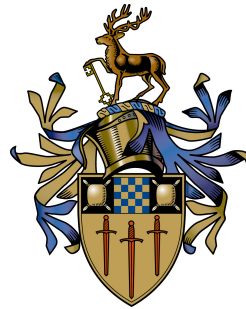


Machine Learning for Robotic Grasping

Rebecca Allday

Submitted for the Degree of
Doctor of Philosophy
from the
University of Surrey



Centre for Vision, Speech and Signal Processing
Faculty of Engineering and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, U.K.

April 2021

© Rebecca Allday 2021

Abstract

Grasping is a fundamental element of robotics which has seen great advances in hardware and engineering over the last few decades. Despite this, most current approaches struggle to generalise to the diverse environments and challenges seen in robotic grasping. This thesis looks at how data-driven deep learning methods can be used to provide this generalisation in various aspects of the pick and place pipeline. Specifically, it explores the static process of detecting repeated objects to be grasped and the dynamic process of grasping a located object.

Engineered solutions offer great accuracy and confidence in grasping technology, but are often brittle and fail as soon as the environment or task changes. Deep learning approaches have been shown to learn representations that provide superior performance over a wide variety of tasks, without the need for hand-engineering. Reinforcement Learning (RL) has the potential to transfer deep learning methods to dynamic problems by learning policies which map from an environment's state to an action to be taken. However, learning these methods end-to-end requires large volumes of training data, which is often impractical and sometimes impossible to collect in robotics applications.

This thesis offers three main contributions. The first contribution presents a method for learning vision based policies, using separate but connected perception and control feedback during training. The proposed Auto-Perceptive Reinforcement Learning (APriL) method allows the perception system to learn the necessary features to interpret the state of the environment, whilst allowing the control policy to focus on how to complete the task at hand using all available state information during training.

In goal driven tasks, with multiple potential targets, a method is required to determine which target to select. In warehouse picking, the type of target object is available at deployment, but is not always known during training and there are often multiple instances available to be selected. To address this, the second contribution proposes a zero-shot repeated object detection method, which can locate instances of similar objects in an image given a conditioning object. This method can also be used with different definitions of similarity, including the ability to distinguish between pickable and unpickable instances for a given policy.

In the final contribution, this thesis shows how these methods can be brought together in a single framework within the Robot Operating System (ROS) to complete a warehouse style picking task, where a given type of object can be picked using a learnt target selection method and grasping policy.

Key words: Robotic Grasping, Deep Learning, Reinforcement Learning, Object Detection, Warehouse Picking

Email: r.allday@surrey.ac.uk

WWW: <https://www.surrey.ac.uk/people/rebecca-allday>

Acknowledgements

Firstly, I would first like to thank both of my supervisors. Prof. Richard Bowden has provided endless support and inspiration, using his unique ability to re-frame problems to play to a researcher's strengths. Dr. Simon Hadfield's enduring patience and willingness to talk through the minute details has given me a deeper understanding and appreciation for the subjects I am passionate about.

Secondly, I am indebted to the EPSRC, the Marion Redfearn Studentship Award, and Tesco Labs for putting their faith in me by funding this research. I would especially like to thank Dr. Paul Wilkinson who supported me with insights from industry during his time at Tesco Labs.

I would also like to acknowledge the many members of CVSSP who I was fortunate enough to learn from and call my friends. Their insightful discussions and encouragement in the lab pushed me to achieve more and become a better researcher.

In addition, I am grateful to my parents who have always believed in me. From my Dad instilling a fascination with technology to my Mum ensuring I knew I could do whatever I put my mind to - I would certainly not be where I am today without their love and support.

Finally, I would like to thank my fiancé, Joe, whose unwavering emotional support has kept me sane and motivated, even during a global pandemic. I cannot thank him enough for enduring this adventure with me and showing me so much love throughout.

Contents

Nomenclature	ix
Symbols	xi
Declaration	xxi
1 Introduction	1
1.1 Contributions	6
2 Literature Review	9
2.1 Deep Learning	11
2.2 Reinforcement Learning	14
2.2.1 RL Algorithms	15
2.2.2 Experience Replay	16
2.2.3 Exploration vs Exploitation	17
2.2.4 Model Based RL	18
2.2.5 Large State Spaces	19
2.3 Application to robotics and grasping	20
2.4 Summary	22
3 Efficient Reinforcement Learning	23
3.1 RL Algorithms	23
3.1.1 Q-Learning	25
3.1.2 Actor-Critic Learning	26
3.1.3 Comparison of Deep Q-Network (DQN) and Asynchronous Advantage Actor-Critic (A3C)	27

3.2	Curriculum Learning	30
3.3	Semantically Understood State Spaces	32
3.3.1	Gaussian Process Dynamics Model	35
3.3.2	Hindsight Experience Replay	41
3.4	Conclusion	44
4	Reinforcement Learning with Varied Observability	47
4.1	Autoencoders in RL	50
4.2	Auto-Perceptive Reinforcement Learning (APRiL)	50
4.3	Auto-Perception Networks	52
4.3.1	Autoencoder	53
4.3.2	Variational Autoencoder (AE)	54
4.4	RL with Auto-Perceptive Networks	56
4.5	Experiments	56
4.5.1	2D Reach Environment	57
4.5.2	Robotic Reach Environment	60
4.5.3	Robotic Pick and Place Environment	66
4.6	Conclusion	68
5	Repeated Object Detection	69
5.1	Background	70
5.2	Recurrent Object Detection	71
5.2.1	Recurrent Segmentation Loss	72
5.2.2	Similarity Network	73
5.2.3	Recurrent Mask Similarity	74
5.3	Feedforward Object Detection	78
5.3.1	Feature Extraction	79
5.3.2	Metric Learning	80
5.3.3	Multi-Scale Features	81
5.3.4	Object Detection	82
5.4	Experiments	83
5.4.1	Many-Shot Learning	85

5.4.2	Zero-Shot Learning	86
5.4.3	Multi-scale features	88
5.4.4	Robotic Grasping Usecase	91
5.5	Conclusion	94
6	Evaluation in an Integrated Framework	97
6.1	Integration of RL with ROS	98
6.1.1	ROS Implementation	98
6.1.2	Network Implementation	101
6.1.3	Evaluation	101
6.2	APRiL with ROS	105
6.2.1	Baxter Implementation	107
6.3	Repeated Object Detection for Grasping	110
6.3.1	Affordances based on Policy	115
6.4	Conclusion	117
7	Conclusions and Future Work	119
7.1	Future Work	121
7.1.1	Reinforcement Learning	121
7.1.2	Repeated Object Detection	121
7.1.3	Object Manipulation and Placement	122
	Bibliography	123

Nomenclature

AP	Average Precision
mAP	Mean AP
IoU	Intersection over Union
sIoU	Soft IoU
GIoU	Generalised IoU
MSE	Mean Squared Error
DoF	Degrees of Freedom
CNN	Convolutional Neural Network
LSTM	Long Short-Term Memory
convLSTM	Convolutional LSTM
AE	Autoencoder
VAE	Variational AE
GAN	Generative Adversarial Network
SGD	Stochastic Gradient Descent
PCA	Principle Component Analysis
ROS	Robot Operating System
RL	Reinforcement Learning
MDP	Markov Decision Process
HER	Hindsight Experience Replay
GP	Gaussian Process
RBF	Radial-basis Function
DQN	Deep Q-Network

A3C	Asynchronous Advantage Actor-Critic
APRiL	Auto-Perceptive Reinforcement Learning
DDPG	Deep Deterministic Policy Gradient
IRL	Inverse Reinforcement Learning
SIFT	Scale Invariant Feature Transform
HOG	Histogram of Oriented Gradients
DPM	Deformable Part Models
YOLO	You Only Look Once
RSIS	Recurrent Neural Networks for Semantic Instance Segmentation
TDID	Target Driven Instance Detection
ROD	Repeated Object Detection
TP	True Positive
FP	False Positive
FN	False Negative
MSCOCO	Microsoft Common Objects in Context
ROS	Robot Operating System
RGB	Red, Green and Blue

Symbols

Introduced in Chapter 3

\mathcal{S}	State space for RL
s_t	State from \mathcal{S} at time t
\mathcal{A}	Action space for RL
\mathbf{a}_t	Action from \mathcal{A} at time t
\mathcal{O}	Observation space for RL
\mathbf{I}_t	Observation image from \mathcal{O} at time t
r_t	Instant reward from environment when action \mathbf{a}_t is taken from state s_t
R_t	Return (sum of discounted future rewards) from time t
$\pi(\mathbf{a}_t s_t)$	Policy giving the probability of action \mathbf{a}_t at state s_t
$V^\pi(s)$	Value of a policy at state s
$Q^\pi(s, \mathbf{a})$	Q-function (action-value function) for state-action pair (s, \mathbf{a})
Ω	Experience replay buffer
M	Maximum size of Ω
$A^\pi(s, \mathbf{a})$	Advantage of action \mathbf{a} taken at state s
H	Entropy
\mathcal{X}	Input space for GP
\mathbf{x}	Point in \mathcal{X}
\mathcal{Y}	Output space for GP
\mathbf{y}	Point in \mathcal{Y}
$m(\mathbf{x})$	Mean function for some point in \mathcal{X}
$\kappa(\mathbf{x}, \mathbf{x}')$	Kernel (covariance) function for all possible pairs of points in \mathcal{X}

\mathbf{X}	Collection of observed data points from \mathcal{X}
\mathbf{Y}	Collection of outputs from function $f(\mathbf{X})$
\mathbf{z}	Position of robotic end effector in \mathbb{R}^3
\mathbf{g}	Position of agents goal in \mathbb{R}^3
α_{HER}	Hindsight Experience Replay (HER) additional episodes added to the replay buffer per unsuccessful episode

Introduced in Chapter 4

ϕ_{enc}	Encoder function
ϕ_{dec}	Decoder function
$\bar{\mathcal{S}}$	Semantic state space for RL
$\bar{\mathbf{s}}_t$	Semantic state from $\bar{\mathcal{S}}$ at time t
ω	Scalar weight used to scale AE conditioning
D_{KL}	Kullback–Leibler divergence
\mathbf{u}	Position of arm end effector in \mathbb{R}^2
a_g	Action determining the state of the parallel gripper

Introduced in Chapter 5

\mathbf{M}_n	Segmentation mask of the n^{th} object in an image
$M_{i,j}$	Element i, j of mask \mathbf{M}_n
\mathbf{F}	Set of features from a fully-convolutional network
$\mathbf{f}_{i,j}$	Element i, j of features \mathbf{F}
$\mathbf{U}_{i,t}$	Output of i^{th} convLSTM at time step t
U_2	Function upsampling by a factor of 2
$\hat{\mathbf{M}}_t$	Binary mask prediction at time t
v_t	Objectness score for object predicted at time t
$\delta_{i,j}$	Assignment matrix from Hungarian algorithm
d_{ij}	L_2 Distance between two points in a latent space
\mathcal{O}	Set of similar object bounding boxes in an image

\mathbf{o}_n	Bounding box of the n^{th} object in an image
$h_{i,j}$	Element i, j of a heatmap output of a cross-correlation
O_p	Set of bounding boxes for similar objects which are pickable
$O_{\bar{p}}$	Set of bounding boxes for similar objects which are unpickable
p	Precision
r	Recall

List of Figures

1.1	Pick and Place Pipeline	1
1.2	Examples of solutions to warehouse based online shopping fulfilment	2
1.3	Current methods used for selecting grasps. Engineered caging grasps such as those proposed by Rodriguez et al. [85] shown in (a) require knowledge of the object geometry and precise location before planning. Methods using deep learning started using static grasp prediction such as Pinto et al. [79] in (b) and have needed large scale data collection using labs like Google Research’s robot array [60] in (c).	3
1.4	Traditional vision tasks from COCO [63] shown in (a) (clockwise from top left) include image classification, object localisation, semantic segmentation and instance segmentation. Both object localisation and instance segmentation give details of where individual objects are in the scene, but require classification on predetermined classes to group the objects, using methods such as YOLO [80] seen in (b).	4
1.5	Robotics simulators used for grasping. The GraspIt! simulator [66] shown in (a) evaluates grasp quality for a given manipulator and shape. OpenAI Gym [15] uses the Mujoco physics simulator shown in (b) for grasping and manipulation with the Fetch robot and Shadow hands. Gazebo [48] (c, inset) allows for easy integration of sensors using the ROS framework with visualisation in RViz (c). This allows smooth transfer of control algorithms to robots which also use ROS.	6
3.1	Reinforcement Learning Overview	24
3.2	2D arm with three joints	27
3.3	Losses and Performance metric during training for the two algorithms	29
3.4	Example of curriculum learning by increasing the difficulty of the problem. The possible goal area (white) is increased gradually from a single point (top left), to half of the play area (top right). Then again gradually increased (bottom right), to the full play area (bottom left).	31
3.5	Performance of A3C during Curriculum Learning - bold lines show a moving average across 100 episodes	33
3.6	Losses of A3C during Curriculum Learning- bold lines show a moving average across 100 episodes	34

3.7	Example of Gaussian Process (GP) in RL data flow	38
3.8	Visualisations of the trajectories from 15 random episodes used to train the GP, showing the physical limits of the agent (e.g. lower bound on the z-axis where the table is and the arc at the front of the top down view where the arms reach is limited).	39
3.9	Example actions at different scales completed by the simulator (left) or the GP (right). The pink arrows show the requested action and the blue arrow shows the change in state.	40
3.10	Example of Hindsight Experience Replay (HER) in RL data flow	42
3.11	Example trajectories of episodes sampled from the initial randomly initialised policy and the converged policy using 1 HER addition per acquired episode and replay buffer maximum size of 10000	45
4.1	Human perception and control are separate systems requiring different feedback.	48
4.2	Overview of APRiL. The optional loss and the $ \mathcal{S} $ determines how much of the encoded latent space is semantically understood. Blue arrows: the data flow in the forward pass, Orange arrows: the data flow in the backward pass.	51
4.3	Data flow through an Autoencoder perception system	52
4.4	Data flow through a Variational AE perception system	54
4.5	Mujoco Reacher simulation with goal (red sphere)	57
4.6	Reconstructions from the auto-perceptive networks for the Reacher environment	58
4.7	Heat maps showing the average number of actions taken to get to goals located across the play area with different policies	59
4.8	Reconstructions from the auto-perceptive network	62
4.9	Fetch simulation with obstacle (box) and goal (sphere)	63
4.10	Reconstructions from the auto-perceptive network for the environment with obstacles - top: semantic features, middle: partially semantic features, bottom: non-semantic features.	65
4.11	Reconstructions from the auto-perceptive networks for Fetch pick and place . .	67
5.1	Similarity network pretraining	74
5.2	Recurrent instance detection with similarity network	75
5.3	Training graphs for pretraining of similarity network with margin of 10	76
5.4	Training graphs of full recurrent system using the pretrained similarity network	77
5.5	Example predictions from the network	77
5.6	Repeated Object Detection	78

5.7	Example heatmaps - (a) Input image with known object bounding box, (b - d) Heatmaps created with crops at three different scales, (e) Pixel-wise max across each scale	80
5.8	Multi-Scale Repeated Object Detection - example values indicate output size of each module	81
5.9	Detailed system diagram with the layers used in these experiments	84
5.10	Detections for networks trained with full dataset - blue box: known target object, orange boxes: predicted detections	87
5.11	Example detections for networks trained on dataset excluding class which we are aiming to detect	88
5.12	Results for multi-scale testing - labels in the legends describe the type of the connection (skip or Repeated Object Detection (ROD)) at different scales (small, medium, large)	90
5.13	Example ground truth robotics data: green boxes show pickable objects (O_p), red boxes show unpickable objects ($O_{\bar{p}}$), blue box show the conditioning object (o_k)	91
5.14	Example detections on robotic grasping dataset for networks trained on dataset excluding the class which we are aiming to detect	93
6.1	System Diagram for A3C setup with ROS multimaster	98
6.2	Top down view of Baxter from Rethink Robotics	99
6.3	Single goal A3C Baxter simulator training - number of actions per episode	100
6.4	Random goal A3C Baxter simulator training - with curriculum learning where the full target area is used from 30,000. The blue line shows the average episode length and the orange line shows the goal variance during training.	103
6.5	Examples of trajectories executed using the Baxter simulator - with top down view on the first row and front view on the second. The image from each state during the episode is overlaid on the next, with the least transparent state being the final one.	104
6.6	System diagrams showing how APRiL can be used for transfer of control system trained in simulation to a physical robot.	106
6.7	Examples of images from the physical robot and reconstructions from the auto-perceptive network	109
6.8	Example sequence of GT state policy and encoded state policy in a similar initial state	111
6.9	Repeated Object Detection with external conditioning image	112
6.10	Examples of ROD with external conditioning. The top row shows the image used to condition the network, each subsequent row shows the output of a single image conditioned on each object respectively.	113

-
- 6.11 Examples of max heatmaps generated by a ROD module. The top row shows the scene and predicted bounding boxes with the image used to condition the network inset in the top right corner, the bottom row shows an elementwise maximum of cross-correlation heatmaps from three scales. 114
- 6.12 Examples of ROD detections with different policies - the first policy uses a policy with the gripper parallel to the the y-axis of the robot, the second policy uses a policy with the gripper parallel to the x-axis of the robot - gripper orientation can be seen in the first row. The conditioning object is shown in the first column. 116

List of Tables

3.1	DQN vs A3C Training with a single fixed target across all episodes. Average actions is the average number of actions for the arm to reach the target per episode.	28
3.2	Curriculum Learning with A3C. Area available shows change during curriculum learning - states the area of the screen where the target may be placed during those training episodes. Average actions is the average number of actions for the arm to reach the target per episode.	32
3.3	Results for use of GP with Fetch Reach environment with a single goal state s_g	41
3.4	Results showing the number of iterations till the average actions per episode converges to below 5 for different quantities of additional HER episodes and different maximum buffer sizes M . Includes episodes until replay buffer is filled to 10% of M (episodes till training starts) and the number of training iterations	44
4.1	Comparison of different works using Reinforcement Learning and AEs	49
4.2	Average episode length (actions to complete task) of system trained on the Mujoco Reacher environment.	58
4.3	Average episode length (actions to complete task) of system trained on the Fetch Reach environment (no obstacle).	61
4.4	Average episode length of system trained on an environment with a randomly placed obstacle.	63
4.5	Average episode length of system trained in the pick and place environment. . .	66
5.1	Example similarity table for predicted masks \hat{M}_i and \hat{M}_j	75
5.2	Repeated object detection - trained with all classes	86
5.3	Repeated object detection - trained with all but one class, validated on the excluded class	89
5.4	Repeated object detection for robotics - trained with all but one class, validated on the excluded class and all data in the validation set	92
5.5	Repeated object detection for robotics - trained with all but one class, Average Precision (AP) values calculated for the unpickable objects - lower AP scores are better	95

5.6	Repeated object detection for robotics - trained with all but one class, AP values calculated for both pickable and unpickable objects	95
6.1	Results from APRiL for the Baxter reach environment - showing the final average episode length and the number of each type of episode use for training	107
6.2	Results from APRiL for the Baxter Pick environment - showing the final average episode length	110
6.3	Results of training on data collected with different policies. The ground truth test data from each policy only includes bounding boxes of objects which were successfully picked by that policy.	115

Declaration

This thesis and the work to which it refers are the results of my own efforts. Any ideas, data, images or text resulting from the work of others (whether published or unpublished) are fully identified as such within the work and attributed to their originator in the text, bibliography or in footnotes. This thesis has not been submitted in whole or in part for any other academic degree or professional qualification. I agree that the University has the right to submit my work to the plagiarism detection service TurnitinUK for originality checks. Whether or not drafts have been so-assessed, the University reserves the right to require an electronic version of the final document (as submitted) for assessment as above.

The work presented in this thesis is also present in the following manuscripts:

- Rebecca Allday, Simon Hadfield, and Richard Bowden. From Vision to Grasping : Adapting Visual Networks. *18th Towards Autonomous Robotic Systems (TAROS) Conference*, 2017,
- Rebecca Allday, Simon Hadfield, and Richard Bowden. Auto-perceptive reinforcement learning (april). In *The 3rd International Workshop on the Applications of Knowledge Representation and Semantic Technologies in Robotics (AnSWeR19) at IROS*, 2019,
- Rebecca Allday, Simon Hadfield, and Richard Bowden. Repeated object detection (rod): Zero-shot generic object detection for recurring instances. In *International Conference on Robotics and Automation (ICRA)*. Under Review, 2021.

Signed:

Date:

Chapter 1

Introduction

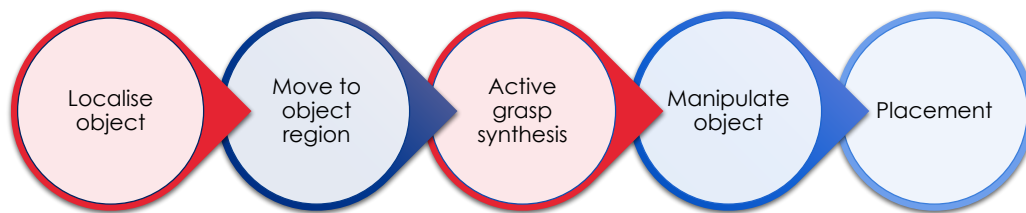


Figure 1.1: Pick and Place Pipeline

Robotic grasping and manipulation is a critical yet challenging task within robotics, which has been approached in many different ways. True human-level manipulation, which includes simple every day tasks such as putting on a jacket, has been a long-term goal in the robotics community. However, with current manipulators and engineered control systems, it remains an unsolved problem. Therefore, approaches typically tackle simpler “structured” or “parameterised” manipulation tasks that are better defined such as cutting, bending, screwing or turning objects. Robots designed for these tasks are often accurate and efficient, but are not able to generalise to other tasks or environments outside their design parameters. The fundamental action linking these tasks is grasping. In order to increase the generality of robotic systems it is necessary to be able to reliably complete the action of grasping in a variety of environments. Most research currently focuses on finding and evaluating the quality of a given static grasp for a previously localised object, without considering how the object may react or what post-grasp manipulation is required. This is an important part of the process but does not consider the full



Figure 1.2: Examples of solutions to warehouse based online shopping fulfilment

act of grasping, including the task of reaching for the object and picking it up for manipulation. In order to move towards true manipulation, the field must move from static “grasp” tactics to active “grasping”, which can generalise to a variety of situations taking into account the dynamic nature of the task.

Grasping objects is one of the key elements of the pick and place pipeline, the other components can be summarised in Figure 1.1. To start with, the object must be localised in the grasping area, which may include many repeated items and distractors in various positions and orientations. The second step involves the robot moving to the desired object, in order to achieve a feasible grasp. The next stage is the grasp synthesis, which is the problem of finding an appropriate grasp configuration which meets a set of criteria defining a stable grasp. These configurations and criteria may be engineered using models to create force-closure grasps, or they can be learnt using real or simulated data to evaluate grasp strategies. In the following step the object is manipulated in some way, for example moving from the grasp location to the place location. Finally, the object must be placed safely in a reliable manner, not damaging the object or the environment around it.

This thesis focuses on this fundamental task of pick and place, with the use case of order fulfilment in online shopping warehouses. There have been a lot of advancements in warehouse technology including scalable storage systems using conveyor belts to allow easy movement of products (Figure 1.2a) and goods-to-person stations, such as the Dematic [22] system seen in Figure 1.2b, allowing the products to be brought to a person to be picked and sorted into customer orders. Currently industrial robots like Baxter, by Rethink Robotics (Figure 1.2c), are

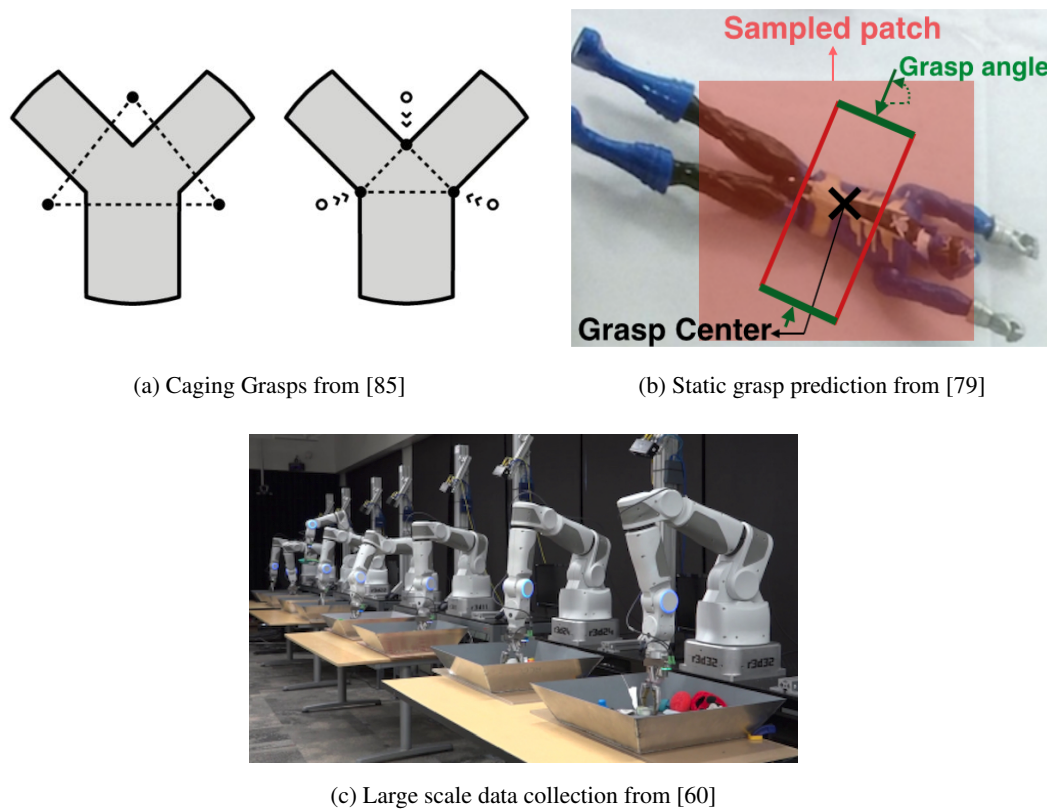


Figure 1.3: Current methods used for selecting grasps. Engineered caging grasps such as those proposed by Rodriguez et al. [85] shown in (a) require knowledge of the object geometry and precise location before planning. Methods using deep learning started using static grasp prediction such as Pinto et al. [79] in (b) and have needed large scale data collection using labs like Google Research’s robot array [60] in (c).

unable to adapt to new environments or constantly changing products, meaning the pick and place task in this application is yet to be automated.

This research aims to use machine learning techniques to aid both the initial step of locating objects and the grasp synthesis. With current robotics and inverse kinematic solvers, moving to a known region and manipulating the object once grasped are relatively trivial. When applying grasping to the task of online shopping fulfilment, the placement of objects is generally to a known location, i.e. within the customer’s box. In contrast, localising the object and grasping it successfully are still challenging research problems. Learning techniques, as opposed to engineered solutions (such as caging grasps seen in Figure 1.3a), offer a greater ability to generalise not only to the objects and environments but to the robotic embodiment used for the

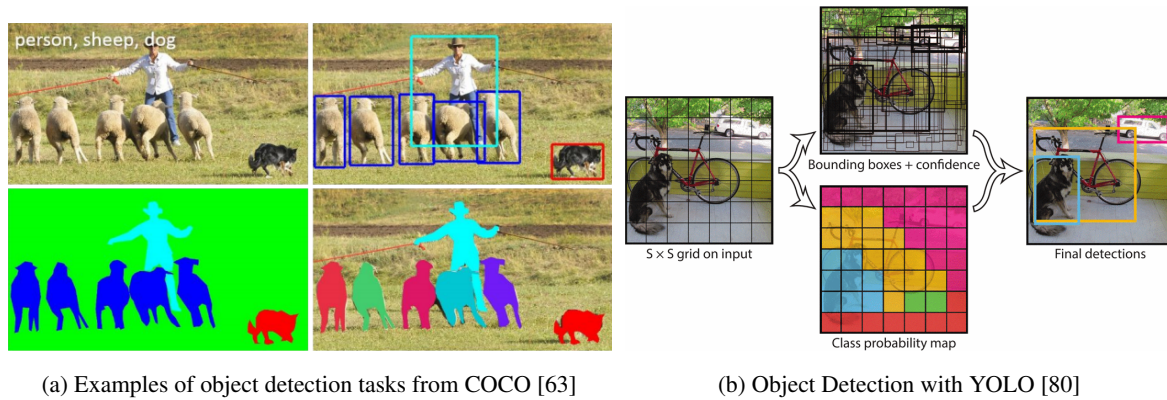


Figure 1.4: Traditional vision tasks from COCO [63] shown in (a) (clockwise from top left) include image classification, object localisation, semantic segmentation and instance segmentation. Both object localisation and instance segmentation give details of where individual objects are in the scene, but require classification on predetermined classes to group the objects, using methods such as YOLO [80] seen in (b).

task. Machine learning, especially deep learning, has demonstrated great success in many fields including computer vision and speech recognition. Due to its powerful ability to learn patterns and generalise when trained appropriately, it is now being applied in many different areas from security to robotics.

Both the task of detecting an object and grasping it, can exploit deep learning methods in different ways. Object detection and classification were some of the first tasks to benefit from modern deep learning for computer vision. The ability to learn relevant features at multiple scales and complexities mean that deep networks are well placed to detect objects at different sizes and orientations in a scene. Most of these algorithms rely on predetermined classes of objects known during training (such as the classes of *sheep*, *dog*, *person*, *bike* seen in Figure 1.4) in order to localise and determine which objects to grasp. In a large warehouse with many different products being added and removed from sale every day, this can make using standard detection algorithms impractical as not all objects can be specified in advance.

In contrast to the methods of learning used for interpreting a static scene, grasping requires the ability to interact with the environment. Many attempts have been made to use the same learning styles for both steps, for example, using a deep network to make a static grasp prediction (in the form of a grasp angle shown in green in Figure 1.3b) from a single image of the object. Whilst

this allows greater generalisation compared to hand engineered techniques, the fundamental lack of closed loop feedback from these vision based methods means that they cannot be used alone. The main learning style for interaction with environments is Reinforcement Learning (RL), which aims to learn a policy which, given the state of the environment, can select the action that should be taken to maximise some reward.

Utilisation of modern physics simulators provides a practical way to develop algorithms and collect the large scale data required for training deep RL methods. Simulators can provide detailed representations of how robots interact with objects (such as the GraspIt! simulator [66] in Figure 1.5a), but require models of objects to generate detailed grasps tactics. Fast physics simulators, such as Mujoco, are used by OpenAI (such as in the Fetch robot and Shadow hand environments seen Figure 1.5b) to train closed loop RL systems, with the ability to collect data efficiently and iterate through learning approaches quickly. These simulated problems often have access to the full state of the environment instead of using the visual observations alone, which makes it harder to transfer these techniques to physical robots where the environment is not always fully observable. Gazebo [48] simulates ROS based robots allowing the simulated and physical robot to share the same systems and controls. It also provides effective simulation of sensors, including RGB and depth cameras, similar to those used in a physical environment, meaning training can make use of visual observations without relying on a fully observable state.

Attempting to transfer RL from simulation to real world problems has still been a challenge, especially since training algorithms to directly learn control from pixels requires a large amount of physical interaction data. Figure 1.3c shows a large scale data collection robotics array at Google Research [60] used for training deep networks with continual feedback. Using this style of data collection is not only expensive, but is slow and impractical for companies wishing to use robotic picking in warehouses. Even with this quantity of data, the relationship between pixel values and the optimal action can be complex and the learning signal from a reward can be too weak for the deep networks to learn to interpret images. Whilst states may not be fully observable at run time, there is often information available during training which goes unused.

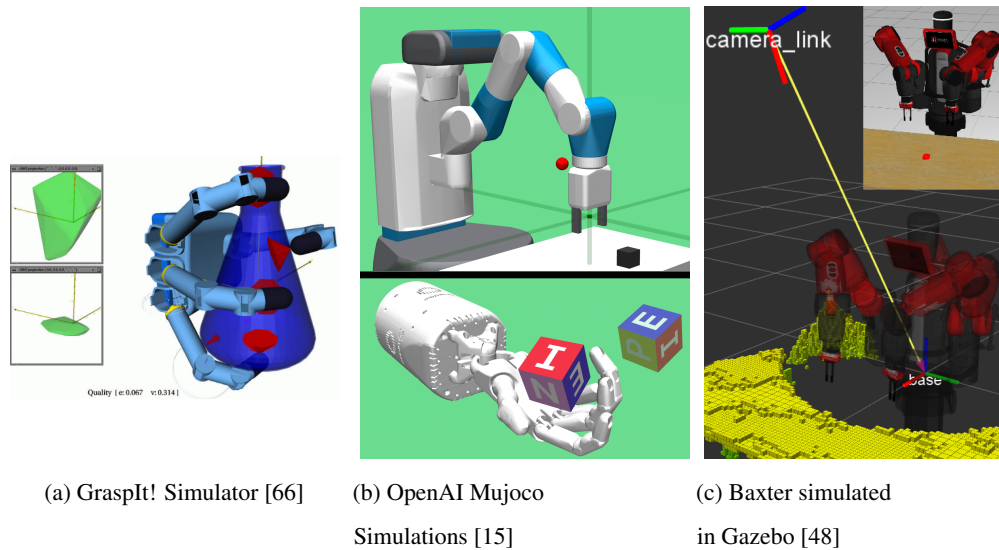


Figure 1.5: Robotics simulators used for grasping. The GraspIt! simulator [66] shown in (a) evaluates grasp quality for a given manipulator and shape. OpenAI Gym [15] uses the Mujoco physics simulator shown in (b) for grasping and manipulation with the Fetch robot and Shadow hands. Gazebo [48] (c, inset) allows for easy integration of sensors using the ROS framework with visualisation in RViz (c). This allows smooth transfer of control algorithms to robots which also use ROS.

The challenges discussed so far provide a natural set of objectives which set a structure to the overall aim of this thesis. These objectives are:

1. To develop an efficient learning method for robotic control which can determine actions from visual data.
2. To develop a technique to locate repeated objects in an image without prior knowledge of object types.
3. To explore how these methods can be used together to perform warehouse style picking.

1.1 Contributions

To achieve these objectives, this thesis presents a selection of connected pieces of work. Chapter 2 will discuss the current state-of-the-art in robotic grasping, deep learning, and computer vision. Specific focus will be placed on different learning styles and how they can be used in robotic grasping.

In order to start addressing the first objective, Chapter 3 looks at how we can improve the efficiency of RL algorithms using both model-free and model based methods. This includes looking at end-to-end methods, which go from images directly to actions, and methods which use a semantically understood state space as input. The chapter starts by employing discrete action spaces to control low level dynamics, and moves to more complex continuous domains with higher level control.

Building on the previous chapter, Chapter 4 addresses the first objective, by proposing APRiL. Auto-Perceptive Reinforcement Learning (APRiL) allows us to deploy a policy which takes an image as input and outputs actions, but uses any semantically understood state space available at training time to introduce structure to the learnt features. This is achieved using deep networks to learn an encoded state space from the image observations by attempting to reconstruct them. The encoded state space can be semantically understood if state data is available, or can be learnt from only the reconstruction loss. This work is inspired by the way we learn as children, where the visual and control skills are learnt in tandem, not in one end-to-end process.

Moving to the second objective, Chapter 5 proposes Repeated Object Detection (ROD), a solution to the initial object localisation step of the pick and place pipeline. We focus on the scenario where we wish to detect similar objects, but do not have knowledge of which types of object this will be in advance. Previous work has been able to detect repeated identical objects. In contrast, the aim of this work is to detect repeated similar objects in the same crate which will be orientated differently and may be of different scales depending on where in the crate they are. This method includes adding a new loss to encourage similarity between features of similar objects across different scales. The results show that, even with object types not included in the training data, it improves on currently available methods.

To address the final objective, Chapter 6 brings these contributions together in a system that allows them to be trained in simulation on the same infrastructure as a physical robot and then deployed with minimal fine tuning on the real robot. This shows proof of concept that a learnt pick and place system can be trained efficiently in simulation, and then deployed in the real world using standard low cost robotics and sensors with minimal additional effort.

Chapter 2

Literature Review

Grasping has been one of the fundamental research areas of robotics for many years. Approaches to grasping are often initially driven by the hardware, with many different robotic end effectors available for interacting with the environment. The most basic of these is the single degree of freedom parallel gripper, often used in industry for basic repetitive tasks with objects of a known size, shape and location. Early ventures away from this included the three fingered Stanford/JPL Hand [89], which eventually led to high degree of freedom hands, such as the 20 degree of freedom anthropomorphic Shadow hand [82]. These more complex hands provide much greater dexterity but are incredibly complex to control, meaning there is a trade-off between adaptability and ease of use.

