Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Imitation Learning with Visual Attention for Robot Control

Author:
Pablo Gorostiaga Belío

Supervisor:
Dr. Edward Johns

Second Marker:
Dr. Antoine Cully

June 15, 2020

Abstract

In recent years, deep reinforcement learning algorithms have emerged as the predominant method to train robotic agents. Due to their exploratory nature, these techniques are too data inefficient, requiring hours or days of interaction time with their environment to learn successful policies, often requiring constant supervision by human operators. In contrast, with *imitation learning*-based training methods, human demonstrations are collected and robot controllers are trained to imitate the expert actions in a much more efficient supervised learning fashion. Controllers trained with this technique, however, suffer from *distribution drift*, where a controller fails to achieve the task it was trained to perform due to the accumulation of prediction errors at test time.

In this thesis, we investigate the use of visual attention to mitigate the effect of distribution drift on robotic controllers trained with imitation learning. With this aim, we propose hard attention, spatial feature-based attention and recurrent soft attention architectures. In addition, we explore meta-learning attention and develop a novel hard attention meta-learning algorithm based on spatial transformer networks. We show that under a wide range of variability conditions across multiple robot control tasks in simulation, our attention-leveraging controllers consistently outperform controllers without attention.

Acknowledgements

I would like to thank my supervisor, Ed, for his excellent advice and encouragement during this project.

I would also like to thank all the researchers who, in their quest for artificial intelligence, inadvertently inspired me to pursue this degree.

To my friends, I thank you for all the good moments in the last four years.

Most importantly, I thank my family for their effort and support throughout this journey, without whom I would not be where I am today.

Contents

Chapter 1

Introduction

Robotics and automation have become prevalent in society in recent years, particularly at industry level [1], while household solutions such as robotic vacuum cleaners have also experienced unprecedented rise [2]. Economical, flexibility-of-use and performance factors have all contributed to this phenomenon. Increasingly, robots are being used to solve complex problems that require them to learn dynamic behaviours to adapt to changes in their environment. Many of these tasks require visual input - the robot utilises vision (a camera) to sense its environment. As an example, we may want a robotic arm to reach and grasp an observed target cube; the location and shape of the target cube may vary, which renders traditional, static robotic behaviours unusable.

To train a robot to perform a certain task, state-of-the-art work employs deep learning-based approaches [3, 4, 5, 6, 7]. One common method is deep reinforcement learning. This is a process of trial and error by which the robot explores and finds the optimal sequence of actions to take in order to achieve a certain task. However, this exploratory learning process takes impractical amounts of time on physical robots, often requiring human operators to be present during training.

A common approach that attempts to mitigate the issue is to combine reinforcement learning with *imitation learning*: we incorporate expert, human knowledge into the reinforcement learning process of the robot in the form of demonstrations [3, 7, 8, 9, 10, 11]. Even though the initial learning speed is increased by grounding the reinforcement learning stage on human demonstrations, these methods still remain inefficient.

Other methods such as [5] explore training uniquely with imitation learning, *without requiring reinforcement learning*. Indeed, obtaining demonstrations allows for training robots directly in a supervised learning fashion, thereby improving the efficiency of the training process; these are known as *behavioural cloning* (BC) approaches.

Behavioral cloning, or training uniquely with demonstrations, brings about the challenge of *distribution drift* - the robot can learn to imitate the human expert, but will make mistakes when predicting which action to take. At test time, accuracy errors will increasingly accumulate during trajectories and will lead the robot into observing images of its environment from a distribution different to its training distribution. High-dimensional inputs exacerbate this situation: higher resolution means bigger variation in sequences of images captured after imprecise actions.

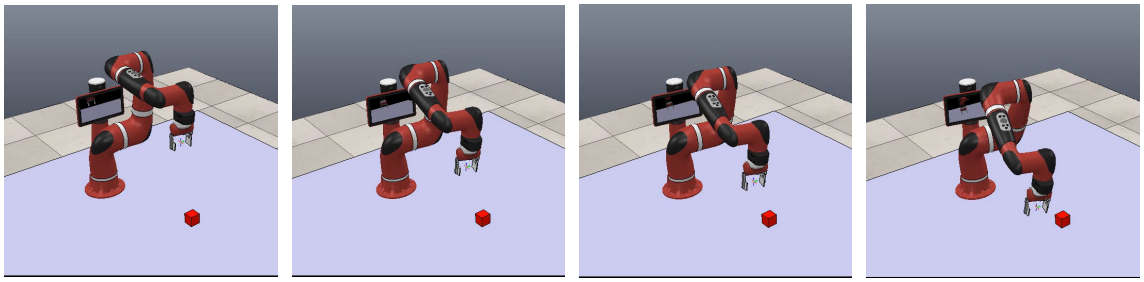
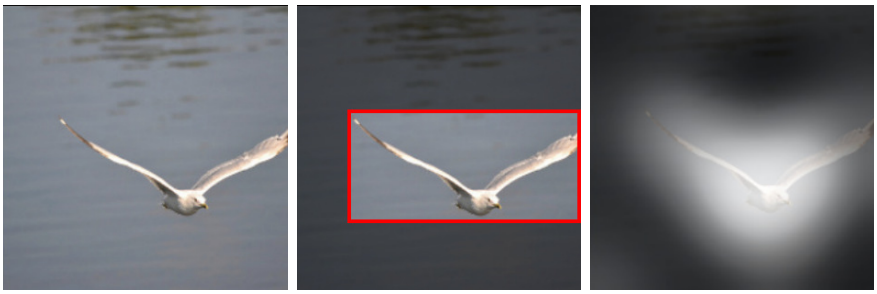


Figure 1.1: Distribution drift experienced by a Sawyer robot when attempting to reach a target cube in simulation. The robot shows progress in the first few frames, but then errors accumulate making it unable to reach the target cube.

This accumulation of errors can often lead to complete failure to achieve the required task. Figure 1.1 shows an example of this behaviour in a task where a Sawyer robot has to reach a target cube, by using images from a camera attached to its arm to predict end-effector velocities. The last frame shows how compounded errors prevented the robot from reaching the target.

We hypothesise that by learning how to focus the input of the neural network to the areas of the input image that matter for the task, we can ensure the controller is less affected by task-irrelevant variability in its input distribution. Focusing, or attention, may take a variety of forms. We show two variants in Figure 1.2.



(a) (b) (c)

Figure 1.2: Different ways of focusing on a region in an image, where (a) is the original image. In image (b) a bounding-box has been used to crop a relevant part of the image (*hard attention*) and in (c) attention is concentrated on the brighter areas in the image without reducing the size of the image (*soft attention*). Image (c) is taken from [12].

Effectively, this would provide stabler performance under the effect of the inevitable inaccuracies of demonstration-trained control models. In the example in Figure 1.1, the robot would learn to focus its visual input on the target cube.

1.1 Objectives

We summarise the main objectives of this project:

- Generate vision-based demonstration datasets in simulation for multiple robot-control tasks. Train supervised models on the full, high-dimensional demonstrations. Validate the effect of distribution drift on robotic behaviour.

- Investigate the effect of focusing on lower dimensional, task-relevant parts of images, used as the input to robot-control networks, when compared to their high dimensional input counterparts. Explore, in addition, various ways of encoding the spatial information associated with these lower-dimensional image crops, and their influence on model performance.

- Propose, implement and evaluate various supervised and reinforcement learning algorithms to learn automatically which features to attend to. This would be done in order to maximise model performance across a variety of demonstration-based robot control tasks.

1.2 Contributions and Structure

In the following, we detail the contributions in this thesis while referencing relevant sections in our report:

- **We generate multiple demonstration based-datasets**. Specifically, we generate ten varied demonstration-based datasets from five different scenes for a control task (Chapter 4) in simulation. Moreover, we generate demonstrations for a fine-grained robotic control task where the aim is to insert a disc into a peg (Section 9.3).

- **We explore the role of hard attention in mitigating distribution drift** (Chapter 5). To do so, we present four different ways of introducing hard attention into neural network controllers, by leveraging predefined relevant regions of each demonstration image (Sections 5.1.1 and 5.1.2). We show that, for the same network backbone, training process and data, including hard attention in behavioural cloning models greatly enhances performance when compared with a variety of full-sized input baseline networks on a sample dataset (Section 5.2). Moreover, we demonstrate that up to 8x less data is required to train some of our attention models to the same performance as full-sized input networks.

- We then shift our focus to learning these relevant locations *automatically* through Chapters 6-8. In Chapter 6, specifically, **we introduce spatial feature-based attention**, and devise a method to learn task-relevant pixel locations using deep spatial autoencoders (Section 6.1). We train controllers using this technique (Section 6.2), obtaining much better performance than non-attentive networks. We then propose two heuristic hyperparameter optimisation-based techniques (Section 6.3) to automatically learn hard attention windows from these spatial locations.

- In Chapter 7, **we examine the suitability of soft attention models** for robotic control tasks and suggest two different recurrence-based soft attention models.

- **We also investigate the applicability of meta-learning techniques** when learning automatically which part of an image to attend to (Chapter 8). In particular, we propose and evaluate two meta-learning hard attention approaches. Firstly, we develop a reinforcement learning agent that controls a cropping window (Section 8.1). Secondly, we present MetaSTN, a novel meta-learning optimisation-based algorithm that learns a hard attention policy in a (sub)differentiable manner via spatial transformer networks (Section 8.2).

- Finally, **we perform an in-depth evaluation** of the methods presented in this thesis on every generated dataset (Chapter 9). We conduct an investigation into the

effect of variability in multiple simulated environments both at training time and at test time and demonstrate that attentive models can cope more effectively than full-sized input networks. To conclude, we discuss the training/inference time and capacity trade-offs between the different approaches presented in Chapters 5-8, and compare their key strengths and weaknesses.

For convenience of the reader, we show in Table 1.1 a summary of the methods we introduce on each chapter along with their key differences.

Chapter	Attention type	Automatic?	Description
Chapter 5	Hard attention	✗	Methods employing hard attention crops, where the crop location is handcrafted and obtained from a simulator.
Chapter 6	Spatial keypoints	✓	Methods that automatically learn and use relevant (x, y) pixel positions on each image.
Chapter 7	Recurrent soft attention	✓	Methods that automatically learn soft attention probabilities that indicate the importance of image regions.
Chapter 8	Meta-learnt hard attention	✓	Methods that automatically learn to control a cropping window to maximise generalisation of an underlying controller.

Table 1.1: Key characteristics of methods presented in each chapter. In the table, the third column denotes whether the attention mechanism is learnt automatically or is human-specified.

Chapter 2

Background

In this section, we introduce deep learning, with an emphasis on convolutional, recurrent and autoencoder models. Moreover, we present both tabular and deep reinforcement learning, the latter being a common method of training robotic agents. Simultaneously, we describe some recent state-of-the-art RL algorithms. Finally, we discuss meta-learning, a growing field of techniques that enable more efficient learning and improved model generalisation.

2.1 Deep Learning

Deep learning is a branch of machine learning (ML) concerning algorithms that use artificial neural networks (ANNs) to encode, decode, transform and generate data, in order to, for instance, learn efficient representations, generate or predict valuable information from training data. Unlike traditional ML techniques, where features are hand-crafted, deep learning models learn both how to generate the desired output and the features to do so directly from data [13, Chapter 1]. Some of the most recent notable successes of deep learning include deep neural network architectures that achieve human-level image classification (ImageNet challenge), networks that are able to translate and caption video streams or models that are able to replicate human artistic skills.

In this section, we will elaborate on some of the underlying mathematical details of artificial neural networks, specifically concentrating on convolutional neural networks, recurrent neural networks and autoencoders.

We distinguish three main paradigms:

- In *supervised learning*, the objective is to learn the best possible approximation to a function f (assumed to exist) such that $f(x_i) = y_i$ for some dataset $D = \{(x_1, y_1), ..., (x_N, y_N)\}$ with the hope that it generalises well to unseen (x, y) data pairs. This requires that the data is labeled: each x_i has some label y_i.
- *Unsupervised learning* aims to detect and extract structure from data. This is done via algorithms such as PCA, which perform dimensionality reduction, or clustering, which finds commonalities between data samples and groups them in different classes.
- *Reinforcement learning* (RL) deals with problems in which the optimal decision is unknown, and as such, there are no labels. Unlike unsupervised learning, the aim in RL is to maximise some reward via interaction with an environment. We provide an overview in Section 2.2.

2.1.1 Artificial Neural Networks

Artificial Neural Networks, originally inspired by biological neurons, apply a sequence of transformations and non-linear functions to some input data, and can be trained to produce a desired output. An example of the use of a neural network could be a model that takes as an input a vector of the joint angles of a robot, and predicts the position of the end-effector of the robot after executing the movement specified by the angles.

Mathematically, a neural network is a transformation f_θ, representing a function $f_\theta(x) = \hat{y}$, where $\hat{y}$ is a predicted output. This function is parameterised by θ (the *weights*), and these can be learned in a process called *backpropagation* that minimises the error between the predicted $\hat{y}$ and the real, desired output y (also known as a label). Going back to the initial example, y would be the actual position of the end-effector of the robot obtained by setting the joint angles of the robot to x; in this case, we have a supervised learning problem.

