savva, Sonia Chernova, and Dhruv Batra. Are we making real progress in simulated environments? measuring the sim2real gap in embodied visual navigation. 2019.

[20] Jonas Schneider Jonathan Ho Peter Welinder Wojciech Zaremba Jerry Tworek Alex Ray, Bob McGrew. mujoco-py. 2018. URL `https://github.com/openai/mujoco-py`.

[21] Avi Singh, Larry Yang, Kristian Hartikainen, Chelsea Finn, and Sergey Levine. End-to-end robotic reinforcement learning without reward engineering. *arXiv preprint arXiv:1904.07854*, 2019.

[22] Alex Irpan. Deep reinforcement learning doesn't work yet, 2018. URL `https://www.alexirpan.com/2018/02/14/rl-hard.html`.

[23] Jim Lambers. Mat 169, lecture 17 notes. 2009. URL `https://www.math.usm.edu/lambers/mat169/fall09/lecture17.pdf`.

[24] Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. Time limits in reinforcement learning. In *International Conference on Machine Learning*, pages 4045–4054. PMLR, 2018.

[25] Ilya Kostrikov, Denis Yarats, and Rob Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels. *arXiv preprint arXiv:2004.13649*, 2020.

[26] Michal Nazarczuk and Krystian Mikolajczyk. V2a-vision to action: Learning robotic arm actions based on vision and language. In *Proceedings of the Asian Conference on Computer Vision*, 2020.

[27] Gunjan Chourasia, Akarsh Shrivastava, Piyush Jaipuriyar, Zeel Shailesh kumar Bhatt, Ali Shahbaaz Khan, and Amrit Das. 7-dof robotic manipulator for autonomous segregation using transfer learning. In *2019 6th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 698–701. IEEE, 2019.

[28] Edward Adelson. Session 3 - (keynote) - the future of camera-based tactile sensing. *ViTax WorkShop ICRA*, 2021. URL `https://www.youtube.com/watch?v=IuCDAjhDnak`.

[29] Daniel Fernandes Gomes, Achu Wilson, and Shan Luo. Gelsight simulation for sim2real learning. In *ICRA ViTac Workshop*, 2019.

[30] Chen Wang, Shaoxiong Wang, Branden Romero, Filipe Veiga, and Edward Adelson. Swingbot: Learning physical features from in-hand tactile exploration for dynamic swing-up manipulation. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5633–5640. IEEE, 2020.

[31] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[32] Peter J Huber. Robust estimation of a location parameter. In *Breakthroughs in statistics*, pages 492–518. Springer, 1992.

[33] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, pages 462–466, 1952.

[34] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Lecture 6a overview of mini–batch gradient descent. *Coursera Lecture slides https://class. coursera. org/neuralnets-2012-001/lecture,[Online*, 2012.

[35] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[36] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[37] Alex Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. Frame skip is a powerful parameter for learning to play atari. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[38] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[39] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[40] Phillip Lippe. Tutorial 4: Optimization and initialization. *UvA DL Notebooks*, 2021. URL `https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial4/Optimization_and_Initialization.html`.

[41] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[42] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Appendix