Once a hardware platform has been selected, methods for autonomous grasp selection can be split into analytic and data-driven methods [88]. Analytical methods use geometric and dynamic criteria to form grasp maps and find force-closure grasps via contact locations. A force-closure grasp is found if there exists a set of contact forces which can resist any external wrench [70]. Methods that focus on using geometric models to find grasps, with no forces considered, are referred to as kinematic grasping. Those that consider both friction and inertial forces to find force-closure grasps are called dynamic grasping. This later category involves having a reliable physics model to determine the forces acting on the object in order to find an equilibrium for grasps, which is difficult when grasping unknown objects. Even when there exists accurate models, these methods are often prone to errors from sensor noise and unreliable robot models. Several branches of grasping research have looked into how to make grasping

robust to these issues, including independent contact regions [74] and caging formulations which, once achieved, guarantee a grasp [85] (see Figure 1.3a).

Alternatively empirical or data-driven methods, focus on sampling potential grasps and evaluating them based on some set of metrics, generally not analytic but encoding perception and prior knowledge. A survey by Bohg et al. [14] divides the area of data-driven grasping into 3 groups:

- **Known objects:** Requiring the use of object recognition and pose estimation to retrieve a suitable grasp from an experience database. Often these methods have access to geometric models of the objects.
- **Familiar objects:** Assume any object encountered is similar to another object and use a measure of similarity (shape, colour, texture) to find a possible grasp.
- **Unknown objects:** Do not assume any access to models or grasp experience and translate the features of an object directly to a grasping method.

Inside the learning framework, each group implicitly encodes the constraints explicitly specified on the previous level. For example, methods using unknown objects can learn how to interpret the similarity of objects (such as shape) within the grasp learning system, without the need for hand-engineered measures of similarity used in methods for familiar objects.

Data-driven methods often use existing knowledge of grasps. This can be as a direct way of mapping grasps onto a known or familiar object. Ekvall and Kragic [24] use human demonstrations of grasps collected using a data glove with magnetic sensors. These grasps are mapped from the human grasp to a robots equivalent grasping posture. At deployment, the system recognises an object and then determines which grasp type can be used for that object. Another method for utilising human grasps, is to observe them visually, as done by Romero et al. [86], where the grasp type and pose is determined from an image of a human grasp and mapped to a robotic hand. These methods do not model the objects explicitly, but do assume a single set of objects, meaning grasps can be transferred directly.

Alternatively existing knowledge can be used as training data for a machine learning method to learn general rules for mapping features of objects to grasps. El-Khoury and Sahbani [25] segment 3D point clouds of objects into approximate super-quadratics labelled as graspable or

not, which are used to train a system to determine which part of an object is graspable, but not how to grasp it. In contrast, Pelosof et al. [77] use a super-quadratic to represent the object and then train a system to take in the object along with a grasp configuration and predict the quality of that grasp. This allows them to directly select a specific grasp which maximises the grasp quality. The use of 2D data has also been employed in systems aiming to learn from grasp experiences, such as Saxena et al. [91] who infer a grasp point on an object directly from an image of said object. They use labelled synthetic data, and create a feature vector of image features from hand crafted filters, which is used to train a logistic regression model to determine successful grasp points.

The methods discussed in this section so far have used classical learning or analytical methods. However, recently one of the biggest advances in autonomy has been the achievements in deep learning. The many different parts of grasping, from the interpretation of the environment to the control of a robotic agent, can make use of these advances. The key areas this thesis focuses on are learning for control, learning for vision, and the intersection of the two.

2.1 Deep Learning

As a tool for machine learning, deep learning [57] has achieved popularity and state-of-the-art results due to its ability to learn both hierarchical representations, combined with powerful nonlinear classification to solve various tasks in one end-to-end process. There are various common styles of learning applied to deep neural networks, each of which use different types of training data. Tasks can be split into two categories - those which are independent and those which interact with some environment. Independent tasks have no causal effect on the input data they act upon. For example, whilst natural language translation is a task which has a time element, the output of the translation does not cause direct changes to the next sentence to be translated. In contrast, dependent tasks require the learnt method to interact with an environment, hence having a causal effect on the next data point. This section will focus on learning styles typically used for independent tasks, whilst Section 2.2 discusses methods for dependent tasks.

The most common style of learning for independent tasks is supervised learning [6], where each element of a dataset has an associated label given by a supervisor and the aim of the learnt

system is to predict the correct label from the input data. In contrast, unsupervised learning [6] is used for tasks such as clustering or feature learning [55], where the aim is to find some unknown pattern in the data. There are also hybrid styles of learning, such as semi-supervised or self-supervised learning. Semi-supervised learning involves using training data for which only a subsection of the data has labels, aiming to improve the precision of learning from the labelled data by exploiting the patterns in the unlabelled data. Self-supervised learning re-frames an unsupervised problem as a supervised one, by using information easily extracted from the data itself. For example, image colourisation can be approached by using colour images as the labels and generating grey-scale images to use as the input [110], this requires no additional labelling but still frames the problem as a supervised one. Another framework used to avoid costly, task specific labels, is weakly supervised learning, where preexisting labelled data from one task may be used to aid in learning another related task. In object localisation, it is extremely time-consuming to annotate images with accurate pose and scale information for each object present. Methods similar to the one proposed by Bency et al. [12], use only image-level class labels and an image-level classifier to predict bounding boxes for individual objects in an image.

The type of input data being used by learning algorithms can help determine the deep learning architectures employed. One common architecture is Convolutional Neural Networks (CNNs) [56] which take input data with spatial information, often in the form of images or depth maps. CNNs use convolution layers which slide a set of kernels (or filters) over the input to detect features in different areas. One of the breakthroughs for CNNs occurred in 2012 when Krizhevsky et al. [51] created AlexNet for use in the ImageNet [23] competition to classify images into 1000 different categories with exceptional results. This inspired researchers to use CNNs with much deeper architectures in order to develop richer representations. However, the gradients applied to the weights of these deeper networks become increasingly small, eventually preventing training. This issue is known as the vanishing gradient problem. Szegedy et al. [103] found that one way to combat this problem is by introducing intermediate loss layers at various depths in their GoogLeNet network.

Another issue with deeper neural networks is that it increases the number of learnt parameters, meaning the models require more memory to train and deploy. In response to this problem Iandola et al. [42] created SqueezeNet which is designed to have far fewer parameters in comparison to AlexNet whilst maintaining accuracy. They achieved this by the use of Fire

modules which are comprised of a *squeeze* convolution layer which feed into an *expand* convolution layer. Fire modules only have 1x1 and 3x3 filters in their convolution layers keeping the number of parameters to a minimum. This makes it a suitable architecture for embedded systems and problems where there is a reduced volume of training data such as robotics.

Whilst deep learning for vision initially saw success in image classification, it has since been used for many different problems. One such application of deep learning in vision is learning compact representations of images. A popular architecture for this is the Autoencoder (AE) [40] which is a self-supervised system using an encoder network to reduce the dimensionality of an input and then decoding this latent representation via an expansion network to reconstruct the original input. Most recent applications of AEs use a CNN as the encoder allowing a compact representation of images, whilst maintaining spatial information. AE style networks have been used for learning generative models able to reconstruct both the original input image but also to transfer domains [39] such as RGB image to semantic segmentation via encoder-decoder style networks like SegNet [10]. One of the uses of AE style networks is generating new outputs by adjusting the latent space. However, this relies upon the representation being interpretable, which is often not the case. Kingma et al. [47] propose Variational AEs (VAEs) to solve this issue, by structuring the encoded space as a distribution from which a value is sampled and passed to the decoder. They apply an additional loss to the encoded representation to push the latent representation towards a unit normal. This encourages the encoded space to be disentangled and allows for much easier interpretation of the space, whilst still maintaining the self-supervised framework.

Generating realistic images is a challenging problem for AEs, with reconstructed images often being blurry. Recently, the most common approach for generative networks are Generative Adversarial Networks (GANs) [36], which use an adversarial network as a critic that attempts to discriminate between the real images and the generated images. The competition between the reconstruction loss and the discriminator loss causes the generator to produce more realistic images whilst the discriminator continually improves its ability to tell the difference. This means that GANs can be used to augment data and generate realistic artificial images, which can be incredibly useful in domains with limited data efficiency. For example, Shrivastava et al. [96] use a GAN based system to refine synthetic images into the real domain, allowing the generation of self-labelled datasets for both RGB and depth images.

Whilst the data efficiency of neural networks has certainly improved, all of the networks discussed here require large quantities of representative data to effectively generalise to their given domain. This is one of the key issues when applying deep learning to other fields. Currently there are many labelled datasets available, especially for classification with the likes of ImageNet [23] and CIFAR-100 [50] and for applications such as segmentation with COCO [63] and Cityscapes [17]. However, these are only readily available at this scale for independent tasks with no causal relationship between the environment being observed and the observer. This is especially true when the solution to a task is inherently tied to the form of the agent trying to complete it, such as the physical properties of a robot completing a grasping task.

2.2 Reinforcement Learning

The styles of learning considered so far have been for independent tasks which infer information from data without interacting with the environment the data comes from. Reinforcement Learning (RL) [101] is an approach to learning that involves sequential decision making for an agent interacting with an environment. The aim of RL is to learn a policy, mapping from the current state of the environment to an action, that can be taken by the agent to maximise the expectation of the total reward across the remainder of the sequence. In contrast to standard supervised learning, RL does not use static labelled ground truth data. Instead, it is trained with data containing: a set of environment states, the actions taken from those states, the immediate rewards achieved by those actions, and the state of the environment following the action (a formalisation of this is presented in Chapter 3).

One of the most important decisions to be made when choosing an RL algorithm, is whether it should be on-policy or off-policy [101]. On-policy methods optimise the same policy which is generating the transitions to be learnt from. In-contrast, off-policy algorithms do not require the data used for learning to have been generated by the same policy being optimised. A simple example of this is optimising a deterministic greedy policy, whilst using a stochastic ϵ -greedy policy to collect the observations. Another option to be considered is online vs offline learning. Online learning uses data as it is made available, either by using the currently available data or increasing the amount of learning data during training as more observations are made. Offline learning (or batch RL [54]) works with static data, where all observations are collected prior

to learning. It is important to note that whilst it is possible to have both online/on-policy and online/off-policy algorithms, it is not possible to have offline/on-policy algorithms, because as soon as the learned policy is different to that used to collect the offline data, it is no longer on-policy learning.

2.2.1 RL Algorithms

There are many approaches to finding an optimal policy, which maps from the current state of the environment to the optimal action for long term rewards. Value-function based approaches aim to learn a value function which finds the expected discounted reward given the current state. The most common value-function method is Q-learning [107] which aims to find the optimal policy via estimating the action-value function (or Q-function), giving the expected discounted reward for a state-action pair when following a given policy. In contrast to value based methods, policy methods aim to optimise the policy directly. One example of such a policy gradient method is REINFORCE [108, 102] where the policy is approximated by a function with parameters which are updated directly by gradient ascent on the expected discounted reward. These two approaches to RL may be combined for actor-critic learning [11], where the parameters of an action-value function are learnt as the ‘critic’, whilst an ‘actor’ is updated to produce a policy based on the output of the ‘critic’.

Traditionally RL methods used with function approximation tend to be highly unstable, especially when using the non-linear function approximations needed for more complex tasks. Deep RL is the combination of RL and deep neural networks used as function approximators. Deep RL has seen notable success including the Deep Q-Network [68] and AlphaGo [97] which brought several improvements to stability and showed impressive results. For example, DQN is used in an off-policy manner with experience replay and a target network to improve the stability of the non-linear CNN used to approximate the value function.

Following these advances there have been many improvements and adaptations of deep learning for RL. Another notable method combining the two is Asynchronous Advantage Actor-Critic (A3C) [67], which uses deep neural networks to approximate both the policy and value functions. This particular algorithm uses parallel actors with separate policies, to encourage exploration of the environment, but are updated by a central update step to improve the stability of learning.

Another actor-critic algorithm which takes advantage of deep learning is the Deep Deterministic Policy Gradient (DDPG) algorithm [61] which applies the techniques learnt in DQN and batch-normalisation [43] from the deep learning community to a deterministic policy gradient approach [98].

A further way to improve the stability of RL algorithms is to restrict the size of the policy update during training. Schulman et al. [93] propose Trust Region Policy Optimisation (TRPO), which uses Kullback–Leibler (KL) divergence to constrain each policy update to a ‘trust-region’. However, applying TRPO to deep neural networks is inefficient and complex since it is not compatible with Stochastic Gradient Descent (SGD). In comparison Proximal Policy Optimisation (PPO) [94] uses a clipped ratio of the probabilities of an action under the new and old policies directly in the objective function, meaning it can be used with efficient SGD methods whilst still ensuring the policy update is relatively small.

2.2.2 Experience Replay

One of the issues with algorithm selection in RL is that whilst on-policy methods allow for exploitation of the current policy, they can often become trapped in a local minima, especially when exploration is not sufficiently built into the policy. Alternatively, off-policy methods evaluate one policy whilst executing another, which allows for much greater flexibility in data collection and exploration, but can still lead to over-fitting to single observed episodes. The use of experience replay [62], where episodes are stored in a buffer and sampled from it for training, has been the key method used to tackle strong correlations in single episodes and increasing the quantity of data available. Sampling from the replay buffer can be uniform but this often leads to replaying episodes which are not significant to improving the policy. To combat this, [92] proposed Prioritised Experience Replay (PER), where the individual transitions are assigned priorities (approximated by the absolute policy loss) which act as weights for sampling from the buffer, meaning that transitions that will have a higher impact on the policy are more likely to be sampled during training.

Experience replay has since been used to combat other issues such as sparse rewards. Reinforcement learning relies on having an appropriate reward function to influence which sequences or actions have a positive or negative impact in the long run. It is often the case for RL problems

that the reward signal is sparse, namely the effects of an action are only known further through the episode. For large state and action spaces, sparse rewards mean that reaching a state which gives meaningful feedback may be rare. In the case of goal orientated RL, Hindsight Experience Replay (HER) [9] can be used to artificially increase the number of episodes in the replay buffer with relevant feedback. This is done by adapting the goal to a state which was achieved by the agent and recalculating rewards before adding an episode to the buffer. Whilst this helps to balance the ratio of successful episodes compared to failures, it does not ensure a good distribution of goals across the state space, meaning we need to consider which episodes HER is applied to. With this in mind, Nguyen et al. [73] proposed HER with Experience Ranking, using the distance between the original and the achieved goal to ensure the newly sampled goals are within a threshold of the original one. This helps stop the buffer being filled with episodes targeting only goals which the original policy was able to reach.

2.2.3 Exploration vs Exploitation

One of the key challenges in RL is the trade-off between exploration of the state space and exploitation of the learnt policy. The agent needs to explore parts of the state space not yet seen to avoid falling into a local optima but also to use the policy to allow it to achieve the task at hand. There have been various attempts to address this issue such as the use of asynchronous actors [67] as mentioned above in the A3C algorithm. Fortunato et al. proposed NoisyNet [32] which encouraged exploration by adding parametric noise directly to the network weights, with the noise parameters also learnt alongside the network.

Another way that exploration has been driven is via the concept of curiosity. Pathak et al. [76] propose an Intrinsic Curiosity Model (ICM) which produces an intrinsic loss comparing the feature space at the next time step to a prediction, via a network that takes in the feature space at the current time step and the action to be taken. To minimise this loss, the agent must take actions which introduce it to parts of the state space which the model has not seen before and therefore cannot predict easily. This can be used with or without the usual extrinsic reward provided by the environment and can help to increase exploration and performance in systems with sparse reward signals.

When looking at the sparsity of a reward signal the connection between action and potential

reward can be too distant. This can often be because the problem is difficult or impractical to learn without any prior understanding, meaning typical exploration will not provide enough context to complete the task. Inspired by the way humans learn as children, curriculum learning [13] is a training strategy which provides structure to the learning process, starting with simple tasks and building upon knowledge learned to complete more complex tasks. However, the curriculum used must be well designed, otherwise the jump in difficulty can cause the network to diverge or if it is too small the agent may not learn and only attempt to exploit its current policy. Some methods apply a curriculum more implicitly by incorporating it into the replay buffer, such as Curriculum-guided HER (CHER) [27], where similarly to Nguyen et al. [73] the episodes which have adapted goals are determined by the distance to the original goals along with a curiosity based exploration metric. They enforce a curriculum by gradually changing the proportion of episodes being selected to have their goals adapted. The aim is to have increased curiosity at the start, even if the achieved goals are far from the original target, then gradually forcing the achieved goals to be close to the originally selected target.

2.2.4 Model Based RL

Given the quantity of data required for training RL algorithms effectively, frequently collecting data from the environment can be slow or impractical, especially in real world environments. All the methods discussed so far have been model-free, meaning they directly learn either a value function or policy without explicitly learning the transition dynamics from state to state for a given action. In contrast, model-based RL attempts directly to learn the dynamics of the system, which can then be used to efficiently create artificial data for training. It is important to note that, whilst many RL approaches use simulation to obtain more data, they are not considered model-based because the simulation is not learnt and they are often slow and computationally intensive.

Efficient learnt models have been shown to be effective at learning the transition dynamics for low dimensional state and action spaces, especially when the model incorporates uncertainty. For example, Black-DROPS (Black-box Data-efficient RObot Policy Search) [16] uses Gaussian Processes (GPs) to learn the dynamics of a 4-Degrees of Freedom (DoF) robotic arm, and then samples additional data from this for training a policy to move to a single position. This meant

learning an effective policy required as little as 5 episodes of physical interaction. Another way to efficiently use model-based RL is via Guided Policy Search (GPS) [59]. This approach learns a dynamics model based on local sampled trajectories, which then uses trajectory optimisation methods to learn an optimal policy for said learnt dynamics. Using this policy to collect more trajectories and then iterating over this system allows efficient continually updating dynamics modelling.

More complex systems modelled in this way can lead to simulation biases, often referred to as “reward hacking”, with errors in the model being exploited by the RL algorithm if they benefit the received reward signal. One way to deal with simulation biases is to use an iterative learning approach where simulation models are continually updated throughout the learning of the policy. Ross et al. [87] assume a good exploration distribution and alternate between collecting new information about the environment via a good policy for the current dynamics model and updating the model. However, methods such as this do not generally scale well to larger domain spaces, such as images.

2.2.5 Large State Spaces

Another challenge with both reinforcement learning and deep learning is the curse of dimensionality. RL methods which work well for problems with small state and action spaces often do not generalise well to large domain spaces, such as robotic arms with high DoF or RGB images of physical environments. Considering this, it is not surprising that dimensionality reduction techniques have been used in conjunction with RL to improve the efficiency of learning.

One such technique proposed is Dimensionality Reduced RL (DRRL) [19] where a chosen dimensionality reduction method is used to transform the state space to some smaller representative state. The results showed that using Principle Component Analysis (PCA) as the dimensionality reduction method improved the speed of convergence. Similarly, Lange and Riedmiller [53] use a deep AE to compress images to a feature space used as the input to a policy.

However, by transforming to much lower dimensional state spaces some low variance, potentially important, data is lost. This can lead to learning a sub-optimal policy. To overcome this Curran et al. suggest Iterative DRRL (IDRRL) [19], which uses transfer learning techniques to apply what

is learnt in a low dimensional state space to learning an optimal policy in a higher dimensional state space. This allows most of the learning to be done efficiently in the low dimensional spaces, but still allows the policy to learn from important data only available in the high dimensional space.

Further work showed that it is possible to use a Cascading AE (CAE) with IDRRL [18] to encode a state space into ever decreasing dimensional spaces and then use transfer learning to build up policies from the smallest dimension back to the full state space. Whilst this allows extremely efficient learning at the lowest dimension spaces, the transfer learning process becomes more complex when increasing the state-space size to that of image observations.

One of the issues with the use of dimensionality reduction in RL is that we often do not have the semantic understanding of the lower dimensional state space to exploit valuable RL techniques such as HER. Nair et al. [71] propose a solution to goal-conditioned RL, using an encoder-decoder system to learn a latent space which can be used to sample goals, provide a lower dimensional, structured input for RL and to compute a reward signal for reaching the goal. Although this allows HER to be used for visual problems, it introduces its own limitations. In using an image as an explicit goal, the agent's flexibility is limited. For example, in a pick and place problem it restrains the final pose of the *robot* when the final position of the *object* is more important.

2.3 Application to robotics and grasping

Robotics has been one of the main application areas for RL research due to its interactive nature between an agent and an environment. However, many of the environments used to benchmark RL are simple software simulators such as video games or basic physics problems (e.g. cart-pole), which do not represent the high complexity or unique challenges of robotics. One of the key differences is the ability to collect meaningful quantities of data. It is easy to collect millions of episodes of data for these benchmarking environments in less than a day. In comparison, using realistic robotic simulators such as Gazebo [48] a thousand episodes could take around 3 days to complete. Attempting to collect large quantities of data with physical robots is often impractical, due to physical and safety restraints, including the need for a human to reset the environment between episodes.

Some works have still attempted to train RL methods with real world data. Levine et al. [58] propose Guided Policy Search with CNN based policies to complete various tasks with a PR2 robot. The CNN is pretrained on ImageNet, followed by a selection of 1000 collected images to regress positions of notable objects in the scene. Then it is used to produce end-to-end policies from image to torque control, using only around 25 physical samples per task. This method performs well on these tasks but the limited quantities of data for the training of a CNN leads to over-fitting, with no results shown on unseen objects or even visually varied objects.

Others have attempted to scale up the quantities of data used to train CNN based networks in more traditional deep learning frameworks. Pinto et al [79] collect 50,000 grasp attempts on a single robot to create a dataset for robotic grasping and train a network within a self-supervised framework. This method uses an AlexNet based CNN to predict grasp position and angles, which show good results on a small selection of unseen objects. The network used in this case has not been adapted from the initial computer vision application, meaning it is not designed to learn features important for grasping, specifically spatial awareness. In contrast Allday et al. [3] adapts the architecture of the network used to predict grasps by removing pooling layers which lose vital spatial information and reducing the size of networks to make them more data efficient. Unfortunately, neither of these methods allow for any correction of mistakes once the initial action has been chosen. Researchers from Google took this a step further collecting 800,000 grasp attempts with 14 robotic arms [60]. These grasp attempts are used to train a network to predict the likelihood of grasp success for different actions. In contrast to the previous work, it uses visual servoing to predict the likelihood of grasp success at discrete time steps through the grasp, allowing it to correct for mistakes. The implementation of this method involves several heuristics such as raising the arm if the end-effector is currently below a given height and there are no moves that currently predict a successful grasp of 50% or more. Whilst this improves the safety of the system, it means that additional knowledge of the environment is required. Since this method uses a fixed dataset the policy cannot be improved through exploration, as in RL, making the assumption that all necessary exploration has been done at data collection.

A key area of research that could make RL methods more viable in robotics is the transfer of policies learnt in simulation to real robots without retraining. In recent works the main method used for sim-to-real transfer is domain randomisation, where various parameters of the simulated domain are randomised during training. This reduces the discrepancy between itself and the

physical world. Domain randomisation can either adjust visual parameters [104] such as those relating to textures and lighting, or physical dynamics parameters [78] such as damping or friction. These methods can stop a policy over-fitting to the simulated world but rather learn how to adapt to a distribution of environments, with the real world laying somewhere in that distribution. It can be difficult to adapt the simulated domain in such a distribution, one way that this has been combated is to use an image conditioned GAN to transform images from their respective domains (real world or randomised simulation) to a common domain which can be used as an input along with the original image to the RL algorithm. Domain randomisation is a powerful tool for transferring learning from simulation but does come at higher computational and implementation costs, especially when access to cloud compute services are not readily available.

2.4 Summary

A lot of the algorithms discussed aim to have a direct vision to action pipeline both during training and evaluation. However, this means that we lose much of the structure provided by additional semantically understood state information at train time. These systems also aim to use torque controls directly on robotic joints, however, this means that the policy must not only learn the higher level method needed to complete the task but also the inverse kinematics of the robot, which in most cases, already has a reliable solution. In order to avoid learning the solution to a problem that has already been solved and to provide more structure to the learning system without providing system specific heuristics, this thesis looks at moving away from end-to-end solutions for robotic grasping, whilst still providing an image to action solution for deployment.

Chapter 3

Efficient Reinforcement Learning

As discussed in Section 2, Reinforcement Learning (RL) has been used within the robotics community for many years with model-based methods [21] and policy learning methods [49]. However, with the recent surge in developments from the area of deep learning, there are many more options for robotics researchers to consider. Deep learning generally requires large quantities of training data, this is also true for RL except the data collection often cannot be done in advance. During the training of online RL algorithms, new data needs to be collected based on the newest learnt policy. We want algorithms to require as little data as possible, meaning one of the most important aspects of RL systems is data efficiency. This chapter provides background formalisation for RL and looks at the effectiveness of different RL algorithms. This includes how they can be made more efficient, especially in the robotics domain.

3.1 RL Algorithms

Most algorithms can be grouped into either value based methods, policy based methods or composite approaches called actor-critic methods. Before describing each type we will formalise RL below. This work focuses on episodic reinforcement learning, where an agent interacts with an environment at discrete time steps, t . Figure 3.1 shows an overview of the data flow in a simple RL system.

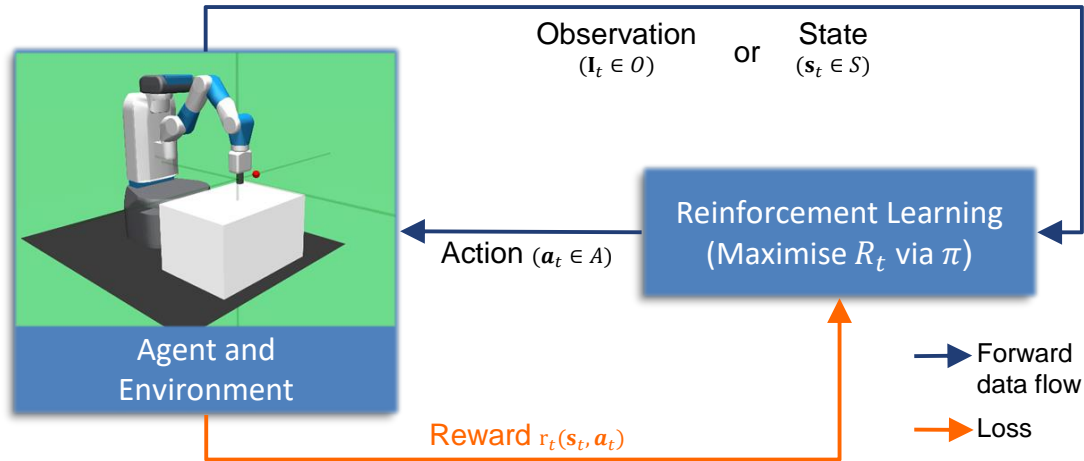


Figure 3.1: Reinforcement Learning Overview

Episodic RL can be formulated as a Markov Decision Process (MDP) involving:

- a set of states $s_t \in \mathcal{S}$,
- a set of actions $a_t \in \mathcal{A}$,
- a transition probability distribution $P(s_{t+1} | s_t, a_t)$,
- a reward function $r_t(s_t, a_t)$ which gives the reward received after transitioning from s_t to s_{t+1} via some action a_t (often denoted as just r_t).

The goal of the agent is to maximise the instantaneous reward, r_t , over time via some policy $\pi(a_t | s_t)$, which maps from \mathcal{S} to \mathcal{A} . The return is defined as $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, which is the accrued reward over time, discounted at each time step k , from time step t , by a discount factor $\gamma \in [0, 1)$. Then the value of a state using policy π is given as $V^\pi(s) = \mathbb{E}[R_t | s_t = s]$. In the case where tasks are of finite time, an episode is the finite sequences of states and the actions taken to transition between those states, ending either when the goal state is reached or when a maximum number of actions, T , have been taken.

Based on these definitions we must choose which function to approximate via learning to obtain the policy. Value function methods focus on estimating the optimal value function, $V^*(s)$. If this is known, the optimal policy at state s_t , is to select the action which maximises $\mathbb{E}[V^*(s) | s = s_{t+1}]$. However, this requires knowing the transition probabilities $P(s_{t+1} | s_t, a_t)$ to find the states s_{t+1} , for all the possible actions at s_t .

3.1.1 Q-Learning

One variant of value based methods are termed Q-learning approaches. In Q-learning the transition probabilities are not known, and do not need to be estimated. The Q-function (or action-value function)

$$Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E} \left[R_t \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}, \pi \right], \quad (3.1)$$

gives the expected return for a given state-action pair and policy, if the action \mathbf{a} is taken in state \mathbf{s} , then the policy π is followed. Q-learning aims to maximise the accrued reward by finding the optimal action-value function

$$Q^*(\mathbf{s}, \mathbf{a}) = \max_{\pi} \mathbb{E} \left[R_t \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}, \pi \right]. \quad (3.2)$$

Mnih et al. [68] approximated the Q-function with a CNN called the DQN. The input to this CNN is an observation of the state of the system in the form of an image. For each action in the discrete action space there is a network output which gives the Q-value for that action, in other words, simultaneously generating $Q^*(\mathbf{s}, \hat{\mathbf{a}})$ for all $\hat{\mathbf{a}} \in \mathcal{A}$. Once the Q-network is trained, it can be used to evaluate the actions at each time step and choose the action with the highest Q-value. This allows the policy to be implicitly learnt, whilst taking the future reward into consideration.

To find the loss function to be used in training we consider how action choice can be related to the action-value function. If the optimal action-value $Q^*(\mathbf{s}_{t+1}, \mathbf{a}_{t+1})$ is known for all possible actions at the next time step, then the optimal policy would be to choose the action with the maximum Q^* value. Hence, Q^* can also be interpreted as recursively maximising $r_t + \gamma Q^*(\mathbf{s}_{t+1}, \mathbf{a}_{t+1})$. Due to this, the difference between $r_t + \gamma Q^*(\mathbf{s}_{t+1}, \mathbf{a}_{t+1})$ and the current Q-value is used for the Q-learning update function

$$L_Q = \mathbb{E} \left[r_t + \gamma Q_{\theta_T}^\pi(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q_\theta^\pi(\mathbf{s}_t, \mathbf{a}_t) \right]^2, \quad (3.3)$$

where θ are the parameters in the Q-network. Therefore, the loss of the CNN is the difference between the expected reward for the full sequence and the instant reward plus the expected reward for the remainder of the sequence, in other words it penalises non-recursive behaviour.

Here θ are the parameters in the Q-network and θ_T are the parameters in a target network which is only updated every C steps. This delay is to make the learning algorithm more stable than

true online learning where an update to Q could change both elements of the loss function at once, possibly causing divergence or oscillations. Another way instabilities in learning are combated is by using experience replay [62] where a set of the agent's experiences are stored, then mini batches are drawn uniformly from this set for training. A limit M is set on the number of episodes to keep, so that as training progresses, older episodes are forgotten and replaced with episodes from the current policy. The experience replay buffer $\Omega = \{e \mid |\Omega| < M\}$ stores each step from an episode in the form $e = \left\{ \left(\mathbf{s}_t, \mathbf{s}_g, \mathbf{a}_t, r_t, \mathbf{s}_{t+1} \right) \mid t = 1 \dots j, j \leq T \right\}$, where \mathbf{s}_g is the goal state and j is the terminating step for that episode (up to a maximum number of actions T). Drawing mini-batches from this selection of episodes smooths the changes in the data and reduces the correlations in single observation sequences.

3.1.2 Actor-Critic Learning

Actor-Critic methods simultaneously predict a policy and a value and use both to learn the optimum solution. An example of this is Asynchronous Advantage Actor-Critic (A3C) learning. This outputs both a policy $\pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)$ and an estimate of the value function $V_{\theta_v}^\pi(\mathbf{s})$ from a neural network. The networks parameterised by θ and θ_v share the main section of the network but have different final layers for output. The advantage of an action is defined by

$$A^\pi(\mathbf{s}, \mathbf{a}) = Q^\pi(\mathbf{s}, \mathbf{a}) - V^\pi(\mathbf{s}). \quad (3.4)$$

This gives insight not only into how well an action performed but also how much better it was than predicted. Since the discounted return R_t can be seen as an estimate to $Q^\pi(\mathbf{s}_t, \mathbf{a}_t)$, and $V^\pi(\mathbf{s}_t)$ is estimated via the network, an estimate of the advantage can be found as $R_t - V_{\theta_v}^\pi(\mathbf{s}_t)$ and used in the loss of both the policy and value estimate. The value loss is

$$L_V = \left(R_t - V_{\theta_v}^\pi(\mathbf{s}_t) \right)^2 \quad (3.5)$$

and the policy loss is

$$L_\pi = -\log \left(\pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t) \right) A^\pi(\mathbf{s}_t, \mathbf{a}_t) - \beta H \left(\pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t) \right) \quad (3.6)$$

where H is the entropy and $A^\pi(\mathbf{s}_t, \mathbf{a}_t)$ is estimated by $R_t - V_{\theta_v}^\pi(\mathbf{s}_t)$. Since the exact R_t is not known, it can also be estimated with

$$R_t \approx r_t + \gamma V_{\theta_v}^\pi(\mathbf{s}_{t+1}). \quad (3.7)$$

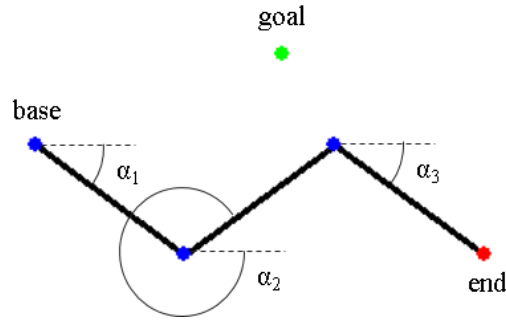


Figure 3.2: 2D arm with three joints

One of the advantages of A3C is its asynchronous nature. A3C uses n instances of an environment, each with their own copy of the networks, parameterised by θ_i and $\theta_{v,i}$ for $i \in [1, \dots, n]$. These networks each feedback to and get updated from the central “global” networks with parameters θ_g and $\theta_{v,g}$. Using many simulators or real life robots in parallel offers a great opportunity to speed up training, which is especially important when data acquisition is slow, such as in robotics. The way in which the networks share updates mean that the exploration and experience from each environment is shared across the system allowing it to learn more efficiently from a diverse sample of data. It also provides the ability to easily vary environments to encourage generalisation across both visual differences and slight behavioural differences which often occur in simulators running on machines with different set-ups. As before, an experience replay buffer can be used by each worker to store a record of the agent’s experiences and sample mini batches for training.

3.1.3 Comparison of DQN and A3C

To compare the performance of the DQN and A3C algorithms, the results in this section look at a mock manipulation problem involving a 2D arm with three joints. The aim is to move the end point of the arm to a goal location in as few steps as possible. The action space for this problem is discrete with 6 different actions. The agent can move the angles α_1 , α_2 , and α_3 (See Figure 3.2) by $\frac{\pi}{8}$ radians in either a positive or negative direction. The maximum number of steps per episode was set to $T = 1000$. The input to both algorithms is an RGB image observation of the system. For this set of experiments, the goal location is fixed for all episodes and the start configuration of the arm is randomly set at the start of each episode.

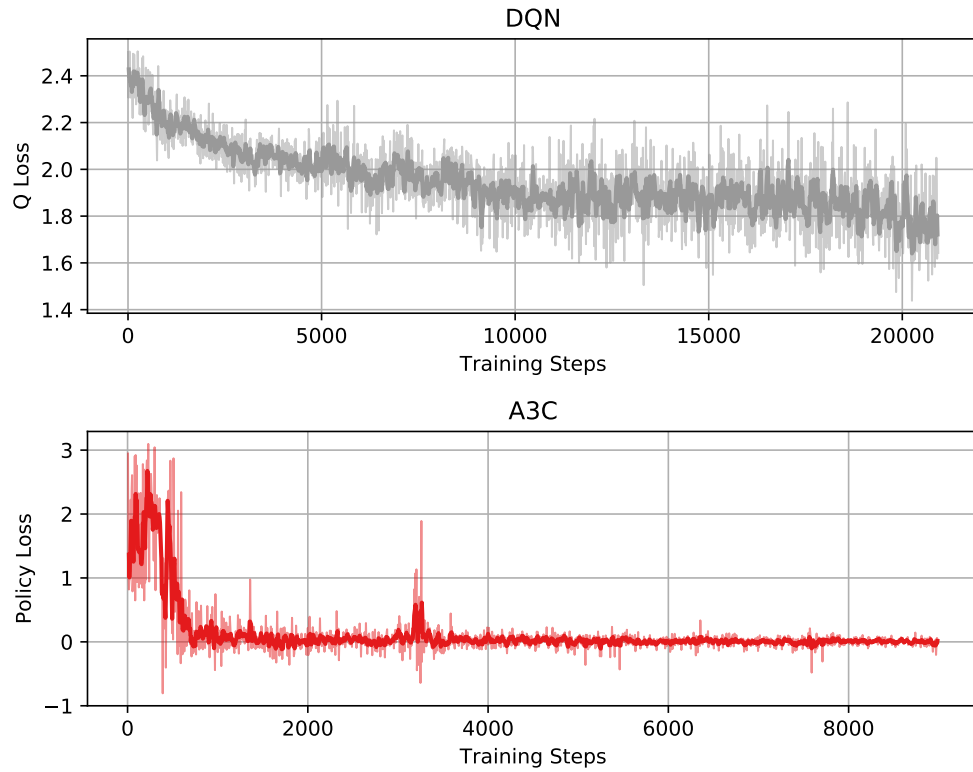
Method	Average Actions	Std Dev
Random	369	± 341
DQN	134	± 99
A3C	15	± 3

Table 3.1: DQN vs A3C Training with a single fixed target across all episodes. Average actions is the average number of actions for the arm to reach the target per episode.