Structurally, artificial neural networks are formed by multiple layers in a pipeline, each of which applies a linear transformation to its input, followed by a non-linear function application. The simplest type of layer in use is the *linear* layer, also known as a *dense* layer. It can be specified as a transformation such that, given input x, prediction $\hat{y}$ is computed in the following way:

$$\hat{y} = \sigma(Wx + b) \tag{2.1}$$

where σ is known as the *activation function* (the non-linear function), W a weight matrix and b a bias vector. When we have multiple layers, we can simply apply these in a pipeline, that is:

$$\hat{y} = h_1(h_2(h_3(...(h_k(x)))) \tag{2.2}$$

$$h_i = \sigma_i(W_i x + b_i) \tag{2.3}$$

Figure 2.1 shows an example neural network with an input layer, a hidden layer (a layer that is not for input nor for output) and an output layer. For this example, the neural network takes a vector $x \in \mathbb{R}^3$ as an input and produces an output vector $\hat{y} \in \mathbb{R}^2$. A connection between two neurons n_i, n_j represents a weight in the matrix W, or the bias weights in the case of the $+1$ neuron. For a given neuron n_i, the output of the previous neuron is multiplied by the weight of the connection to n_i, summed along the other inputs to n_i. The activation function is applied, and a value is produced and supplied to the next layer. When all neurons are taken into account, and following our formulation, this can be summarised as matrix multiplication and addition in two layers h_1 and h_2 as in Equation 2.3.

The importance of activation functions relies on the fact that they introduce non-linearities into the network - effectively, this allows neural networks to approximate the real underlying function, even if it is highly non-linear. Without activation functions, neural networks would just be a sequence of linear transformations. A very common activation function is ReLU (rectified linear unit), specified as $ReLU(x) = \max(0, x)$.

The output layer is a special case for the activation function - for regression tasks as in the example, no activation function is used in the final layer. In other scenarios, such as classification, where we would want to classify the input into one of C classes, we would use

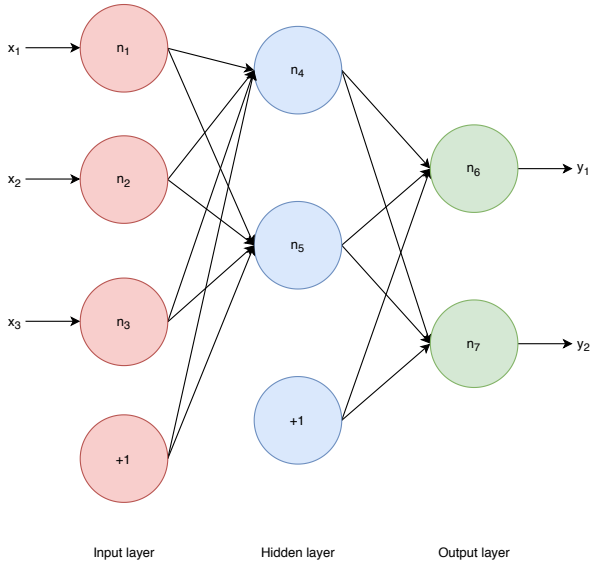


Figure 2.1: Artificial neural network with one hidden layer, 3 inputs and 2 outputs.

a *softmax activation* on the output of the final layer. The softmax function normalises its input and generates a pseudo-probability distribution as a vector with components $\sigma(x)_i$ from an input x with components x_i, and is given by:

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{C} e^{x_j}} \qquad \text{(Softmax)}$$

As aforementioned, the objective is to learn the weights θ (W, b for the above cases), by minimising a prediction error. Depending on the type of prediction we want to make, we may need to specify the mentioned error in different ways. This is done via a *loss function*. In a regression setting, where the objective is to predict a continuous variable (as in our example), we specify a squared error loss $\mathcal{L}_{SE} = (y - \hat{y})^2$ for a single sample (x, y), and can generalise for multiple samples by computing the mean squared error (MSE) $\mathcal{L}_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$. In a alternative setting, like a classification setting, where we would want to classify the input into one of C classes, we would use a different loss function (for instance, *categorical cross-entropy*). In general, depending on the type of problem we are solving we may need other loss functions.

To train the network, as hinted at in the paragraphs above, we employ the backpropagation algorithm. In a nutshell, the output of the network is evaluated for some input data x_i from a training dataset $D = \{(x_1, y_1), ..., (x_N, y_N)\}$. The loss function is computed with respect to the corresponding label y_i, and the gradient is propagated backwards through the layers. The weights of each layer can be updated using stochastic gradient descent, but nowadays more advanced optimisation algorithms are often applied, such as Adam [14]. In addition, in general batch operations are performed - instead of evaluating the network for some vector x, we evaluate it for the matrix X representing a batch of samples. Thus, the batch size becomes a *hyperparameter*. Other possible hyperparameters include the learning rate of the optimiser, the number of layers or the number of neurons in a hidden layer. For each prediction task, these often need to be tuned empirically via *hyperparameter search*.

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have prevailed in computer vision in recent years, and refer to neural networks that use a special type of layer called a *convolutional layer*. The main difference between convolutional layers and linear/dense layers is the type of operation performed on the input to the layer. In the convolutional case, the layer performs a convolution over the input (usually an image) to build a 2/3-dimensional feature map.

Figure 2.2 shows an example convolution operation with a kernel of size $3 \times 3 \times 1$, in which the weights in the kernel are multiplied by the corresponding pixels in the image, and the results are summed to produce a unique value. The operation is repeated for every pixel in the image, producing a 2-D feature map.

Extending this, a flattening operation can be applied to the output map and linear layers can then be utilised to produce a prediction. Likewise, for kernels with multiple channels (such as a $3 \times 3 \times 16$ kernel, 16 channels), the above operation is executed as many times as there are channels, generating instead a 3-D feature map with depth equal to the number of channels. In addition to the kernel size, other parameters of the convolution operation, such as *stride*, *padding* or *dilation* can also be specified.

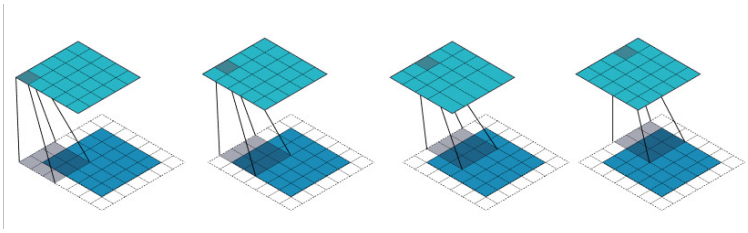


Figure 2.2: Convolution operation on an image with kernel size 3x3x1 [15].

We distinguish four main advantages of using convolutional layers over linear layers for visual inputs. Firstly, we typically specify a much smaller kernel than the input image. When compared to a linear layer, this implies that convolutional layers are much more sparse than linear layers, leading to faster computation and reduced memory usage.

Another big benefit of convolutional layers is *weight sharing*. The weights are the values stored in the convolutional kernel, from which feature maps are produced. Since the same set of weights is applied to all the pixels in the image, the networks learns to generalise much more easily than in the linear case. As a result, *overfitting* is prevented, a problem that occurs when a network or model finds a signal in the noise of training data, which prevents it from generalising successfully to new, unseen samples.

A greater degree of flexibility is also provided by convolutional layers. Unlike with linear layers, where the matrix operation depends on a fixed size, a convolutional layer can handle input of any size.

Last but not least, convolutional layers provide the property of equivariance to translation [13], allowing the network to generalise features learnt in one location of the image (for instance, edges or textures) across the image.

Typically, convolutional layers are used alongside pooling and batch normalisation lay-

ers [16]. In the former case, the pooling layer uses a kernel to pool and reduce the number of features (size of) in a feature map. This can be helpful when input images to a network are high-dimensional to reduce the number of parameters of the network, particularly those of linear layers after flattening of feature maps. In the latter, by adding batch normalisation layers before the activation of convolutional layers, we normalise the input to the activation function, leading to faster training. For more information on CNNs, see [13, Chapter 9].

2.1.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of networks that are well suited to data that is related across time. This may be musical, natural language, time-series data or a robot demonstration. Essentially, recurrent neural networks keep a *memory*, often known as the *hidden state*, denoted h_t for the hidden state at time step $t \in [0, T]$ in a sequence of length T. At each time step, RNNs apply a function to the input for that time step and their memory to obtain the next hidden state, i.e. $h_t = f(x_t, h_{t-1})$. There is no restriction on the sequence length T; RNNs can be trained with sequences of data of different lengths. However, it is known that their performance can degrade when tested on sequences of lengths larger than seen in training.

The first RNNs suffered heavily from the *vanishing gradient* problem. Due to their inherent recurrence, when computing gradients using the chain rule in the backpropagation algorithm, many multiplications between the gradients are performed. When the gradients are small (i.e. < 1), these multiplications cause the gradients to become vanishingly small. As a consequence, the neural network weights are not updated, and the network stops learning. In order to deal with this problem, a variety of solutions have been proposed. A notable one are Long Short-Term Memory networks, also known as LSTMs, introduced in [17].

2.1.4 Autoencoders

In the domain of unsupervised learning, an *autoencoder* performs dimensionality reduction by learning a low-dimensional representation of an input from which such input can be reconstructed. In that sense, it does not need any labelled data, hence it is an unsupervised method.

A simple formulation for an autoencoder consists of two networks: an encoder network $E(\cdot)$ and a decoder network $D(\cdot)$. These may be multi-layered feed-forward, convolutional or even recurrent networks - any type of differentiable model may be used. The encoder projects some input x to a lower dimensional space, called the latent space, i.e. $E(x) = z$, $z \in \mathbb{R}^{latent}$, where *latent* is the dimensionality of the latent space. The decoder D decodes z to obtain $D(z) = \tilde{x}$ and aims to reconstruct the input x. Therefore, typically a *reconstruction* loss is used, such as the loss $||x - \tilde{x}||_2^2$. Note that to prevent the encoder and decoder from learning the identity function, if $x \in \mathbb{R}^n$, $latent << n$. In this way, only the most important features of x are kept.

2.2 Reinforcement Learning

Reinforcement Learning (RL), a sub-field of machine learning, is a set of computational methods, techniques and algorithms that can be used to solve control problems. Overall,

these are problems in which there is a system (an agent) that needs to learn how to behave in an environment, by choosing the optimal action given the current state of the world, in order to achieve a specified goal or task.

For the learning process to work, the agent must be able to interact with such environment, or *act* on it. Upon the act of an agent, the state of the environment changes, and it emits a *reward* signal to the agent. Note that the only feedback the agent obtains from the environment is this signal and the state of the environment after the action of the agent has taken place. Consecutively, the agent learns from the reward/experience, and keeps interacting with the world, in a closed-loop fashion. Intuitively, the *immediate reward* obtained by the agent on every interaction with its surroundings guides the agent towards success in the task and provides a form of goal specification - the agent learns how to achieve the task by maximising the rewards obtained throughout its interaction with the world. Figure 2.3 shows an example of this configuration.

Examples of Reinforcement Learning problems can be:

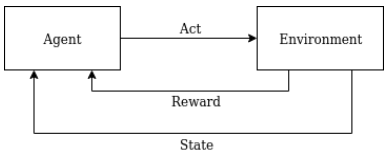


Figure 2.3: Example interaction of an RL agent with an environment

- Controlling a robotic arm to pour water into a glass, where the glass might be of different sizes, types, and be positioned at multiple locations. The agent, the robotic arm controller, could act on the environment by issuing commands to the robotic arm. This could potentially lead to a change in the position of the end-effector of the arm (part of the new state, holding a water bottle) and a reward for such action. These commands would be very difficult for a (non-learning) algorithm to generate, even more by considering the variability of the environment.

- Playing against and defeating masters at board games where the state space of game configurations is massive, rendering brute-force approaches impractical and poorly performing. A famous example of this is AlphaGo [18], an RL agent that learnt to play and win against the best human players at the game of Go (a game with an estimate of more than 10^{170} possible board configurations).

Recently, reinforcement learning has been under the spotlight due to the impressive advances of the work in [18, 19, 20, 21]. In the following sections, we will present some preliminaries on the theory of reinforcement learning along with tabular RL and its limitations. Furthermore, we will expand on the more practically applicable, state-of-the-art deep reinforcement learning algorithms. For a more in depth overview of reinforcement learning, however, we refer the reader to [22].

2.2.1 Tabular Reinforcement Learning

Reinforcement learning problems can be modelled as Markov Decision Processes (MDPs). A Markov Decision Process $\mathcal{M}$ is specified by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where $\mathcal{S}$ is the set of possible states the agent can be in, $\mathcal{A}$ is the set of actions the agent can take, $\mathcal{P}$ is the

probability transition function, $\mathcal{R}$ is the *immediate/instantaneous reward function* and γ is a quantity called the *discount*.