Gantt Chart

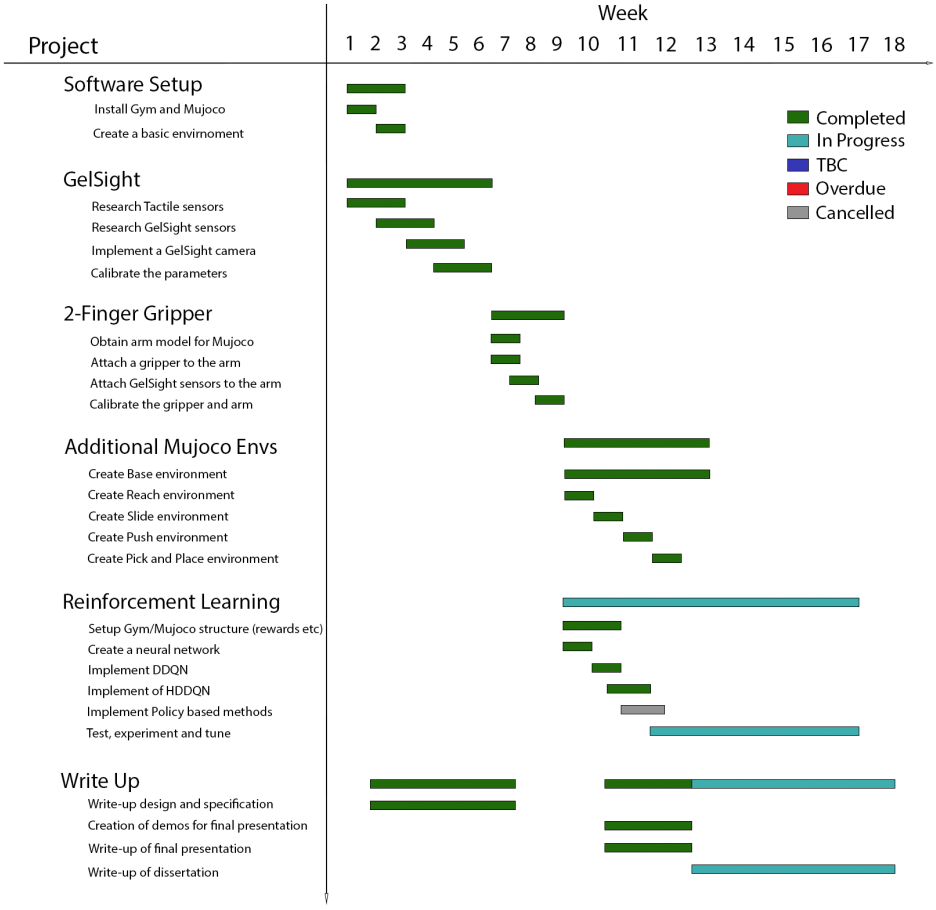


Figure 15: An update of the Gannt chart found in the Design and Specification Documentation.

Mujoco Windows 10 Installation

Installation of Mujoco within windows a relatively, although it should be noted that as of version 200, the platform is no longer officially supported. Therefore, some additional steps are necessary to enable the complete feature set.

Within your python environment:

- git clone https://github.com/openai/mujoco-py
- edit the file mujoco-py/mujoco/_py/generated/wrappers.pxi:
  - change isinstance(addr, (int, np.int32, np.int64))
  - to hasattr(addr, '__int__')
- cd mujoco-py
- pip install -e . −−no-cache

Main.py's Model

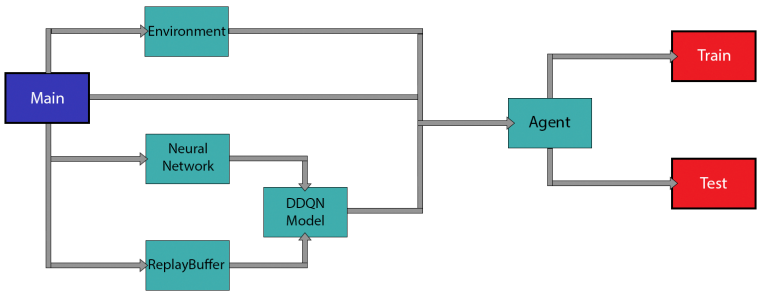


Figure 16: Main process flow of the software. Agent, Environment, Neural Network and ReplayBuffer and DDQN Model must be called separately. Neural Network and ReplayBuffer objects must be fed into the DDQN Model(s). Similarly, Environment and DDQN objects must be fed into an Agent, which can then be trained or tested.

Relevant Python Packages utilzed

Name	Version
anaconda-client	1.7.2
anaconda-navigator	2.0.3
conda	4.10.1
conda-build	3.21.4
conda-verify	3.4.2
Box2D	2.3.10
glfw	2.1.0
dm-control	0.0.364896371

Name	Version
dm-env	1.5
dm-tree	0.1.6
gym	0.18.3
gym-retro	0.8.0
h5py	3.1.0
HeapDict	1.0.1
ikfast-pybind	0.1.1
keras-nightly	2.5.0.2021032900
Keras-Preprocessing	1.1.2
keras-tuner	1.0.4
mujoco_py	2.0.2.5
numpy	1.19.5
tensorflow	2.5.0

Table 1: A comprehensive listing of all installed python packages