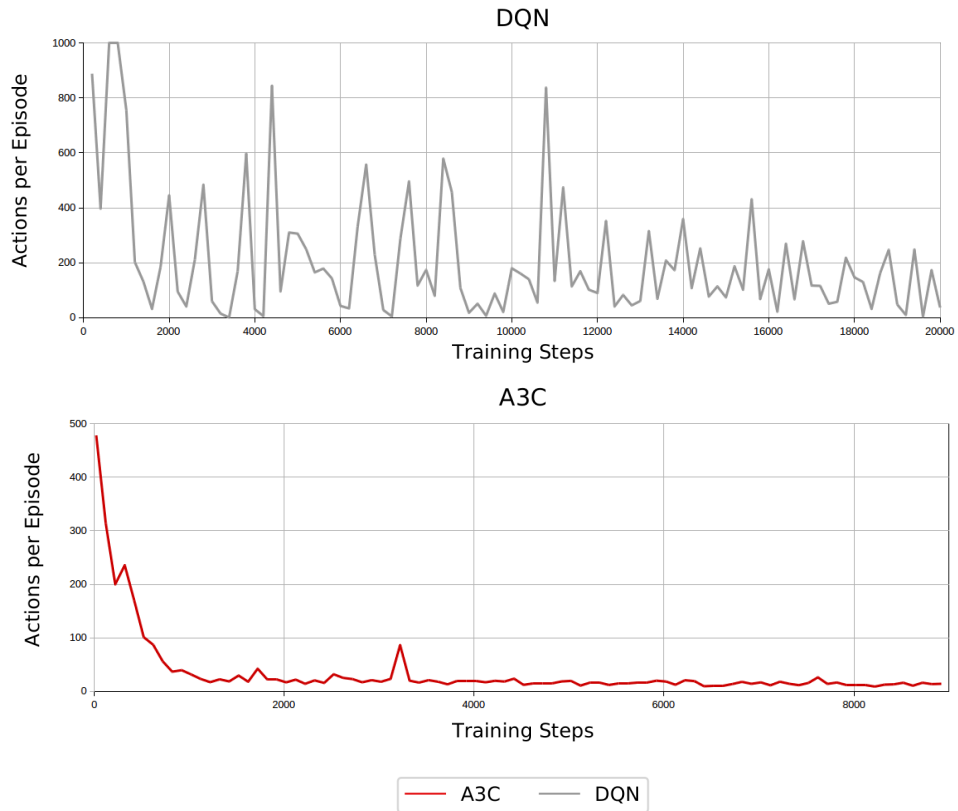
The average number of steps required to reach the goal per episode is used as a metric to judge the performance of trained networks. The networks were trained using Adam optimisers [46] in a Tensorflow [1] implementation. As a base line, it took an average of 369 actions per episode, to reach the goal using a random walk.

The DQN and A3C algorithms both used the experience replay buffer as described above. DQN uses the delayed target network updates while A3C uses the global network with multiple workers each with their own local network.

Table 3.1 shows the final results from both algorithms after convergence. With this single goal we can see that DQN improved on the random policy, reducing the average actions per episode to 134. On the other hand, A3C converges to a much more effective policy, only requiring an average of 15 actions per episode. The graphs in Figure 3.3a show the losses during training for these networks. For DQN it shows the Q-network loss, L_Q , from Equation 3.3 and for A3C it shows the policy loss, L_π , for Equation 3.6. At first the DQN loss trends downwards, seeming more stable than the A3C initial loss. However, after the first 500 iterations of training the A3C loss dramatically reduces, this is likely due to the separate actor and critic networks allowing the algorithm to have a stronger drive of whether the actions taken were better or worse than expected. In Figure 3.3b we can see that the average length of episode during A3C training converges quickly and is less noisy than the same metric during DQN training. The asynchronous nature of the A3C system, with multiple workers, means it receives a greater variance of interactions with the agent and environment over the same number of training iterations. This allows it to generalise quicker to the varied starting states of the arm in comparison to DQN.



(a) Losses



(b) Performance

Figure 3.3: Losses and Performance metric during training for the two algorithms

Next the problem was made more difficult by changing the position of the goal. At the start of each episode, the goal is placed randomly in the play area, within reach of the arm. The RL algorithm now needs to generalise to both the random start point and the random goal placement. When the systems were trained on this problem, from randomly initialised policies, neither converged. The additional variance in the states means there is a more complicated relationship between actions and reward and with no further guidance, the system cannot learn this relationship.

3.2 Curriculum Learning

With a more complex system in mind, a key challenge in reinforcement learning is the trade-off between exploration and exploitation. The aim is to explore the space as much as possible, in order to find more optimal policies, whilst also exploiting the learnt policy to avoid exploring sub-optimal policies. There is also the issue of a varying initial environment which then means more exploration is needed to discover the effects on different states.

Curriculum learning can be used to guide the exploration of the agent, gradually increasing the difficulty of the problem by allowing different initial states. Curriculum learning [13] encourages the network to learn simpler tasks first and then build upon those experiences to solve more complex tasks, just as humans do. This means that the network can exploit its current knowledge during exploration, whilst still being able to gradually learn to generalise to more complex tasks in a structured manner.

For the problem of moving an arm to a target, the task initially solved in Section 3.1.3 was simplified by using a target in a single location across all episodes, with the initial arm state random. Then instead of trying to solve the random target problem from random initialisation, the newly sampled episodes can gradually be made more challenging. This means as training continues, the position of the target for each episode is allowed to be in an increasingly larger space in the play area. The variance of the complexity of the task may be increased in steps or continuously. A combined strategy may gradually increase the complexity up to a threshold, then allow the system to re-converge before gradually increasing complexity again, until the full problem is solved.

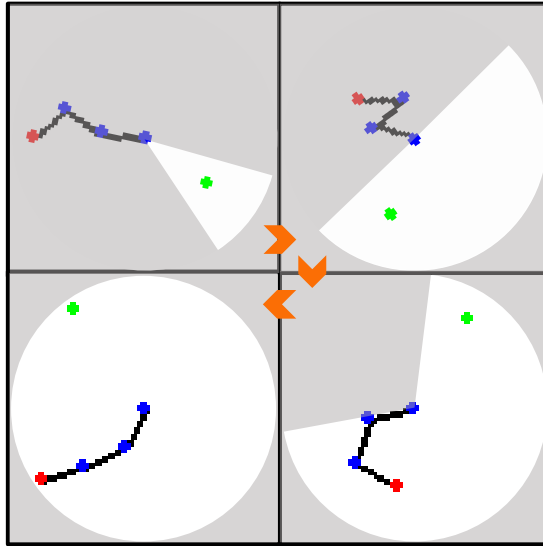


Figure 3.4: Example of curriculum learning by increasing the difficulty of the problem. The possible goal area (white) is increased gradually from a single point (top left), to half of the play area (top right). Then again gradually increased (bottom right), to the full play area (bottom left).

To show the use of curriculum learning, we use our mock problem again. The aim is to be able to move the arm to any goal position in the play area. Training starts with the goal set in a single position, using the networks which were trained on a single goal from Section 3.1.3. Then the position for each episode is randomly initialised to be in a gradually more varied position until it can be placed anywhere in half of the space. Once this stage is reached, the network is allowed to converge. Then it is gradually increased again until the whole play area is available (see the increasing goal area show in white in Figure 3.4). This set of results use the A3C algorithm due to it's much higher performance in Section 3.1.3.

As before, the system had two asynchronous workers running on separate threads on the same machine. The initial training with a single target position from Section 3.1.3 achieved an average of 15 actions. Table 3.2 shows that using the curriculum learning schedule described above, A3C is able to converge to effective policies at each stage of learning. Finally, when the target can be placed anywhere in the play area, an average of only 25 actions per episode is achieved. It is interesting to note that placing the goal only within a quarter of the screen, can actually be seen as a more challenging task than both the whole screen and the single goal. This is because the arm is less likely to be randomly initialised near the goal compared to the whole screen scenario and there is no single solution that can be memorised as in the case of a single goal.

Method	Area Available	Average Actions	Std Dev
A3C	Single goal	15	± 3
A3C	1/4 screen	42	± 12
A3C	1/2 screen	13	± 2
A3C	Whole screen	25	± 6

Table 3.2: Curriculum Learning with A3C. Area available shows change during curriculum learning - states the area of the screen where the target may be placed during those training episodes. Average actions is the average number of actions for the arm to reach the target per episode.

The plots in Figure 3.5 show how, at each point where the goal variance is increased, the performance of the policy gets worse (the actions per episode increases and the reward per episode decreases). It is important to note that each time the problem is made more difficult, it doesn't diverge as far as the initial randomly initialised policy from the single target mode. Figure 3.6 shows the losses from the same set of training. As expected, the policy and value losses diverge each time the problem gets more difficult. It is interesting to see that the entropy loss, H from Equation 3.6, also increases each time the complexity is increased. This indicates a greater level of exploration in the policy, meaning that it is able to find better solutions for the complex cases it has not seen, allowing it not to get stuck in a local minimum from the previous level of complexity.

3.3 Semantically Understood State Spaces

The methods used thus far have used visually simple problems with an image as input. However, the aim of this work is to apply these methods to more complex robotic problems. Even with these simple problem domains the algorithms presented have taken hundreds of thousands of episodes worth of training to converge. When approaching more complex problems, the training steps needed will increase and each episode will take longer to collect, especially where it includes the rendering of visually complex scenes from simulators.

For example, using a fast physics simulator, such as the Mujoco physics simulator [105], it takes approximately 0.01 seconds per rendered simulation step. For the 1,000,000 episodes it took

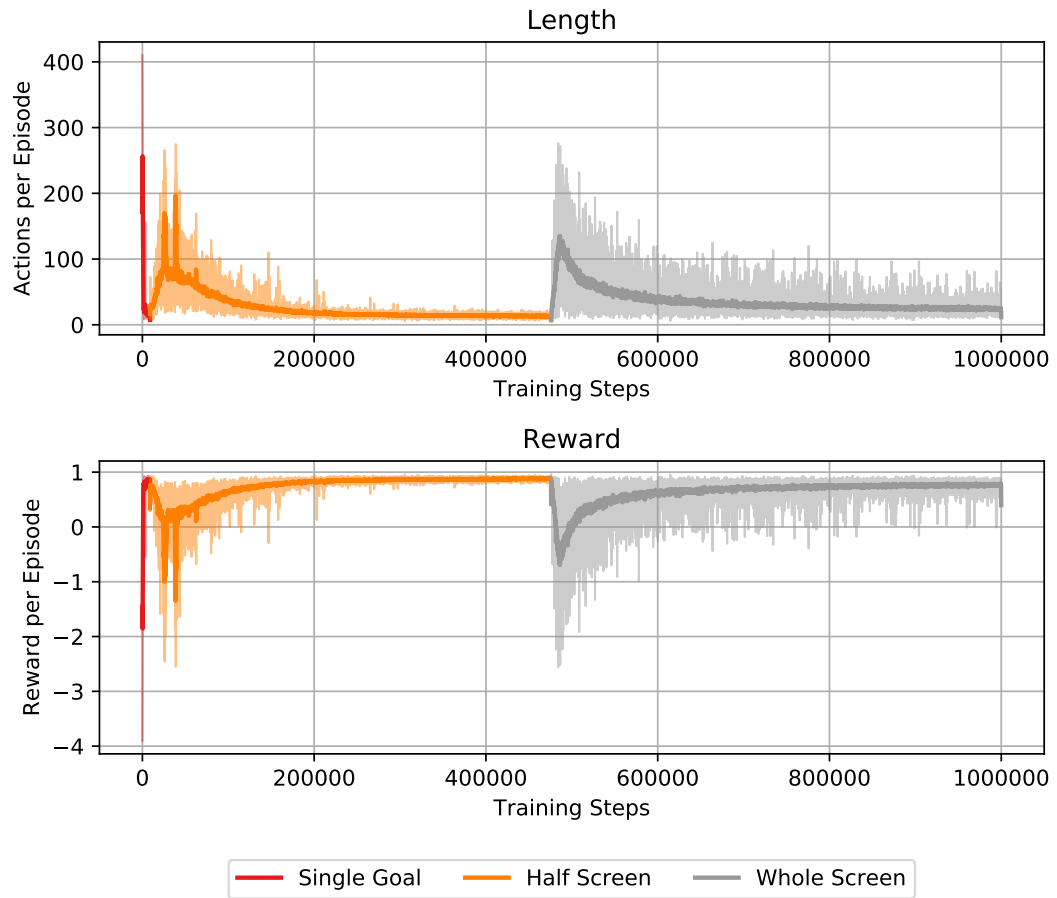


Figure 3.5: Performance of A3C during Curriculum Learning - bold lines show a moving average across 100 episodes

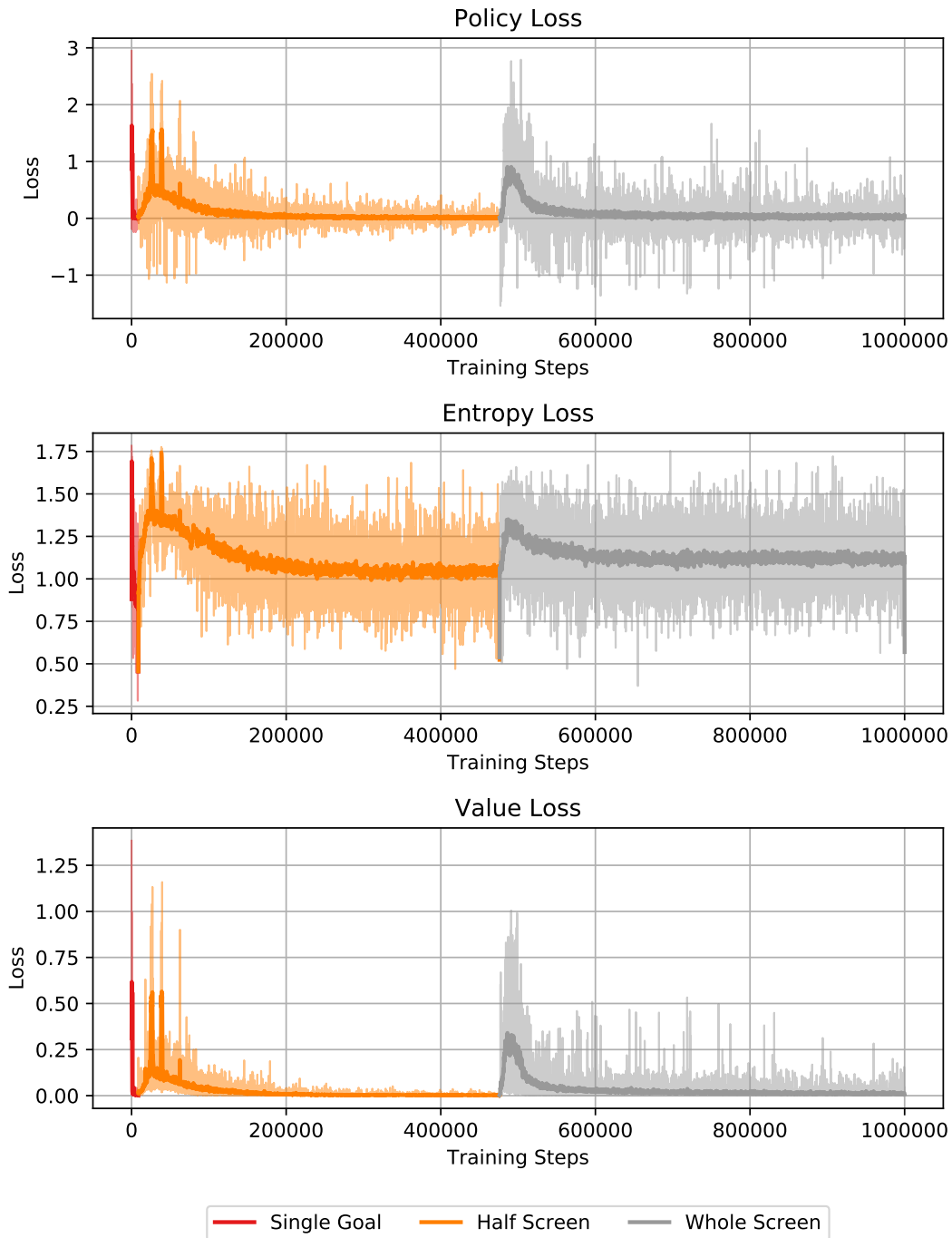


Figure 3.6: Losses of A3C during Curriculum Learning- bold lines show a moving average across 100 episodes

to achieve the final result in Section 3.2, if each episode took a maximum of 50 steps it would still take approximately 5 days just to collect the episodes to learn to solve a similar problem. This makes implementing and testing the techniques above, as they currently are, in a robotics simulator extremely impractical and almost impossible on a physical robot.

In order to use these RL methods with more complex robotics problems, techniques to improve the data efficiency of policy learning need to be employed. Traditionally, approaches used in the RL community use a semantically understood, parameterised state space which fully describes the system, such as euclidean poses or angular positions. This fully observed state, with all relevant information to complete the task, will often be much smaller than an image observation, meaning less data to capture and store. Therefore, this section will focus on using semantically understood state spaces as input, rather than images. This style of input space also allows the implementation of the techniques presented in this section, which aim to reduce the number of agent-environment interactions needed for convergence.

To demonstrate the benefits of these techniques the experiments in this section will use the OpenAI [15] framework with the Mujoco physics simulator [105]. The agent will be the simulated Fetch robot [109] which has an arm with 7 degrees of freedom and a parallel gripper. The environment used in this section is the Reach environment, which involves moving the end effector to a goal location, represented as a red sphere. The maximum number of actions per episode is set to $T = 50$.

3.3.1 Gaussian Process Dynamics Model

In order to reduce the number of costly agent-environment interactions we use Gaussian Processes (GPs) to approximate the dynamics of our system and give uncertainty information. A GP defines a distribution over functions, $P(f)$, for a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ mapping from an input space \mathcal{X} to an output space \mathcal{Y} . This distribution is defined by a mean function $m(\mathbf{x})$ and a covariance function $\kappa(\mathbf{x}, \mathbf{x}')$, also known as the kernel, such that

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), \kappa(\mathbf{x}, \mathbf{x}')), \quad (3.8)$$

where $(\mathbf{x}, \mathbf{x}')$ are all possible pairs in the input domain \mathcal{X} . In most cases, the mean function can be assumed to be zero and all observed data can be zero centred to make calculations easier.

This can then be used to build a regression model, where the GP is treated as a prior and a posterior distribution $P(f | \mathbf{X})$ can be created given some data. Assume we have n previously observed data points $(\mathbf{X}, \mathbf{Y}) = (\mathbf{x}_i, \mathbf{y}_i)_{i=1}^n$ where $\mathbf{y}_i = f(\mathbf{x}_i) + \epsilon$ and $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is Gaussian noise.

In order to fit the GP to the observed data, the kernel κ_θ is parameterised by hyper-parameters, θ . These are fitted by maximising the marginal likelihood $P(\mathbf{Y} | \mathbf{X}, \theta)$ of the GP given the observed data (\mathbf{X}, \mathbf{Y}) . So the optimal parameters are given by

$$\theta^* = \arg \max_{\theta} P(\mathbf{Y} | \mathbf{X}, \theta). \quad (3.9)$$

Assuming we have $\kappa = \kappa_{\theta^*}$, for some new set of input points \mathbf{X}_* we want to make predictions for $\mathbf{Y}_* = f(\mathbf{X}_*)$. We know that \mathbf{Y} and \mathbf{Y}_* should come from the same function, meaning they are jointly Gaussian so we can state

$$\begin{bmatrix} \mathbf{Y} \\ \mathbf{Y}_* \end{bmatrix} \sim \mathcal{N} \left(\begin{bmatrix} m(\mathbf{X}) \\ m(\mathbf{X}_*) \end{bmatrix}, \begin{bmatrix} \kappa(\mathbf{X}, \mathbf{X}) + \sigma^2 I & \kappa(\mathbf{X}, \mathbf{X}_*) \\ \kappa(\mathbf{X}_*, \mathbf{X}) & \kappa(\mathbf{X}_*, \mathbf{X}_*) \end{bmatrix} \right), \quad (3.10)$$

where I is the identity matrix. From this we can calculate the conditional distribution

$$P(\mathbf{Y}_* | \mathbf{Y}, \mathbf{X}, \mathbf{X}_*) = \mathcal{N}(\mu_{*|\mathbf{X}}, \sigma_{*|\mathbf{X}}^2), \quad (3.11)$$

where

$$\mu_{*|\mathbf{X}} = \kappa(\mathbf{X}_*, \mathbf{X}) [\kappa(\mathbf{X}, \mathbf{X}) + \sigma^2 I]^{-1} \mathbf{Y} \quad (3.12)$$

$$\sigma_{*|\mathbf{X}}^2 = \kappa(\mathbf{X}_*, \mathbf{X}_*) - \kappa(\mathbf{X}_*, \mathbf{X}) [\kappa(\mathbf{X}, \mathbf{X}) + \sigma^2 I]^{-1} \kappa(\mathbf{X}, \mathbf{X}_*). \quad (3.13)$$

This may be sampled from to get estimates of the function $f(\mathbf{X}_*)$ at the new points.

Figure 3.7 shows how a GP can be used to artificially increase the number of agent interactions for training. In detail, the observed data is collected via a small number of interactions with the agent and environment, and stored in a replay experience buffer as before, shown in the centre of Figure 3.7. This observed data is then taken from the replay buffer and used to optimise a GP. The input domain for the GP is $\mathcal{X} := \mathcal{S} \times \mathcal{A}$, so $\mathbf{x} = (\mathbf{s}, \mathbf{a})$ where \mathbf{s} is the current state, and \mathbf{a} the action to be taken. We represent the dynamics of the simulator as

$$\mathbf{s}_{t+1} = \mathbf{s}_t + D(\mathbf{s}_t, \mathbf{a}_t) + \epsilon, \quad (3.14)$$

with ϵ (Gaussian system noise) and $D(\mathbf{s}_t, \mathbf{a}_t)$ (unknown transition dynamics). The Gaussian Process will be used to regress an estimate of the transition dynamics $D(\mathbf{x}_t)$ with the output being the change in the state, $\mathbf{y}_t = \Delta \mathbf{s} = D(\mathbf{x}_t)$. The GP is written as

$$D(\mathbf{x}_t) \sim \mathcal{GP}\left(m(\mathbf{x}_t), \kappa_D(\mathbf{x}_t, \mathbf{x}'_t)\right), \quad (3.15)$$

where m is the mean function (which we will assume to be zero) and κ_D is the kernel function. With a set of observed state-action pairs $\mathbf{X}_{1:n} = \mathbf{x}_1, \dots, \mathbf{x}_n$ and transitions $\mathbf{Y}_{1:n} = D(\mathbf{x}_1), \dots, D(\mathbf{x}_n)$, we can optimise the hyper-parameters of our kernel via Equation 3.9. Then the GP, with optimised hyper-parameters, is queried at a new data point $\mathbf{x}_* = (\mathbf{s}_*, \mathbf{a}_*)$, with state \mathbf{s}_* and action \mathbf{a}_* , to obtain a distribution over expected state updates,

$$P(D(\mathbf{x}_*) | \mathbf{X}_{1:n}, \mathbf{Y}_{1:n}, \mathbf{x}_*) = \mathcal{N}\left(\mu_{*|\mathbf{X}_{1:n}}, \sigma_{*|\mathbf{X}_{1:n}}^2\right), \quad (3.16)$$

where $\mu_{*|\mathbf{X}_{1:n}}$ and $\sigma_{*|\mathbf{X}_{1:n}}^2$ are calculated as in Equations 3.12 and 3.13.

With a given start point, and actions obtained from the RL policy, we can sample from this Gaussian allowing the rapid creation of more episodes to train the RL system. As shown in Figure 3.7, these artificial episodes can be added to a GP generated replay buffer. The same reward calculations as the normal environment are used so these episodes can be added directly to main replay buffer, Ω , which contains the initial agent interactions. This replay buffer can be sampled from to create mini batches of episodes to train the networks.

When using images as the observation, GP modelling is impractical. Firstly, it increases the dimensionality of the input domain, making the time taken for the inverse covariance matrix calculations required in Equations 3.12 and 3.13 prohibitive. Secondly, calculating rewards for the artificial episodes relies on having a semantic understanding of the state, for example, knowledge of where the goal is and if it is reached. This information would not be available to the system if the only state information is the image observation.

Reach Environment

For the Fetch Reach environment, the RL input state is the position of the arm $\mathbf{z} \in \mathbb{R}^3$ and the goal position $\mathbf{g} \in \mathbb{R}^3$. The actions available to the agent are of the form $\mathbf{a} = \Delta \mathbf{z}$. Unfortunately, the transition dynamics of the system are not perfect, so the new position after an

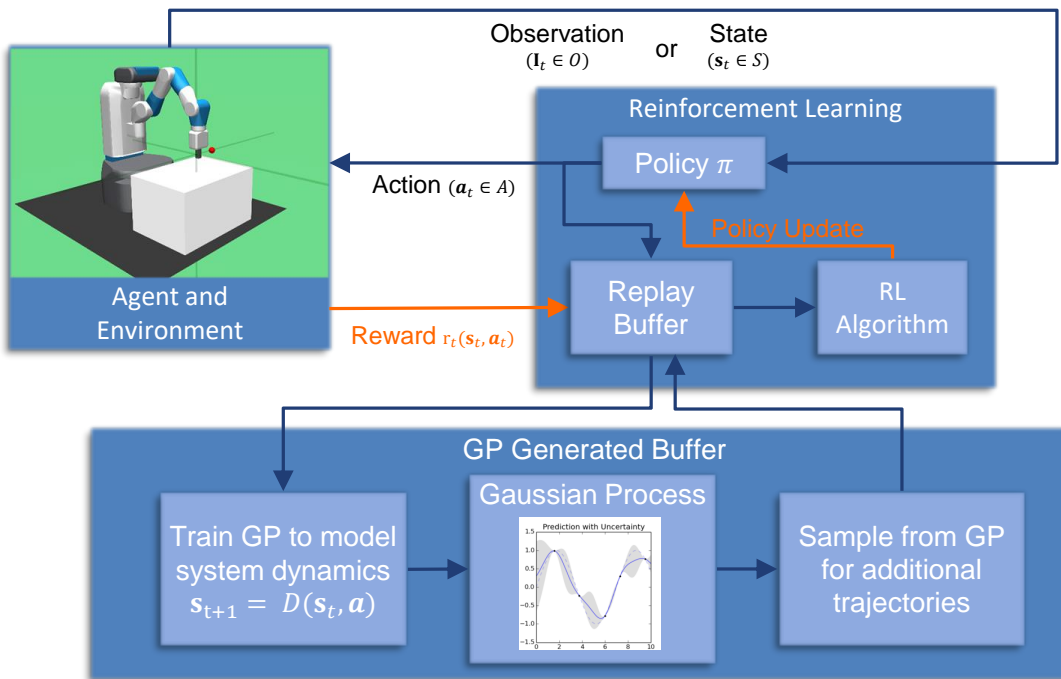


Figure 3.7: Example of Gaussian Process (GP) in RL data flow

action is performed is not always $z + \mathbf{a}$, meaning each episode step requires agent-environment interaction in the simulator in order to get an accurate next state. This means it is an excellent problem to test the benefits of the GP modelling.

Since the only part of the state that changes when an agent performs an action is the arm position z , this is what we use as the input to the GP along with the action, giving $\mathbf{x} = (z, \mathbf{a})$. To start with a random policy is used to collect the initial data consisting of 15 episodes. This is used to fill the replay buffer and to train the GP. Visualisations of the trajectories from these episodes can be seen in Figure 3.8, showing the restrictions of the agent that need to be modelled by the GP, such as the limit at approximately 0.4 on the z -axis where the table is placed. This also shows how much of the state space has not been explored by the initial episodes, meaning the GP needs to generalise to these states.

The GP was trained using the Python GPy package [37] with a standard Radial-basis Function (RBF) kernel. In order to check that the GP is modelling the transition dynamics as expected, a series of scaled unit actions in each direction from the simulator were collected from various positions. Both the requested action and the actual change in position were recorded.

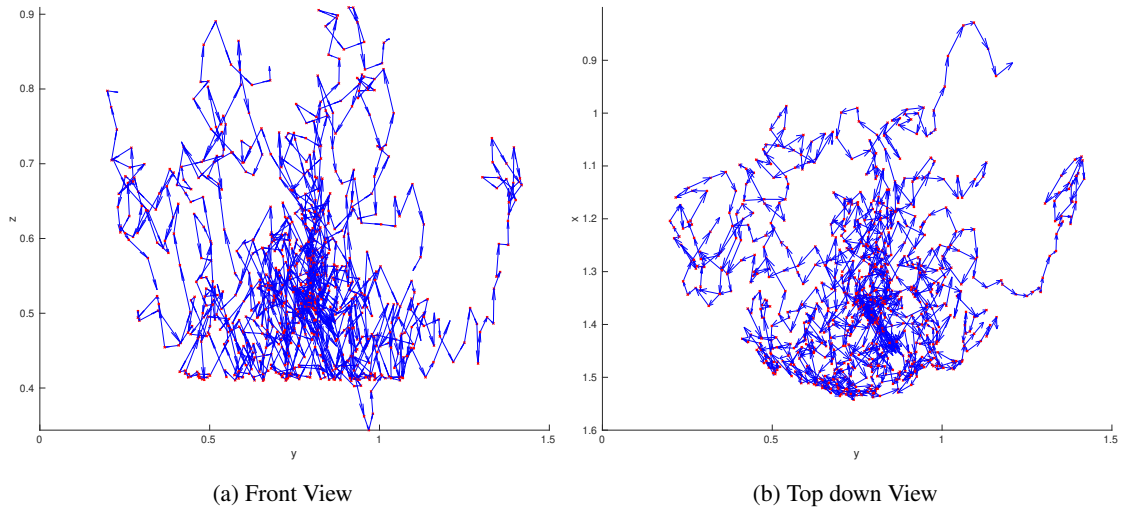


Figure 3.8: Visualisations of the trajectories from 15 random episodes used to train the GP, showing the physical limits of the agent (e.g. lower bound on the z-axis where the table is and the arc at the front of the top down view where the arms reach is limited).

Visualisations of these can be seen in the left column of Figure 3.9, with the pink arrows showing the requested action and the blue showing the change in state. The same data was then collected using the GP and can be seen in the right column of Figure 3.9.

Comparing the two sets of actions it is clear that larger actions taken by the agent are modelled accurately by the GP, with both sets of blue arrows showing that the action tends to be completed in the correct direction but at a smaller step than requested. However, as the actions get smaller the simulator does not always act as expected. Figure 3.9a show that actions taken further from the centre of the state space are not executed as expected with the tendency to move further from the centre, no matter what the requested action was. Figure 3.9b shows that the GP is attempting to model this behaviour but not as accurately as the larger actions. This set of results show that our GP is able to model the transition dynamics of the system with only 15 episodes worth of interactions with the simulator. It also shows that the simulator is not capable of executing small actions effectively, so from now on we limit the minimum action size to a scale of 0.4 of the maximum possible action. Not only can the GP be used to create episodes more efficiently, it has also given an excellent way to inspect and understand the dynamics of the agent, allowing adaptations to the base policy.

Once the GP is trained it is used to sample additional training episodes to fill the replay buffer

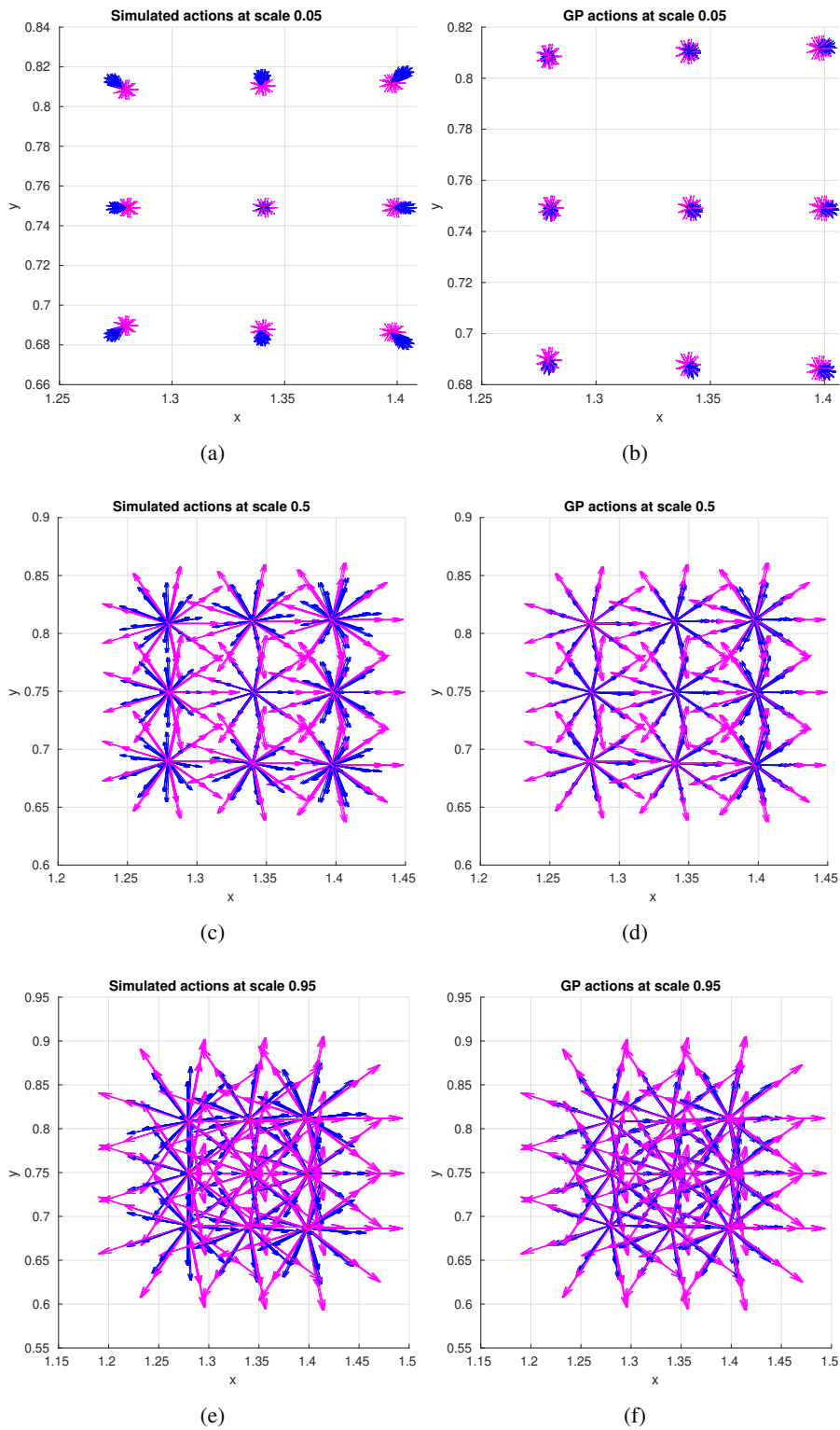


Figure 3.9: Example actions at different scales completed by the simulator (left) or the GP (right). The pink arrows show the requested action and the blue arrow shows the change in state.

and train the neural network without needing further simulation. It takes approximately 0.01 seconds per rendered simulation step, but only 0.0025 seconds to sample a single step from the GP. This equates to saving 75% of the time that would have been spent on collecting simulation examples.