$\mathcal{P}$ models the environment dynamics: at time step t, $\mathcal{P}^{a_t}_{s,s_{t+1}}$ represents the probability $p(s_{t+1}|s = s_t, a = a_t)$ of the agent reaching some state s_{t+1} in the next time step from its current state s_t by taking action a_t. Similarly, $\mathcal{R}^{a_t}_{s_t,s_{t+1}}$ represents the instantaneous reward obtained by the agent for transitioning from state s_t to s_{t+1} by taking action a_t. Furthermore, we define an *episode* to be a sequence of transitions $(s_t, a_t, s_{t+1}, r_{t+1})$ that either continues infinitely or ends at some terminal state. These transitions are generated in the following way: at time step t, the agent is in state s_t, chooses action a_t and the successor state s_{t+1} is sampled from the environment dynamics. Upon arrival to state s_{t+1}, the agent receives the immediate reward r_{t+1}.

Overall we want to maximise the expected return from our current state. The return R_t, starting from time step t is defined as the total discounted reward obtained from time step t:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \tag{2.4}$$

Finally, as we can see from Equation 2.4, the discount $\gamma \in [0, 1]$ tunes the short-sightedness/far-sightedness of the agent with respect to rewards. With $\gamma \approx 0$, immediate rewards are emphasized; with $\gamma \approx 1$, long-term, delayed rewards are preferred. Moreover, the actual decision making process is performed by a *policy*, which we denote by $\pi(a|s)$, the conditional probability distribution for the agent taking action a at state s. A policy can also be deterministic, i.e. $\pi(s)$.

Value and Q functions

The RL agent needs a way to gauge how "good" a state is, or a particular state and action taken from that state, by following a specific policy π. To that end, we define the *value function* and the *Q function*, respectively:

$$V_\pi(s) = \mathbb{E}[R_t|S_t = s] \tag{2.5}$$
$$Q_\pi(s, a) = \mathbb{E}[R_t|S_t = s, A_t = a] \tag{2.6}$$

Given the previous description of an MDP, the value function for a given state s and policy π can be estimated by the (Bellman Equation):

$$V_\pi(s) = \sum_{a\in A} \pi(a|s) \sum_{s'\in S} \mathcal{P}^a_{ss'}(\mathcal{R}^a_{ss'} + \gamma V_\pi(s')) \qquad \text{(Bellman Equation)}$$

where $s' \in S$ are all the successor states (reachable) from the current state s. The relationship between the value function and the Q function is given by:

$$V_\pi(s) = \sum_{a\in A} \pi(a|s) Q_\pi(s, a) \tag{2.7}$$

RL algorithms have to keep track of how good states/state-action pairs are, since at every state we want to choose the action that leads to the highest expected return. As a consequence, we call *tabular RL* algorithms to the algorithms that maintain a table-like data

structure to keep track of the values of $Q_\pi(s, a)$. Once we have computed the optimal Q values, obtaining the optimal *greedy* deterministic policy is trivial - it can be obtained by, for a given state, choosing the action with maximum Q value i.e. $\pi^*(s) = \arg\max_a Q^*(s, a)$. Note that usually the Q function is stored, as it is more meaningful than the value function.

Note that we may not know $\mathcal{P}$ or $\mathcal{R}$, a case in which we cannot use the (Bellman Equation) and we need to sample the environment to be able to infer these. Monte-Carlo and TD-learning methods are some of these *model-free* value estimation methods. Within TD-learning methods, perhaps the most famous one is Q-learning [23], from which papers like [20] derive their work. Furthermore, we classify RL algorithms as either *on-policy* or *off-policy*. On-policy algorithms learn a policy π, and use this same policy to sample the environment. Off-policy algorithms, as opposed to on-policy, use a different policy to sample the environment (a behaviour policy) from the one learnt. Examples of on-policy algorithms and off-policy algorithms are SARSA and Q-learning, respectively.

2.2.2 Deep Reinforcement Learning

Tabular reinforcement learning suffers from two main issues:

- For real world tasks, such as controlling a robot, we often require continuous state and action spaces. We can discretise these spaces at high resolution so as to employ the tabular formulation of RL. This, however, increases the size of the table we need to maintain in memory to impractical sizes.

- Lack of generalisation across states and actions of tabular methods means that every state and action have to be visited/tried at least once in order to learn about them. This, in turn, implies that for large state-action spaces the learning process will take long amounts of time. It is also highly inefficient, as similar states and actions might have similar expected returns - tabular RL does not exploit this property.

The solution to these problems lies in the usage of *deep learning*. By employing neural networks to approximate the Q function, we avoid having to keep track of Q values for all states and actions. Instead, this is encoded in the parameters of the neural network, the number of which is orders of magnitude smaller. Similarly, generalisation is inherently provided by neural networks, hence solving the efficiency problem of tabular methods.

Deep Q Networks

Deep Q Networks (DQNs), in their most basic formulation, address the problem of continuous state spaces by employing a neural network that takes the state as an input, and predicts the Q value of each discrete action available to the agent. This formulation can be extended to continuous action spaces, but we do not address this in this section. The parameters of the neural network are optimised by minimising a squared error loss with respect to the tabular Q-learning formulation:

$$\theta \leftarrow \theta - \alpha \nabla_\theta (r + \gamma \max_a Q_\theta(S', a) - Q_\theta(S, A))^2 \tag{2.8}$$

where Q_θ represents the network, α is the learning rate and (S, A, S', r) is a sampled transition from the environment.

Building on top of DQNs, two interesting extensions called *experience replay buffer* and

target networks were used by [20, 21], and led to the unprecedented results reported in both papers. They are based on two realisations:

- Data collected by sampling the environment is heavily correlated, and leads to non-uniform training of the neural network. By storing sampled transitions in a buffer, and sampling batches of transitions from this buffer in a uniform way, the correlations between transitions are broken. Thus, the stability of the training of the network is enhanced.
- The max operation, applied to contiguous states repeatedly (as the agent moves between correlated states), may cause an overestimation of Q values. As a fix, the network used in $\gamma \max_a Q_\theta(S', a)$ is substituted by $\hat{Q}_\theta$, the target network. $\hat{Q}_\theta$ can be kept frozen for a number of iterations, after which the updated parameters from Q_θ are copied over. Alternatively, a moving average can be used.

It is important to highlight that more recent state-of-the-art algorithms, such as the ones we will describe in subsequent sections, still apply these extensions.

Policy Gradients

Policy gradients optimise the parameterised policy directly. They employ a neural network with input the state of the agent, and output the action the agent must perform, representing the policy directly. An example of policy gradient algorithms is the REINFORCE algorithm [24, 25], which uses gradient ascent with the following gradient to optimise the policy:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) R^{(i)} \qquad \text{(REINFORCE)}$$

$$R^{(i)} = \sum_{t=0}^{T} r(a_t^{(i)}, s_t^{(i)}) \qquad (2.9)$$

Using the current policy π_θ (initially randomly initialised, with parameters θ), N trajectories are collected. The policy gradient is computed by summing the gradient of the log probabilities of the actions given the states, over all steps in the trajectory, weighting that quantity by the sum of rewards of the trajectory, and finally averaging this quantity over all N collected trajectories. This process is then repeated in a loop.

One major drawback of policy gradients is that they have high variance [25, 26]. This can be effectively reduced by exploiting the temporal structure of trajectories - past rewards at time $t', t' < t$ should not affect an action a_t. Another variant of the policy gradient estimation method substracts the expected return (a *baseline*) from the obtained return of the trajectory, thereby reducing the variance. This quantity is often referred as the *advantage*. As a baseline, the value function at the current state in the trajectory, $V(s_t)$ can be chosen - in fact, any value independent of the actions can be used as an unbiased baseline [22, pg. 329].

Actor-Critic architectures

Actor-Critic methods distinguish themselves by their use of two elements, an *actor* and a *critic*. The actor represents the policy, and is often modelled by a neural network π_θ.

Typically, the actor is updated via policy gradient based methods. The critic, in contrast, provides an appraisal of the value of the action taken at the current state of the agent, following the current policy. This can be modelled as a different neural network, Q_θ or V_θ, and the parameters of the actor can be updated according to its evaluation. Note that often, since policy gradients can be used to update the actor, baseline-based (baseline as the critic) policy gradient methods are considered actor-critic methods, particularly when the baseline is learnt from experience or is state-dependent [27].

Overall, many of the current state-of-the-art algorithms use these methods and frameworks - algorithms such as Proximal Policy Optimisation (PPO, Schulman et al., 2017, [28]) or Soft Actor Critic (SAC, Haarnoja et al., 2018, [29]) are based on the aspects presented throughout the last sections.

2.2.3 Soft Actor Critic

Haarnoja et al. introduced the Soft Actor Critic in 2018 through [29]. SAC is a model-free, off-policy, actor-critic algorithm that provides stable and sample-efficient learning. The latter aspect is a negative aspect of on-policy algorithms - an off-policy structure allows for reuse of old collected data, for example by means of the previously introduced experience replay buffer.

Maximum Entropy Framework

An innovative aspect of the SAC algorithm is its basis on the maximum entropy framework. This framework was introduced in [30] by Ziebart, and includes entropy into the standard RL objective of maximising the expected sum of rewards. In an episodic setting, the aim then becomes to maximise:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)}[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t))] \tag{2.10}$$

where $\mathcal{H}$ is the entropy, and the expectation operator is applied over the trajectory distribution of policy π. Parameter α weights the contribution of the entropy to the objective, and is called the *temperature* parameter. This can be considered a hyper-parameter. The authors, however, show in the follow-up paper [31] a method to learn this parameter without a big impact in the performance of the algorithm.

The benefits of this formulation when compared to the original objective are threefold:

- The policy is encouraged towards high-reward regions of the state space, but also towards unexplored regions. Low-reward, explored regions of the state space will be ignored.
- Actions that have similar rewards will be assigned similar probabilities.
- Empirically, the paper shows that this formulation improves the training speed of the algorithm with respect to other state-of-the-art algorithms that do not use the maximum entropy framework. In fact, when compared on a dexterous hand manipulation task with the PPO algorithm, the authors in [31] reported 3 hours of learning time, 4.4 hours less than PPO.

For simplicity, we omitted the discount in the equation - such a formulation can be found in the appendix of the original paper, and would be required for *infinite-horizon* cases, where episodes are non-terminating.

Gradients and Algorithm

The algorithm is based on the concept of soft Q function, defined by the repeated application of an operator $\mathcal{T}^\pi$ to the Q function under a fixed policy π:

$$\mathcal{T}^\pi Q(s_t, a_t) \triangleq r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim \mathcal{P}}[V(s_{t+1})] \tag{2.11}$$

$V(s_{t+1})$ is called in this case the *soft value function*, and is defined by the following expression:

$$V(s_{t+1}) = \mathbb{E}_{a_t \sim \pi}[Q(s_t, a_t) - \alpha \log \pi(a_t|s_t)] \tag{2.12}$$

The algorithm employs a parameterised policy π_ϕ, usually represented by a neural network that predicts the mean and covariance of a Gaussian distribution. As a consequence of this representation, the algorithm provides naturally both a deterministic and a stochastic policy: if we wish to use a stochastic policy, we simply sample $a_t \sim \mathcal{N}(\mu, \Sigma)$, where $\pi_\phi(s_t) = (\mu, \Sigma)$. For deterministic policies, we can take $a_t = \mu$. We also have a neural network that represents the soft Q function, Q_θ. In the latter case, in a similar way to DQNs, we want to minimise the MSE loss with respect to equations (2.11) and (2.12). The gradient for the cost function $J_Q(\theta)$ can be estimated by:

$$\nabla_\theta J_Q(\theta) = \nabla_\theta \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}}[\frac{1}{2}(Q_\theta(s_t, a_t) - r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim \mathcal{P}}[\hat{V}_\theta(s_{t+1})])^2] \tag{2.13}$$

In the above equations, $\mathcal{D}$ is a replay buffer, and $\hat{Q}_\theta$ is the target network for Q_θ, as presented in section 2.2.2. Finally, the gradient for the policy can be updated using the loss:

$$\mathcal{L}_{\pi_\phi} = \alpha \log \pi_\phi(\tilde{a}_t|s_t) - Q_\theta(s_t, \tilde{a}_t) \tag{2.14}$$

where $\tilde{a}_t$ is a sample from $\pi_\phi(\cdot|s_t)$ obtained via the *reparameterisation trick*. The pseudocode of the algorithm is given in Figure 2.4, as provided by the authors. The gradient steps in the equations above are batched, with batches sampled from the replay buffer. For more detailed pseudocode, you may visit [32].

2.3 Meta-Learning

Meta-Learning, or *learning to learn*, is a paradigm that has recently gained traction throughout the deep learning research community. In traditional learning methods, one may train a neural network to achieve a particular task. In meta-learning, in contrast, the aim is to improve the training speed, performance or generalisation of these models across a variety of tasks. As the name suggests, through meta-learning we aim to learn how to augment the learning capabilities of models. Therefore, meta-learning procedures are often executed through multiple learning episodes.