The initial aim is to move the arm from a fixed start position to a fixed goal, which are both the same for every training episode. This is the simplest problem since the system can over-fit to a single trajectory solution. The RL algorithm was able to converge in 1500 episodes of interaction with the GP and only 4 interactions with the simulator during RL training.

The next aim is to move from a random start point to a fixed goal. This requires the policy to learn to adapt to the different potential starting points. Using the 400,000 episodes of interactions with the GP our system converged to 6.69 average actions per episode. In comparison to the number of GP episodes, only 79 episodes of interactions with the actual simulator were used. To collect 400,000 episodes worth of simulator data would have taken up to 55.5 hours assuming each episode took the maximum number of steps, whereas using the GP this upper bound is only 13.8 hours. This magnitude of time savings allows the applications of methods to more complex problems. However, even with these time savings we find that our system does not converge for randomly placed goals. With randomly placed goals, the agent rarely reaches the goal whilst exploring, often reaching the maximum number of steps for the episode. The sparseness of this positive feedback is not sufficient for RL algorithms to converge.

Environment Setup	Average Actions per Episode	Simulator Interactions	GP Interactions
Fixed s_0	2.00	4	1500
Noisy s_0	6.69	79	400000

Table 3.3: Results for use of GP with Fetch Reach environment with a single goal state s_g

3.3.2 Hindsight Experience Replay

Hindsight Experience Replay (HER) [9] is a powerful technique which allows us to learn from unsuccessful episodes in learning, especially where rewards are sparse and success from random exploration may be limited. Using HER, the goal of an episode is adjusted to be a state the

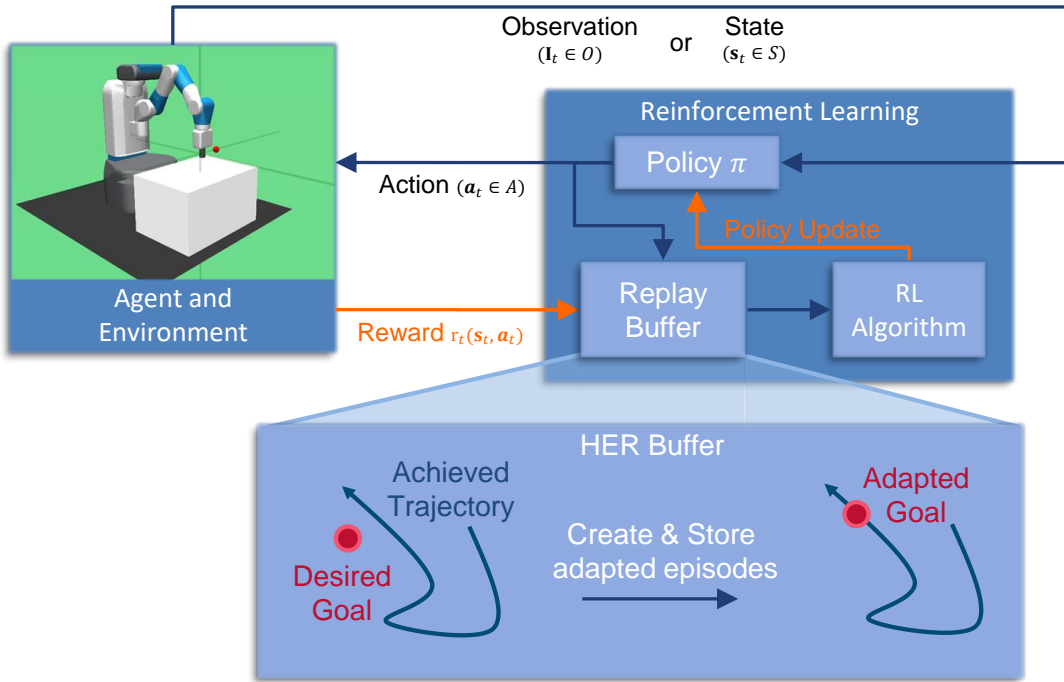


Figure 3.10: Example of Hindsight Experience Replay (HER) in RL data flow

agent achieved during the data acquisition - meaning a successful episode is artificially created. For a given unsuccessful episode $(s_t, s_g, a_t, r_t, s_{t+1})_{t=1}^T$, the goal state $s_g \neq s_1, \dots, s_T$. In this case, we may “replay” this episode with $s_g = s_i$ for some $1 < i < T + 1$ and recalculate the rewards for each step, knowing that it will achieve the adapted goal. Adding these adapted episodes to the experience replay, Ω , as shown in Figure 3.10, means the episode buffer then has more successful episodes to learn from and has a more balanced ratio of successful episodes without needing excessive exploration.

In this section the number of HER additions, α_{HER} , is the number of new episodes created from one unsuccessful episode, by sampling goals without replacement from the achieved states. The number of HER additions per episode can be as large as the maximum length of an episode, T , meaning every state achieved is used as a goal for a new episode added to the HER replay buffer. If an episode has some length, m , where $m < \alpha_{\text{HER}}$, then the HER additions for that episode would be bound to m since there are only m states to sample new goals from.

Using images as the observation for HER is a challenge since there needs to be an understanding of how the goal, and the current state in relation to the goal, is represented in the observation

space in order that it can be adapted for the artificial episodes and to recalculate rewards. Using a state space with some level of semantic understanding allows this to be possible. Whilst it is possible to set a goal as a visual representation as in [71], in use cases like robotics, where there are many ways to potentially achieve a goal, the system is forced to aim for one which is visually similar to the visual goal specified. This also requires either a predetermined set of example goal images, further limiting the potential goal states, or the use of image generation networks to produce realistic goal images, which would require even more data to train.

Reach Environment

This section shows HER used with the same environment described in Section 3.3.1. In this situation, the goal state s_g is randomly placed at the start of each episode. In this set of experiments, the trained GP from section 3.3.1 is also used to reduce the number of interactions with the simulator.

As stated in Section 3.3.1, before the introduction of the HER buffer, the RL system failed to converge for the environment with random goals. The results in Table 3.4 show the number of iterations taken to achieve a result of less than 5 actions per episode. In these experiments, the HER buffer is set to have maximum replay buffer sizes M , which can be varied. Training does not start until 10% of the replay buffer is filled with either episodes sampled from the simulator, the GP, or from the HER additional episodes. The number of episodes taken to fill the buffer to 10% capacity is also shown in Table 3.4. The number of HER additions, α_{HER} , is also varied, with 1 meaning that for each sampled episode a single additional artificial episode is created using a random state from that episode as the goal state.

We can see that using a single HER addition per episode leads to the shortest number of training iterations required to converge. Using only one additional HER episode means that the replay buffer contains successful episodes for positive reinforcement but still maintains the diversity of trajectories needed for the policy to learn how to reach the goal. However, it does require more episode interactions to initially fill the replay buffer. Using a larger replay buffer does not seem to have any significant impact to the speed of convergence but does again make the initial data collection period (before training starts) take much longer. From these experiments it is clear that using a smaller replay buffer with a small number of HER additions gives a good compromise

HER additions α_{HER}	Buffer Size M	Episodes till $ \Omega \geq \frac{M}{10}$	Training Iterations
1	10000	550	12700
10	10000	100	22700
50	10000	300	114100
1	1000000	50150	13100
10	1000000	9450	21900
50	1000000	21600	66200

Table 3.4: Results showing the number of iterations till the average actions per episode converges to below 5 for different quantities of additional HER episodes and different maximum buffer sizes M . Includes episodes until replay buffer is filled to 10% of M (episodes till training starts) and the number of training iterations

by keeping the initial data collection phase and the number of required training iterations small enough to be possible to collect in a reasonable time. Figure 3.11 shows example trajectories collected during training. On the left we see trajectories from the randomly initialised policy used for data collection. On the right we see trajectories from the converged policy from the first line of Table 3.4. The majority of these converged episodes reach their respective goal point via a relatively direct trajectory. However, we can see two of the trajectories do not smoothly reach the goal point, despite the fact that they move in the correct direction initially. It is encouraging to see that even when these trajectories do not perform optimally, they do not diverge completely but stay within the area of the goal, for example, the purple trajectory in the bottom right of Figure 3.11d starts moving towards the goal effectively, then deviates by moving to the side, before self correcting and moving in the direction of the goal again.

3.4 Conclusion

The techniques explored within this chapter are the basis for how RL can be effectively applied to robotics problems. Initially looking at appropriate RL algorithms to use, the results showed that A3C outperformed DQN in our simple problem, both in terms of performance once converged

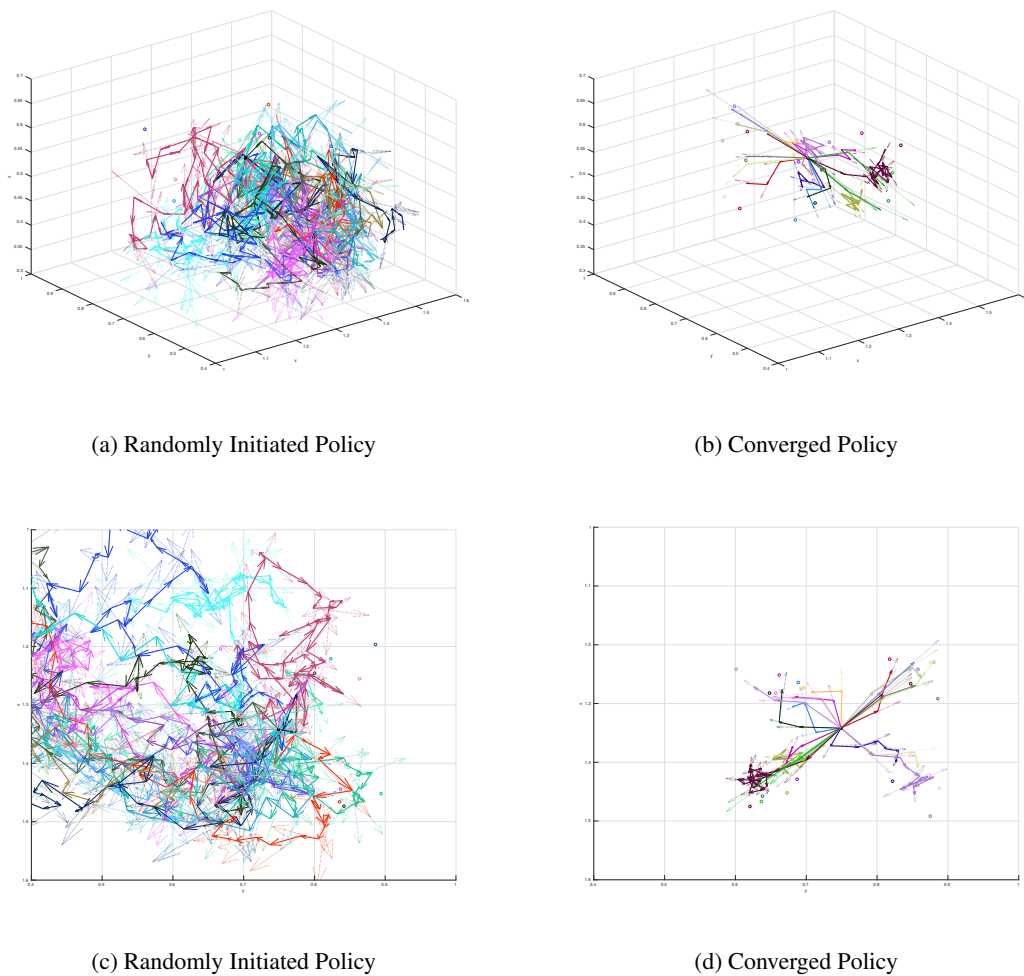


Figure 3.11: Example trajectories of episodes sampled from the initial randomly initialised policy and the converged policy using 1 HER addition per acquired episode and replay buffer maximum size of 10000

and the speed of the convergence. However, the issue of applying this to more complex problems means that a structured learning approach is needed. The use of curriculum learning along side A3C is shown to be incredibly useful in guiding the system, exploiting knowledge from easier problems as it learns. The mock problem shows that the system can be used with image based inputs, however, this slows down both data collection and learning.

To improve the efficiency of the system when moving to a more complex robotic domain, Section 3.3 used a semantically understood state. The time taken to collect data in a physics simulator means that the standard A3C technique takes too long to converge for practical use. Using the example of the Fetch robot Reach environment, the work in this chapter showed that the use of GP modelling provides up to a 75% speed increase.

The problem of sparse positive rewards and lack of sufficient exploration means that the system needs more help. Introducing HER artificially increased the number of successful episodes in the replay buffer. The results show that adding a single additional positive episode for every other episode collected allows the system to converge in just over 12,000 training iterations. The results also showed that adding more positive episodes can have a negative effect on the speed of convergence, with the addition of 10 HER episodes causing the performance to decrease by approximately 44%. This shows the importance of variation and exploration in the training episodes for RL.

The draw back of these methods is that they use a semantically understood state at all times. This can be less practical in the robotics domain where these states may not be available at deployment. Instead, we often have access to image observations as in the results from Section 3.1, but these cannot be used with GP modelling and HER which are necessary for the convergence on more complex tasks.

Chapter 4

Reinforcement Learning with Varied Observability

So far we have considered using RL in situations where the only available observation is an RGB image or some semantic state. The first case is convenient since RGB sensors are readily available and relatively simple to simulate. However, even small images lead to large observation spaces, with a small 600×600 RGB image giving 1,080,000 inputs per frame. Using images with low dimensionality, learning a policy with a discrete action space is challenging but feasible, as shown in Section 3.2. However, when introducing continuous action spaces, the large dimension of observations mean that we require significantly more data.

Where a semantically understood state is available, the RL input tends to be much smaller and more practical for training complex systems. This form of input also allows the methods of Section 3.3 to be used. HER is impractical with images due to the lack of semantic understanding for creating new episodes, with respect to both adapting the goal and recalculating the reward. The same is true for GP modelling and GPs do not perform well for high dimensional data.

Unfortunately, most environments in which tasks need to be solved are not fully observable. This means some of the semantically understood state, which is needed to complete the task every episode, is not available. However, this partial observation can be enough to complete the task some of the time. Instead of relying on purely image based inputs when the full semantic state is unavailable, using the available semantic information along with features extracted from

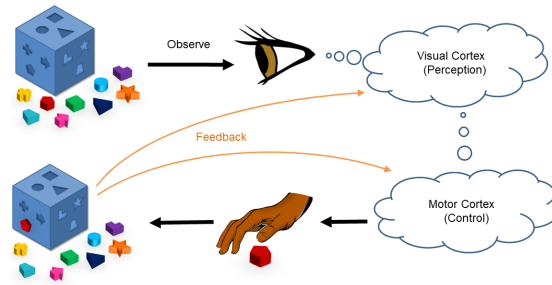


Figure 4.1: Human perception and control are separate systems requiring different feedback.

image observations can improve the efficiency of our training for a specific task.

It is perhaps unsurprising that even incomplete semantic data helps, since training directly from perception through to control is a complex relationship with poorly constrained supervision. For infants, interaction with the environment allows the visual parts of the brain to learn how to perceive the world, even if they fail to complete the task at hand. Learning to interpret our surroundings and then to interact with them are learnt simultaneously but not necessarily as one continuous system. Most visual input is passed through the primary visual cortex, then proceeds forward in two “streams”: ventral and dorsal. The ventral carries information such as object types, whilst the dorsal stream carries information involving the location and movement of objects via the parietal lobe to the pre-motor cortex [52]. There are different types of feedback to train these two different streams. This is visualised in Figure 4.1, where the perceptual understanding is learned in the visual cortex, and the control in the motor cortex, with connections between the two. Inspired by this, this work considers using interaction with the world to co-train separate visual perception and control systems, each with their own feedback but learning from the same interactions.

This chapter shows how an auto-perception network can be used to learn an effective state space for RL. We call this approach Auto-Perceptive Reinforcement Learning (APRiL). Unlike previous RL techniques which try to learn an encoded state space, the proposed solution generalises across all levels of state observability. When the semantically understood state space is completely or partially observable at training time, it is used to condition the learning of a representative feature space. The conditioned auto-perception network can be used to create a feature space which includes this knowledge but is not limited to it - allowing retrieval of other auxiliary information from the observations which have not been considered by the developer.

Table 4.1: Comparison of different works using Reinforcement Learning and AEs

	Use of AE	RL method	RL Input Space	Goal conditioned	Semantics in RL input
Lange and Reidmiller [53]	Encode image	FQI [26]	Latent space	No	None
Finn et al. [29]	Encode image	Gaussian Controller + Guided Policy Search [58]	Robot state + latent space	Implicit in image encoded into the latent space	Partially
Stadie et al. [100]	Encode image for augmented reward	DQN [68]	Images	Implicit in image	-
Kimura [45]	Pretrain RL network	DQN [68]	Images	Implicit in image	-
Nair et al. [71]	Encode image, Calculate reward, Generate data	TD3 [33]	Latent space	Conditioned with a point sampled from the state space	None

4.1 Autoencoders in RL

As discussed in Chapter 2 (Section 2.2.5), there have been many different techniques used to reduce dimensionality and extract information from high dimensional state spaces like images. One of the most effective tools for extracting lower dimensional representations of images are deep Autoencoders (AEs). Table 4.1 compares the uses of AEs in RL systems. Finn et al. [30] use an AE to create a set of feature points representing positions in the image that describe the environment, for example object locations. Stadie et al. [100] encode a state for training a dynamics model in order to improve exploration by increasing curiosity, but still use the raw observation as the input to the learning system. Lange and Riedmiller [53] use a deep AE to compress a visual input to a low dimensional feature space, which is not semantically understood. This improves the reinforcement learning data-efficiency. Kimura [45] uses AEs as pre-training for a DQN system. However, none of these approaches can exploit valuable RL techniques such as HER. For example, Lange and Reidmiller’s work does not have sufficient semantic understanding in the features in order to adapt the episode to a new HER goal. Kimura’s requires images, for fine-tuning of the network, which again cannot be adapted for a new goal.

Nair et al. [71] use an encoder-decoder system to learn a latent space which can be used to sample goals, provide a lower dimensional, structured input for RL and to compute a reward signal for reaching the goal. As discussed in Section 2.2.5, this introduces potentially undesirable limitations on how the agents can reach the goal. Nair et al. also assume that only the image is available to the RL system, they do not consider cases where we may want to make use of any other state information that is available - meaning information is wasted.

4.2 Auto-Perceptive Reinforcement Learning (APRiL)

Fig. 4.2 shows the outline of the proposed APRiL approach. The blue arrows show the data flow at deployment. The observation image of the agent and environment is passed to the encoder which provides an encoded state, which may be completely or partially semantically understood. This is then passed to the trained reinforcement learning system which selects an action which is passed back to the agent to be executed.

The flow of the data during training is depicted as the orange arrows in Figure 4.2. The optional

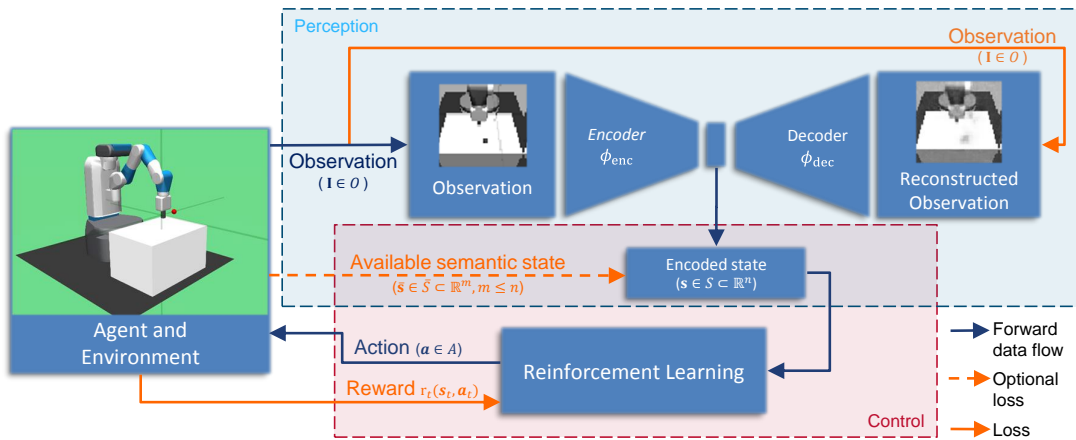


Figure 4.2: Overview of APRiL. The optional loss and the $|\mathcal{S}|$ determines how much of the encoded latent space is semantically understood. Blue arrows: the data flow in the forward pass, Orange arrows: the data flow in the backward pass.

loss can be used if semantically understood knowledge of the state is partly or fully available. The perception network is trained independently on data collected with an initial random walk policy from the RL system. The RL block can be trained using data from the agent and from a pre-trained Gaussian Process (GP) which models the dynamics of the system (as in Section 3.3.1). This means that the RL system can obtain vast quantities of data points without having to run them all through a physics simulator, speeding up the process.

The RL system and the auto-perception network are independent networks, which can be trained concurrently with much of the same data but do not need to be trained end-to-end as they exploit different types of supervision. Each of the following auto-perceptive cases, which have different levels of semantic observability, can be integrated with the RL in the same training framework.

In the case of the encoded feature space being entirely semantically understood, the auto-perceptive network is trained with data collected for the initial random exploration. The same data can be used to train the GP to learn the dynamics, in order to train the control network. The auto-perceptive network and the control network can be co-trained in parallel and tested individually before being integrated. The perception network infers an approximated state from an observation and then passes this approximate state to the RL network without the RL needing to see any images during training. This means the system does not need access to the robot state at run time and can predict actions with only visual input, but it is not necessary for it to be

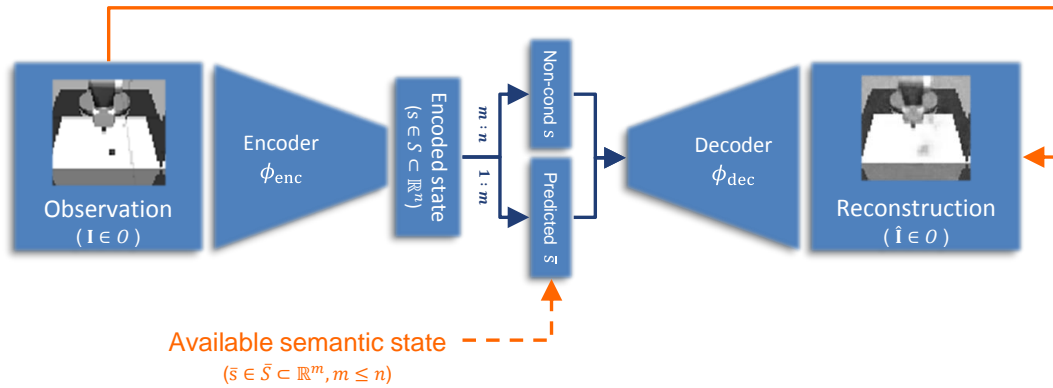


Figure 4.3: Data flow through an Autoencoder perception system

trained in an end-to-end manner, allowing RL to benefit from HER and GP modelled transition dynamics.

When the encoded feature space is partially semantically understood then the auto-perceptive network is still pretrained on random data but the encoder arm is used to get the encoded state for input to the RL system. Therefore, the RL system only has to interpret the low dimensional feature space coming from the auto-perceptive network, not the raw pixel values. This means that training is focused on solving the control problem. Techniques such as HER are still feasible since we have a predetermined understanding of some of the feature space being used by RL.

The final case is where there is no semantically understood state available. This is similar to Lange and Riedmiller’s work [53] where the encoder feature space had no predetermined semantic meaning. This case still allows a lower dimensional state space to be learnt from the visual input even when there is no semantic state available during training.

4.3 Auto-Perception Networks

To use visual observations from the environment, without needing to train an RL system with a high dimensional state space, the observations need to be embedded into a smaller intermediate state. Modern deep learning techniques such as AEs or Variational AEs can be used to do this.

4.3.1 Autoencoder

An Autoencoder (AE) allows the visual space to be mapped to a lower dimensional latent space. The encoder uses the observations of the agent and environment in the form of an image, transforming it to the latent space with the function $\phi_{\text{enc}} : \mathcal{O} \rightarrow \mathcal{S}$, whilst the decoder arm transforms from the latent space to a reconstructed image $\phi_{\text{dec}} : \mathcal{S} \rightarrow \mathcal{O}$.

In the RL notation, the AE takes the image observation at time t as input and compresses it down to the latent space $\mathbf{s}_t = \phi_{\text{enc}}(\mathbf{I}_t)$ and the output is a reconstruction of that image at time t , $\hat{\mathbf{I}}_t = \phi_{\text{dec}} \circ \phi_{\text{enc}}(\mathbf{I}_t)$. To train the network, the reconstruction loss is a pixel-wise loss against the input

$$L_{\text{recon}} = |\mathbf{I}_t - \hat{\mathbf{I}}_t|. \quad (4.1)$$

We denote the semantic state space as, $\bar{\mathbf{s}}_t \in \bar{\mathcal{S}}$, which contains all available information from the environment that has a predefined semantic meaning. The optional conditioning loss is the absolute difference between a section of the encoded state space and the semantically understood state. The slicing of the encoded state space, into predicted $\bar{\mathbf{s}}_t$ and non-conditioned \mathbf{s}_t is illustrated in Figure 4.3. In the case where $\mathcal{S} \subset \mathbb{R}^n$ and $\bar{\mathcal{S}} \subset \mathbb{R}^m$, with $m \leq n$, then the conditioning loss is

$$L_{\text{cond}} = |\mathbf{s}_t^{1:m} - \bar{\mathbf{s}}_t|. \quad (4.2)$$

The full loss for the visual perception network is

$$L_{\text{VP}} = L_{\text{recon}} + \omega L_{\text{cond}}, \quad (4.3)$$

where ω is a weighting which determines how strong the conditioning is. The learnt feature space can be:

1. entirely conditioned to be semantically understood as the observable state ($m = n$, $\omega \neq 0$),
2. partially conditioned with some learnt features relating to the observable state and some auxiliary features with no predetermined semantic meaning ($m < n$, $\omega \neq 0$),
3. or not conditioned with learnt features having no predetermined semantic meaning ($\omega = 0$).

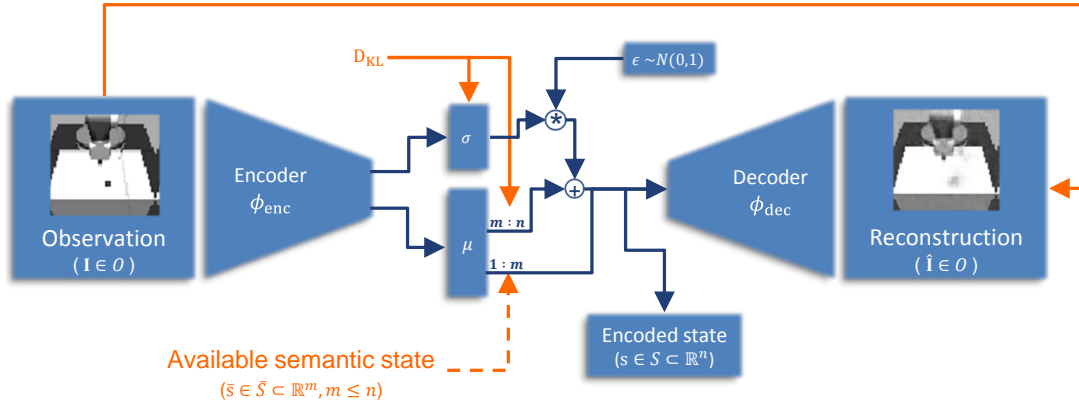


Figure 4.4: Data flow through a Variational AE perception system

This network can be trained using data from initial random exploration and fine-tuned during reinforcement learning if needed.

The encoder, ϕ_{enc} , in this work uses 6 convolution layers each of which halves the height and width of the image. A single fully connected layer then produces the encoded state. The decoder, ϕ_{dec} , takes this encoded state as input into a single fully connected layer, from which the output is reshaped to allow 6 de-convolution layers outputting the reconstructed observation.

4.3.2 Variational AE

APRiL can also employ a Variational AE (VAE) to encode the latent space. VAEs learn a probability distribution as the latent space allowing greater generalisation and a smooth latent space. The encoder gives a set of means and standard deviations such that $\phi_{\text{enc}}(\mathbf{s}_t | \mathbf{I}_t) = \mathcal{N}(\mu, \sigma^2)$, where μ and σ are of size n . The decoder gives a reconstruction of the input image $\hat{\mathbf{I}}_t = \phi_{\text{dec}}(\mathbf{s}_t)$, where $\mathbf{s}_t \sim \phi_{\text{enc}}(\mathbf{s}_t | \mathbf{I}_t)$. In addition to our reconstruction loss, the assumption of some prior $P(\mathbf{s}_t)$ is enforced. This is done by minimising the Kullback–Leibler divergence (D_{KL}), which shows the divergence between two distributions. The prior used over the latent variables is $P(\mathbf{s}_t) = \mathcal{N}(0, 1)$. This gives the variational loss as

$$L_{\text{var}} = D_{\text{KL}}\left(\phi_{\text{enc}}(\mathbf{s}_t | \mathbf{I}_t) \parallel \mathcal{N}(0, 1)\right) = \frac{1}{2} \sum_{i=1}^n (\sigma_i^2 + \mu_i^2 - \log(\sigma_i^2) - 1). \quad (4.4)$$

In order to use this VAE with the semantic state space $\bar{\mathcal{S}}$, part of this state must be conditioned.

The L1 latent space conditioning loss from Equation 4.2 can be used. So the loss for the semantic state conditioned elements in the VAE are

$$L_{\text{cond}} = |\mu_t^{1:m} - \bar{s}_t|. \quad (4.5)$$

The remaining elements will use the standard variational loss, giving the full conditioned variational loss as

$$L_{\text{VAE}} = L_{\text{cond}} + \beta D_{\text{KL}} \left(\phi_{\text{enc}} \left(\mathbf{s}_t^{m:n} \mid \mathbf{I}_t \right) \parallel \mathcal{N}(0, 1) \right), \quad (4.6)$$

where β is a constant used to vary the variational loss. Finally, the full visual perceptible loss for the VAE is given by

$$L_{\text{VP}} = L_{\text{recon}} + \omega L_{\text{VAE}}. \quad (4.7)$$

Again this method can be conditioned differently depending on how much semantic data is available during training.

It is important to note that despite the naming of the conditioning loss, this is different to Conditional Variational Autoencoders (CVAEs) [99]. CVAEs take wanted attributes (equivalent to our known semantic state) as input to both the encoder and decoder. In comparison we want our encoder to be able to learn this attribute along with the variational latent space from the image. Then the decoder can use the learnt attribute and the sample from the learnt latent space as the input.

The encoder, ϕ_{enc} , for the VAE in this work uses 6 convolution layers each of which halves the height and width of the image. A single fully connected layer then produces both μ and σ . The semantic state conditioned part of μ is used directly from this output. The σ output is multiplied by the value sampled from the unit normal and then added to the non-conditioned part of μ . This means the predicted σ is the same size as only the non-conditioned state. This non-conditioned state and the semantically conditioned state are then concatenated back together to form the full encoded state. The decoder, ϕ_{dec} , takes this fully encoded state as input into a single fully connected layer, from which the output is reshaped to allow for 6 de-convolution layers outputting the reconstructed observation as before. The flow of data between the end of

the encoder and the input of the decoder, as described above, is illustrated for clarity in Figure 4.4.

4.4 RL with Auto-Perceptive Networks

The encoded state from the auto-perceptive network can be used with any standard RL algorithm. In this work we focus on the actor-critic method detailed in Section 3.1.2.

We have two ways to train APRiL. In the first case where $m = n$, then the policy loss from Equation 3.6 becomes

$$L_{\pi} = -\log \left(\pi_{\theta} \left(\mathbf{a}_t \mid \bar{\mathbf{s}}_t \right) \right) A^{\pi} \left(\bar{\mathbf{s}}_t, \mathbf{a}_t \right) - \beta H \left(\pi_{\theta} \left(\mathbf{a}_t \mid \bar{\mathbf{s}}_t \right) \right), \quad (4.8)$$

using the semantic state $\bar{\mathbf{s}}_t$. This allows the RL system to be trained with the ground truth semantic state but then run with the state inferred from observations at deployment.

In the second case, where $m < n$, the full encoded state is needed as the input to the RL system with the policy loss becoming

$$L_{\pi} = -\log \left(\pi_{\theta} \left(\mathbf{a}_t \mid \phi_{\text{enc}}(\mathbf{I}_t) \right) \right) A^{\pi} \left(\phi_{\text{enc}}(\mathbf{I}_t), \mathbf{a}_t \right) - \beta H \left(\pi_{\theta} \left(\mathbf{a}_t \mid \phi_{\text{enc}}(\mathbf{I}_t) \right) \right), \quad (4.9)$$

with the observation \mathbf{I}_t at time t . In this case the policy and value networks cannot be fully trained before the auto-perceptive system is trained since the encoder is needed to find the losses. However, the policy and value networks may be trained with whatever semantic state is available to start learning, via Equation 4.8. These networks may then be adapted to take input from the additional non-semantic encoded state by increasing the size of the input layer and loading the learnt weights to the appropriate slice of the first fully connected layer. This means that the pretraining can make use of the techniques like HER presented in Section 3.3, but the final RL system can be fine-tuned to take advantage of any additional information available from the image.

4.5 Experiments

To evaluate APRiL we use the OpenAI [15] framework with the Mujoco physics simulator [105]. The networks are trained using Adam optimisers [46] in a Tensorflow [1] implementation.

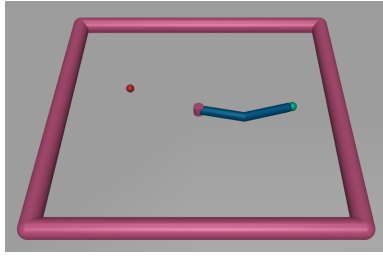


Figure 4.5: Mujoco Reacher simulation with goal (red sphere)

4.5.1 2D Reach Environment

This first set of experiments uses a Reacher problem similar to that in Section 3.2. This environment only has two joints on the arm and has a more visually complex rendering (see Figure 4.5). The goal is to move the end point of the arm to the goal (red sphere). The action space is a vector of the change in joint angles $\Delta \alpha \in (-0.1, 0.1)$. The available semantic state is the 2D pose of the end point of the arm u and the 2D pose of the goal. The maximum number of steps per episode is 50 and the average episode length using a random policy is 49.46.

The first variation of APRiL tested in this section uses an unconditioned AE as the auto-perception network with latent space size $n = 8$ and $\omega = 0$, which is similar to [53]. Some example reconstructions can be seen in Figure 4.6d, showing the ability of the network to reconstruct the position of the arm relatively well. Using the latent space from this auto-perceptive network as input improves on the random policy, but only achieves an average of 44.30 actions to reach the goal. The heatmap in Figure 4.7b shows that the policy has settled to almost a single trajectory and has not generalised to different goals, with the average steps taken per episode reaching the maximum of 50 for most goal locations.

To make the use of the semantic state available, the next experiment uses a fully conditioned AE as the auto-perception network. This correlates to the first case from Section 4.3.1, where $m = n = 4$. Initially the policy is trained with the ground truth semantic state as the RL input. Due to the small action sizes, the converged system takes an average of 31.02 actions to reach the randomly placed goal. The heatmap in Figure 4.7c shows that this policy has generalised much better than the perception based network, with most goal locations having an average episode length of less than the maximum episode length.

The fully-conditioned auto-perceptive network is trained on the initial random buffer episodes.

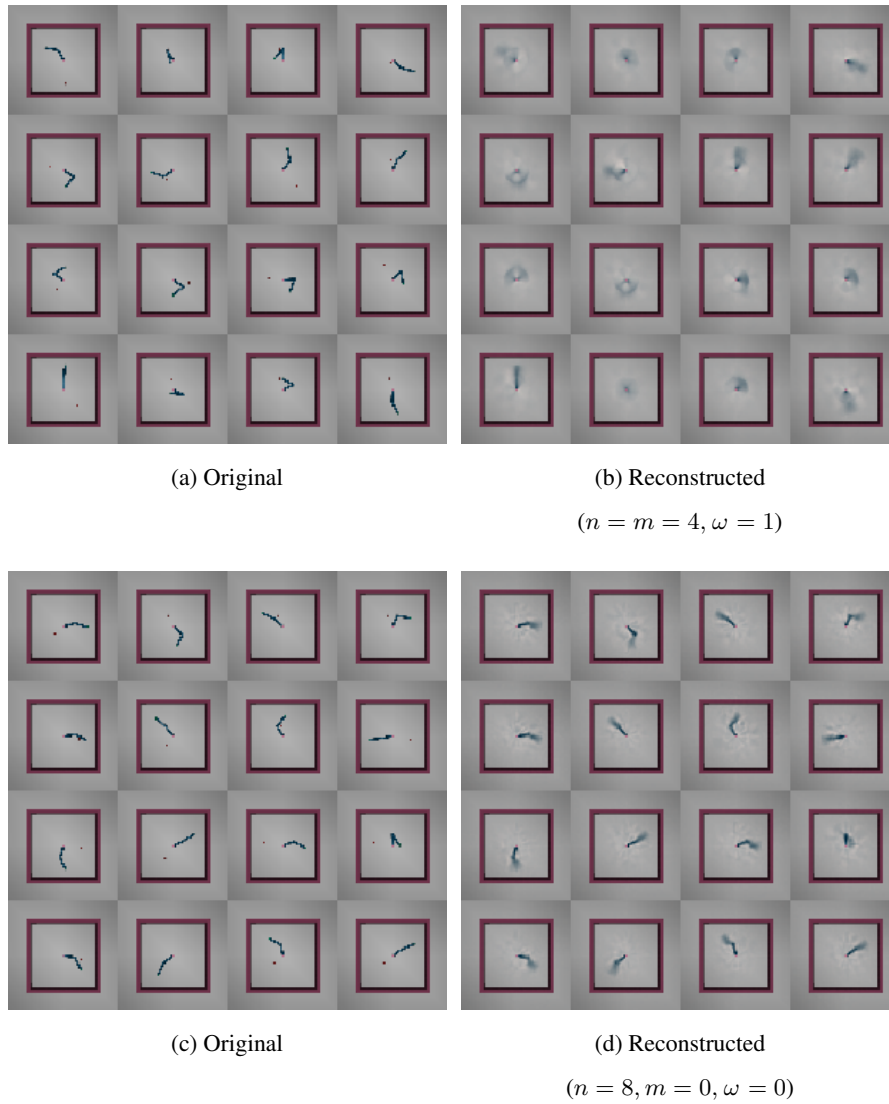


Figure 4.6: Reconstructions from the auto-perceptive networks for the Reacher environment

Runtime RL Input	n	m	Average Episode Length	Std Dev
Random Policy	-	-	49.46	± 4.49
Ground truth $\bar{\mathcal{S}}$	-	-	31.02	± 16.46
Perceived \mathcal{S} [53]	8	0	44.30	± 12.94
Perceived \mathcal{S}	4	4	35.02	± 16.51

Table 4.2: Average episode length (actions to complete task) of system trained on the Mujoco Reacher environment.

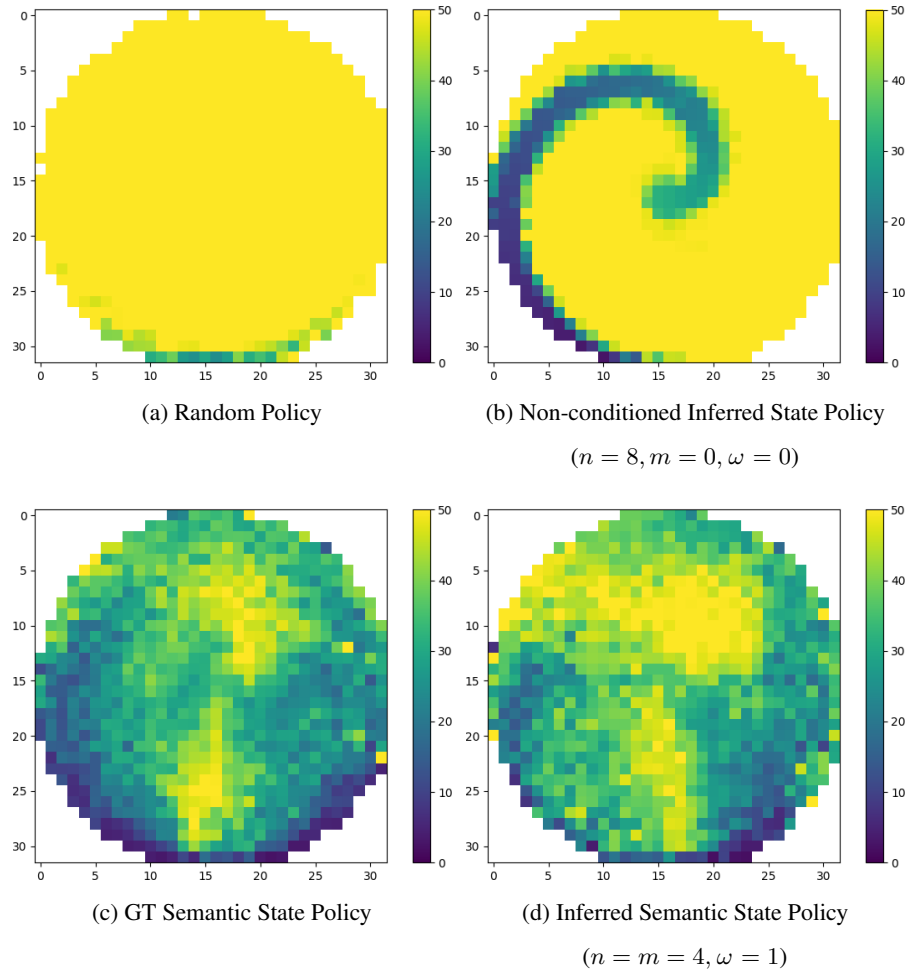


Figure 4.7: Heat maps showing the average number of actions taken to get to goals located across the play area with different policies

Reconstructions from this network can be seen in Figure 4.6b, whilst they are not as clear as the reconstructions from the non-conditioned network, they are able to indicate the position of the arm. Finally, using the latent space of the trained auto-perceptive network, the learnt policy achieves 35.02 average actions per episode. The heatmap in Figure 4.7d shows that the generalisation achieved by the policy has mostly been maintained, with a small drop in performance due to the uncertainty in predicting the images. This auto-perceptive network and policy allows us to complete the problem using only the visual input at run time, whilst we have made use of the semantic information available during training.

4.5.2 Robotic Reach Environment

This set of experiments uses a variation of the Fetch robot reach environment because it has a continuous action space and has a visually interesting environment to test the auto-perceptive system. The aim is to direct the end-effector of the Fetch arm to a goal, g , represented visually by a red sphere. The action space is defined with actions Δz , where z is the position of the end-effector, and the maximum episode length is set as $T = 50$. This set of experiments use an AE as the auto-perceptive network, with grey-scale input images.

Fully Semantic Features

The first experiment uses a fully observed, semantically understood state $\bar{s}_t = (z_t, g)$. We use a random policy to collect an initial experience replay buffer. This data can be used to train multiple aspects of the system. Initially, we train a GP on the transitions taking in $(z_t, \Delta z)$ and outputting z_{t+1} . This allows us to create extra episodes to train our RL system as described in Section 3.3.1. The advantage actor-critic RL system is trained with data created from both the GP and from the agent, including the HER additions to the replay buffer. The data from the random policy and any episodes collected using the simulator are used to train the perception network. In this case the perception network is co-trained such that $\bar{s}_t = s_t = \phi_{\text{enc}}(\mathbf{I}_t)$, which is the first case from Section 4.3.1, when $m = n$ and $\omega \neq 0$. Finally, at test time, the networks can be used together to go directly from vision to actions, following the data flow shown by the blue arrows in Figure 4.2. We compare this to a latent space with no conditioning loss, where $\omega = 0$, which is similar to [53].

Runtime RL Input	Average Episode Length
Ground truth $\bar{\mathcal{S}}$	3.10
Perceived \mathcal{S} ($\omega = 1.0$)	12.42
Perceived \mathcal{S} ($\omega = 0.5$)	17.06
Perceived \mathcal{S} ($\omega = 0.0$) [53]	30.94

Table 4.3: Average episode length (actions to complete task) of system trained on the Fetch Reach environment (no obstacle).

The training of the RL system, using the semantically understood state space directly, converges with only 15 episodes of random policy interactions with the simulator, the rest of the data used is collected from our trained GP. It takes approximately 0.01 seconds per rendered simulation step, but only 0.0025 seconds to sample a single step from the GP. This equates to saving 75% of the time that would have been spent on collecting simulation examples. This is a saving that would not be possible using a traditional end-to-end visual RL algorithm.

Table 4.3 shows the policy achieves an average episode length of 3.1 actions when using the ground truth state space as input. The perception network is trained alongside this. Examples of the reconstructions from the AE can be seen in Figure 4.8, along with reconstructions from the AE without the semantic conditioning (ω). Even though we fully constrain the encoded feature space, and do not enable the system to encode any independent visual properties, the decoder is able to learn how to produce realistic images of the scene from a semantic intermediate state. This includes how to correctly place a fully textured robotic arm. These reconstructions are certainly comparable to the reconstructions without the conditioning loss. However, the reconstruction accuracy is unimportant, the key is that the reconstruction loss encodes meaningful information into the latent space for RL.

At test time we can see the performance benefits from using the visual encoder network to produce the latent space, providing an approximation of the semantically understood state space, as the policy achieves an average episode length of 12.4 actions. The increase in episode length, when compared to using the ground truth state as input, is largely due to the goal or end point

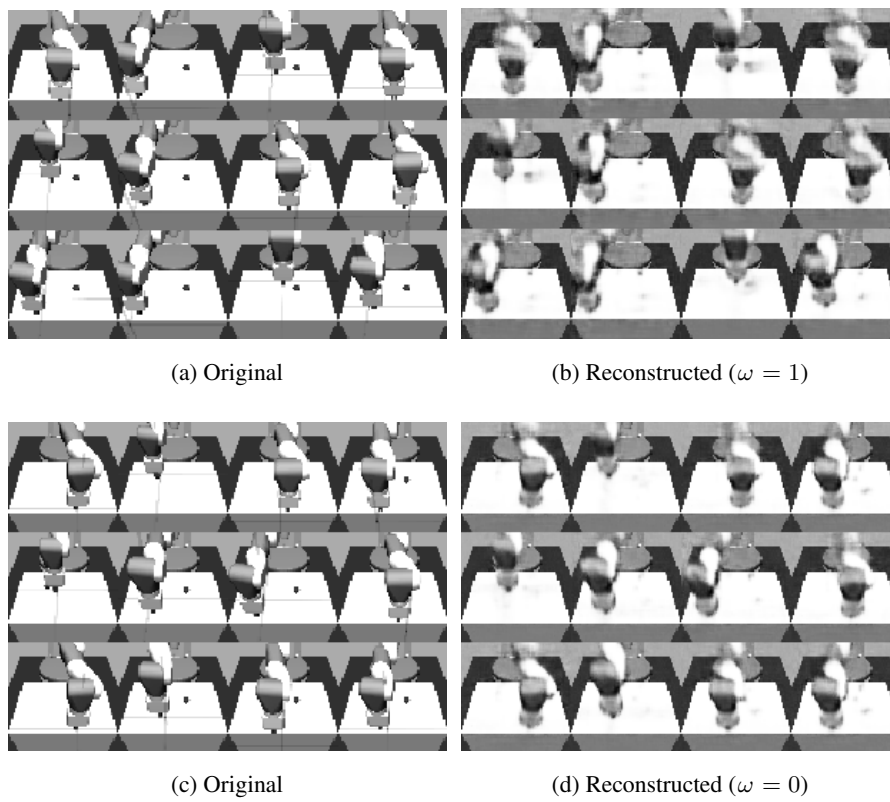


Figure 4.8: Reconstructions from the auto-perceptive network

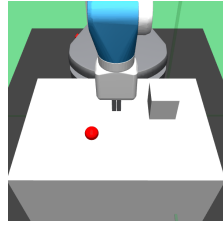


Figure 4.9: Fetch simulation with obstacle (box) and goal (sphere)

Runtime RL Input	n	m	Average Episode Length
Ground Truth $\bar{\mathcal{S}}$	-	-	8.45
GT $\bar{\mathcal{S}}$ and Perceived \mathcal{S} [30]	6	0	5.44
Perceived \mathcal{S} [53]	16	0	37.04
Perceived \mathcal{S}	6	6	28.55
Perceived \mathcal{S}	8	6	20.83

Table 4.4: Average episode length of system trained on an environment with a randomly placed obstacle.

being occluded or out of the field of view, in which case the arm must move to attempt to gather more information about its current state. In these situations, the ground truth algorithm is an unrealistic comparison for a vision based system which will never have full access to the state. However, APRiL provides a more effective system than using the perceived state, \mathcal{S} , not conditioned on the semantic state, $\bar{\mathcal{S}}$, which is similar to [53]. This conditioning allows for a practical image to action method when the semantic state is not known at deployment.

Partially Semantic Features

The next set of experiments introduces an element such that the state is not fully observed via the semantically understood state space. A randomly placed obstacle (box) is added which can affect exploration and the potential solutions to reaching the goal (red sphere), see Figure 4.9. Again we compare the results in this section to a network trained with no access to the available semantic state which is similar to [53]. We also train a system which takes the ground truth semantic state and a separate latent space (in a similar way to [30]) to show that if both are available, the system has all the information it needs.

We first train APRiL on the same state that was available in the previous set-up - the position of the end-effector z and the goal position g . This means that the RL system does not receive any information about the obstacle. As expected, we see a reduction in performance compared to the environment with no obstacle. From 3.10 average actions per episode with no obstacle to 8.45 with obstacles - this equates to approximately a 2.5 times increase in the number of actions. Examples of the reconstructions from the perception network are seen in Figure 4.10b. These reconstructions are comparable to those in Figure 4.8b, with some slight degradation because the scene is more complex but no additional degrees of freedom have been provided in the latent space. It is interesting to note that the decoder arm attempts to reconstruct the obstacle even though it is theoretically not present in the intermediate state.

When testing with the image to action system we see that this leads to worse performance with an average episode length of 28.55 actions (See Table 4.4). This is in comparison to 12.42 actions with no obstacles, equating to approximately a 2.5 times increase in the number of actions which is similar to the decrease in performance seen without perception. This is likely because it has no way of knowing about the obstacle in the encoded state and often mistakes it for the goal, especially if the goal is occluded by the arm.

Next we allow the encoded feature space to be only partially semantically understood. We used a feature space of size $n = 8$, with the semantically understood state \bar{s}_t conditioning only the first 6 elements (i.e. $m = 6$). The rest were driven purely by the reconstruction loss, allowing it to learn what is relevant to understanding the environment. Examples of reconstructions from the perception network can be seen in Figure 4.10d. This trained perception network does a better job of modelling the obstacle and goal as independent objects, however the robot arm has lost a significant amount of visual fidelity. This may be because all systems have been trained for the same number of iterations, despite having more network parameters. Regardless, a high fidelity image of the robotic arm is not important for RL, as long as the position is known.

The proposed RL system using our partially semantically understood feature space as input performs better than the system using the semantic state alone, with an average of 20.83 actions (See Table 4.4). In comparison to the 12.42 actions in the environment with no obstacles, this is only a 1.68 times increase for a more difficult problem. This is approximately a 30% improvement compared to 28.55 average actions taken when using the semantic feature space.

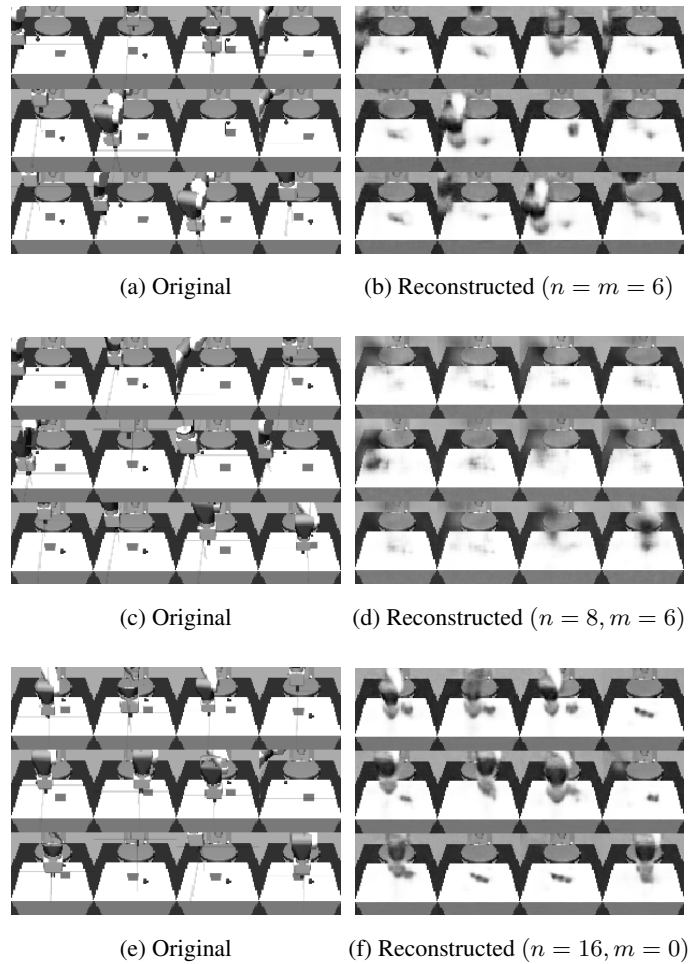


Figure 4.10: Reconstructions from the auto-perceptive network for the environment with obstacles - top: semantic features, middle: partially semantic features, bottom: non-semantic features.

This shows that when we do not have access to the full semantically understood state, our feature space can encode the additional auxiliary information necessary to solve the task better than with the semantic state based perception alone.

Finally we give the perception network complete freedom to encode a state space based purely on the reconstruction loss in a similar manner to [53]. Figure 4.10f shows that this improves the reconstruction as expected since the only feedback given to the encoder-decoder network is reconstruction. However, as we can see from Table 4.4, the performance of the system is significantly worse than those incorporating semantic understanding.

Runtime RL Input	n	m	Network	Average Episode Length
Ground Truth $\bar{\mathcal{S}}$	-	-	-	33.57
GT $\bar{\mathcal{S}}$ and Perceived \mathcal{S} [30]	7	0	AE	74.20
GT $\bar{\mathcal{S}}$ and Perceived \mathcal{S}	7	0	VAE	47.80
Perceived \mathcal{S}	9	9	AE	80.17
Perceived \mathcal{S}	16	9	AE	71.50
Perceived \mathcal{S}	16	9	VAE	67.80

Table 4.5: Average episode length of system trained in the pick and place environment.

4.5.3 Robotic Pick and Place Environment

The next set of experiments use a variation of the Fetch robot pick and place environment. The aim is to direct the end-effector of the Fetch arm to grasp an object and move it to a goal, g , represented visually by a red sphere. The action space is defined with actions $(\Delta z, a_g)$ where z is the position of the end-effector and $a_g \in [-1, 1]$ is the action determining the state of the parallel gripper, with 1 being open and -1 being closed. Since the task is more complex, the maximum episode length is set as $T = 100$. This set of experiments uses both types auto-perception networks (AE and VAE), with RGB input images since it is difficult to differentiate between the goal and object in grey-scale images.

In order to speed up training of the RL system for this set of experiments, we initially collect data using a simple hand-crafted heuristic policy, which gives an average episode length of 65.33 actions. This data is used as demonstration for imitation learning. Therefore the initial RL policy is not random as it was before but pretrained using the initial demonstrations. This gives the RL algorithm a better chance of interacting with the object during trial episodes. When using HER, additional modified episodes are only included in the HER buffer when object interaction has occurred, otherwise the replay buffer is imbalanced with episodes containing no meaningful feedback for the task at hand.

Table 4.5 shows results for the pick and place environment. First we show the results using the ground truth semantic state as input to the network. As expected this is better than the heuristic policy but still requires the semantic state available at run time. We also show results using the

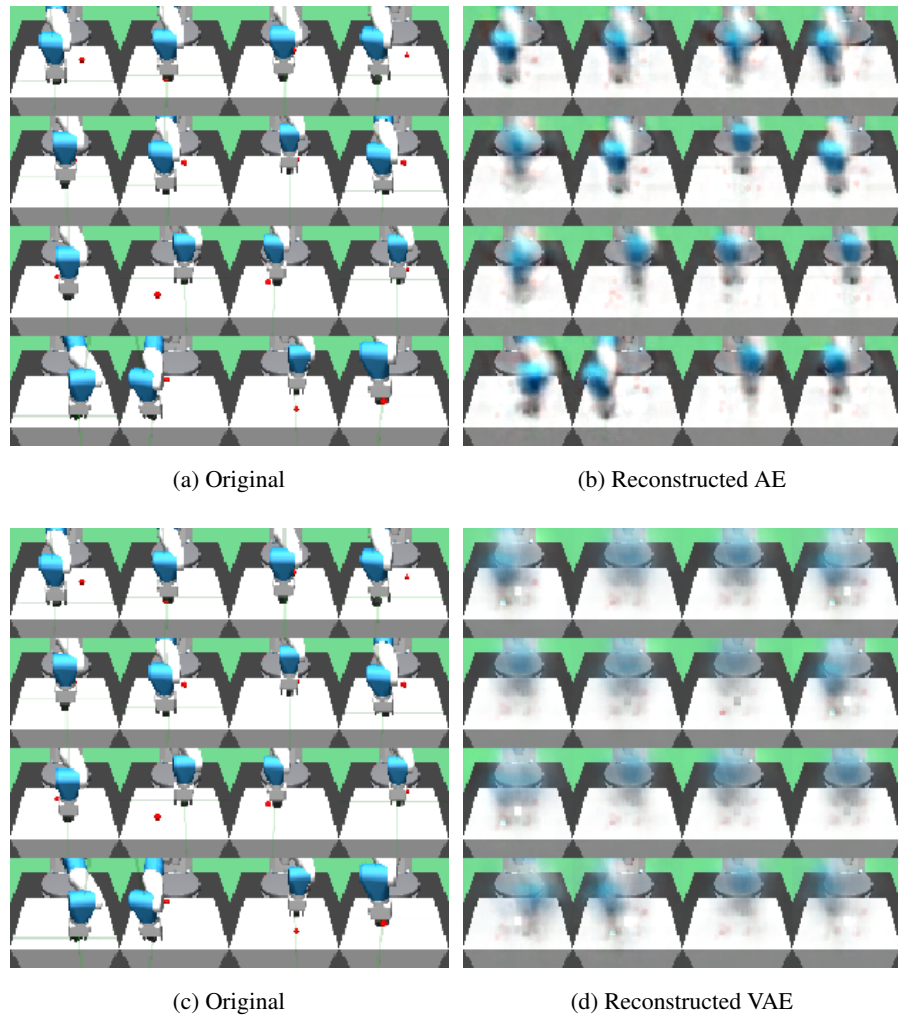


Figure 4.11: Reconstructions from the auto-perceptive networks for Fetch pick and place

ground truth semantic state along with a learnt perceived state. In this environment this does not add value since the task can generally be completed with the ground truth semantic state.

Finally we show results for the system with only the perceived state input at run time, conditioned with the semantic state during training. We can see that with just the semantic state conditioning we have an average of 80.17 actions per episode. The drop in performance is explained by noise in the perceived conditioned state, such as when the arm obscures the goal or the object from the cameras view. Using a partially conditioned state with an additional 7 unconditioned features, the system improves to 71.50 actions per episode for an AE perceptive network and 67.80 actions per episode for a VAE perceptive network. The additional features allow the system

to compensate for the lack of visual representation. Whilst this does not match the ground truth semantic state input results, it does complete the task without the need for any semantic state input at run time. Some example reconstructions from both the AE and VAE networks are shown in Figure 4.11. It is interesting to note that the AE reconstructions are clearer than the VAE reconstructions, despite the fact that the VAE latent space performs better in the RL system.

4.6 Conclusion

In this chapter we have shown that the bio-inspired separation of percepts and control at training time allows superior performance in reinforcement learning. The proposed system, APRIL, can predict actions purely from visual data. We showed that allowing the perception system to encode additional properties into the feature space improves the performance over a system using just the approximate perceived state.

This demonstrates the value in allowing the visual system to encode additional features into the RL algorithm input. In addition, the splitting of perception and control allows other techniques to be used, which are typically challenging to implement in the high dimensional image domain, such as HER and transition dynamics modelling with GPs. Whilst we still have a system which allows us to go from visual observations to action - the training does not need to be end-to-end.

Chapter 5

Repeated Object Detection

So far, this work has focused on how to effectively learn policies for control, with both semantic and perception based inputs. These policies are effective when we know what the goal is. For example, the experiments from 4.5.3 show that policies can be learnt to grasp an object and move it to a desired location, assuming that the object location is known. However, when we have a selection of possible objects in an environment, such as a robot picking a product in a warehouse, then there needs to be a method to determine which object, from many, we should attempt to grasp. The method used needs to be able to detect potentially graspable objects, given the type of object required.

Object detection has been a key challenge for computer vision. Historically, object detectors focused on identifying multiple instances of a single object, such as faces [106]. In contrast, deep learning favours a shared feature representation, meaning training a single network for multiple objects has obvious benefits, where early layers of the network are more generic and shared across classes. However, it is not always known during training what set of objects need to be detected at deployment.

This chapter focuses on situations where we do not have a predefined set of object classes. Instead, we develop an architecture that is capable of recognising arbitrary objects, given only a single example at run time. The required object class will often not have been seen during training, hence requiring zero-shot detection. The use case of robotic grasping motivates this work as follows: having successfully grasped an object, we want to find other objects of that

type in a container. In this situation we do not want to retrain our object detection system for every new type of object encountered. Therefore, we need a system which can locate repeated instances of an object without needing prior knowledge of the object class.

To solve this problem we investigate both recurrent and feedforward networks, finally proposing a feedforward network that can be trained in a zero-shot framework. This means the model is trained on generic instance detection examples and used to detect any object, even those of classes not included in the training dataset. There has been relatively little work that addresses this problem, a notable example being Target Driven Instance Detection (TDID) [7] which aims to detect multiple instances of the same object. However, unlike TDID, we do not explicitly train on the types of object we wish to detect, hence supporting zero-shot detection.

5.1 Background

Many of the traditional approaches to object detection have relied upon hand-crafted features such as the Scale Invariant Feature Transform (SIFT) [65] and Histogram of Oriented Gradients (HOG) [20] used in detection methods such as a sliding window Deformable Part Models (DPM) [28]. There have also been approaches to zero-shot detection based on objectness scores, such as Alexe et al. [2] who uses various image cues to determine the likelihood of an image window containing an object. However, more recently, deep convolutional neural networks have dominated object detection. One of the milestones was the Regions with CNN (R-CNN) [35] feature detector, which is a two stage detector that relies on a region proposals stage followed by classification and regression of the location for the proposed region. This was used as the basis for further two-stage detectors, such as Faster-RCNN [83] which introduces a region proposal network to obtain the regions instead of the slow selective search algorithm used previously. In-addition to predicting bounding boxes with a two stage detector, Mask R-CNN [38] takes it a step further and predicts binary masks for each region of interest, giving a more detailed location and shape of each instance. However, with all these systems, there are two stages to detection meaning running the actual detection network multiple times (once for each proposed region) often making it slow.

Some end-to-end CNN based solutions to this problem use recurrent neural networks. One such example is Recurrent Neural Networks for Semantic Instance Segmentation (RSIS) [90] which

uses a convolutional encoder and a recurrent decoder to produce a sequence of binary object masks, along with class predictions for those masks. This is inspired by the human inspection of images, where we scan sections of images in a sequential manner. In contrast, recent single stage detectors such as the Single Shot MultiBox Detector (SSD) [64] and You Only Look Once (YOLO) [80] do not perform region proposals or use recurrence, but predict bounding boxes of instances directly. This allows a faster and more streamlined detector. With advances in deep learning, such as residual networks and batch normalisation, further improvements have lead to the latest version of this network: YOLOv3 [81].

The only way to use semantic detectors like YOLO or RSIS to find repetitions of a target is to assign it a class, get the object classifications for the scene and assume the target class was available in training. If we do not know the class of object we are aiming to detect or it was not available during training then these detectors will not work. Target Driven Instance Detection (TDID) [7] aims to detect instances in images based on one or more target examples. It uses the same feature extractor on the scene image and the target image, then employs both correlation and the difference between the features to determine where the instances are located. Unlike a traditional object detector, TDID does not predict classes but aims to pick-out the specific target instance based purely on the conditioning images. TDID aims to solve a similar task to us, however we do not aim to detect a single specific instance in an image but detect all objects similar to the target object. TDID adapts well to this but is only suitable within limited domains which are very similar to those in the training dataset. The datasets used by TDID, such as Active Vision Dataset (AVD) [8] and GMU Kitchens [34], all focus on a small set of objects found in homes. In contrast, we aim to detect objects outside of a specific domain, meaning our method can be applied to many different applications.

5.2 Recurrent Object Detection

To solve this problem, our initial solution uses a recurrent system similar to RSIS [90]. This system uses a feature encoder followed by a recurrent network to predict a sequence of instance masks with a stop flag to indicate when it has found all the objects in the image. The idea of being able to detect objects in order lends itself well to detecting repeated objects, but needs a learning signal to enforce the similarity of objects without using a class predictor.

An image contains a set of m ground truth instance masks all of the same type. We define the masks for an image of height h and width w as

$$\mathbf{M}_n = \left\{ M_{i,j} = \{0, 1\} \mid i = 1, \dots, w, j = 1, \dots, h \right\}, \quad (5.1)$$

for each object $n=1, \dots, m$. We also have a set of features \mathbf{F} from a fully convolutional encoder

$$\mathbf{F} = \left\{ \mathbf{f}_{i,j} \mid i=1, \dots, w_f, j=1, \dots, h_f \right\}, \quad (5.2)$$

of size (w_f, h_f) with $|\mathbf{f}_{i,j}| = c_f$ being the number of channels. We can take the features from various depths of this encoder giving a list of features $[\mathbf{F}_0, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{F}_4]$ where \mathbf{F}_0 is the output from the deepest layer in the encoder.

The decoder is made up of 5 convolutional Long Short-Term Memory (LSTM) modules. The i^{th} convLSTM layer at time step t gives the output

$$\mathbf{U}_{i,t} = \text{convLSTM}_i \left(\left[U_2 \left(\mathbf{U}_{i-1,t} \right) \mid \mathbf{F}_i \right], \mathbf{U}_{i,t-1} \right), \quad (5.3)$$

where U_2 is upsampling by a factor of 2, and $\mathbf{U}_{i-1,t}$ is the output of the previous convLSTM. For the first convLSTM layer the input is just \mathbf{F}_0 . A final convolution layer is applied to the output of the final convLSTM with sigmoid activation to give a binary mask prediction $\hat{\mathbf{M}}_t$, with the same dimensions as the image for each time step t . The outputs $\mathbf{U}_{i,t}$ for each time step are also pooled and concatenated before passing through a single fully connected layer to give a single objectness score v_t .

5.2.1 Recurrent Segmentation Loss

To train this system we need a loss to guide the segmentation and a loss to inform the network when to stop detecting objects. The segmentation loss uses a version of Intersection over Union (IoU) known as Soft IoU (sIoU) between the predicted mask $\hat{\mathbf{M}}$ and the ground truth mask \mathbf{M} giving

$$\text{sIoU} \left(\hat{\mathbf{M}}, \mathbf{M} \right) = 1 - \frac{\langle \hat{\mathbf{M}}, \mathbf{M} \rangle}{\| \hat{\mathbf{M}} \|_1 + \| \mathbf{M} \|_1 - \langle \hat{\mathbf{M}}, \mathbf{M} \rangle}. \quad (5.4)$$

The order of the objects should not be imposed, so to match each predicted mask to a ground truth mask, the Hungarian algorithm is used to find the optimal assignment of predictions to

ground truth objects. This means that for a set of predicted masks $\hat{\mathbf{M}} = \hat{\mathbf{M}}_1, \dots, \hat{\mathbf{M}}_T$ and a set of ground truth masks $\mathbf{M} = \mathbf{M}_1, \dots, \mathbf{M}_m$ the segmentation loss is

$$L_{\text{seg}} = \sum_{i=1}^T \sum_{j=1}^m \text{sIoU}(\hat{\mathbf{M}}_i, \mathbf{M}_j) \delta_{i,j} \quad (5.5)$$

where $\delta_{i,j}$ is the matrix of assignment from the Hungarian algorithm (with value 1 if masks $\hat{\mathbf{M}}_i$ and \mathbf{M}_j are matched and 0 otherwise). The stop loss is calculated using binary cross entropy as

$$L_{\text{stop}} = - \sum_{t=1}^T \alpha_t \log v_t + (1 - \alpha_t) \log(1 - v_t), \quad (5.6)$$

where $\alpha_t = 1$ for $t \leq m$ and $\alpha_t = 0$ for $t > m$.

5.2.2 Similarity Network

In order to provide a learning signal that enforces the similarity of objects, our proposed system uses the features \mathbf{F}_0 from the encoder and the predicted masks $\hat{\mathbf{M}}_t$ to determine how similar masks are with a similarity network. The similarity network passes the features \mathbf{F}_0 , masked with \mathbf{M}_i and \mathbf{M}_j , through the network to give points q_i and q_j in a latent space. The distance between these two points is $d_{ij} = |q_i - q_j|_2$. If the class labels of the objects represented by masks \mathbf{M}_i and \mathbf{M}_j are l_i and l_j respectively, then the network is trained with a contrastive loss

$$L_{\text{con}} = \begin{cases} d_{ij} & \text{if } l_i = l_j, \\ \max(0, \rho - d_{ij}) & \text{if } l_i \neq l_j, \end{cases} \quad (5.7)$$

for some margin ρ . This encourages masks of objects of the same type to be close in the latent space and those of different types to be further apart. The similarity network can be trained using the ground truth masks \mathbf{M}_n and a fixed feature encoder pretrained on ImageNet. Figure 5.1 gives an overview of the pretraining system. A single input image (shown in orange) is passed through the encoder backbone to get the features \mathbf{F}_0 , then two instance masks are individually applied to these features. The masked features are each passed through the similarity network (shown in green). The resulting latent representations are the input to the contrastive loss in Equation 5.7, such that if the applied masks are of similar instances the latent representations will be pushed together and if they are not they will be pushed apart.

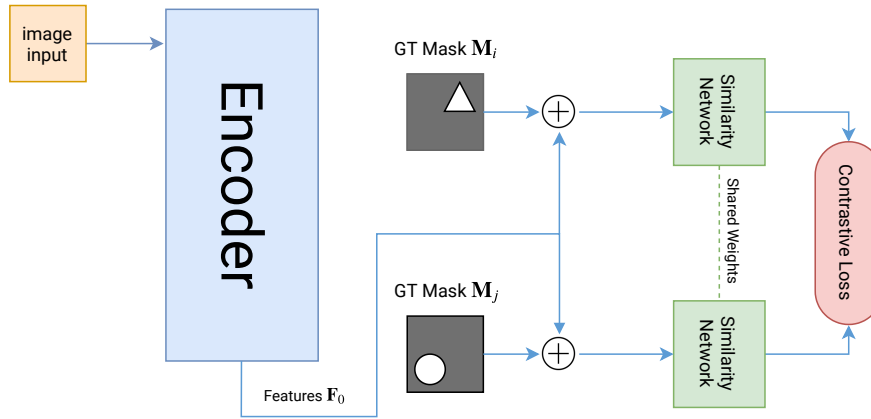


Figure 5.1: Similarity network pretraining

5.2.3 Recurrent Mask Similarity

When training the recurrent instance detection network, the similarity network can be used to determine the similarity of two masks from the sequence and penalise masks which are not similar to a mask already detected. For prediction \hat{M}_i and \hat{M}_j , where $i < j$, we calculate the distance between q_i and q_j from the latent space to give d_{ij} . Table 5.1 helps visualise this. For each $j > 1$ we take the minimum d_{ij} and create a weighted sum of these to form the repetition loss

$$L_{\text{rep}} = \sum_{n=1}^T \gamma^n \min(d_{in}, 0 \leq i < n), \quad (5.8)$$

where T is the total number of predictions made by the system from the recurrent network and $\gamma \in [0, 1]$. Lower values indicate that later predictions are close to previous predictions in the latent space. We aim to minimise this value since the objective is to predict similar objects to those already detected. Figure 5.2 gives an overview of how the similarity network is used in the instance detection training. The similarity network (shown again in green) uses predicted masks from the recurrent decoder (shown in purple), rather than the ground truth masks used for pretraining, in order to create a similarity table as described above. The full loss for instance detection is then calculated as

$$L_{\text{MASK}} = L_{\text{seg}} + \lambda_s L_{\text{stop}} + \lambda_r L_{\text{rep}} \quad (5.9)$$

with constants λ_s and λ_r used to weight the different losses.