A classic example of a problem where meta-learning is applicable is N-way k-shot classification. To solve this problem, we aim to obtain a trained model that is able to perform

Algorithm 1 Soft Actor-Critic

Input: θ_1, θ_2, ϕ ▷ Initial parameters
$\theta_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$ ▷ Initialize target network weights
$\mathcal{D} \leftarrow \emptyset$ ▷ Initialize an empty replay pool
for each iteration **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$ ▷ Sample action from the policy
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ ▷ Sample transition from the environment
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$ ▷ Store the transition in the replay pool
  end for
  for each gradient step **do**
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$ ▷ Update the Q-function parameters
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ ▷ Update policy weights
    $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ ▷ Adjust temperature
    $\bar{\theta}_i \leftarrow \tau\theta_i + (1-\tau)\bar{\theta}_i$ for $i \in \{1, 2\}$ ▷ Update target network weights
  end for
end for
Output: θ_1, θ_2, ϕ ▷ Optimized parameters

Figure 2.4: Soft Actor Critic (SAC) algorithm [31]

well on a test set given a training dataset with k samples of each of N classes. The test set is restricted to those N classes. For large k, this is the standard classification problem; nowadays, neural networks have surpassed humans on image classification challenges such as ImageNet [33]. However, when k is small, or even $k = 1$, traditional learning algorithms struggle on these *few-shot* classification problems.

Meta-learning can solve this problem by considering multiple few-shot learning episodes. These form the *meta-training* set, composed of multiple (D^{train}, D^{test}) few-shot tasks. By learning how to learn effectively on the meta-training dataset, when presented with new, unseen few-shot tasks of the form $(D^{train}_{new}, D^{test}_{new})$ learning performance is increased. These testing tasks form the *meta-testing* dataset.

While meta-learning can be found in multiple forms, we differentiate between three main types[1]:

- Metric-based: This type of meta-learning algorithms learn a metric/embedding space from learning episodes that allows performance to generalise to new, unseen tasks. An example of this type of method are Prototypical Networks [35], where each class is embedded into a prototypical space. Classification of a test sample can be computed by using its distance to each prototypical space in an approach akin to K-nearest neighbours.

- Model-based: Using the meta-training dataset, model-based methods predict the parameters θ that when employed as the weights for another model on D^{test} give good performance. Different approaches vary the parameter prediction network, as well as the complexity/type of the model whose parameters are being predicted.

- Optimisation-based: Also known as gradient-based, these approaches differ from model-based ones in that, rather than using a model to predict the parameters, an optimisation procedure such as stochastic gradient descent is used directly on these

[1]For a more in-depth review, see [34].

parameters. It is often framed as a bi-level optimisation problem, where in an *inner loop*, parameters are optimised on the training data D^{train} from the meta-training set. At the same time, an *outer loop* optimisation step uses D^{test} to ensure the parameters learnt allow for generalisation.

An example of a well-known gradient-based meta-learning algorithm is Model-Agnostic Meta-Learning, or MAML for short [36]. It is model-agnostic in the sense that it can be readily applied to supervised or even reinforcement learning settings. MAML learns some parameters θ for a model f_θ using a distribution of meta-training tasks. When learning an unseen task, the goal is that these parameters adapt to the new task as quickly as possible, while giving the best possible performance. In their setting, they consider an inner loop where they sample repeatedly meta-training tasks $(D_i^{train}, D_i^{test})$, and compute the parameters θ_i' in the following way:

$$\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}(f_\theta) \tag{2.15}$$

where α is a step size hyperparameter, and the gradient is computed using the loss on D_i^{train}. Then, the *outer loop* optimisation, or *meta-update*, updates the θ parameters as follows:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_i \mathcal{L}(f_{\theta_i'}) \tag{2.16}$$

with β is another step size hyperparameter, and the gradient in this case is computed using the loss on D_i^{test} for each i sampled in the inner loop, with respective parameters θ_i'. Note that this optimisation involves a double gradient, as we compute ∇_θ over the function f_θ', with θ' containing itself a gradient (Equation 2.15). For a more comprehensive overview of the algorithm, see the original paper.

Chapter 3

Related Work

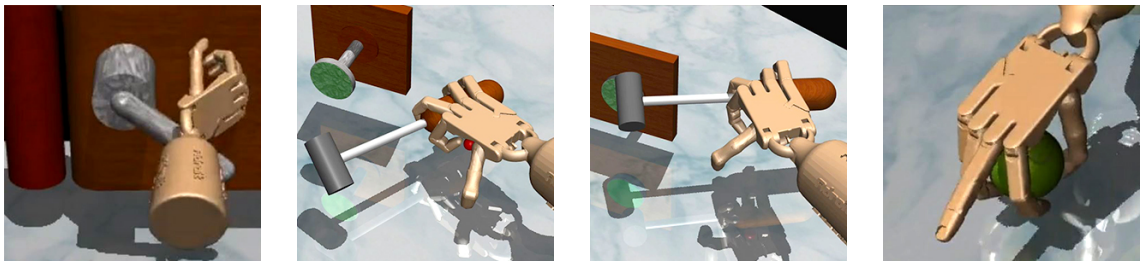
In the following chapter we introduce some previous work linked to our project. Specifically, we motivate and survey the use of demonstrations with supervised and reinforcement learning. In addition, we give a brief introduction to attention layers and their use in the literature, with a focus on robotic applications.

3.1 Imitation Learning

Whether on-policy or off-policy, model-free reinforcement learning algorithms are very sample inefficient. These algorithms start their learning process with no knowledge whatsoever about their environment, and they must explore this environment in order to find the optimal policy. For robotic tasks, for example, this translates into unrealistic, impractical training times, policies that lack robustness to variability in the environment and behaviours that are hardly human-like [3]. Figure 3.1 shows example unnatural behaviours learnt by a robot that are generally preferred to be avoided.

Furthermore, a lot of effort is required to encourage the RL agent into regions of the state-action space with higher rewards, performed by carefully engineering the reward function, and the addition of demonstrations can mitigate this [7].

In general, *imitation learning* addresses these limitations of RL by learning the optimal policy from expert demonstrations. These are often provided by a human through physical/virtual operation of the agent [5], but can also be generated without human intervention, directly in simulation [9].



(a) (b) (b) (c)

Figure 3.1: Example unnatural behaviours learnt by a dexterous robotic hand in different tasks: (a) opening a door, (b) using a hammer to hit a nail, (c) picking up a ball [3].

3.1.1 Behaviour Cloning and Distribution Drift

Imitation learning can be expressed in its simplest form as a *behaviour cloning* problem. The aim is to copy the behaviour of an expert at some specific task, where the behaviour is specified by demonstrations, as mentioned previously. This can be solved by optimising the following MSE loss:

$$\mathcal{L}_{BC} = \frac{1}{2}(\pi_\theta(s_{E_t}) - a_{E_t})^2 \qquad \text{(BC Loss)}$$

where s_{E_t}, a_{E_t} are the expert action and the state at which such expert action was taken, at time step t. π_θ is the parameterised policy that we want to optimise with respect to the expert actions. s_{E_t}, a_{E_t} are obtained from a dataset $\mathcal{D}_E$ containing expert trajectories of the form $\{(s_{E_0}, a_{E_0}, s_{E_1}, a_{E_1}, ..., s_{E_T}, a_{E_T})\}$. An expert reward r_{E_t} may also be required for every (s_{E_t}, a_{E_t}) pair.

The main problem found while training uniquely following a behaviour cloning approach is *distribution drift* [37]. Our model, even though it has been trained with expert demonstrations, will perform slightly inaccurate predictions. The accumulation of these errors throughout a trajectory of length T means there is a different distribution over the states seen by the model when compared to the distribution over the "expert" states the model was trained on. This drift impacts negatively the performance of our model at test time, and often leads to failure to achieve the task the model was trained on.

In the upcoming sections, we will present some common features of different recently proposed ways of combining imitation learning, including behavioural cloning, and supervised/reinforcement learning algorithms. Overall, these achieve superior performance (combating distribution drift) and training speed in control tasks, including robotic ones.

3.1.2 RL/IL and Supervised Approaches

We call *RL/IL approaches* those that combine RL with some form of imitation learning loss, and incorporate expert knowledge into the optimisation process of the deep RL agent.

Overall, these approaches use the expert demonstration as a way of guiding an RL agent into a region of the state space close to the optimal policy, and fine-tune the obtained policy by incorporating sampled transitions from the environment. Supervised approaches do not employ reinforcement learning, relying on complex loss functions and *plenty* of demonstration data to obtain good performance.

In general, for RL/IL algorithms we notice that papers on the topic seem to include some or most of the following features:

- **Pre-training the RL agent on expert demonstrations before rolling out the policy**. Hester et al, 2017, [8] demonstrate faster initial learning for their combination of DQN with demonstrations (Deep Q learning from Demonstrations, DQfD) in Atari benchmarks with discrete actions. Goecker et al, 2019, [11] show that including a pre-training stage in a quadrotor control task improves the initial training speed. They report better results with their framework than with DDPGfD [7] in two control tasks. This is an extended version of DDPG [38] for demonstrations, which as opposed to DQfD can handle continuous actions. Nevertheless, a pre-training stage need not improve the long-horizon performance of the agent as shown in [11], and other approaches show improved performance without this step [9].

- **Demonstrations are stored in a replay buffer, kept and sampled throughout training**. DQfD uses *prioritised experience replay*, a non-uniform version of experience replay which assigns higher probability of being sampled to transitions with a larger TD-error. Using this method, DQfD samples from a single replay buffer containing both expert and environment transitions. In contrast, Goecker et al. keep a constant ratio (25%/75%) of transitions sampled from two separate buffers $\mathcal{D}_E, \mathcal{D}_R$. These contain the expert and environment transitions respectively. Nair et al., 2018, [10] also use two separate buffers, combining DDPG with *hindsight experience replay* (HER) for $\mathcal{D}_R$. This is a particular type of replay buffer that aims to incorporate transitions leading to the final state of a failed sample episode as if reaching that state was the original goal. Similarly to [11], a constant ratio is used to sample from both buffers (128/1024 samples).

- **A behavioural cloning loss usually integrated into the RL algorithm's Q/actor network optimisation process**. Hester et al. use for DQfD a *large margin classification loss* on the Q network so that the values of unseen actions in demonstration states are grounded during pre-training. This ensures expert actions have higher Q values. The researchers in [10] note however that this may prevent the agent from finding a better policy than the expert-provided one. A BC loss is used both by [11, 10], although the latter assumes that demonstrations may not be perfect. Their algorithm uses an actor-critic architecture similar to DDPG, and counteracts this by employing a modified BC loss. This loss is only applied when the critic network determines that the expert action is better than the actor policy's. Finally, Rajeswaran et al., 2018, [3] enhance NPG (Natural Policy Gradients), an extension of the (REINFORCE) rule, giving raise to Demonstration Augmented Policy Gradients (DAPG). In their formulation, the NPG gradient is generalised to include a weighted BC term that encourages actions to be close to the demonstration ones. They also introduce a decay on this weight to avoid continuously forcing the agent towards the demonstration policy.

- **Including the IL loss into the reward function**. Zhu et al., 2018, [9] modify the reward of the RL agent to include an IL loss/reward, as well as the task reward. They express the reward function as $\lambda r_{IL}(s_t, a_t) + (1-\lambda) r_{task}(s_t, a_t)$ where r_{IL}, r_{task} are the reward functions for the imitation process and the task, respectively, and $\lambda \in [0, 1]$ is a hyperparameter. In their case, they use GAIL (Generative Adversarial Imitation Learning) for the IL process, and use PPO as the reinforcement learning algorithm.

An example of the supervised case is given by Zhang et al., 2018, [5]. In their setting, they train a robot to perform single and multi-stage tasks uniquely with demonstrations and supervised learning. They use the (BC Loss), as well as an alignment loss for velocities and L2 regularisation for the neural network weights. More importantly, they add the loss of auxiliary tasks, such as predicting the gripper pose (stored in the demonstration dataset) after an action. This is shown to have a very big impact in the performance of the agent, and is also applied by [9].

3.2 Attention Models

It is well known that, while observing their environment, human beings visually identify, focus and attend to specific visual features [39]. Attention to visual stimuli can be understood in a cognitive setting as dependent on goals or expectations, but can also be due to

simpler, salient visual stimulation. In these cases, we distinguish *top-down* and *bottom-up* attention, respectively, the combination of which is believed to form the attention mechanism humans have developed.

In recent years, visual attention has been leveraged in multiple ways across the computer science research community, leading to improvements in areas such as image captioning, object recognition or robot learning. The upcoming sections summarise some of the main contributions and give an overview of the techniques applied to simulate attention across diverse models.

3.2.1 Attention and Natural Language Processing

The application of attention has proven very successful in the field of *natural language processing* (NLP), particularly in areas such as image captioning (generating a sentence summarising the content of an image) or visual question answering (VQA, answering a question by using information from a supplied image).