$i \backslash j$	\hat{M}_0	\hat{M}_1	\hat{M}_2	\hat{M}_3
\hat{M}_0	-	d_{01}	d_{02}	d_{03}
\hat{M}_1	-	-	d_{12}	d_{13}
\hat{M}_2	-	-	-	d_{23}
\hat{M}_3	-	-	-	-

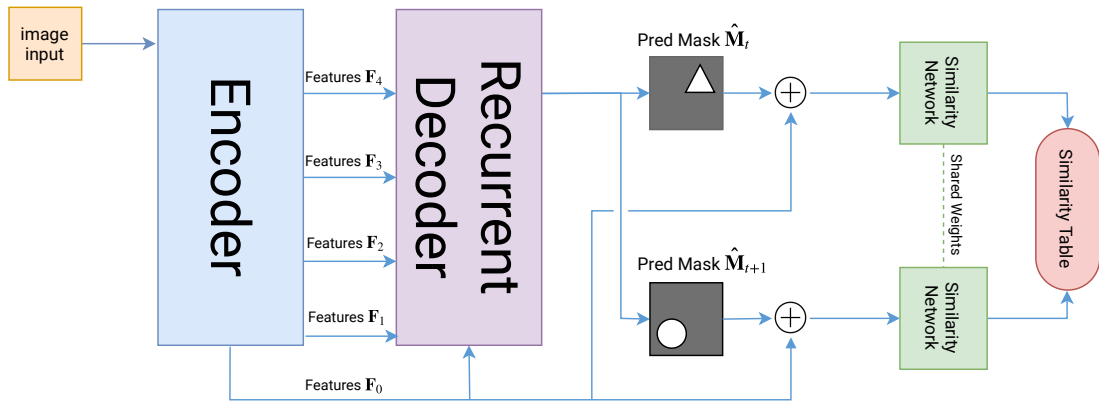
Table 5.1: Example similarity table for predicted masks \hat{M}_i and \hat{M}_j .

Figure 5.2: Recurrent instance detection with similarity network

To test the proposed recurrent repeated object detection system, the network was trained on a subset of the CityScapes dataset [17] with only images including multiple instances of the same class of objects. Whilst the CityScapes dataset is from a different domain to robotic grasping, it has many repeated objects with labelled masks, especially repeated cars and people.

Firstly, we pretrained our similarity network with the ground truth masks and features from the fixed encoder. Figure 5.3 shows the average euclidean distance between points in the latent space with matching masks and non-matching masks. As desired, the matching pairs are closer in the latent space than the non-matching pairs. Whilst it does over-fit to the training set slightly, we can see that there is still a distinct difference between the matching and non-matching encoded masks in the validation set.

Next we used this similarity network in our full system with the recurrent decoder predicting masks. We use curriculum learning to introduce the different losses at various intervals starting with the segmentation loss, then the repetition loss and finally the stop loss. The max number

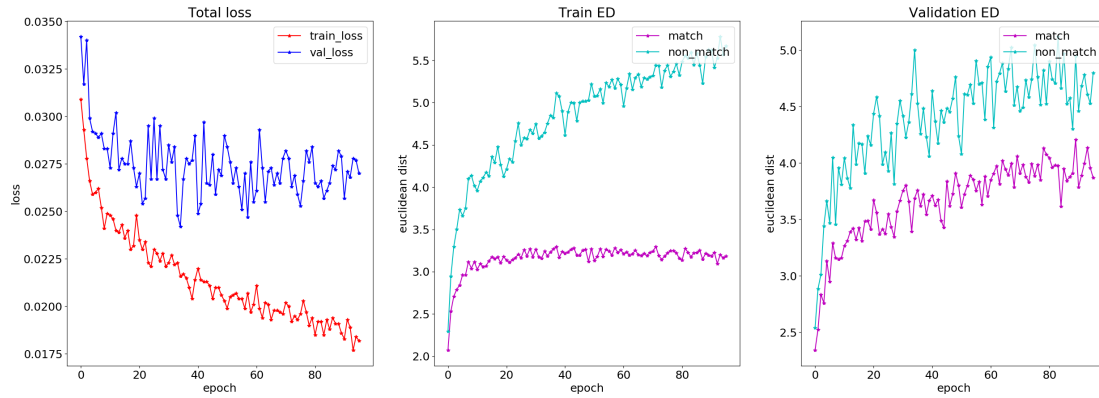
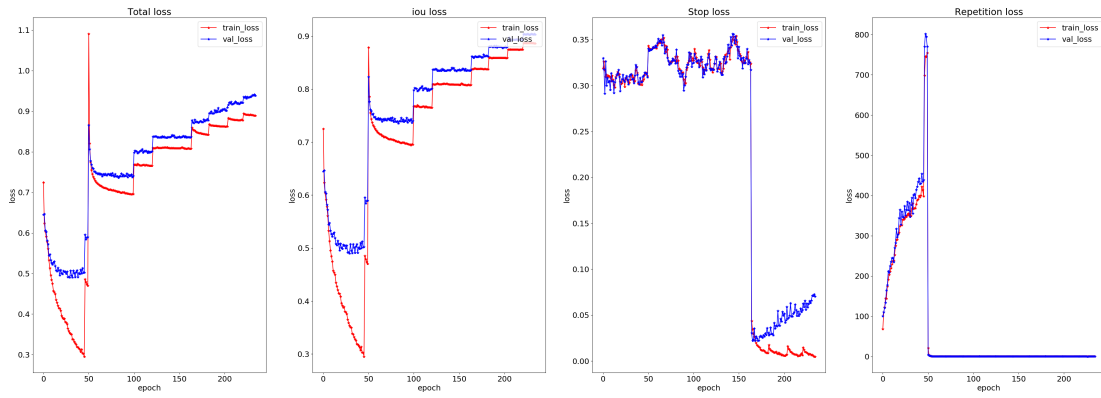


Figure 5.3: Training graphs for pretraining of similarity network with margin of 10

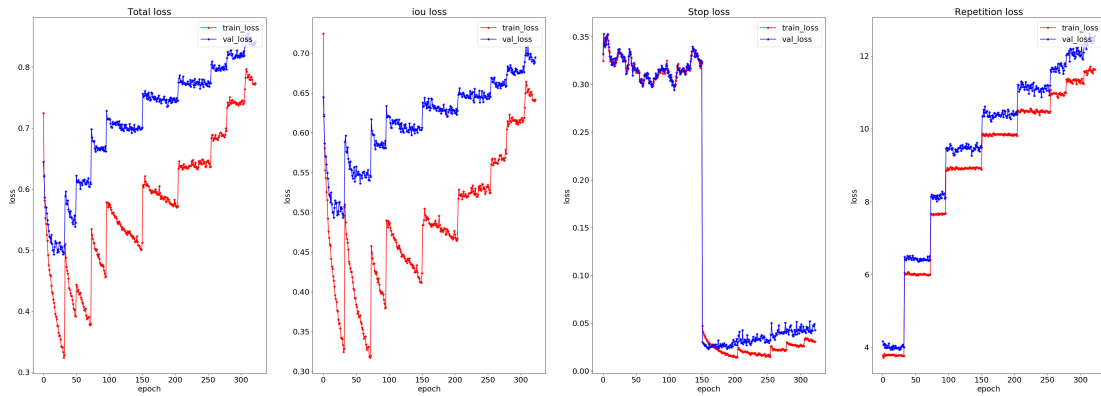
of objects detected is also gradually increased making the problem harder as training proceeds. Figure 5.4a shows the recurrent systems losses during training. As each loss is introduced, or the the maximum number of objects detected increases, we see a jump in the total loss (as expected). Once introduced, the repetition loss converges very quickly. However, the network achieved this by predicting the same mask at each time step, as can be see from Figure 5.5a. This is the easiest way for the network to produce similar masks, hence minimising the similarity loss which overpowers the other losses.

In order to combat this issue, we use the ground truth masks which were sorted with the Hungarian algorithm to calculate the similarity table and the repetition loss. This means that the network predicting the same masks will not reduce the repetition loss, because each predicted mask will have a different ground truth mask assigned. Figure 5.5b shows some examples from the network when using this method. We can see that it has stopped predicting the same mask at each step. However, when we look at Figure 5.4b we can see that the repetition loss never converges. Since we do not use the masks the network has predicted, there is no output of the decoder used to calculate the repetition loss. This means the network has no way of back-propagating the repetition loss though the decoder weights.

Due to these issues, the recurrent system is not suitable for detecting repeated objects. The recurrent system falls into the trap of many two stage detectors, with multiple networks required, needing multiple stages of training for a single task. In contrast a single stage, feedforward network should be much quicker, not requiring the roll out of the recurrent modules. We also found that using the sequential predictions did not allow the network to look at all of the objects



(a) Using predicted masks to calculate repetition loss



(b) Using ground truth masks to calculate repetition loss

Figure 5.4: Training graphs of full recurrent system using the pretrained similarity network



(a) Using predicted masks to calculate repetition loss

(b) Using ground truth masks to calculate repetition loss



(c) Order of predictions

Figure 5.5: Example predictions from the network

in the image to find the similar ones. This supports the move to a feedforward approach such as YOLO where the whole image is viewed at once to predict the objects. Using a feedforward network will hopefully make it easier to condition on a previously detected object.

5.3 Feedforward Object Detection

The feedforward system proposed in this section aims to take advantage of the ability to predict all repeated objects at once, using correlations between features, and predict bounding boxes to detect both visual and affordance similarity. An overview of our Repeated Object Detection (ROD) approach is shown in Figure 5.6 and our full detection system is shown in Figure 5.8. The main feature of ROD is that it actively drives the object detection network to learn similar features for similar objects across different scales. This section details the components that achieve this, which can be summarised in the following three points:

1. Multiple scales of resized crops for use in cross-correlation (Part C Figure 5.6).
2. A correlation loss L_h to enforce feature similarity (Equation 5.15).
3. Application of ROD at multiple depths of the backbone network (Part C-D Figure 5.8).

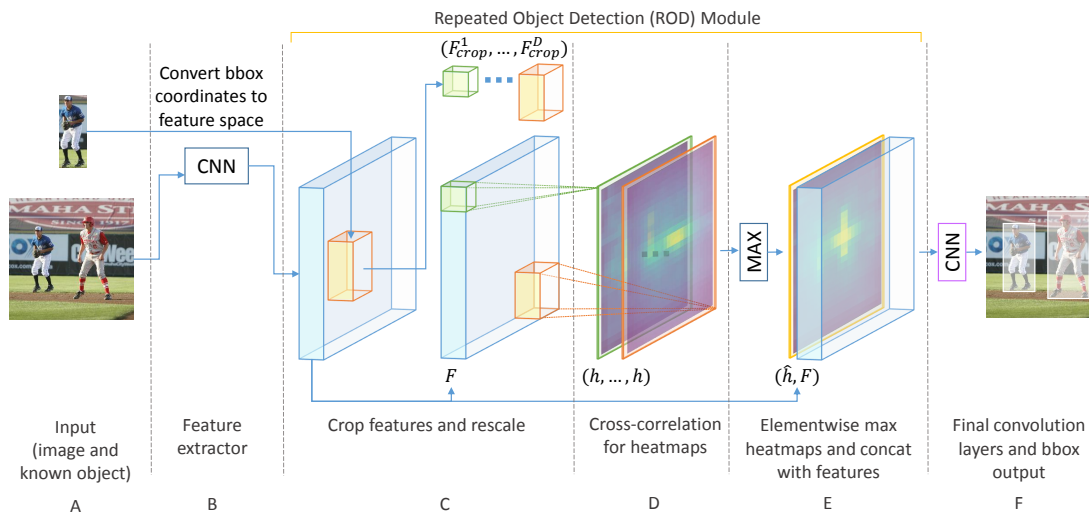


Figure 5.6: Repeated Object Detection

To formalise the feedforward approach, an image contains a set of m ground truth instance bounding boxes all of the same type, which need to be detected without detecting other objects in the image. We define this set as

$$O = \{o_n = (x_n, y_n, w_n, h_n) \mid n=1, \dots, m\}, \quad (5.10)$$

where (x_n, y_n) is the top left corner of the bounding box and (w_n, h_n) is the size. Of these, a single instance, $o_k \in O$, is known. The aim is to predict all the other instances which are similar to the known instance.

5.3.1 Feature Extraction

The first step is to extract important features from the image. To do this, the image is passed through a fully-convolutional neural network (step B in Figure 5.6), creating a set of features \mathbf{F} , as in Equation 5.2. The known bounding box o_k is transformed from the image space to the feature space giving $\hat{o}_k = (\hat{x}_k, \hat{y}_k, \hat{w}_k, \hat{h}_k)$. This is used to create a crop of the features (step C in Figure 5.6)

$$\mathbf{F}_{\text{crop}} = \left\{ \mathbf{f}_{i,j} \mid \begin{array}{l} i = \hat{x}_k, \dots, \hat{x}_k + \hat{w}_k \\ j = \hat{y}_k, \dots, \hat{y}_k + \hat{h}_k \end{array} \right\} \subset \mathbf{F}. \quad (5.11)$$

The crop is used as the filter in cross-correlation across the rest of the feature map, giving a heatmap with elements $h_{i,j} = (\mathbf{F}_{\text{crop}} \star \mathbf{F})_{i,j}$. The points in the image similar to the known instance should ideally have similar features, creating peaks in the heatmap.

However, this alone would struggle to cope with instances which are at vastly different scales to the known instance. To combat this, the crop is resized to D different scaled crops $\mathbf{F}_{\text{crop}}^s$ for $s = 1, \dots, D$ which are then each used to create D new heatmaps (step D in Figure 5.6). Taking the pixelwise maximum of these D heatmaps creates a final maximum heatmap

$$\hat{h}_{i,j} = \max_{s=1, \dots, D} [(\mathbf{F}_{\text{crop}}^s \star \mathbf{F})_{i,j}], \quad (5.12)$$

which can show similar objects at different scales across the image. An example of this can be seen in Figure 5.7.

Finally, the heatmap is concatenated to the features (step E in Figure 5.6) and passed through a further set of convolution layers (step F in Figure 5.6) to support detection. Step C - E are collectively referred to as a Repeated Object Detection (ROD) module.

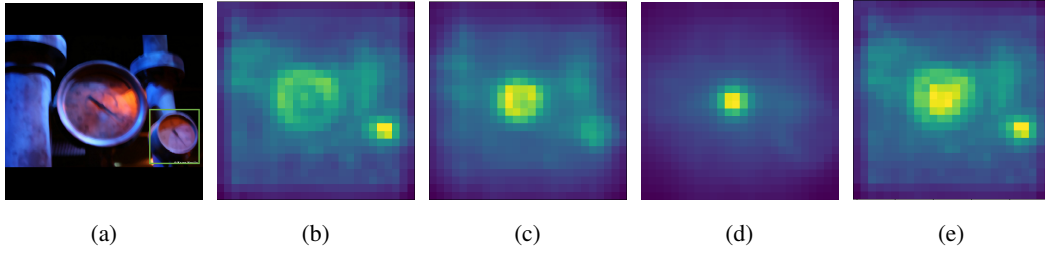


Figure 5.7: Example heatmaps - (a) Input image with known object bounding box, (b - d) Heatmaps created with crops at three different scales, (e) Pixel-wise max across each scale

5.3.2 Metric Learning

Ideally the initial features from the fully-convolutional network will be similar for different instances of the same object type across scales, orientation or instance. To encourage this, a correlation loss is proposed which promotes higher peaks at points in the heatmap where similar instances occur. This loss aims to create a Gaussian in the heatmap around the locations of the ground truth instances. So we define our target heatmap to have pixel values

$$g_{i,j} = \sum_{n=1}^m G(\hat{x}_n - i, \hat{y}_n - j) \quad (5.13)$$

where

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (5.14)$$

The heatmap correlation loss is then calculated as the Mean Squared Error (MSE) between the generated heatmap and the target heatmap giving

$$L_h = \frac{1}{w_f h_f} \sum_{i=1}^{w_f} \sum_{j=1}^{h_f} (\hat{h}_{i,j} - g_{i,j})^2. \quad (5.15)$$

This loss is applied to the maximum heatmap $\hat{h}_{i,j}$ in the ROD module i.e. part E of Figure 5.6. The pixelwise maximum of the heatmaps $\hat{h}_{i,j}$ is not produced directly by the network but is the result of correlation between learned features, providing a learning signal indicating that similar objects to the scaled crop should result in similar features.

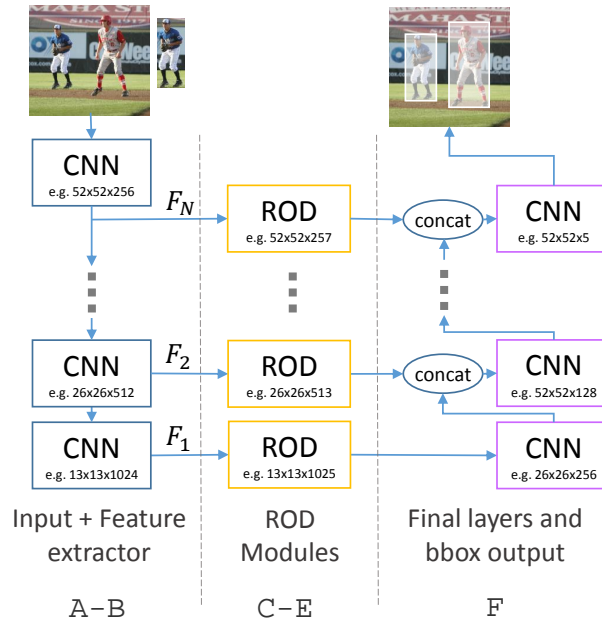


Figure 5.8: Multi-Scale Repeated Object Detection - example values indicate output size of each module

5.3.3 Multi-Scale Features

The feature extractor used in part B of Figure 5.6 gives a final output of features at a certain scale. However, at intermediate points in the fully convolutional feature extractor backbone there are features at different scales which we can make use of. To take advantage of this we introduce skip connections, where ROD modules may be applied, allowing the later parts of the network access to features from different scales.

With the correlation loss L_h being applied at each ROD module's maximum heatmap, we encourage the correlation of similar objects' features to be consistent across scales. Figure 5.8 shows how ROD modules can be applied at different feature levels from an extractor network and then joined later in the CNN as a skip connection.

In comparison to the different scales of the heatmaps in Figure 5.7 which uses one set of features and adjusts the crops to different scales, here we are also using sets of features from different depths of the feature extractor. If we use feature maps ($\mathbf{F}_1, \dots, \mathbf{F}_N$) at N different depths of the feature extractor, then each of these has heatmaps created at D different crop scales. This means we encourage features of similar objects to be correlated over $N \times D$ scales.

5.3.4 Object Detection

The output layer where the bounding boxes are predicted uses a grid and anchor system similar to that proposed by [81]. The output is an $S \times S$ grid over the image, where each cell predicts 5 values. This gives us an output in the form $\mathbf{t}_i = (t_i^x, t_i^y, t_i^w, t_i^h, t_i^o)$ for each cell $i = 1, \dots, S^2$.

The output t_i^o is the confidence in the predicted bounding box for cell i . The terms (t_i^x, t_i^y) are passed through a sigmoid function σ and give the location of the predicted box relative to the top left corner of that cell (c_i^x, c_i^y) . The B anchor sizes give a prior on the size of the bounding boxes p_j^w, p_j^h where $j = 1, \dots, B$ and t_i^w, t_i^h are used to calculate the predicted width and height of the box. This gives the final box predictions for cell i and anchor j as $\hat{\mathbf{t}}_{i,j} = (x_{i,j}, y_{i,j}, w_{i,j}, h_{i,j})$ where

$$x_{i,j} = \sigma(t_i^x) + c_i^x \quad (5.16)$$

$$y_{i,j} = \sigma(t_i^y) + c_i^y \quad (5.17)$$

$$w_{i,j} = p_j^w e^{t_i^w} \quad (5.18)$$

$$h_{i,j} = p_j^h e^{t_i^h} \quad (5.19)$$

Since there are $S^2 \times B$ predictions, which is likely to be greater than the number of objects to predict, a single cell-anchor pair is selected to be responsible for predicting each object $\mathbf{o}_n \in O$. This selection is based on which prediction achieves the highest Generalised IoU (GIoU) with \mathbf{o}_n . GIoU [84] is given by

$$\text{GIoU}(a, b) = \frac{|a \cap b|}{|a \cup b|} - \frac{|c \setminus (a \cup b)|}{|c|}, \quad (5.20)$$

where c is the smallest enclosing box for a and b . From this we get a mask of binary values $r_{i,j}$

$$r_{i,j} = \begin{cases} 1 & \text{if } \exists \mathbf{o}_n \in O \text{ s.t.} \\ & \arg \max_{(a,b)} (\text{GIoU}(\hat{\mathbf{t}}_{a,b}, \mathbf{o}_n)) = (i, j), \\ 0 & \text{otherwise} \end{cases} \quad (5.21)$$

where $i = 1, \dots, S^2$ indicates the cell and $j = 1, \dots, B$ indicates the anchor. This will set $r_{i,j}$ to 1 for cell-anchor pairs which are responsible for predicting an instance and 0 for all others.

We can then formulate our bounding box loss as:

$$L_{\text{bbox}} = \sum_{i=1}^{S^2} \sum_{j=1}^B r_{i,j} |\hat{\mathbf{t}}_{i,j} - \mathbf{o}_{ij}|_2^2, \quad (5.22)$$

where $\mathbf{o}_{ij} \in O$ is the ground truth instance for which cell-anchor pair (i, j) is responsible. The object confidence losses are defined as

$$L_{\text{obj}} = \sum_{i=1}^{S^2} \sum_{j=1}^B [r_{i,j}(1 - t_i^o)^2 - \beta(1 - r_{i,j})(t_i^o)^2], \quad (5.23)$$

where β is a weight to balance the fact that there will always be more cells without objects than those with objects. The total loss is then calculated as

$$L_{\text{ROD}} = L_{\text{bbox}} + L_{\text{obj}} + L_{\text{h}}. \quad (5.24)$$

It is worth noting that since we do not care about the class of the objects we are detecting, we do not predict a class or have a class loss. This means that ROD can focus on the patterns that make instances similar, rather than what features relate to a specific class. This means that ROD can be used in zero-shot learning - since it can be used to predict repeated objects which it has not seen during training.

5.4 Experiments

The experiments employed to evaluate our approach use a subset of the Microsoft Common Objects in Context (MSCOCO) dataset [63], including all images which have multiple objects of the same class. This dataset gives bounding boxes for every object in an image. For each epoch of training, each image in the training set has one object randomly selected as the query object. This means that at each epoch, the same images will be seen but with potentially different objects being used to condition the detections.

All the networks tested use the YOLOv3 backbone called Darknet-53 [81], this is a fully-convolutional network with residual layers. A detailed architecture is shown in Figure 5.9. Since the baseline (TDID) can be used with any base object detection network we have re-implemented this using the same Darknet-53 backbone in order to make valid comparisons

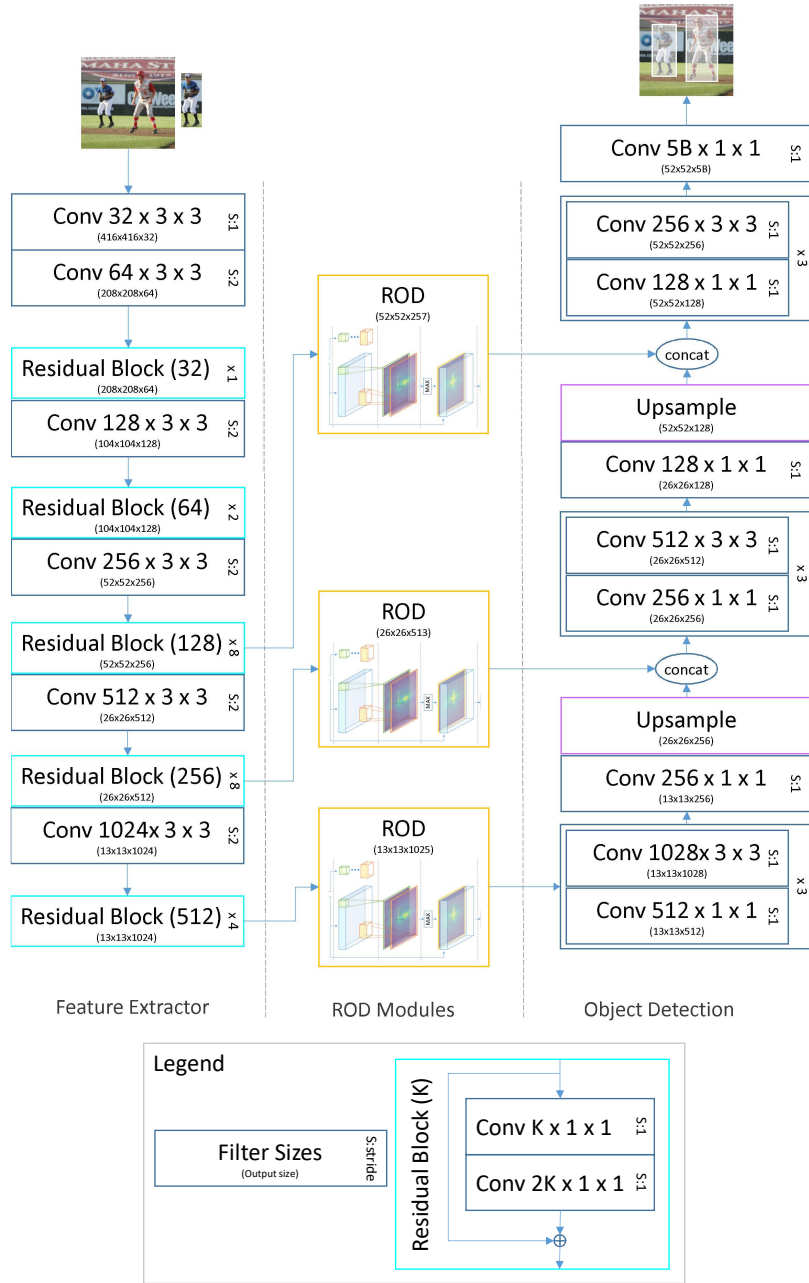


Figure 5.9: Detailed system diagram with the layers used in these experiments

between the approaches. Our implementation of TDID uses the crop of the known instance as the target image. All networks are implemented in PyTorch version 1.1.0 [75].

The evaluation metric we use is Average Precision (AP). To calculate AP we consider the following concepts

- True Positive (TP) - detections that are actually ground truth objects,
- False Positive (FP) - detections that are not ground truth objects,
- False Negative (FN) - ground truth objects that are not detected.

Precision is then defined as

$$p = \frac{TP}{TP + FP}, \quad (5.25)$$

giving the ratio of correctly detected objects to the total number of detections. Recall is defined as

$$r = \frac{TP}{TP + FN} \quad (5.26)$$

giving the ratio of correctly detected objects to the total number of ground truth objects. AP is given by finding the area under the precision-recall curve plotted for a list of detections sorted by objectness score, with a given IoU threshold determining which objects are considered a TP.

This can be approximated using

$$AP \approx \sum_{i=1}^j (r_{i+1} - r_i) p_{interp}(r_{i+1}), \quad (5.27)$$

where

$$p_{interp}(r_i) = \max_{\tilde{r} \geq r_i} p(\tilde{r}). \quad (5.28)$$

In all experiments $AP_{0.5}$ is the AP with the threshold that provides a TP at IoU = 0.5 and mAP is the mean AP across thresholds IoU = [0.5, 0.55, ..., 0.95]. This follows the protocol of the MSCOCO detection challenge [69].

5.4.1 Many-Shot Learning

First we ran experiments including all images from the training data that contained repetition, which was around 74,000 images. We then validate on a set of 5,000 unseen images containing

	AP _{0.5}	mAP
TDID	0.4481	0.2716
Ours (without L_h)	0.5733	0.4616
Ours	0.6181	0.4935

Table 5.2: Repeated object detection - trained with all classes

repeated instances. Table 5.2 shows that our correlation technique performs better than TDID. It also shows that using the additional correlation loss L_h further improves the results.

Figure 5.10 shows examples of detections with different conditioning images for both TDID and our method. We can see that our network can successfully pick out repeating instances of the conditioned object even when other classes are present (e.g. top row in Figure 5.10). The middle row of Figure 5.10 shows our method not only detecting objects in clear images, such as the baseball players, but also in distorted images, such as the buses blurred by rain. The detections in the bottom row show our method successfully conditioning with different scaled objects by detecting both elephants no matter which is used as the target.

5.4.2 Zero-Shot Learning

In comparison to TDID, we expect ROD to perform better in zero-shot transfer, where the network is asked to find repeated objects of a type it has not seen during training. The additional L_h loss encourages similar objects to have correlated features without any classifier. Therefore, as long as the training set has diverse enough training examples, the model will be able to learn features which relate different objects together, even if they are not visually exactly the same. This loss provides a much stronger learning signal at various scales throughout the network to reinforce the concept of similarity regardless of the object type.

In order to show that ROD improves the ability to detect arbitrary unknown repeated objects, it is trained excluding one of the classes from the adapted MSCOCO dataset. We do this for a random selection of classes for both our network and TDID. The results in Table 5.3 show the AP_{0.5} and mAP scores on the validation data containing objects of the excluded class. ROD



Figure 5.10: Detections for networks trained with full dataset - blue box: known target object, orange boxes: predicted detections



Figure 5.11: Example detections for networks trained on dataset excluding class which we are aiming to detect

outperforms TDID on every class in both metrics. As expected we do see a drop in performance compared to the many shot task. However, in some cases, our zero-shot detection results actually exceed the performance of TDID when performing many-shot detection (i.e. Table 5.2). Table 5.3 shows that ROD has the ability to generalise to detecting repeated unseen objects compared to TDID.

Examples of the networks detecting objects which were not seen in training are shown in Figure 5.11. Whilst TDID does detect some objects it had not seen before, such as cars, ROD manages to pick out more of these objects. ROD shows impressive viewpoint invariance for unknown objects in the chair detection example. We can also see that ROD shows invariance to deformable pose changes when detecting dogs. Even though the performance of all networks on the person dataset was significantly weaker due to a dramatic reduction in the quantity of training data when not including people, we still see our network detecting some people in the baseball examples in Figure 5.11.

5.4.3 Multi-scale features

The next set of experiments aim to show the effect of the ROD module at different scales. Networks were trained with ROD modules applied at different stages of the feature extractor.

Excluded Class	AP _{0.5}		mAP	
	Ours	TDID	Ours	TDID
Dog	0.5670	0.4955	0.4257	0.3492
Car	0.3890	0.3814	0.2752	0.2308
Chair	0.3150	0.2603	0.1747	0.1203
Bowl	0.4058	0.3341	0.2960	0.1674
Person	0.2217	0.1841	0.1209	0.0833

Table 5.3: Repeated object detection - trained with all but one class, validated on the excluded class

Our full system (shown in Figure 5.9) has a ROD module at all three scales, one at the large scale ($13 \times 13 \times 1025$), one at the medium scale ($26 \times 26 \times 513$), and one at the small scale ($52 \times 52 \times 257$). We denote this as RRR, with a ROD module at each scale - going from smallest scale on the left to largest on the right. In these tests the ROD modules can be replaced with a standard skip connection denoted as S. A network with a ROD module at the large scale and a skip connection at the medium and small scale would be denoted as SSR. The results showing the AP_{0.5} of each network at different scales can be seen in Figure 5.12. We separate the objects into different scales based on their bounding box size. Objects in a range a to b have a bounding box with area between a^2 and b^2 .

The first plot in Figure 5.12 shows the results of using only one ROD module at the various scales. This shows that the different networks all peak at the smaller scales, probably due to this being the range with the most training data available. The network SSR, with the ROD module at the large scale, performs better than the others at the extreme scale of 0.8-0.9. Whilst the network RSS, with the ROD module at the small scale, performs the worst at this level.

The second plot in Figure 5.12 shows the results of using two ROD modules at various points in the network. Again the difference in the scale of the features where the ROD modules are applied has an effect on their ability to detect objects at different scales. The use of the mid scale aids the transition between the two scales, in the 0.9-1 object scale we can see that both the networks with the medium ROD module perform better than the one without. This is important because we will often be conditioning an image on an object which appears smaller or larger

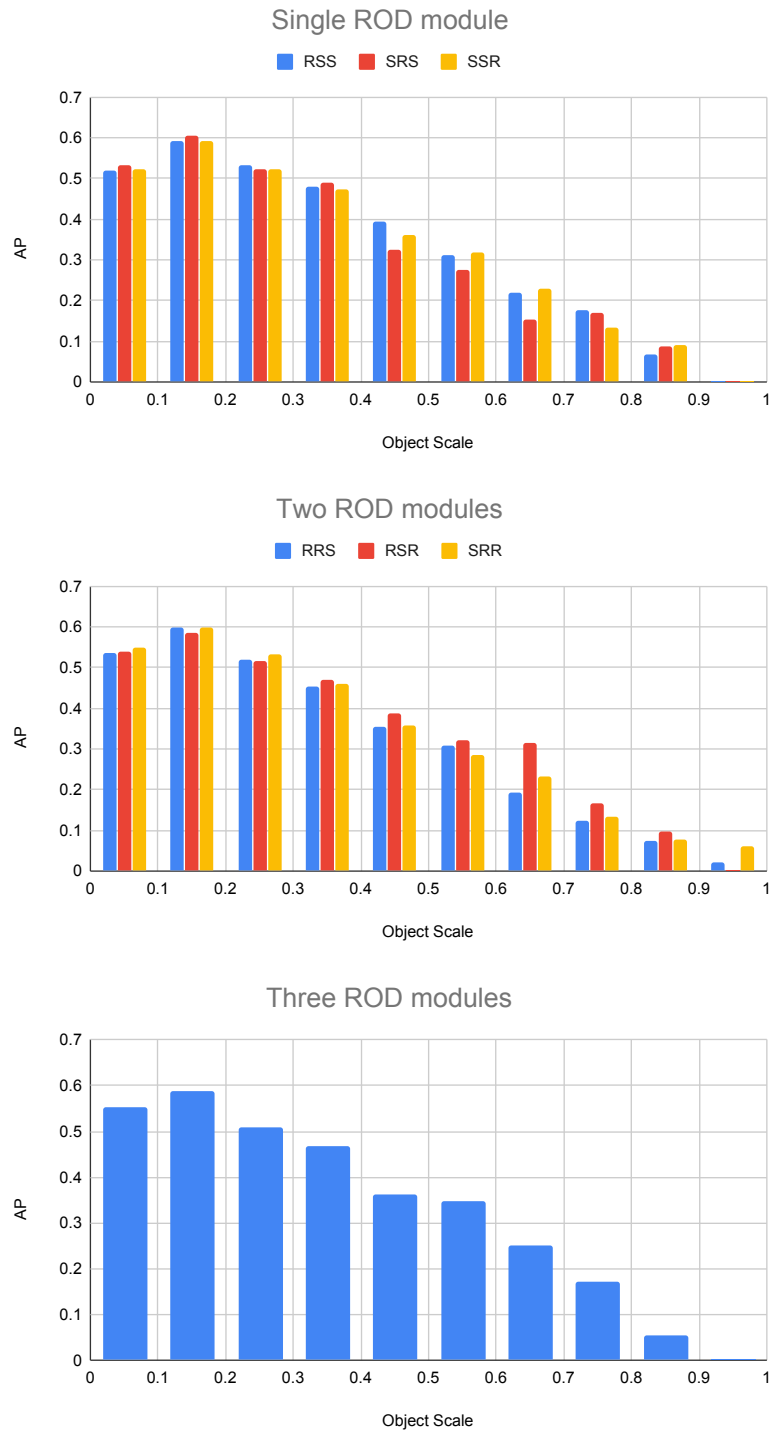


Figure 5.12: Results for multi-scale testing - labels in the legends describe the type of the connection (skip or ROD) at different scales (small, medium, large)

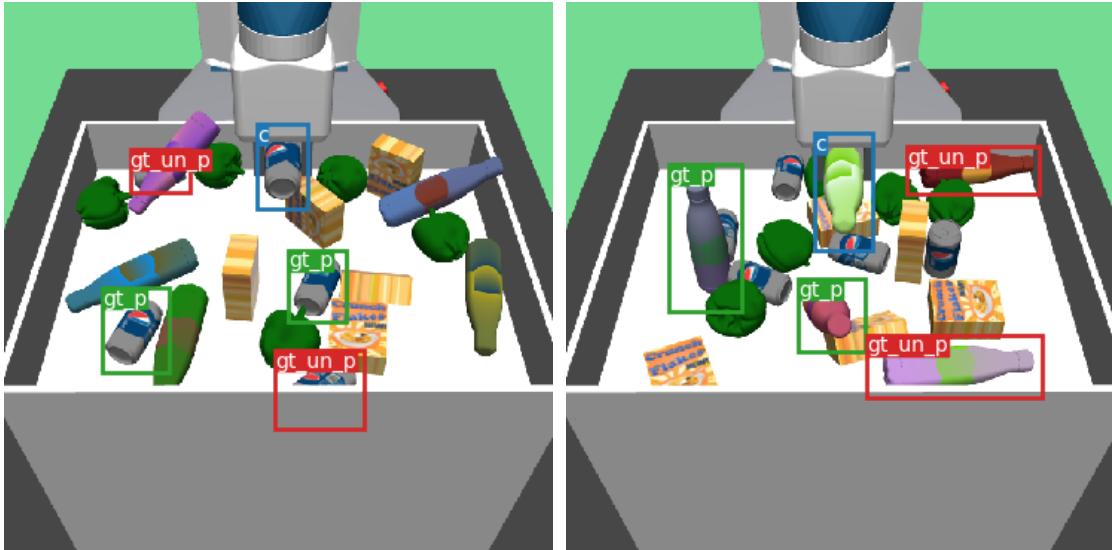


Figure 5.13: Example ground truth robotics data: green boxes show pickable objects (O_p), red boxes show unpickable objects ($O_{\bar{p}}$), blue box show the conditioning object (o_k)

than those in the image which we are aiming to detect.

5.4.4 Robotic Grasping Usecase

This section shows the application of ROD to robotic grasping. In this scenario we have a set of data collected from the Mujoco physics simulator [105]. The simulation includes the Fetch robot attempting to pick up a selection of objects in a cluttered environment. Once an object is picked successfully, the task is to determine where other similar objects are and also which of them are pickable. This requires the system to be view point invariant and deal with the occlusion to determine objects which are both visually similar but also have similar affordances in their current state in the environment. To support this we redefine the concept of “similar” objects to be not only those of the same class, but those which are also graspable in their current configuration. To relate back to our previous formalisation, the full set of objects of the same type is

$$O = O_p \cup O_{\bar{p}} \cup \{o_k\}, \quad (5.29)$$

where O_p is the set of similar pickable objects, $O_{\bar{p}}$ is the set of unpickable objects and o_k is the known object that has already been successfully picked.

Excluded Class	Results on Excluded Class				Results on All Classes			
	AP _{0.5}		mAP		AP _{0.5}		mAP	
	Ours	TDID	Ours	TDID	Ours	TDID	Ours	TDID
Apple	0.6182	0.5689	0.3384	0.3094	0.6024	0.6056	0.3146	0.3197
Bottle	0.4525	0.4186	0.1434	0.1415	0.7497	0.6711	0.4807	0.3799
Can	0.5830	0.5237	0.3521	0.2923	0.6248	0.5955	0.3260	0.3241
Cereal	0.5557	0.5218	0.3134	0.2367	0.6323	0.6432	0.3542	0.3237
Pepper	0.6594	0.5809	0.3388	0.2503	0.6509	0.6236	0.3689	0.3199
None	-	-	-	-	0.9056	0.8543	0.8479	0.6196

Table 5.4: Repeated object detection for robotics - trained with all but one class, validated on the excluded class and all data in the validation set

To this end, the dataset contains the ground truth bounding boxes of all objects of the same type that the robotic arm successfully picked when placed in that position. We annotate this pickable affordance automatically by repeatedly letting the robot attempt to grasp one of the similar items after the initial successful pick. If the pick is successful it is added to O_p , otherwise it is added to $O_{\bar{p}}$. Then simulation is reset back to the initial successful pick before attempting to grasp the next object. Examples of these annotations can be seen in Figure 5.13. In this task we want to detect all objects in the set O_p . To allow the generic repeated object detectors, which are pretrained on MSCOCO, to gain an understanding of the pickable affordance, they are finetuned for 4 epochs on this robotic dataset.

Table 5.4 shows that when trained with all of the data, ROD outperforms TDID by approximately 37% mAP. When classes are excluded from training, the results on the excluded classes show ROD outperforming TDID in all cases. With the full data set we achieve similar or higher AP. This shows that ROD achieves similar performance on the seen objects but consistently generalises better to unseen objects compared to TDID. These results show that ROD can easily be extended to abstract non-visual concepts of similarity.

Figure 5.14 shows some example detections from the network trained without knowledge of the



Figure 5.14: Example detections on robotic grasping dataset for networks trained on dataset excluding the class which we are aiming to detect

specified object. In the bottle examples we can see that ROD prioritises pickable bottles. The undetected bottles are unpickable since they are in awkward positions for the gripper to get to or too close to other objects in the scene. The examples with cereal boxes show that ROD can pick up similar pickable objects even when they are in different orientations leading to very different visual appearances.

To further show the ability of ROD to implicitly learn affordances, Table 5.5 shows the AP scores of the detections using the unpickable objects as the ground truth. For our system to be successful we wish to detect all pickable objects first before detecting any of the unpickable objects. This means that the area under the precision-recall graph (which gives us the AP score) should be small. We can see that for the unpickable objects on average ROD achieves a lower AP, especially in the experiments with objects that have never been seen before. This indicates that ROD is detecting the affordance of unseen objects better than the TDID system. Both ROD and TDID have the highest AP for the unpickable bottles, this is because it has the most complex relationship between affordance and visual appearance. The results show that ROD does not perform as well as TDID when the systems are trained on all the available data, however ROD outperforms TDID in the zero-shot learning situation. This means that ROD is generalising better than the TDID system to unseen objects and their affordances.

Table 5.6 shows the AP values for all objects regardless of affordance. We either out perform or achieve similar results to TDID. Combined with the results from Tables 5.4 and 5.5 this indicates that ROD has greater capacity to detect repeated objects in new environments with a small amount of finetuning, including completely unseen objects.

5.5 Conclusion

The work in this chapter started by investigating a recurrent repeated object detection framework, which was found not to be suitable for the task at hand. Following this, the chapter presented Repeated Object Detection (ROD) - a feedforward technique able to find repeated instances of a query object, outperforming other generic target driven object detectors. ROD can be used in a zero-shot training framework to query images for classes of objects not present in the training set. We have shown that using feature correlation fused across scales in a zero-shot

Excluded Class	Results on Excluded Class				Results on All Classes			
	AP _{0.5}		mAP		AP _{0.5}		mAP	
	Ours	TDID	Ours	TDID	Ours	TDID	Ours	TDID
Apple	0.5086	0.5643	0.2946	0.3222	0.4463	0.4735	0.2004	0.2251
Bottle	0.6443	0.6499	0.5285	0.4540	0.5700	0.5405	0.3381	0.2717
Can	0.4623	0.4991	0.2862	0.2765	0.4519	0.4561	0.2025	0.2115
Cereal	0.5111	0.6704	0.3666	0.4320	0.4601	0.5114	0.2289	0.2375
Pepper	0.5047	0.5533	0.2932	0.3302	0.4655	0.4719	0.2164	0.2100
None	-	-	-	-	0.7499	0.7011	0.7097	0.4715

Table 5.5: Repeated object detection for robotics - trained with all but one class, AP values calculated for the unpickable objects - lower AP scores are better

Excluded Class	Results on Excluded Class				Results on All Classes			
	AP _{0.5}		mAP		AP _{0.5}		mAP	
	Ours	TDID	Ours	TDID	Ours	TDID	Ours	TDID
Apple	0.6572	0.6780	0.5050	0.4848	0.6024	0.6056	0.3146	0.3197
Bottle	0.8704	0.8306	0.7628	0.6475	0.7497	0.6711	0.4807	0.3799
Can	0.6777	0.6788	0.5194	0.4645	0.6428	0.5955	0.3260	0.3241
Cereal	0.6906	0.7918	0.6020	0.6068	0.6323	0.6432	0.3512	0.3237
Pepper	0.6746	0.6877	0.5094	0.5087	0.6509	0.6236	0.3689	0.3199
None	-	-	-	-	0.9056	0.8543	0.8479	0.6196

Table 5.6: Repeated object detection for robotics - trained with all but one class, AP values calculated for both pickable and unpickable objects

framework results in successful detection of previously unseen objects. Our results show that further refining a learned feature space, by using an added conditioning loss which guides it to features that are invariant to the size and orientation of an object, improves the average precision of predicted detections.

ROD has the potential to be applied to any task where we wish to find repeated objects with similar characteristics in an image, knowing only one of the instances. Our results show that ROD can be used to detect objects for robotic grasping - successfully recognising where other similar graspable objects are after performing an initial grasp. This shows the potential for not just detecting objects that are visually similar but have similar affordances. For a given grasping policy, whether it is designed or learned, ROD will be able to assist in determining which objects are most likely to be graspable and similar to the previously picked object.

Chapter 6

Evaluation in an Integrated Framework

The previous chapters of this thesis have provided data driven methods to improve the generalisation of two parts of our pick and place pipeline described in Chapter 1. In order to localise objects, the previous chapter proposed ROD for finding repeated objects in the scene without the need for the object or class to be known at training time. Additionally, to complete active grasp synthesis, APRiL, which was presented in Chapter 4, can be used to select actions based on the visual input with a known goal. However, in order to use these methods for a pick and place pipeline in a realistic robotic environment, such as a warehouse, they need to be integrated together.

To do this, it is necessary to consider how to effectively use these methods with a communication system for robotics such as ROS, ensuring it integrates with both RL and deep learning frameworks. Increasing the speed of simulated data collection is essential in order to have sufficient data to train deep RL systems to convergence. Transfer of knowledge between each separate element of the pipeline must be done effectively in order to complete the task from start to finish. This chapter will explore the best way to do this for the problem of warehouse grasping.

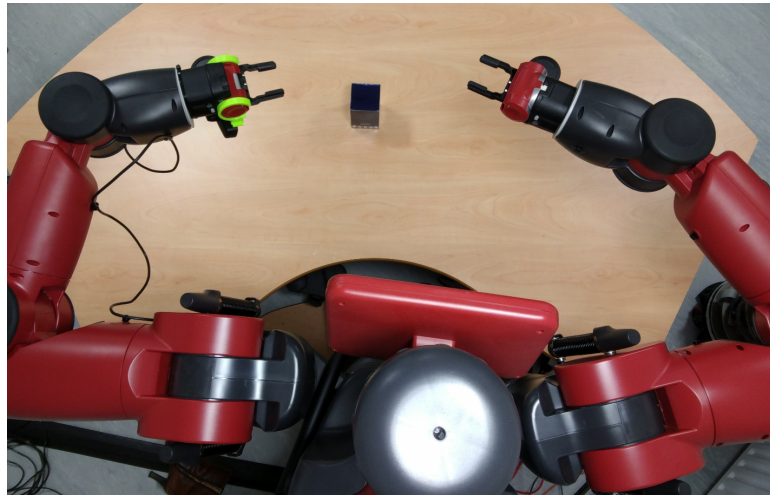


Figure 6.2: Top down view of Baxter from Rethink Robotics

Rethink Robotics as the agent. Baxter uses the ROS platform and has its own ROS master for all communication and control. Rethink Robotics' Baxter model is implemented so that it can be simulated in Gazebo and integrates fully with ROS just as the real robot does. This also involves running an individual ROS master per simulator. Multiple Gazebo simulators can be run from the same machine, using different ports for the ROS masters. However, even running two instances of Gazebo on a typical desktop is impractically slow, causing a bottle neck for training. Also ROS masters are designed to be independent systems meaning that ROS nodes can typically only communicate with a single ROS master decided by environment variables. Dynamically switching between these different ROS masters is a non-trivial task.

The proposed solution uses a separate machine for each simulator, each with its own ROS master which communicates with a master on a central deep learning machine via a ROS multi-master discovery node from the package `multimaster_fkie` [44]. This package includes a `master_discovery` node and a `master_sync` node. The `master_discovery` node polls the ROS environment for changes and publishes these changes over the network. The `master_sync` node connects the `master_discovery` nodes, requesting the ROS states and registering the remote topics and services to the local ROS master. This system means that each worker machine only needs to be able to run a single simulator and does not require the level of GPU needed for deep learning.

By default, all topics are synced across the remote systems - this introduces two issues. Firstly,

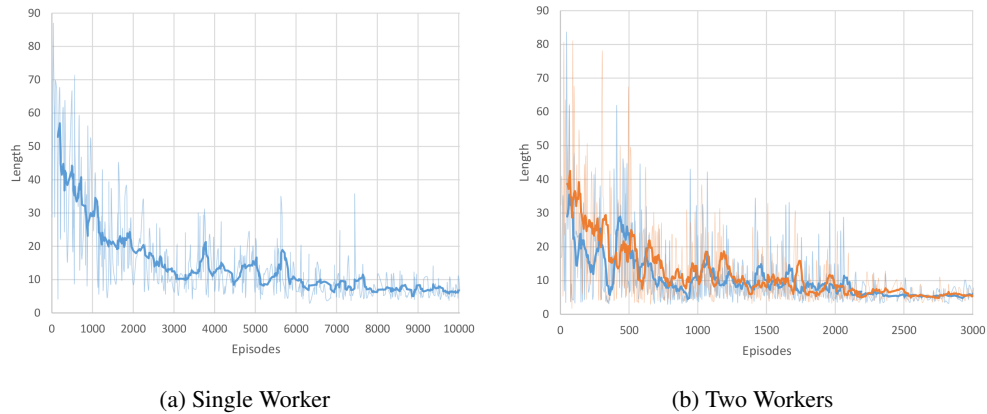


Figure 6.3: Single goal A3C Baxter simulator training - number of actions per episode

duplicate simulators generally publish their states under the same topic name, meaning each `master_sync` will try and sync all the robot state topics. Secondly, complex robots have many topics publishing and subscribing to large quantities of data, so if this is being synced across the network, it creates a lot of network traffic which can slow down the system.

To combat these problems the system makes use of the ROS relay feature to remap the necessary topics to new topics with a namespace specific to each worker machine (shown at the top of Figure 6.1) to ensure no confusion between simulator topics between workers. Next a ROS node which subscribes to the topics needed for communication across the network is created on each simulator machine. The name of these nodes are passed to the `master_sync` node on the deep learning machine (at the bottom of Figure 6.1), meaning that only topics subscribed to by this node are synced, hence reducing the data being sent over the network.

In the case of a Baxter simulator, the data that needs to be transferred depends on how the simulator is controlled. Typically Baxter is controlled via the MoveIt! controller interface directly. However, syncing all the topics and services to control the simulator in this manner from the deep learning machine is impractical. Instead a ROS action server is used for a subset of actions needed to perform the task. The ROS action server only has five topics associated with it: goal, cancel, status, feedback, and result. Syncing these topics allows the system to send goals and wait for updates on both the progress and the final result - meaning all control can remain on the worker machine. This also allows the parsing of reasons for failed actions to be done on the worker machine allowing simplified messages to be sent back to the deep learning

machine.

Depending on the type of actions being completed by the Baxter robot (controlled by MoveIt!) they can take several seconds to complete. This is due to both the physical limitations and the safety limitations in place to avoid over straining the joints or damaging the environment. In simulation there is an option to disable these safety limits, however this dramatically decreases the reliability of the actions completing successfully. The option taken in this work to increase the speed of simulation is to increase the “real time” factor in the physics simulator itself. By increasing the real time updates and the maximum step size for updates we increase the speed by a factor of approximately 5 - depending on the limits of the machine the simulator is running on. Whilst this factor can be increased further, $\times 5$ was found to be the highest setting where the physics simulation is still reliable.

6.1.2 Network Implementation

In order to test the functionality of this framework, we train a deep CNN using the A3C algorithm. The input to the network is an observation of the state of the environment in the form of an Red, Green and Blue (RGB) image. The first section of the network involves 5 convolution layers and three fully connected layers. In comparison to the network used by [67] in the original A3C implementation, this backbone network is much deeper because the images from the Baxter simulator are much closer to photo-realistic images than those from arcade games. These initial layers are pretrained on the ImageNet dataset [23].

The later section of the network uses a 256 unit LSTM and finishes with two fully connected layers - one for the policy output and one for the value prediction. The activation function for the policy output is the softmax function, whilst the value prediction is linear. The number of policy outputs is the number of possible actions - the action with the highest output from the softmax is selected and passed to the action client to be converted to an action goal message. This network is implemented in Tensorflow [1] to facilitate integration into a ROS node.

6.1.3 Evaluation

For these experiments the system described above is used with a simulated Baxter robot. In the Gazebo simulator there is a cube goal on a table within reachable distance from Baxter’s left

end effector. The aim of the problem is to move Baxter's left end effector to within 10cm of the target. We will start by attempting to reach a single goal across all episodes, then gradually expand the range of possible goals. In this set of experiments, the actions are six discrete options given as a positive or negative change of 3 possible angles - the shoulder, elbow and wrist. The performance of our system is evaluated using the average length of episodes.

Initially the system is evaluated based on the average number of actions needed to reach a single goal point. Using the system as described above we show the advantage of using multiple simulators even when they are distributed across machines. Figure 6.3 shows the length of episodes as training progresses. Our system manages to achieve an average of 7 actions to get to the single goal with one simulator but takes around 10,000 episodes to achieve this. In comparison, a system using just one more worker, is able to achieve an average of 6 actions to get to a single goal in only 3000 episodes each. This increase in speed of convergence is extremely useful when using real-time robotics systems.

Now that the system is able to reach a single goal, the next experiment uses curriculum learning (as discussed in Section 3.2) to gradually expand the potential area for the goal allowing the network to generalise to any goal. The network trained on a single goal is used to initialise this experiment and the available area increased gradually from a single target to a quarter of the table then again up to the whole table. Using the pretrained network, allows the system to exploit the single goal policy for goals close to the first goal, which prevents the network from moving back to random exploration. Figure 6.4 shows that the average number of actions per episode never goes back above 30. It can also be seen that once the goal variance (orange line) finished increasing, the fluctuation in the number of actions taken starts to decrease and the final average number of actions comes down to 6, which is the same as the single goal problem. There is a greater amount of variation in the number of actions required. However, given the increased variance in the start position (meaning that there is more variation in the distance to the target), this is expected. Figures 6.5b and 6.5d show an example close to optimal performance achieved by this network in comparison to random walk in Figures 6.5a and 6.5c.

This section shows a new way of connecting ROS based robots on different machines to a single learning system, meaning that more realistic simulations can be used in ROS to train RL policies with multiple workers. As shown, multiple workers increase the speed of data collection during

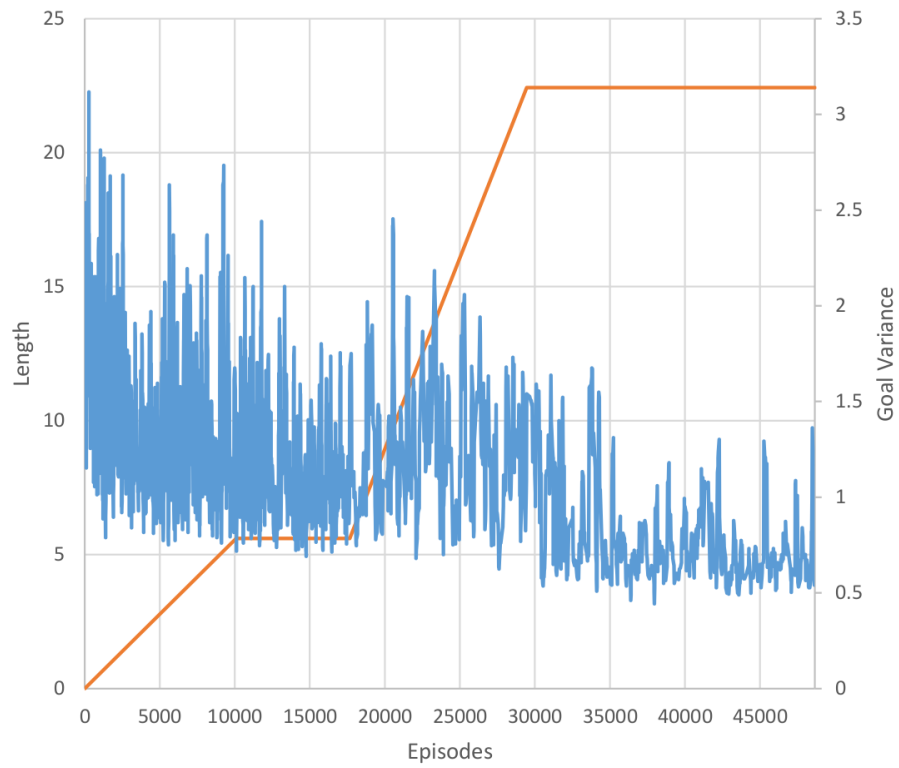


Figure 6.4: Random goal A3C Baxter simulator training - with curriculum learning where the full target area is used from 30,000. The blue line shows the average episode length and the orange line shows the goal variance during training.

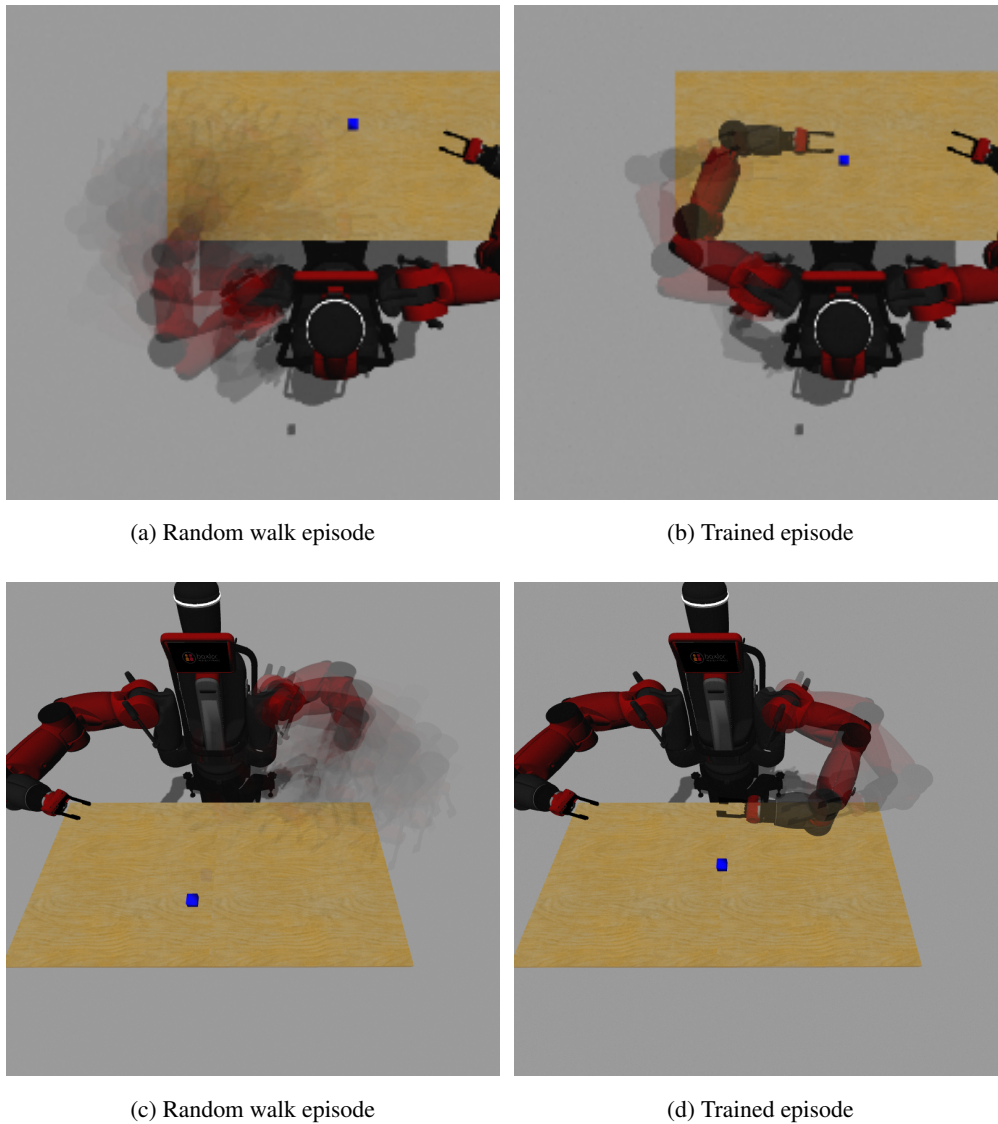


Figure 6.5: Examples of trajectories executed using the Baxter simulator - with top down view on the first row and front view on the second. The image from each state during the episode is overlaid on the next, with the least transparent state being the final one.

RL which is vital when using live systems and more computationally demanding simulations. This framework can now be used to integrate more complicated learning methods, such as APRiL, with ROS based robots.

6.2 APRiL with ROS

The integration of ROS into RL methods with a simulator, such as Gazebo, immediately means that there is greater potential for transferring control systems learnt entirely through simulation to physical robots. The ROS communication methods which allow the RL system to communicate with the simulated robot during training are identical to those used by the equivalent physical robot.

This thesis has already discussed the large quantity of data required for training an RL policy. Generating this scale of data is generally not realistically possible on most physical robotic setups due to the requirement for human supervision. During training, the agent needs to explore the state space, including failure states in order to learn from the them. However, with physical robots, these failures could be dangerous to both the robot and the environment. In simulation, these failures can be recorded and learned from safely.

In contrast, visual data can be more safely collected on real hardware without the worry of damaging the robot or other objects in the scene. The data for training perception networks can be collected once and then used to train networks repeatedly with different network architectures and other hyper-parameters. Due to the partially online nature of the RL algorithms used, this is not the case for data collected to train the control systems.

The separation of perception and control systems as described in APRiL, Section 4.2, allows us to complete the time consuming control data collection in simulation whilst training the vision system on data from the physical robot. This means that at run time, a physical robot can infer a state from a perception network trained on physical data, and take actions from a policy which uses this inferred state but was trained on exclusively simulated data. This reduces the task of transferring a system to a physical robot to only transferring the perception problem. Hence, reducing the risk to the robot and environment since it does not need to explore the failure cases.

The system diagrams in Figure 6.6 show how APRiL can be used to transfer learning to a

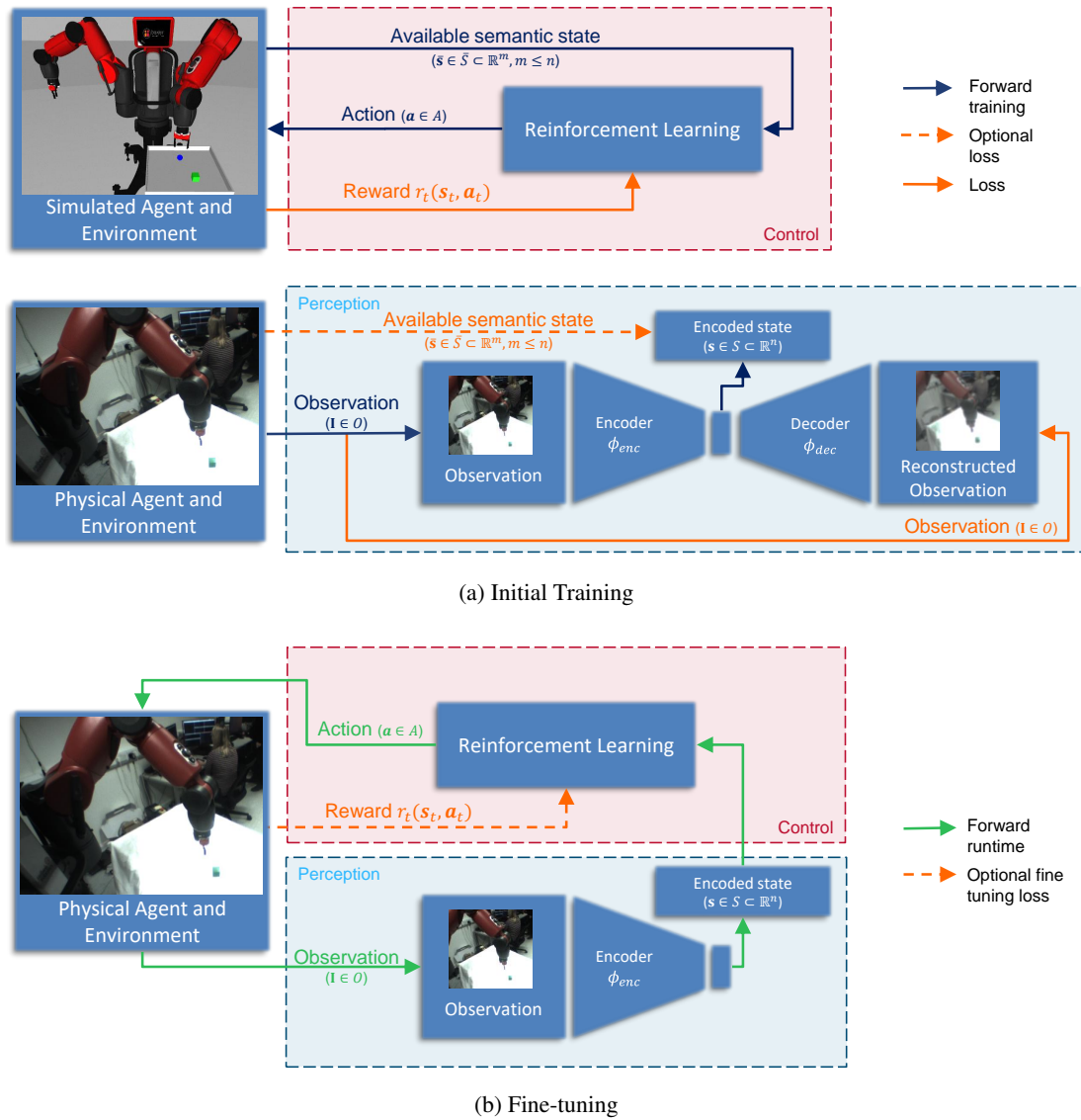


Figure 6.6: System diagrams showing how APRiL can be used for transfer of control system trained in simulation to a physical robot.

Agent Type	Policy Input	Number of Training Episodes			Average Episode Length	Std Dev
		Simulator	GP	Physical		
Simulated	GT $\bar{\mathcal{S}}$	15	2800	-	3.38	± 2.37
	Enc \mathcal{S}	65	2800	-	10.18	± 10.84
Physical	GT $\bar{\mathcal{S}}$	15	2800	0	3.52	± 2.45
	Enc \mathcal{S}	65	2800	23	8.14	± 12.75

Table 6.1: Results from APRiL for the Baxter reach environment - showing the final average episode length and the number of each type of episode use for training

live robot. The first diagram (Figure 6.6a) shows the initial training cycle where the control and perception systems can be completely separated, with the simulated robot training the RL networks and data from the physical robot used to train the perception network. The communication between the simulator and the RL system can be facilitated by the methods described earlier in this chapter with ability to expand to multiple simulators. The second diagram (Figure 6.6b) shows the system used at run-time with the physical robot, with an additional option to continue to fine-tune the RL policy if necessary but keeping the perception network static.

6.2.1 Baxter Implementation

To demonstrate the use of APRiL with ROS and the ability to transfer to a physical robot, this section starts with the reach environment, aiming to move the arm from a start position to a random goal.

Reach Environment

Initially the RL control network is trained with data from the simulator and GP episodes to improve efficiency. Then the auto-perception network is trained with additional images collected from the simulator. Examples of the images and reconstructions can be seen in the first row of Figure 6.7. The reconstruction shows that the encoded features have a reasonable understanding

of the arm position and for this application, the lack of detail in reconstruction is not important. The first two rows in Table 6.1 show the number of episodes required to train the systems to convergence. Whilst the perception only input reduces the performance slightly, it still shows good performance.

To transfer the control system to the physical robot, no more physical data is needed. The network trained on the simulated data can be used directly when using the ground truth semantic state. To transfer the perception system, a small number of episodes are needed from the physical robot for retraining. The last two rows in Table 6.1 show the number of different types of episode required to train this system. Using no perception it is possible to transfer the policy trained entirely on simulation and GP episodes directly to the physical robot without losing performance.

Using a perceived state input it was possible to transfer the the same policy to the physical robot with only 23 episodes of physical robot interaction, which are used to train the perception network. To encode a perceived state, images were collected from an RGB camera in the robots right hand, this meant a consistent camera position could easily be set. Examples of the images collected and reconstructions from the perception network can be seen in Figure 6.7. Whilst the reconstruction of the arm is not clear, the reconstructions do indicate the same position of the arm as in the original image.

As in Chapter 4, using the encoded state as input for control is noisier than using the ground truth semantic state but still achieves an average of under 10 actions per episode using only a perceived input. Interestingly this actually outperforms the simulated perception system, likely due to the images in the simulation being flat with minimal shadows making it hard to predict a position from them. In comparison, the images from the physical simulation give more information about depth from shadows and using a “side-on” camera position. This shows how the use of ROS based simulation environments, which mimic the communication of a physical robot, allow the transfer of control with no further training and only a small amount of data collection is needed for perception.

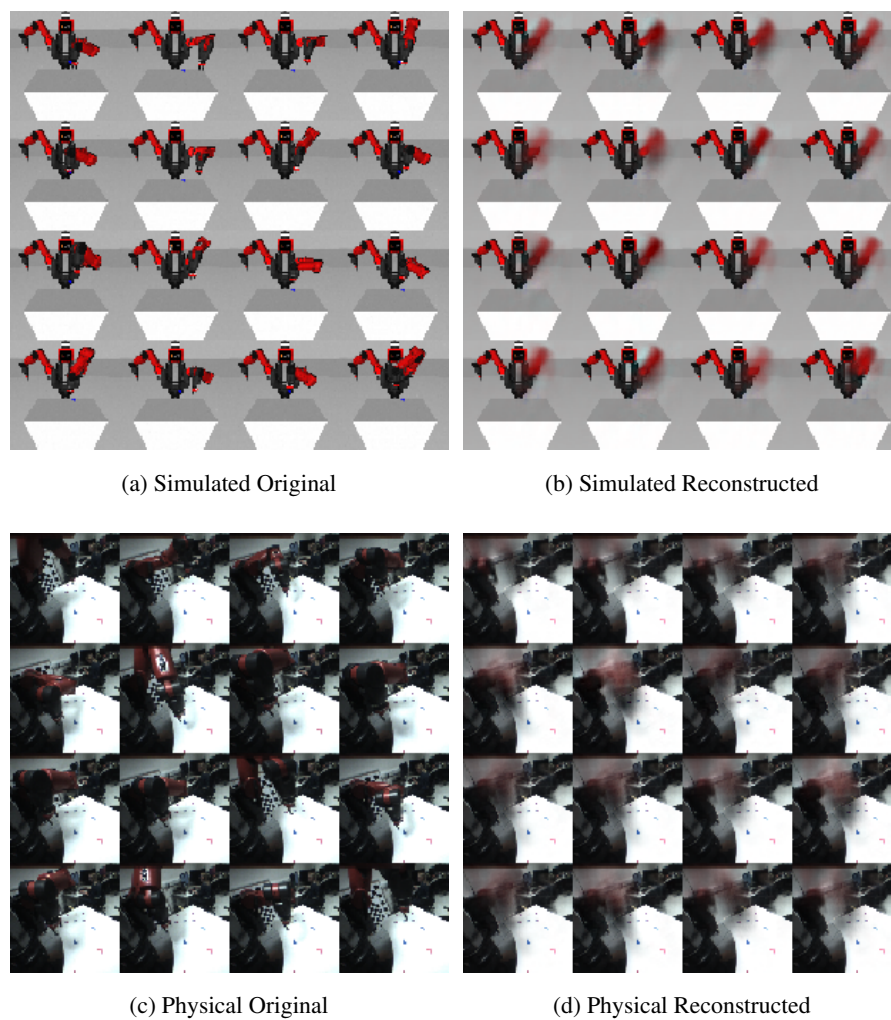


Figure 6.7: Examples of images from the physical robot and reconstructions from the auto-perceptive network

RL Input	Number of Training Episodes	Average Episode Length
GT $\bar{\mathcal{S}}$	429	40.91
Enc \mathcal{S}	429	48.65

Table 6.2: Results from APRiL for the Baxter Pick environment - showing the final average episode length

Pick Environment

This next set of experiments uses the same Baxter robot but in a picking environment. The aim of the agent is to pick up a cube from a random position and move it to a goal location in as few moves as possible. The same ROS based communication framework is used as before. The action space is again defined as, $(\Delta z, a_g)$, a change in the end-effector position and a gripper action $a_g \in (-1, 1)$. In this environment $a_g > 0.5$ sets the gripper to open, $a_g < -0.5$ sets the gripper to fully closed (or until a gripped object is detected), any other value causes no change in the gripper position. The maximum length of this task is set to $T = 100$ and the policy is initially pretrained in a supervised manner using data collected with a basic heuristic approach.