In order to take advantage of visual input for NLP tasks, images have to be suitably encoded. For image captioning, Xu et al., 2015, [12] extract visual features from an image using the fourth layer of a pre-trained VGG image classification network. In a similar captioning task, You et al., 2016, [40] realised that depending on how deep of a CNN layer is utilised to extract visual features, we might extract either solely high-level (top-down, deeper layers) or low-level (bottom-up, shallow layers) features from an image. Combining a CNN layer with other visual attributes extracted from the image, such as words related to concepts in the image, allowed the authors to integrate both top-down and bottom-up features. For VQA-based challenges, the question is also encoded in addition to the visual input, and both encodings are used throughout [41, 42]. Overall, note that visual features tend to be combined with the encoding produced by an RNN.

Attention is commonly understood by these NLP methods as finding the probabilities α_i that can then be used to weight the contribution of each feature i, whether that be visual or language-oriented. As an example, the researchers in [12] divide an image into regions and assign attention probability α_i referring to the importance of location i in the image (out of a limited number of locations) with respect to the visual features obtained.

Different types of attention are common. The authors of [12] distinguish two types of attention: *soft attention* and *hard attention*. For a visual depiction of the differences between these two types of attentions, see Figure 1.2.

In their case, with soft attention, a softmax operation (Softmax) is used in combination with the state of an RNN and backpropagation to learn attention probabilities. These attention probabilities are then used to weight the different regions of the image, and a *context vector* is obtained from this weighted average. Hard attention, in contrast, is employed by the researchers to highlight a single image region, ignoring the remaining other regions of the image. Note that with soft attention every region of the image will have a non-zero weight, and therefore irrelevant regions will not be completely ignored.

Although the authors show that hard attention can obtain better performance, it is also harder to optimise. In particular, hard attention is not a differentiable operation in the

form presented in the paper[1], and therefore the researchers use a procedure equivalent to (REINFORCE) to optimise the attention mechanism.

You et al., use *input and output soft attention* to focus the input and output to an image caption generating RNN. A softmax operation on the output of a bilinear learnt function is used to estimate *input attention*. This type of attention is employed to compute the weight of the different visual attributes and features fed into the RNN. When generating a probability map which specifies the word most likely to appear in the next time step for the caption, *output attention* is also applied. This is performed using the bilinear-softmax approach of the input attention, but in combination with the state of the RNN and the visual attributes of the image, with the objective to help the probability generation stage.

Yang et al., 2015, [41] performed the initial investigation into employing attention for VQA, and used multiple stacked attention layers to focus the search for answers to questions based on images. Visual and question-derived features are then combined via a single-layer neural network and softmax is applied to the output to generate the attention map. In a similar spirit to the aforementioned papers, the attention is used to produce a weighted combination of the visual features of the image, and these are added to the question features. For this particular application, however, the authors note that the focused attention may emphasize relevant regions of the image, but this may not be enough to model more complicated relationships often needed to answer image-based questions. To that end, they feed the combination of attention and visual features through multiple attention layers, eventually applying more weight on the elements of the image leading to the answer to the question.

Other papers on the area such as [42] use two separate spatial and semantic soft attention pipelines, combining the produced attention to surpass the performance of the method of Yang et al.

In any case, utilisation of attention provides a major advantage: by visualising the attention map, intuition can be gained regarding which aspects of the visual input the model was concentrating on when generating specific words. As an example, the usefulness of attention-based visualisation is highlighted when the model fails to produce a good caption for an image, as we can often directly "see" why the model made such a mistake. An illustration of this benefit (with soft attention) is given in Figure 3.2. In the figure, attention allows us to see what the captioning system was concentrating on at every time step, in addition to realising that the model confused the two giraffes with a bird, generating a mistaken caption.

3.2.2 Combining Attention with Reinforcement Learning

We now explore applications of attention as used by reinforcement learning agents, where the attention map is learnt through the reinforcement learning process.

Already introduced in Section 2.2.2, DQNs offer a solid framework easily extendable with attention. In particular, Mousavi et al., 2016, [43] learn and apply the soft attention mechanism introduced in the previous section (from Xu et al. [12]) to Atari benchmarks. As

[1] More recent papers have proposed ways of making this a differentiable operation via (sub)differentiable mechanisms or the reparameterisation trick, see Sections 8.2.1 and 10.1.2

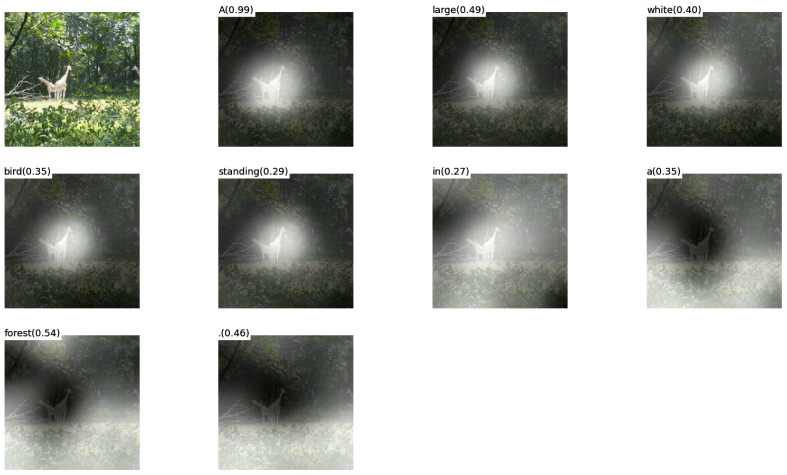


Figure 3.2: Soft attention applied on an image with two giraffes across time in order to produce a caption (mistaken in this case) [12].

the underlying DQN, they integrate a CNN to handle the visual input and utilise the state of an RNN and a softmax activation to estimate attention probabilities. As before, these are then used as weighting terms for the visual features, eventually fed to several linear layers to predict the Q-values of actions. Although the researchers show improvement when comparing the generated attention to the regions of the game interface humans attend to, they do not provide a comparison with standard deep Q-learning. Indeed, no insight is provided as to whether attention helped the model perform better than default Q-learning.

More human-like attention methods are devised by Mnih et al, 2014, [44], by simulating the property of human attention whereby only a focused region in an image is attended to at a time. As we move away from the center of focus, resolution is traded for capacity - the human vision system dedicates higher resolution to these centers, and blurs (lower resolution) the wider visual information on its surroundings. This "glimpse"-like attention is computed by taking images with multiple resolutions at a certain location. Both the location itself and the glimpse are fed to several linear layers to produce a suitable encoding. Figure 3.3 shows an example glimpse taken on training data in the MNIST handwritten digit classification dataset.

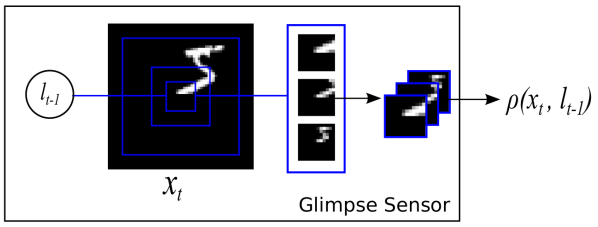


Figure 3.3: Glimpse with three different resolutions/scales taken of a handwritten digit from the MNIST dataset [44].

Another aspect inspired by human attention is its sequential nature. As Mnih et al. point

out, humans attend to features in a sequential fashion, one after the other. In their system, this is mimicked by an RNN which takes an initial glimpse, and produces an action and a location where the next glimpse should be taken at. The process is repeated in a loop for a fixed number of glimpses, where such number can be learned. The whole system is trained using (REINFORCE) in addition to a baseline for variance reduction.

An advantage of Mnih et al.'s method is concerned with the lower dimensionality of the input to the network. Unlike [43] where a weighted combination of visual features extracted from the entire, high dimensional input is supplied to the agent, in [44] partial, lower dimensional observations (*foveated representations*) are experienced by the agent at every time step, improving the efficiency of the method. In fact, the trade-off between taking more glimpses and the efficiency of the method can be automatically balanced by learning how many glimpses to take, in conjunction with adding negative rewards per additional glimpse.

3.2.3 Visual Attention in Robotics

The field of robotics has also benefitted from the inclusion of attention, although the interpretation of what constitutes attention varies with respect to the previous two sections.

In Finn et al., 2015, [6], an autoencoder is used to extract a low dimensional representation of images taken by the camera of a PR2 robot, keeping only the features that better represent the image. These features are then supplied to an RL agent instead of the full, high-dimensional image.

In this case, learning where to attend is understood as an unsupervised learning problem - only the most important features that summarise the image should be focused on. Moreover, the features obtained in their approach are computed through a *spatial softmax layer*: instead of storing the feature values, they represent images by the estimated pixel location of these features, where the expectation is computed using the softmax output. The intuition behind this approach is that the most representative features will be the locations of objects a robot has to interact with in a scene.

Figure 3.4 shows how the spatial features are computed. Note that a CNN is used to extract a 3-d feature map from the image. From each channel, one spatial feature is always extracted, after the softmax operation. This implies that this approach cannot handle scenes where there are no objects (no spatial features), or scenes where the same object appears in multiple areas of the scene (multiple spatial features on the same channel).

Devin et al, 2017, [4] apply a similar object-centric approach, but distinguish *meta-attention*, attention that can be applied to any task, from *task-specific attention*. Meta-attention is designed to be object-centric - a region proposal network proposes crops (location features) of objects in the image, and visual features for these are obtained from a pre-trained image classification network.

For each task, task-specific attention is trained from demonstrations by predicting the trajectory of the demonstration. Taking the visual features $f(o_i)$ from each object crop o_i (i-th object) of the meta-attention network, attention weights w are trained to produce an attention map with attention to object i being proportional to $e^{w^T f(o_i)}$. By grouping w into a matrix W, this formulation can be extended to attend to multiple objects. The

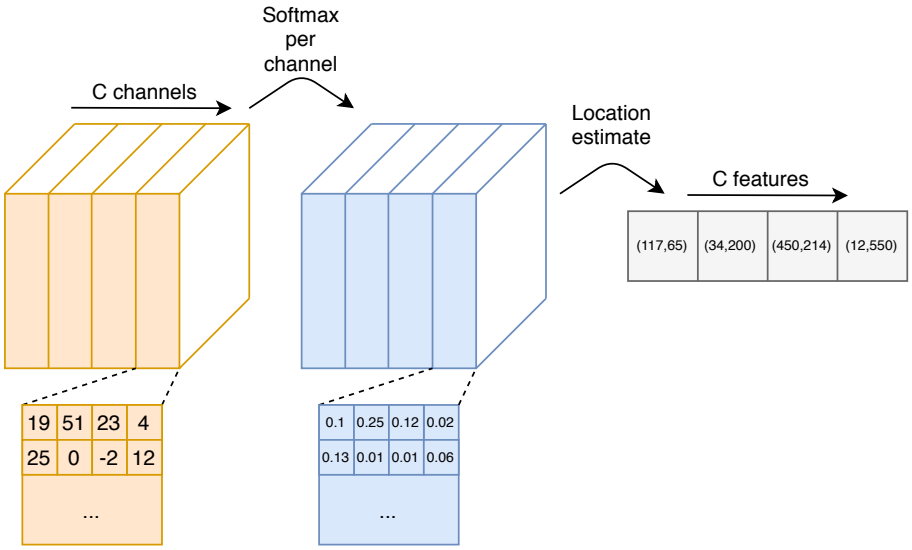


Figure 3.4: Spatial softmax operation performed on a 3-d feature map to produce location features.

attention map is then employed in a soft attention fashion to weight the contribution of the location features of each object, producing a bounding box for where the robot should focus on. With this soft estimate of bounding box coordinates, and the state of a PR2 robot (end-effector positions and joint/end-effector angles and velocities) trajectories are predicted in order to train the attention weights.

Once these weights are known, an RL agent is trained to produce a policy. At this stage, however, instead of obtaining an estimate location as in soft attention, the location with maximum attention probability is attended to. Concatenated with the state of the robot, they constitute the input to the RL agent. Figure 3.5 shows this process.

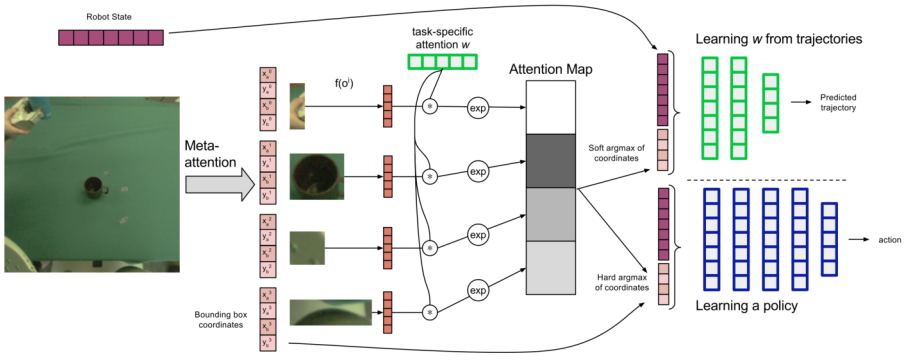


Figure 3.5: Usage of meta attention and learnable task-specific attention with an RL agent to train a robot with demonstrations [4].

It is worth highlighting that while both this method and [6] have the drawback of requiring pre-training with external datasets such as ImageNet, an extra object detector network needs to be trained for the former, and the approach is not end-to-end. Nonetheless, Devin et al.'s method allows for easier generalisation across scenes, object classes and instances as it learns to distinguish which types and instances of objects are useful to attend to for specific tasks. The autoencoder-based attention system only looks at the features that best

summarise the image, which may vary depending on the setup of the scene and training data, restricting its generalisation, especially when large amounts of training data are not available.

Following other lines of research, Abolghasemi et al., 2018, [45] learn attention via a natural language task specification ("pick up the box", for instance) and the visual input to a robot. Their approach uses a *generative adversarial network* (GAN), a neural network that learns to generate realistic images, in combination with a *variational autoencoder* (VAE). In essence, they apply an attention mask to the input frame based on the encoded task specification, and using the output frame, they attempt to regenerate the task specification, learning the attention weights in the process. These are then used on visual features at test time, fed to an RNN that outputs the joint angles for the robot to progress in the task.

Chapter 4

Setting the Scene

In this section, we introduce the main task we will be training neural network-based controllers to achieve. We then give an overview of the experimental setup. Finally, we showcase the datasets we generated and worked with during this project.

4.1 Main Task

Throughout this thesis, we will consider the following task: we have some object (like a cube) on a surface, such as a table, and we want a robotic arm to reach such object. The arm is also located on the surface, and has a camera attached to its wrist, just before the end effector, pointing towards the object. In order to reach the target object, the robotic arm's controller will need to process the input image from the camera, and rotate each of the arm's joints with a certain velocity, for a fixed amount of time. This process is repeated until the end-effector of the robotic arm reaches the object. Note that from now on, we will refer to the object the arm is trying to reach as the *target object*.

An example setup and trajectory can be seen in Figure 1.1, where the target object is the red cube on the table. For this particular case, the controller fails to achieve the desired trajectory due to the previously described *distribution drift* problem.

We further assume we may want to reach the target object by setting the arm's end effector at different positions and orientations, as long as the target object is visible from the wrist camera. Moreover, we impose no restrictions on the DoF (*degrees of freedom*) of the robotic arm. For this task, the controller will process RGB data and will output the Cartesian velocity and rotation to be applied to the tip of the end-effector. Given these, the *inverse kinematics* module of the robotic arm can be used to output the joint velocities required to move it.

Given a dataset with demonstrations for any such scene, a neural network-based robotic arm controller for this type of scene can be trained via supervised learning, using a behavioural cloning (BC Loss) approach. In its simplest form, the required dataset $\mathcal{D} = \{d_1, d_2, ..., d_N\}$ will be composed of demonstrations d_i, where $i \in [1, N]$. Each demonstration may be of different length, and will *at least* contain image-action pairs. These will represent the input image to the controller at time step t in the trajectory, and the action the robot must execute at that same step. Thus, $d_i = \{(I_t^i, a_t^i)\}$, for $t \in [1, T^i]$, with $I_t^i \in \mathbb{R}^{H \times W \times 3}$ being the input RGB image and T^i representing the length of demonstration i. We also define the action $a_t^i = [v_t^i, r_t^i]^T$ as an $\mathbb{R}^6$ vector containing the tip velocity v_t^i

and rotation component r_t^i, both in $\mathbb{R}^3$.

4.2 Experimental Setup

But how can we generate datasets similar to the above? A valid option would be to use a real-life robotic arm, setting up a physical scene, with a real target object. We could then manually move the arm towards the target object, recording in the process the images taken from the wrist camera, as well as the tip velocity and rotation components. This is, however, very time-consuming and would prevent quick experimentation.

Therefore, we perform most of our experiments directly in simulation (Section 4.2.1), where we also generate the datasets we train our controllers on. While it is possible to transfer controllers trained in simulation to the real world, this is outside of the scope of this project. Rather, we aim to explore approaches that are easily reproducible in a real-life setting, for which obtaining a dataset through manual control is feasible and practical.

4.2.1 V-REP/CoppeliaSim and PyRep

We choose CoppeliaSim (formerly named V-REP) [46] as our simulator of choice. It is relatively simple to create a scene on, and includes robotic arm models with it. These can be set up on the scene by a simple drag-and-drop operation. With regards to the simulation API, it provides a C/C++ API, although Lua scripts may be embedded to control every step of the simulation.

We choose Python as our programming language for this project due to its prevalence in the DL research community. We decided to use PyRep's Python API for V-REP. PyRep [47] is a project providing ease-of-use through a clean Python API for CoppeliaSim and faster communication with the simulator. A lot of the boilerplate that we would have had to write if we had used the original APIs was prevented by using PyRep: this allowed us to be more productive during the project.

4.2.2 PyTorch

Since we need to train neural network-based models, we require an automatic differentiation framework. We selected PyTorch for this project. PyTorch [48] is a Python framework for gradient computation that allows low-level operation control on tensors. An important advantage when compared to a similar, yet perhaps more established framework such as TensorFlow [49] is the way gradients are computed and stored. In PyTorch, gradients flow through a dynamic graph, whereas in TensorFlow a static graph has to be created during model initialisation. This makes it easier to debug your models and test the result of operations while development is ongoing. Another heavily-used Python framework for deep learning is Keras [50], although it has been designed to abstract away the internal tensor operations, and has a much higher level API. Since our goal is to conduct research, we require the flexibility of a low-level API, while keeping the boilerplate to a minimum to enable effective progress; PyTorch strikes the right balance.

4.3 Datasets

In this section, we give details regarding the different datasets we employed throughout the project, and how we generated them.

4.3.1 Dataset Generation

As discussed earlier, we generate datasets to train our controllers in simulation. To do so, we set up several scenes on CoppeliaSim. Each contains a target object, which may be a cube, a sphere or a similar primitive, sitting on a textured table. We also load a different texture on each target object; the texture is asymmetric such that when viewed from a rotated camera, a controller may be able to infer the difference between its current pose and the target object pose. In addition, we enable shadows in the scene. Finally, we add *distractor objects*: these are a fixed number of objects per scene that are set up on the table. Their shape, size, colour, position and orientation may vary between demonstrations: we generate datasets both without and with distractors. Unlike these distractors, the target object is kept at the same position across demonstrations.

An example can be seen in Figure 4.3a, where the first image of a demonstration is shown. The center cube, which has an asymmetric texture is the target object, while the other three objects are the distractor objects.

Early in the project, we set up a Sawyer robot on the table, and controlled it while recording the images obtained from a camera in order to move the end-effector towards the target cube. This involved using the inverse kinematics module, as we mentioned earlier, and we found that generating varied, high-quality demonstrations was not trivial. Therefore, we decided to simplify the task and remove the robot from the scene, leaving only the camera remaining. Since our approach is only vision-based, and relies on the velocity and rotation of the tip of the arm, it would remain valid. By only controlling a camera, the demonstration generation process became much easier and faster.

We show the overall demonstration generation process in Figure 4.1. In every scene, to simulate the end effector reaching the target object, we set up a point 5cm above it as the target point (green ⊗ in the figure). When the camera reaches this point, we consider the task to have been achieved.

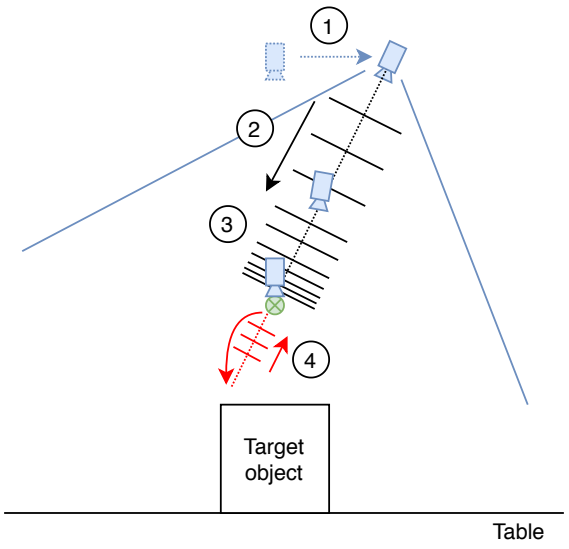


Figure 4.1: Demonstration generation for `Scenes 1-5`.

In order to generate a varied set of demonstrations, we randomly sample position/rotational offsets. At the beginning of each demonstration, we apply them to the initial position/pose of the camera, which is set above the target cube, looking down, towards it. This is shown in the first step of Figure 4.1, where we visualise applying an offset to the position, although generally we also apply orientation offsets. This allows us to generate diverse training images which contain the target object in different parts of the image, with different poses with respect to the camera. Some examples of initial images in generated demonstrations are shown in Figure 4.2.

The velocity and orientation components are generated taking into account the distance from the camera to the target point. In particular, velocity is obtained by computing the direction of the vector joining the current camera and target positions in the world frame. The rotational component is computed from the difference between the camera's current world frame pose and the target world frame pose, which is also set to look down towards the object.

At the beginning of the trajectory (step 2 in Figure 4.1), the magnitude of the velocity vector is fixed and unit-normalised. The orientation difference vector is normalised by the distance to the target point. To obtain the next image, we multiply the velocity and orientation components by the length of each simulation step (50ms) to obtain the position and orientation change. We add these quantities to the position and current pose of the camera. Once the camera reaches a point within a small threshold of the target point, we set the velocity and rotation components to zero, as we want our controller to naturally stop when it reaches the target.

Each dataset also has associated a set of 100 *test trajectories*. These are specified as an initial offset for the position and the orientation of the camera, alongside descriptors (including size, position, and so on) for the distractor objects. These trajectories are generated independently from the original dataset, and we use these to estimate the performance of our controllers throughout. Moreover, we generate trajectories with and without random distractors. In the former case, even if distractors are generated randomly, at test time they are kept fixed for comparability.

In our initial experiments, we trained controllers with the above setup. We evaluated them on the mentioned test trajectories, and found that they were unable to stop once they reached the target point. Indeed, there is an important discontinuity at the target point, where the predictions must shift from non-zero velocities/rotations to zero velocities/rotations. It is hard for a controller to learn that behaviour, particularly due to the accumulation of errors that will happen along a trajectory, but also when sufficient data to model the discontinuity is not available.

We therefore aimed to achieve a "hover" type of behaviour for the controller, where, once reached the target point, the controller stays as close to the target point as possible. To do so, we needed to provide more data around the target point discontinuity. For that reason, during the demonstration generation process, after the controller reaches a certain distance to the target point, we start slowing it, as shown in step 3 of Figure 4.1. This helps provide more data around the discontinuity, and improves the interpolation capability of our models.

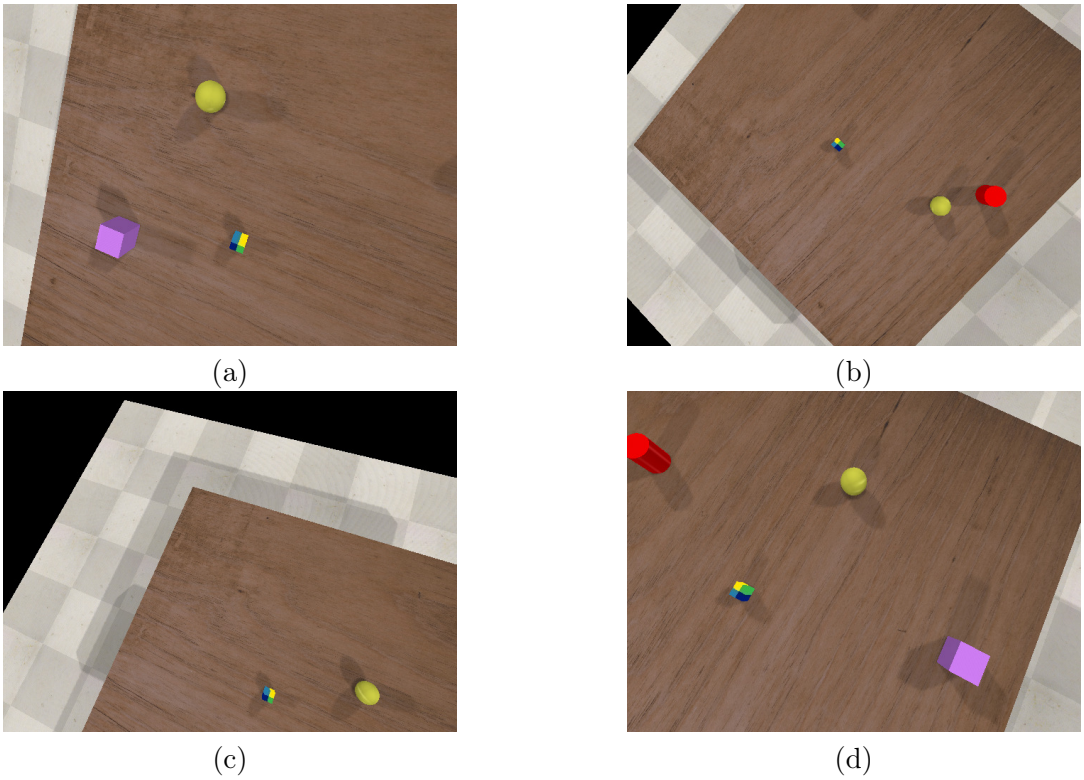


Figure 4.2: Initial images for several demonstrations in dataset `Scene 1`. In this particular instance, the same three distractor objects are used across demonstrations, but they are positioned at different, random locations in each demonstration.