The results in Table 6.2 show that on average we are able to complete the pick task in half the time of the maximum episode length and that using only the encoded visual state, we only lose a small amount of performance. An example episode for both the ground truth state policy and the encoded state policy can be seen in Figure 6.8. Both types of state input show similar performance, but it is interesting to note that the ground truth state input attempts closing the gripper earlier, having to reopen them to complete the grasp. Here we only give results for simulation performance, since the robotics labs were inaccessible at the time of completing this work due to COVID-19 restrictions. The previous results show how using APRiL with ROS allows for smooth transition from simulation to a real robot, once you have a trained policy.

6.3 Repeated Object Detection for Grasping

The previous section provides a method for learning a grasping policy once the object to be picked is known. In order to identify which object is to be grasped during warehouse picking, an

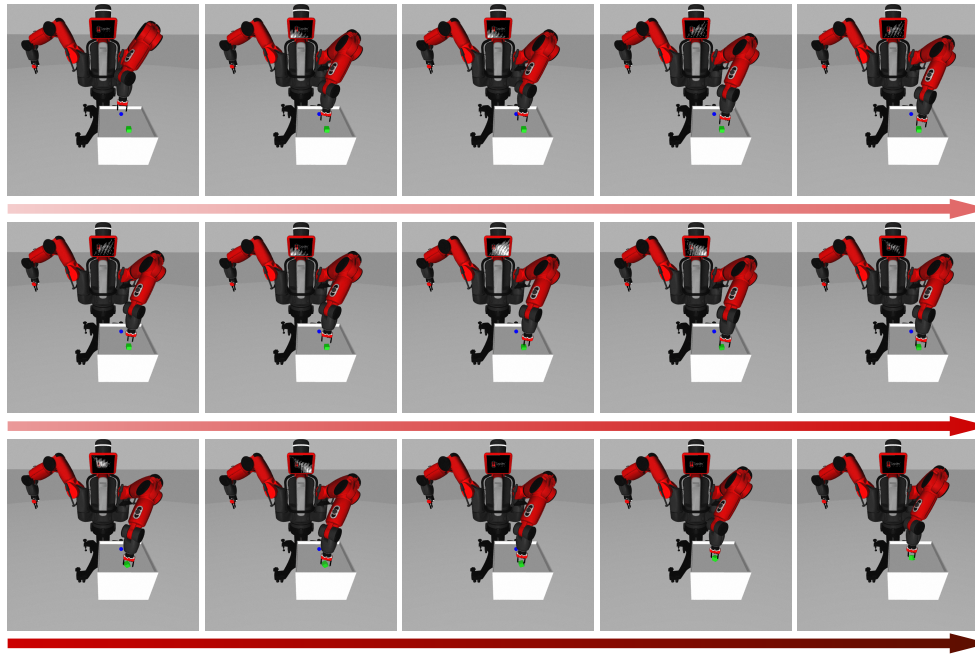
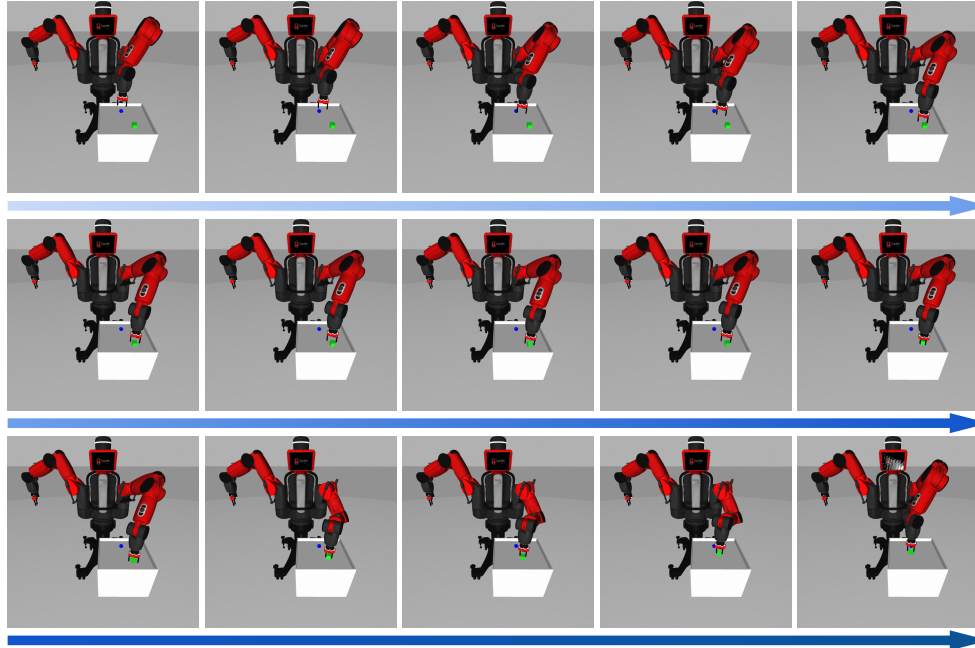
(a) GT \bar{S} Policy(b) Enc S Policy

Figure 6.8: Example sequence of GT state policy and encoded state policy in a similar initial state

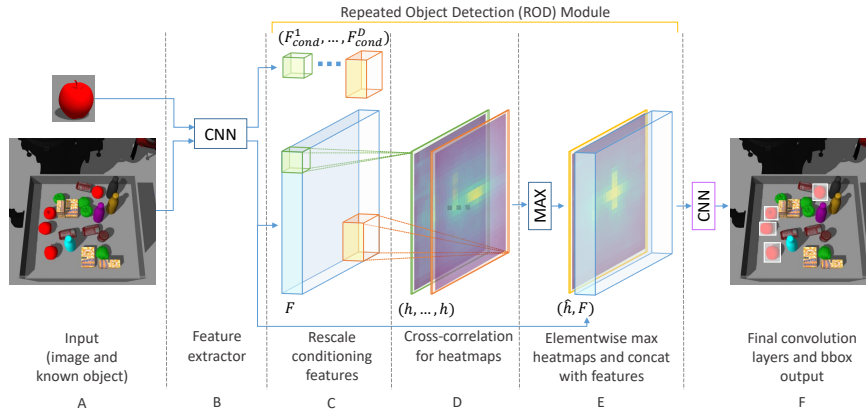


Figure 6.9: Repeated Object Detection with external conditioning image

object of the same type as the purchased item must be selected from a bin with several repeated instances, often with distractors. The potential objects will be varied, changing regularly depending on what is available from the sellers at any given time. This means techniques requiring 3D models or many training images of all available products are impractical, however, online shops will generally have at least one image of the item being sold to show to the customer on the store website. As set out in Chapter 5, ROD is well suited for this problem, given that it can be used in a zero shot manner, and has the ability to implicitly infer affordances of objects.

In this section we will show how ROD can be used to select objects to be grasped using a given policy. Online stores provide consumers with an image of an item being purchased. This image can then be used to condition the network, instead of the crop in Section 5.3.1, allowing the system to find all instances of that item in the scene. Figure 6.9 shows how ROD has been adapted to take in a target image. Both the scene image and the conditioning image are fed through the CNN backbone network to acquire two sets of features, scene features F and target features F_{cond} . The target features are resized to D scales, F_{cond}^s for $s = 1, \dots, D$, which are used to create a correlation heatmap as in Equation 5.12. The remainder of the system is used as described in Chapter 5.

Initially we show ROD being used to detect objects of the correct type in the scene. Figure 6.10 shows ROD is able to predict different selections of objects depending on the conditioning image, which is shown in the top row. The green boxes show the ground truth bounding boxes for the given object, whilst the orange boxes show the network predictions. Generally, the network will pick out the repeated objects with tight bounding boxes, without many false positives, even though there are lots of distractor objects. However, there are some false positives where

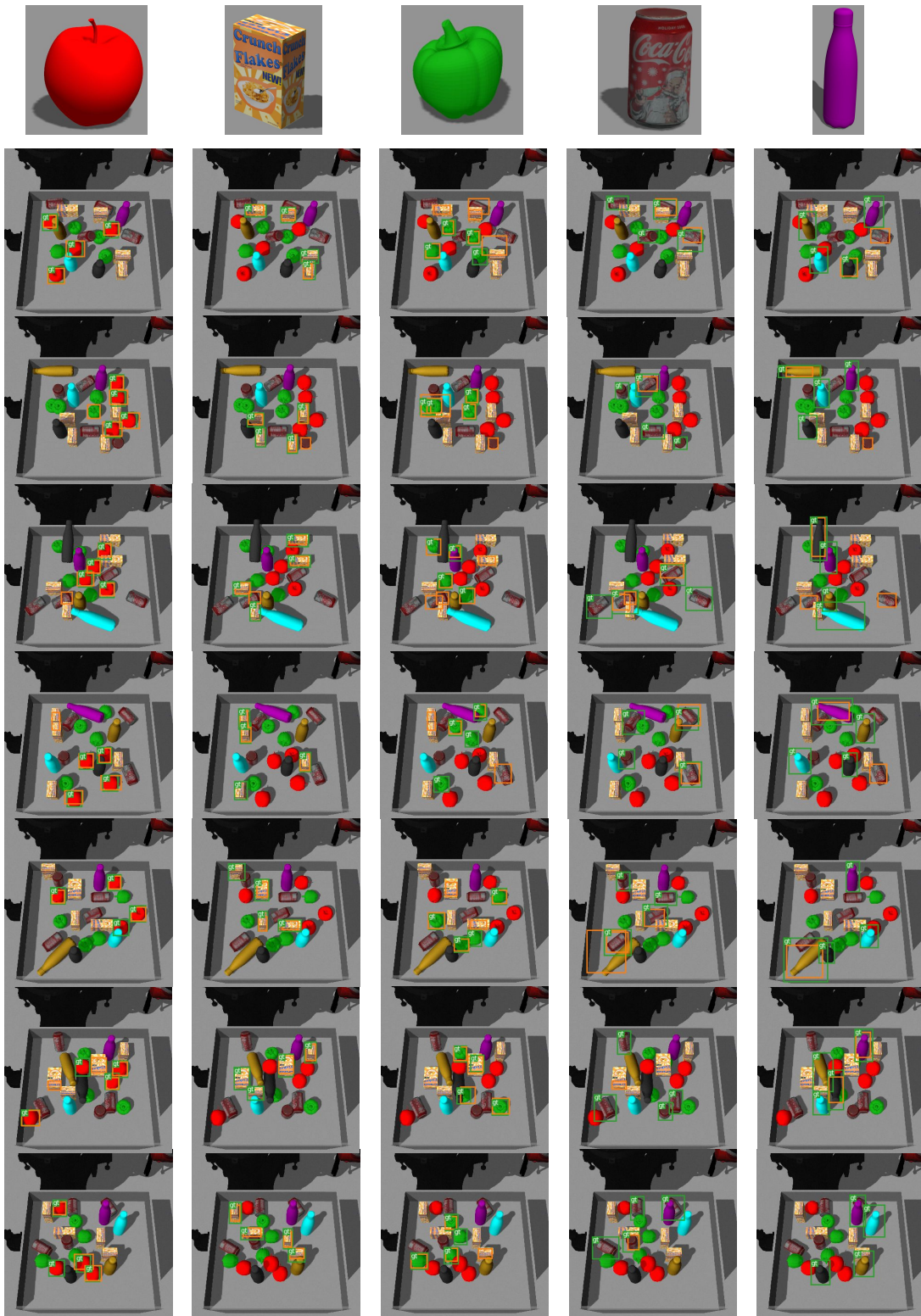


Figure 6.10: Examples of ROD with external conditioning. The top row shows the image used to condition the network, each subsequent row shows the output of a single image conditioned on each object respectively.

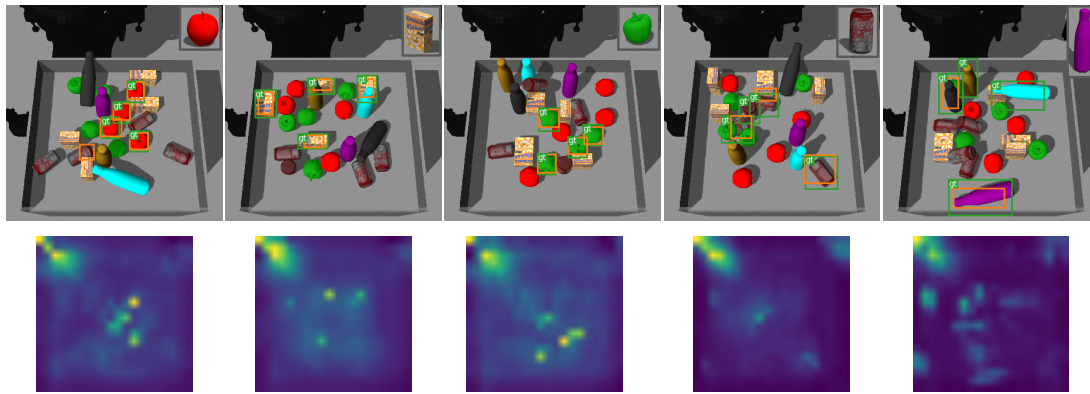


Figure 6.11: Examples of max heatmaps generated by a ROD module. The top row shows the scene and predicted bounding boxes with the image used to condition the network inset in the top right corner, the bottom row shows an elementwise maximum of cross-correlation heatmaps from three scales.

objects with similar properties are detected, including shape (the pepper detected in the second row when conditioning on an apple) and colour (part of the red can detected in the third row when conditioning on an apple). When conditioning on the can in the first row, it is interesting to note that despite only detecting two of the cans in the scene, one of the cans is detected despite being partially occluded and the other predicted bounding box is more accurate than the ground truth. The most challenging object in this dataset is the bottle, since it is the most visually different between instances and can have drastic variances in appearance depending on orientation. However, in the final column we can see several different detections of bottles in different orientations and of different colours to that of the object pictured in the conditioning image.

In Figure 6.11 we show examples of the cross-correlation heatmaps from a ROD module. The heatmaps show that features of similar objects are causing peaks in the heatmaps, even when the object is in a different orientation from the conditioning image. The cereal box example in this figure shows that these peaks in the heatmap are still present even when the object is partially obscured. The heatmap in the final column, which is conditioned on the bottle, shows peaks where the bottles occur, but also where the other cylindrical objects are laying next to each other. This is likely because the two cans end-to-end are very visually similar to a bottle.



Training Data	Test Data	$AP_{0.5}$	mAP
	Policy 1	0.5896	0.4277
	Policy 2	0.4840	0.4198
	Policy 1	0.3040	0.2701
	Policy 2	0.5014	0.4945

Table 6.3: Results of training on data collected with different policies. The ground truth test data from each policy only includes bounding boxes of objects which were successfully picked by that policy.

6.3.1 Affordances based on Policy

Depending on the policy used to collect the training data for ROD, the network can implicitly determine which objects are pickable. Section 5.4.4 showed ROD could determine which objects were pickable for a single heuristic policy. This section will use different learnt policies to collect the training data, where each object will be labelled as pickable or not depending on the outcome of the attempted grasp. The first policy is the one learnt in Section 6.2.1, which has the gripper parallel to the y-axis of the robot. The second policy, learnt via the same method, has the gripper parallel to the x-axis of the robot. Given the restraints of the parallel gripper, different objects will be pickable with different grasping angles. For example, if the bottle is parallel to the y-axis it is unlikely to be pickable via the first policy. However, it is likely that the second policy will be able to pick the object.

Table 6.3 shows quantitative results from ROD trained with the two policies described above. These results show both trained networks have higher AP values for detecting the objects which were pickable for the policies they were trained for. Figure 6.12 shows example detections of ROD trained with data collected from the two different policies. The columns show the results for different policies, with the gripper orientated as shown by the images in the first row.

The model trained with the second policy (with the gripper parallel to the x-axis of the robot) predicts that the apple with boxes either side of it is unpickable, which is correct since the gripper would hit a box whilst attempting to approach the object in either direction. The model trained with the first policy fails to detect any pickable apples, the only false negative here is the apple at the front of the bin which will often be detected as unpickable since it is at the extreme of the



Figure 6.12: Examples of ROD detections with different policies - the first policy uses a policy with the gripper parallel to the the y-axis of the robot, the second policy uses a policy with the gripper parallel to the x-axis of the robot - gripper orientation can be seen in the first row. The conditioning object is shown in the first column.

arms reach. The example conditioned on the cereal box shows how the different orientations of the box could determine pickability, with the boxes laying down not being predicted at all by the second policy. The first policy correctly detects the unobstructed box that is laying down and reasonably does not detect the two stacked boxes given the position of the blue bottle blocking the grippers approach to objects.

The example conditioned on a can shows that neither network detects the can that is partially obscured by a cereal box, which would obviously impair the ability of any policy to pick it. We can again see the effect of orientation with only one can pickable by both policies. The bottle is the most challenging object in this set, since it has the least amount of successful picks in the training data. The first model detects the shadow rather than the pickable blue bottle, whilst correctly not detecting the other bottles which are not pickable by the first policy. The purple bottle is pickable with the second policy as the gripper orientation would not knock it over on approach. There is some ambiguity around pickability when it comes to objects in close proximity to others, for example, the yellow bottle could potentially be pickable by the second policy but the cereal box could obstruct the grippers ability to secure a reasonable grasp. In this case, it is preferable for the detector to be cautious and predict other objects which are more likely to be pickable first.

6.4 Conclusion

This chapter has brought together the various pieces of work detailed in this thesis to complete the pick and place pipeline. We have shown how ROS can be used with RL in order to train policies with multiple realistic simulators. This allows efficient training of policies using APRiL which can be transferred to the real robot.

With a policy learnt in this manner, this work has shown how ROD can be used to detect repeated pickable objects depending on the policy used. This showed that depending on the policy selected, the goal object can be different in order to increase the likelihood of a successful grasp. This method of repeated object detection with implicit affordances, together with the policies learnt via APRiL and the ROS communication methods, provides the elements necessary to complete the grasping pipeline in a warehouse picking environment.

Unfortunately due to restricted access to the robotics lab because of COVID-19 the physical testing of this work was limited, but the concept was shown using the reach problem on a physical Baxter robot.

Chapter 7

Conclusions and Future Work

The work in this thesis was driven by the problem of product picking in the context of online shopping fulfilment warehouses. To address this challenge it breaks the problem down into the pick and place pipeline as shown in Figure 1.1 from Chapter 1, aiming to progress both the active grasping and object localisation stages. In order to achieve this, the main objectives of the thesis were:

1. To develop an efficient learning method for robotic control which can determine actions from visual data.
2. To develop techniques to locate repeated objects in an image without prior knowledge of object type.
3. To explore how these methods can be used together to perform warehouse style picking.

This chapter will briefly summarise the conclusions from each chapter of this thesis and discuss how it addresses these objectives.

Chapter 3 discussed Reinforcement Learning and how it could be applied to robotics problems. It showed how techniques such as HER and GP modelling can be used together to increase data efficiency and performance. The results showed promise at addressing the first objective of developing an efficient learning method for robotic control, but required direct access to a semantically understood state space at both training and deployment.

Chapter 4 addressed the challenges raised in Chapter 3 by introducing Auto-Perceptive Reinforcement Learning (APRiL). The proposed method is inspired by how children learn visual motor tasks, in one system but with different processes. APRiL can predict actions directly from images at deployment, whilst making use of any available semantically understood state during training. The value of allowing the visual system to encode additional information into the state space, over and above the information in the available semantic state, was shown by improvement in performance in a robotic picking task. Nevertheless, the use of the available semantic information allowed the use of the efficient RL techniques that were explored in Chapter 3, giving greater performance than just encoding the state from images without conditioning. APRiL provides both an efficient learning method for robotic control and the ability to predict actions from visual data, concluding the first objective.

Chapter 5 moved from investigating the active grasping stage of the pick and place pipeline, to investigating the object localisation problem described in the second objective. Initially, a method using recurrent networks to sequentially select repeated objects from scene was investigated. However, after some initial work, it was concluded that it did not effectively allow for comparison of objects within the network in order to determine similarity. To solve this issue Repeated Object Detection (ROD) was proposed which employed a feedforward network and cross-correlation at various scales to detect similar objects in a scene. We showed how ROD improves performance over TDID for detecting repeated objects on both generic instance datasets and robotics datasets - where the definition of similarity includes the affordance of pickability. ROD can be used in a zero-shot manner without needing knowledge of all the objects during training, as shown with both styles of datasets. This addressed the second objective of the thesis.

Chapter 6 aimed to explore how the methods from the previous chapters could be combined to complete our initial problem of warehouse picking. This work started by describing a system to smoothly integrate RL methods with robotics platforms in ROS, which is necessary for deployment on real world robots. Using this framework it was shown how APRiL can effectively be used to transfer a policy learnt in simulation into the real world with only minimal additional data. In order to use these policies in a warehouse picking environment, this chapter then went on to show how ROD could be used to detect pickable objects for a given policy and hence select a target object to be picked. This means we are able to localise a desired object,

move to it and pick it with a learnt policy. This allowed us to complete the pick and place pipeline for warehouse style picking as described in the third objective.

7.1 Future Work

This thesis has addressed various aspects of robotic grasping using multiple forms of deep learning. To conclude, this section will focus on several aspects of important future work which could improve both the base performance of these algorithms but also the specific application of warehouse picking.

7.1.1 Reinforcement Learning

The first area of future work to be considered would be to continue exploring the transfer of simulated policies learnt using APRiL onto physical robots for more complex tasks, especially once COVID-19 restrictions have been lifted making access to hardware possible again. In terms of other challenges in reinforcement learning, for applications such as robotic grasping, the sparsity of rewards is one which needs further consideration. Whilst APRiL provides an efficient method for learning a vision based grasping policy, as well as allowing techniques which improve learning for sparse rewards such as HER, it still uses a reward which only gives feedback once it achieves the goal. This makes it incredibly difficult for the RL algorithm to assign credit to the actions taken throughout an episode. However, if we have some expert examples, a smooth, shaped reward signal could potentially be learned without hand engineering using Inverse Reinforcement Learning (IRL) [72]. Finn et al. [29] use IRL for a selection of robotics tasks by parameterising the reward using a neural network which takes the raw state in terms of the gripper pose and velocity relative to the target position. Combining this method of learning with APRiL would be an interesting line of work, in order to efficiently use IRL to learn a vision based policy.

7.1.2 Repeated Object Detection

ROD offers a solution to the repeated instance localisation problem at the start of the pick and place pipeline which does not require classification as part of the network. This still requires

the training data to have instance level class annotations. A future line of work would be to investigate how to improve the accuracy of this method without requiring labelled data via unsupervised or weakly supervised training. In addition, ROD uses fixed scales with each correlation module and could be improved by introducing learnable scales, in order to create correlation maps with the conditioning features resized to the most informative scales.

To improve the integration of ROD with policy learning methods such as APRiL, the next steps would involve considering reworking the integration of ROD and the policy learning method. For example, we could examine how ROD could be used as part of the perception network, and allowing the policy to select the object to be picked based on the conditioned features learnt from detection.

7.1.3 Object Manipulation and Placement

In order to use the methods in this thesis with all possible objects within an online shopping fulfilment warehouse, we must consider how to manipulate and place objects safely once they have been picked. This problem has been tackled far less than grasping in the robotics community and deep learning has not been utilised. There have been approaches to placement which look at finding a clutter free flat surface [95], however, these require the use of 3D information from a sensor such as a tilting laser range finder (LIDAR). There are also model based methods [41] which use environmental constraints to find valid placement methods but also require accurate 3D information about the environment and only consider very restricted configurations.

In order to avoid the need for accurate 3D data, we believe deep learning could be used to assess the most successful method for the placement of objects, especially into already cluttered environments such as those in crates for online shopping fulfilment. This could be achieved by building on APRiL using IRL to bypass the need for an engineered scoring function for safe placements. Another interesting line of work would be to explore the most efficient way to pack objects. This would involve making the most of the space in the customers crate, whilst still making sure the object that is being placed and those already in place do not get damaged. To do this “optimal packing” autonomously would require a step-change in forward planning. Not only considering placement, but all future placements.

Bibliography

- [1] 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. Software available from tensorflow.org.
- [2] Bogdan Alexe, Thomas Deselaers, and Vittorio Ferrari. Measuring the objectness of image windows. *IEEE transactions on pattern analysis and machine intelligence*, 34(11):2189–2202, 2012.
- [3] Rebecca Allday, Simon Hadfield, and Richard Bowden. From Vision to Grasping : Adapting Visual Networks. *18th Towards Autonomous Robotic Systems (TAROS) Conference, 2017*.
- [4] Rebecca Allday, Simon Hadfield, and Richard Bowden. Auto-perceptive reinforcement learning (april). In *The 3rd International Workshop on the Applications of Knowledge Representation and Semantic Technologies in Robotics (AnSWeR19) at IROS, 2019*.
- [5] Rebecca Allday, Simon Hadfield, and Richard Bowden. Repeated object detection (rod): Zero-shot generic object detection for recurring instances. In *International Conference on Robotics and Automation (ICRA)*. Under Review, 2021.
- [6] Ethem Alpaydm. *Introduction to machine learning*, volume 1107. 2014.

-
- [7] Phil Ammirato, Cheng-Yang Fu, Mykhailo Shvets, Jana Kosecka, and Alexander C. Berg. Target driven instance detection. *arXiv*, 2018.
- [8] Phil Ammirato, Patrick Poirson, Eunbyung Park, Jana Kosecka, and Alexander C. Berg. A dataset for developing and benchmarking active vision. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2017.
- [9] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.
- [10] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 39(12):2481–2495, 2017.
- [11] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.
- [12] Archith John Bency, Heesung Kwon, Hyungtae Lee, S Karthikeyan, and BS Manjunath. Weakly supervised localization using deep feature maps. In *European Conference on Computer Vision*, pages 714–731. Springer, 2016.
- [13] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. *Proceedings of the 26th Annual International Conference on Machine Learning - ICML '09*, pages 1–8, 2009.
- [14] Jeannette Bohg, Antonio Morales, Tamim Asfour, and Danica Kragic. Data-Driven Grasp Synthesis: A Survey. *IEEE Transactions on Robotics*, 30(2):1–21, 2014.
- [15] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.
- [16] Konstantinos Chatzilygeroudis, Roberto Rama, Rituraj Kaushik, Dorian Goepp, Vassilis Vassiliades, and Jean-Baptiste Mouret. Black-box data-efficient policy search for robotics. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 51–58. IEEE, 2017.

-
- [17] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [18] W. Curran, R. Pocius, and W. D. Smart. Neural networks for incremental dimensionality reduced reinforcement learning. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1559–1565, 2017.
- [19] William Curran, Tim Brys, David W Aha, Matthew E Taylor, and William D Smart. Dimensionality reduced reinforcement learning for assistive robots. In *AAAI Fall Symposia*, 2016.
- [20] Navneet Dalal and Bill Triggs. Histograms of Oriented Gradients for Human Detection. In *International Conference on Computer Vision & Pattern Recognition (CVPR '05)*, volume 1, pages 886–893. IEEE Computer Society, June 2005.
- [21] M. P. Deisenroth and C. E. Rasmussen. Pilco: A model-based and data-efficient approach to policy search. *International Conference on Machine Learning (ICML)*, 2011.
- [22] Dematic. White paper: Automated storage systems optimize goods-to-person order fulfillment. Technical report, SupplyChain247, June 2019.
- [23] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. ImageNet: A large-scale hierarchical image database. *Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.
- [24] Staffan Ekvall and Danica Kragic. Learning and evaluation of the approach vector for automatic grasp generation and planning. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 4715–4720. IEEE, 2007.
- [25] Sahar El-Khoury and Anis Sahbani. Handling objects by their handles. *IEEE/RSJ International Conference on Intelligent Robots and Systems, Workshop on grasp and task learning by imitation*, pages 58–64, 2008.
- [26] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(Apr):503–556, 2005.

-
- [27] Meng Fang, Tianyi Zhou, Yali Du, Lei Han, and Zhengyou Zhang. Curriculum-guided hindsight experience replay. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems* 32, pages 12623–12634. Curran Associates, Inc., 2019.
- [28] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE transactions on pattern analysis and machine intelligence*, 32(9):1627–1645, 2009.
- [29] Chelsea Finn, Sergey Levine, and Pieter Abbeel. Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization. *Icml 2016*, 48(2000), 2016.
- [30] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Deep spatial autoencoders for visuomotor learning. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 512–519. IEEE, 2016.
- [31] C. Fitzgerald. Developing baxter. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6, 2013.
- [32] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy Networks for Exploration. 2017.
- [33] Scott Fujimoto, Herke van Hoof, and Dave Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.
- [34] Georgios Georgakis, Md Alimoor Reza, Arsalan Mousavian, Phi-Hung Le, and Jana Košecká. Multiview rgb-d dataset for object instance detection. In *2016 Fourth International Conference on 3D Vision (3DV)*, pages 426–434. IEEE, 2016.
- [35] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.
- [36] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

-
- [37] GPy. GPy: A gaussian process framework in python. <http://github.com/SheffieldML/GPy>, since 2012.
- [38] Kaiming He, Georgia Gkioxari, Piotr Dollar, and Ross Girshick. Mask r-cnn. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.
- [39] Geoffrey E Hinton, Alex Krizhevsky, and Sida D Wang. Transforming auto-encoders. In *International conference on artificial neural networks*, pages 44–51. Springer, 2011.
- [40] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, 2006.
- [41] Anne Holladay, Jennifer Barry, Leslie Pack Kaelbling, and Tomas Lozano-Perez. Object placement as inverse motion planning. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 3715–3721, 2013.
- [42] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH 50X FEWER PARAMETERS AND <0.5MB MODEL SIZE. *arXiv*, pages 1–5, 2016.
- [43] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15*, page 448–456. JMLR.org.
- [44] Sergi H. Juan and Fernando H. Cotarelo. Multi-master ros systems. Technical Report IRI-TR-15-1, Institut de Robòtica i Informàtica Industrial (IRI), Universitat Politècnica de Catalunya (UPC), January 2015.
- [45] Daiki Kimura. DAQN: Deep Auto-encoder and Q-Network. *arXiv*, abs/1710.06542, 2018.
- [46] Diederick P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.
- [47] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

-
- [48] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE.
- [49] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 3, pages 2619–2624 Vol.3, April 2004.
- [50] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.
- [51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, pages 1–9, 2012.
- [52] Michael Land and Benjamin Tatler. *Looking and acting: vision and eye movements in natural behaviour*. Oxford University Press, 2009.
- [53] S. Lange and M. Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2010.
- [54] Sascha Lange, Thomas Gabel, and Martin Riedmiller. *Batch Reinforcement Learning*, pages 45–73. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [55] Quoc V Le. Building high-level features using large scale unsupervised learning. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8595–8598. IEEE, 2013.
- [56] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel. Handwritten Digit Recognition with a Back-Propagation Network. *Advances in Neural Information Processing Systems*, pages 396–404, 1990.
- [57] Yann LeCun, Bengio Yoshua, and Hinton Geoffrey. Deep learning. *Nature*, 521(7553):436–444, 2015.

-
- [58] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-End Training of Deep Visuomotor Policies. *Journal of Machine Learning Research*, 17:1–40, 2016.
- [59] Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*, pages 1–9, 2013.
- [60] Sergey Levine, Peter Pastor, Alex Krizhevsky, and Deirdre Quillen. Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection. *arXiv*, 2016.
- [61] 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*, pages 1–14, 2015.
- [62] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.
- [63] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing.
- [64] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.
- [65] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.
- [66] Andrew T Miller. *GraspIt!: A Versatile Simulator for Robotic Grasping*. PhD thesis, COLUMBIA UNIVERSITY, 2001.
- [67] 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, 2016.

-
- [68] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dhharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518, 2015.
- [69] MSCOCO Dataset detection evaluation. <http://cocodataset.org/index.htm#detection-eval>. Accessed: 2019-10-15.
- [70] Richard M Murray, Zexiang Li, and S Shankar Sastry. *A Mathematical Introduction to Robotic Manipulation*, volume 29. CRC Press, 1994.
- [71] Ashvin V Nair, Vitchyr Pong, Murtaza Dalal, Shikhar Bahl, Steven Lin, and Sergey Levine. Visual reinforcement learning with imagined goals. In *Advances in Neural Information Processing Systems*, pages 9209–9220, 2018.
- [72] Andrew Ng and Stuart Russell. Algorithms for inverse reinforcement learning. *Proceedings of the Seventeenth International Conference on Machine Learning*, 0:663–670, 2000.
- [73] H. Nguyen, H. M. La, and M. Deans. Hindsight experience replay with experience ranking. In *2019 Joint IEEE 9th International Conference on Development and Learning and Epigenetic Robotics (ICDL-EpiRob)*, pages 1–6, 2019.
- [74] Van-Duc Nguyen. Constructing Force- Closure Grasps. *The International Journal of Robotics Research*, 7(3):3–16, 1988.
- [75] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.
- [76] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.
- [77] Raphael Pelossof, Andrew Miller, Peter Allen, and Tony Jebara. An SVM Learning Approach to Robotic Grasping. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE*, 2004.

-
- [78] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 1–8. IEEE, 2018.
- [79] Lerrel Pinto and Abhinav Gupta. Supersizing Self-supervision: Learning to Grasp from 50K Tries and 700 Robot Hours. In *2016 IEEE international conference on robotics and automation (ICRA)*, volume 2016-June, pages 3406–3413. Institute of Electrical and Electronics Engineers Inc., jun 2016.
- [80] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
- [81] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.
- [82] M Reichel. Transformation of shadow dextrous hand and shadow finger test unit from prototype to product for intelligent manipulation and grasping. In *Intelligent Manipulation and Grasping, Inter Conf., Genova, Italy*, volume 70, 2004.
- [83] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.
- [84] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.
- [85] A Rodriguez, M. T. Mason, and S. Ferry. From Caging to Grasping. *The International Journal of Robotics Research*, 31(7):886–900, 2012.
- [86] Javier Romero, Hedvig Kjellstrom, and Danica Kragic. Modeling and evaluation of human-to-robot mapping of grasps. In *2009 International Conference on Advanced Robotics*, pages 1–6. IEEE, 2009.
- [87] Stephane Ross and J. Bagnell. Agnostic system identification for model-based reinforcement learning. *Proceedings of the 29th International Conference on Machine Learning, ICML 2012*, 2, 03 2012.

-
- [88] A. Sahbani, S. El-Khoury, and P. Bidaud. An overview of 3D object grasp synthesis algorithms. *Robotics and Autonomous Systems*, 60(3):326–336, 2011.
- [89] J Salisbury. Design and control of an articulated hands. In *Proc. of the International symposium on Design and Synthesis, Tokyo*, 1984.
- [90] Amaia Salvador, Miriam Bellver, Manel Baradad, Ferran Marques, Jordi Torres, and Xavier Giro-i Nieto. Recurrent Neural Networks for Semantic Instance Segmentation. *arXiv*, 2017.
- [91] Ashutosh Saxena, Justin Driemeyer, and Andrew Y. Ng. Robotic Grasping of Novel Objects using Vision. *The International Journal of Robotics Research*, 27(2):157–173, 2008.
- [92] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.
- [93] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.
- [94] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [95] Martin J. Schuster, Jason Okerman, Hai Nguyen, James M. Rehg, and Charles C. Kemp. Perceiving clutter and surfaces for object placement in indoor environments. In *2010 10th IEEE-RAS International Conference on Humanoid Robots, Humanoids 2010*, pages 152–159, 2010.
- [96] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Joshua Susskind, Wenda Wang, and Russell Webb. Learning from simulated and unsupervised images through adversarial training. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2107–2116, 2017.
- [97] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever,

-
- Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
- [98] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *ICML'14*, page I–387–I–395. JMLR.org, 2014.
- [99] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. *Advances in neural information processing systems*, 28:3483–3491, 2015.
- [100] Bradly C. Stadie, Sergey Levine, and Pieter Abbeel. Incentivizing exploration in reinforcement learning with deep predictive models. *CoRR*, abs/1507.00814, 2015.
- [101] Richard S Sutton and Andrew G Barto. *Reinforcement Learning : An Introduction*. The MIT Press, 1998.
- [102] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.
- [103] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June:1–9, 2015.
- [104] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 23–30. IEEE, 2017.
- [105] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

- [106] Paul Viola, Michael Jones, et al. Rapid object detection using a boosted cascade of simple features. *CVPR (1)*, 1(511-518):3, 2001.
- [107] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
- [108] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.
- [109] Melonee Wise, Michael Ferguson, Derek King, Eric Diehr, and David Dymesich. Fetch and freight: Standard platforms for service robot applications. In *Workshop on Autonomous Mobile Service Robots*, 2016.
- [110] Richard Zhang, Phillip Isola, and Alexei A Efros. Colorful image colorization. In *European conference on computer vision*, pages 649–666. Springer, 2016.