In addition, for each demonstration we add data *past* the discontinuity: we randomly sample a point between the target point and the target object, and move the camera backwards to its original target, visualised in step 4 of the figure. Including this data allows the controller to learn to backtrack towards the target point, even when it has gone beyond it due to prediction errors.

Alongside the image and the action data, we also store other important information in the dataset. Specifically, we keep track of the relative position and orientation of the target object with respect to the camera frame. Furthermore, we record the pixel obtained from the projection of the center of the target object onto camera space. These pieces of information will be useful for the experiments in Chapter 5.

Other details include the fact that all the demonstrations are run at 20Hz, i.e. the controller takes an image every 50ms of simulation time. Moreover, images are stored at a 640×480 resolution, although we train our models at downscaled resolutions such as 128×96.

4.3.2 Dataset Examples

For the duration of this project, we generate 5 main, distinct, datasets for the task described in Section 4.1. While the task is the same, we vary the type of target object, textures and number of distractor objects in the scene. This enables us to test the generalisation of our approaches effectively. Table 4.1 shows details of each dataset. In addition, in all of these

datasets, the average number of samples (image-action pairs and extra information) per demonstration is 35, requiring in average 29 steps to reach the target point. An extra 6 datapoints are generated on average to help model the discontinuity, as described previously.

Scene	# dem.	# images	# distractors
`Scene 1`	151	5265	3
`Scene 2`	150	5252	3
`Scene 3`	151	5267	11
`Scene 4`	151	5274	7
`Scene 5`	151	5270	8

Table 4.1: Main characteristics of demonstration (dem.) datasets used to train controllers.

A downwards view of each scene is shown in Figure 4.3, and sample demonstrations for each scene are provided in Appendix A.

Figure 4.3: View from the top of scenes (a) `Scene 1`, (b) `Scene 2`, (c) `Scene 3`, (d) `Scene 4` and (e) `Scene 5`. For each of these scenes, the target object is the object in the center of the images above.

Chapter 5

Encoding Hard Attention

Controllers trained with behavioural cloning (BC Loss) suffer from *distribution drift*: at each step within a trajectory, a controller will output a control action that will not be perfect. As the controller steps through the trajectory, these errors accumulate. Eventually, they lead the controller into seeing an input distribution very different from the distribution it was trained on. This further exacerbates the magnitude of the errors in the outputs of the controller, and it fails to complete the task it was trained on. A visualisation of this effect can be seen in Figure 1.1.

In order to alleviate distribution drift, controllers trained with behavioural cloning require very large demonstration datasets. This amount of data may not be available, and even if it can be generated, the generation process may take a long time. For example, in the setup we described earlier, manually managing a physical robot to generate many demonstrations is often impractical. Nonetheless, this process can be facilitated, with an example being researchers employing virtual reality for robot operation [5].

As mentioned earlier, we hypothesise that if a controller learns to attend to the relevant parts of an image for a task, the variability in the input observations to the controller will be reduced. Thus, less data will be required to learn new tasks. Moreover, by reducing the input size, overfitting would be prevented. All in all, we would expect such a controller to perform better at the task at hand than a controller processing full-sized visual inputs.

We thus assume we have been given a task-relevant point of interest on an RGB image, specified by its pixel coordinates (x, y). By directly cropping around the location of this point and training controllers with such crop, we aim to find out whether performance of a controller can be improved. Note that this feature *is not available naturally* to a controller at test time or in the real world. Indeed, we handcraft and generate this feature during dataset creation in simulation, and obtain it at test time from the simulator itself. Our aim in this chapter, however, is not to find this location, but rather investigate the effect on performance of attending to a task-relevant position[1].

More specifically, can a model employing hard attention around a task-relevant object or location perform well on unseen trajectories under the presence of distribution drift? In addition, is it sufficient to feed into a smaller network the cropped part of the image? Or is there a particular way of encoding the location of the hard attention crop in the image

[1] Methods that learn this location explicitly and automatically are discussed in Section 6.3.1 and Chapter 8.

that empirically provides better performance?

Finally, we experiment and compare the performance of hard attention models with a number of baselines, when trained with different number of demonstrations. In this way, we will also explore whether a hard attention-based controller can learn more effectively with a smaller number of demonstrations than controllers without attention.

5.1 Attention Encodings

As mentioned in Section 4.3.1, our datasets contain the pixel position of the target object for each image. We will use this position as the point of interest - since the task is to reach the cube, only the position and orientation information about the object should be required to solve this task.

All of our attention networks take as input the cropped section with the interest point at the center. The size $W_c \times H_c$ of this cropping window is manually fixed; for this task, we experiment with cropping window sizes of 64×48 and 32×24. The cropped image is fed to a CNN, which will output a 3-D feature map. This feature map is flattened and fed into an MLP, which outputs the vector tip velocity and rotation components.

Since we may crop anywhere within an image, we need to provide to the network some encoding of the location of this crop. The following sections will explore multiple ways of encoding this information.

5.1.1 Naive Attention Encodings

Our initial attempt consisted in utilising the top left pixel of the cropping window directly, which we call `Attention Network V1`. We computed this position by computing the top left pixel of a given window centered at the relevant provided point. The CNN received the input cropped image, and outputted visual features which were fed into an MLP. The pixel is then concatenated to the output of the first hidden layer of the MLP, which will produce a predicted action.

The main issues with this approach are that by only providing a single pixel, no information regarding the size of the attention window is provided to the network. In addition, it is very hard for a network to learn the relationship between the position and the visual features from the signal of a single pixel, as the visual features themselves do not have information corresponding to which area of the image they were extracted from.

In order to attempt to fix the first issue described above, for a given pixel representing the center of a crop we computed the top left and bottom right pixels associated with the cropping window. These are then normalised, and fed into the same hidden layer of the MLP as in the previous approach. We call this approach `Attention Network V2`.

5.1.2 Tiling and CoordConv-based Encodings

In our next approach, we aim to solve the second issue described above. Similarly to the previous attention model, we still compute the top left and bottom right normalised pixels. This time, however, we tile both the top left and bottom right pixels to form two 3-D feature maps with dimensions $\mathbb{R}^{H_c \times W_c}$ each. We concatenate them, and instead of

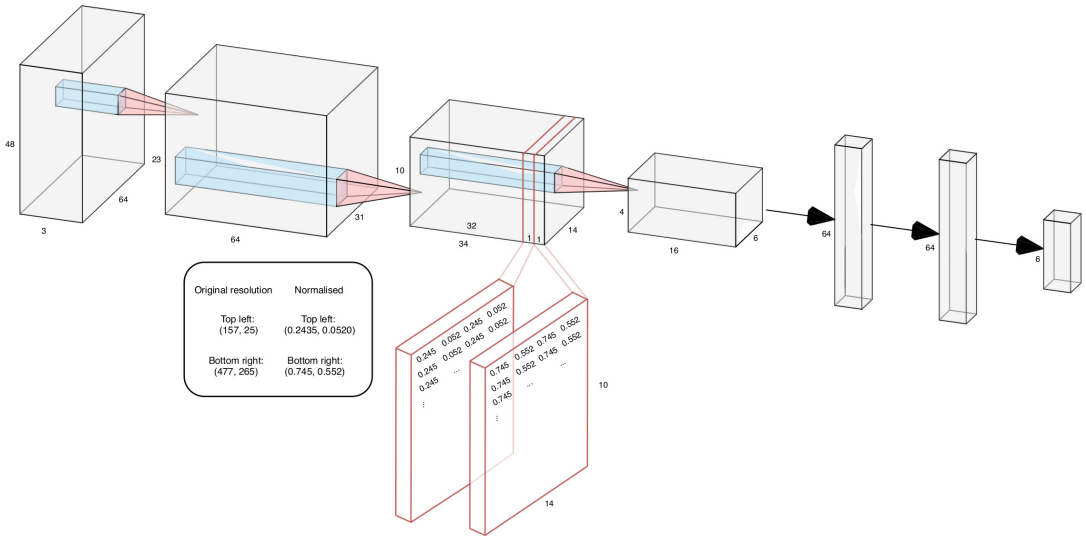


Figure 5.1: Tiling-based hard attention network.

flattening them and supplying them to the MLP, we concatenate them to the output of a convolutional layer in the CNN network. This ensures that visual features produced by the CNN. This approach is inspired by [51], where tiling is used to encode the state and action taken by a robot. We visualise the tiling operation and show a sample architecture using tiling in Figure 5.1. Note that for illustrative purposes, we show feature maps with values in [0, 1]; we would normalise them so they lay in the range $[-1, 1]$.

Furthermore, we propose an alternative approach that also augments the signal from the crop position input. This is mainly based on the CoordConv solution, presented by Liu et al., 2018, [52]. CoordConv is a modification to the standard convolutional layer employed in CNNs, with the aim of including coordinate information into the convolution operation. In particular, a CoordConv layer performs a normal convolution, but concatenates two $\mathbb{R}^{H \times W}$ feature maps before the operation, where H and W are the height and width of the input to the layer. Therefore, for an input tensor in $\mathbb{R}^{C \times H \times W}$ with C channels, the convolution is performed over the augmented tensor in $\mathbb{R}^{(C+2) \times H \times W}$.

These two feature maps represent the coordinates of each feature at any given height and width. In the paper, the rows of the first feature map represent the y coordinate, and each row is filled with its height index. In this way, the first row of the feature map is composed of W zeros, the second row is full of W ones, and so on, with the final row filled with the value $H-1$. The second feature map represents the x coordinate, and in a similar way, its first column is filled with zeros, its second column with ones, and its final column with the value $W-1$. Note that the order in which these feature maps are concatenated is irrelevant. In all of our experiments, we always concatenate the x feature map before the y feature map, to be consistent with our pixel/point of interest representation. As a final step, the two feature maps are normalised so they lay within the range $[-1, 1]$. An example CoordConv layer is shown in Figure 5.2, where the i and j maps represent the y and x coordinate maps, respectively.

This effectively enables a network using CoordConv layers to learn translation dependance,

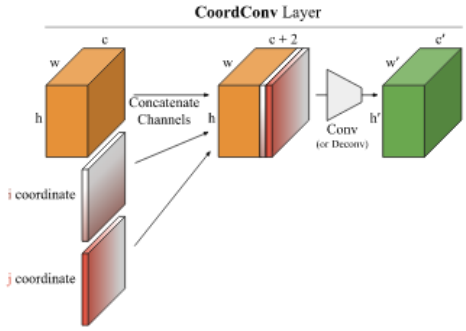


Figure 5.2: CoordConv layer [52].

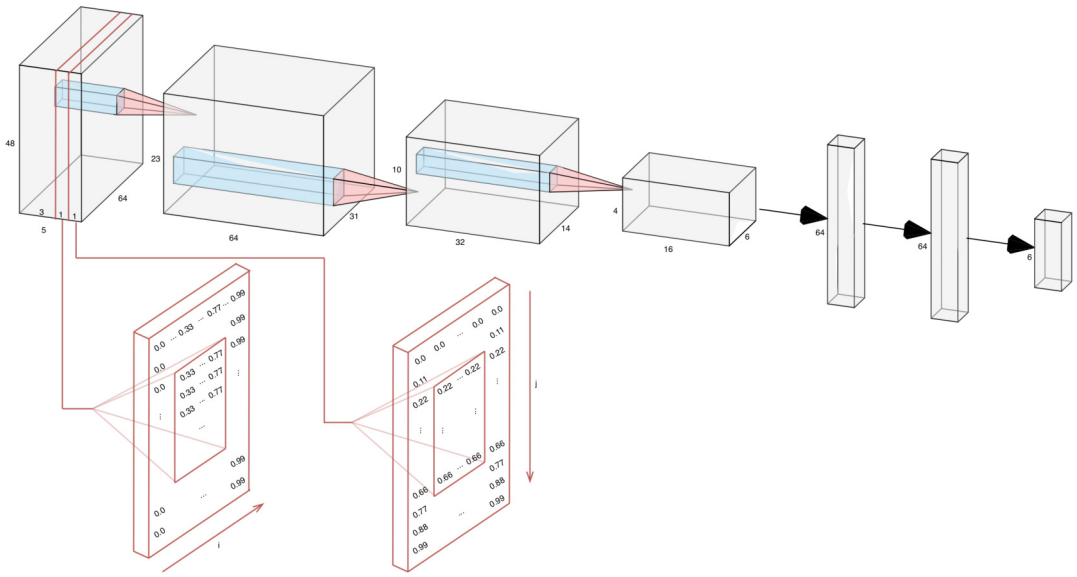


Figure 5.3: CoordConv-based hard attention network.

a property really valuable for our setting. By associating the input features with these coordinates, our controllers should be able to estimate the position of the target object more accurately. Unlike the standard CoordConv feature maps in the paper, since the input to our network is a crop around the pixel for the target object, we provide the part of the original CoordConv feature maps associated with the crop. This is performed by augmenting the original, full image with CoordConv feature maps, and cropping the resulting $\mathbb{R}^{5\times H\times W}$ image to obtain the corresponding $\mathbb{R}^{5\times H_c\times W_c}$ crop around the provided pixel, which is then fed into a CNN-MLP pipeline. Figure 5.3 shows a sample network architecture using these crops, with an attention window of 64×48. In the figure, i and j represent the x and y coordinate axes respectively. Much like in the tiling case, we show dummy unnormalised coordinate maps. Normally, we would convert them into the range $[-1, 1]$.

5.2 Evaluation

We perform a *preliminary* evaluation of the hard attention methods described so far. As the evaluation task/scene, `Scene 1` is chosen. We use three distractor objects, and fix

all their attributes except for the position - distractors are positioned at random at the beginning of each demonstration, and kept fixed throughout its duration. For testing, each test trajectory contains the same three distractors seen during training but at different positions on the table.

5.2.1 Metrics

Up until now, we have treated performance in an abstract sense - we now specify which performance measures we will employ to gauge the generalisability of our methods when evaluating them on test trajectories.

Throughout this thesis, we concentrate on two main metrics:

- Success rate: We define the success rate as the percentage of trajectories for which the final distance to the target point is smaller than some threshold. This distance is taken after a fixed number of steps in the environment, which we take to be 29 for `Scenes 1-5`. This is due to the fact that this is the average number of steps to reach the target object in the demonstration datasets. In addition, unless otherwise stated, we take the threshold to be 3cm.

- Final distance: Since the success rate has a somewhat arbitrarily defined threshold of 3cm, we also want to inspect the final distance to the target point as described above. This allows us to quantify whether our models consistently get close to the target point, even if distribution drift prevents them from reaching points within the 3cm threshold.

Apart from the above, we may introduce other ad-hoc performance metrics. Furthermore, whenever reporting these measures, unless otherwise stated, we will perform three replications of the training process. From each of these, we will obtain the performance metrics as described above, compute the average and standard deviation of the means and report these together.

5.2.2 Baselines

In order to be determine whether hard attention can provide a performance improvement on a certain task, we need some baseline networks to compare our attentive variants against. We propose the following:

- Relative Baseline Network: A network that takes as input the relative position and orientation of the target object (a vector in $\mathbb{R}^6$) with respect to the camera's frame of reference. This network does not require vision, but is not readily applicable in the real world - we can only obtain these quantities on a simulator. Since these relative quantities are very low dimensional, and directly specify the task we are aiming to solve, we expect this network to perform really well on our setup.

- Full Image Network: This network takes as input the full RGB image, rather than the attended region. We explore different resolutions for the full image, such as 128×96, 64×48 and 32×24, and implement alternatives that add CoordConv maps to these images. In doing so, we will explore whether downsampling is enough to reduce distribution drift. Furthermore, we will consider whether adding a CoordConv map is enough to obtain better performance, rather than restricting the image to a particular region via hard attention.

For comparability, we employ the same CNN and MLP architecture for every network, be that relative, full image or attention networks. More specifically, in our experiments we will use a 3-layer batch-normalised CNN with 64, 32 and 16 channels, kernel sizes of 5, 7 and 5 respectively, with stride 2 and padding set to 1. We chose this architecture based on the validation loss obtained after a few experiments with the full image network. The MLP architecture with two hidden layers used throughout is shown in Figure 5.3; 64 neurons are used in each hidden layer. Activation-wise, ReLU non-linearities are used throughout. The only aspect of the architecture that varies is the dimensionality of the flattened visual features fed to the feed-forward network, which is determined by the size of the input image. In the case of the relative baseline network, we use the same MLP as for the visual networks, but with the required input size of $\mathbb{R}^6$.

5.2.3 Preliminary Results

We train baselines and attention networks as described throughout this section with the same dataset split: d_{train}%-10%-10%, where we set d_{train} to values between $5 - 80\%$ to analyse the performance of our networks for various training dataset sizes. We use the same validation and test datasets for every experiment to be able to draw a fair comparison between these methods. We show the success rates for every experiment run in Table 5.1.

Scene 1						
Network/% training data	80%	40%	20%	15%	10%	5%
Relative Baseline	98.33±0.47	98.67±0.94	65.33±2.62	43.33±6.02	16.33±6.34	2.33±0.94
Full Image Network (128 × 96)	37.0±2.83	22.0±2.16	16.33±1.25	7.33±0.47	4.33±1.25	0.33±0.47
Full Image Network + Coord (128 × 96)	34.0±7.48	22.67±2.49	15.67±2.36	4.67±0.47	4.67±0.94	0.33±0.47
Full Image Network (64 × 48)	31.0±5.89	16.0±0.82	10.67±2.87	4.33±1.25	2.33±2.05	0.0±0.0
Full Image Network + Coord (64 × 48)	31.33±5.56	19.0±2.94	13.67±2.87	5.33±1.25	1.67±0.94	0.0±0.0
Attention Network V1 (64 × 48)	71.0±5.1	35.0±5.1	10.67±7.93	4.67±3.09	2.67±1.89	1.67±2.36
Attention Network V2 (64 × 48)	69.0±5.72	56.0±1.41	31.33±2.36	17.33±1.7	8.33±2.05	4.33±2.49
Attention Network Coord (64 × 48)	**81.33±2.05**	**60.67±3.09**	**45.0±3.27**	**25.0±0.82**	**14.33±1.89**	3.67±0.47
Attention Network Tile (64 × 48)	75.0±2.83	57.33±1.25	31.33±4.03	18.0±1.63	11.67±3.3	**4.67±1.7**
Full Image Network (32 × 24)	22.67±2.05	15.0±0.0	7.67±0.47	2.33±0.47	2.67±1.25	0.33±0.47
Full Image Network + Coord (32 × 24)	24.67±1.25	15.33±3.86	6.67±3.68	2.33±1.25	1.0±0.82	0.0±0.0
Attention Network V1 (32 × 24)	82.0±4.97	57.67±6.24	12.0±5.72	9.0±7.26	1.0±0.82	1.33±1.25
Attention Network V2 (32 × 24)	87.0±0.82	**68.0±1.63**	34.33±3.3	**27.0±5.72**	9.33±1.25	3.33±1.7
Attention Network Coord (32 × 24)	**89.0±1.63**	66.0±2.16	**45.0±4.08**	20.67±4.5	**13.33±4.78**	4.0±0.82
Attention Network Tile (32 × 24)	84.33±3.86	62.67±1.25	44.0±6.38	24.0±5.72	13.0±2.16	**4.67±1.25**

Table 5.1: Success rate after 29 steps across 100 trajectories for multiple networks, trained with different percentages of data and different attention resolutions.

Overall, we observe that attention variants perform much better than full input networks, no matter the dimensionality of the input or attention window. Even for the simplest attention encoding, `Attention Network V1`, where only the center pixel is fed to the network, 34 more successful trajectories are completed on average (with 80% of training data) than a standard full image network.

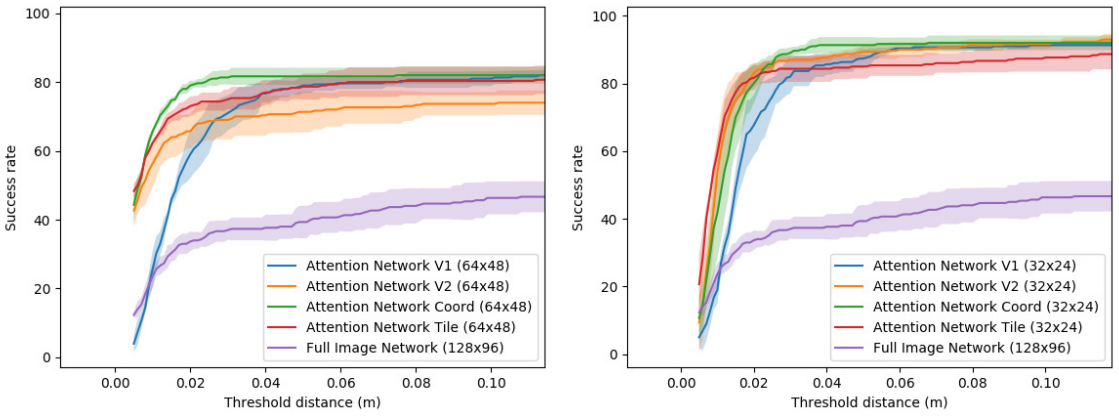
Moreover, networks taking as input downsampled images do not perform better than networks with the same resolution focused on relevant parts of the image. We can see

that, for this setup, every attention network outperforms by a large margin their full input counterparts for both 64×48 and 32×24 inputs. This statement remains valid whether a CoordConv layer is used in the full image networks; simply adding CoordConv coordinate maps does not solve the task. Furthermore, we highlight that capacity does not play an important role here, as for this experiment every network with the same input size, e.g. 64×48 inputs has a similar number of parameters.

In terms of the attention variants, `Attention Network V1` models perform well when sufficient data is available (80%). For smaller amounts of data, the model cannot learn from the poor quality attention encoding, and performance drops severely. In fact, full image networks are competitive with these networks for training dataset sizes smaller than 40%. In contrast, `V2`, `Coord` and `Tile` variants outperform full image networks at every dataset size, showing the benefit of focusing on relevant parts of the image. Out of these, the CoordConv variant obtains the best results for 64×48 input sizes. For 32×24 inputs, both `V2` and `Coord` do, with the CoordConv model performing well most consistently across dataset sizes.

As expected, the relative baseline performs the best across all networks thanks to its handcrafted input. Note that we do not pay much attention to the 5% case, as the success rate for every network on that dataset size is close to zero. This indicates that 5% (9 demonstrations) is not sufficient to even begin to learn in this task. Interestingly, CoordConv variants are quite competitive with the performance obtained by the relative baseline on 10% training data.

We also look at the performance of these networks for different success rate distance thresholds, as we only considered a 3cm threshold previously. The plot is shown in Figure 5.4 for the 80% training case. We observe that the previous analysis holds for other thresholds, and confirm that `Coord` hard attention networks perform consistently above other methods.



(a) (b)

Figure 5.4: Success rates for different threshold distances for networks (a) with 64×48 attention and (b) with 32×24 attention. The shaded area represents the interval one standard deviation away from the performance mean.

5.3 Other Variations

5.3.1 Sinusoidal Positional Encodings

Sinusoidal positional encodings gained popularity from the release of "Attention is All you Need", by Vaswani et al., 2017, [53]. This paper introduced the transformer architecture within an NLP setting, and applied it to solve several machine translation and constituency parsing tasks. In their setup, they needed to encode positional information of each word in a sentence before feeding it into their model, i.e. where the word is in the sentence, including its order with respect to the rest of the words in the sentence. Thus, they compute a positional encoding and add it to the embedding of each word in the input sentence. As presented in the paper, the positional encoding is computed according to the following formula:

$$\gamma(p)_{2i} = \sin(p/10000^{2i/d_{model}}) \tag{5.1}$$

$$\gamma(p)_{2i+1} = \cos(p/10000^{2i/d_{model}}) \tag{5.2}$$

where $p \in \mathbb{N}$ is the position of the word, i is the dimension in the encoding and d_{model} is the total dimensionality of the encoding, i.e. $\gamma(p) \in \mathbb{R}^{d_{model}}$.

Other research areas have also benefitted greatly from this type of encodings. In [54], a much more recent paper, the authors are able to obtain new views of a scene by learning the radiance field (Neural Radiance Fields, or NeRF) of the scene using an MLP. This allows them to generate new views of the scene, or even to extract a mesh from the scene itself. In their work, they show that using a positional encoding greatly enhanced the quality of the radiance field representation learnt by their feedforward network. In their case, they compute the positional encoding for a given spatial point coordinate $p \in [-1, 1]$ as:

$$\gamma(p)_{2i} = \sin(2^i \pi p) \tag{5.3}$$

$$\gamma(p)_{2i+1} = \cos(2^i \pi p) \tag{5.4}$$

As an alternative to CoordConv-based feature maps, we experiment with the encodings from NeRF. To do so, we directly apply Equations 5.3 and 5.4 to the normalised CoordConv maps. For a positional encoding dimension L, we obtain $4L$ positional maps ($2L$ from each coordinate map) that are concatenated to the input RGB image, in a similar way to our CoordConv approach.

We train several of these variants in an experiment. We employ the 32×24 CoordConv hard attention architecture we have used so far and apply the positional encoding. Otherwise, the networks are trained exactly with the same procedure and data. We show the results in Table 5.2 for various training dataset sizes and encoding dimensions.

We find that adding a positional encoding of the form given in Equations 5.3 and 5.4 worsens performance for all training dataset sizes when compared to the baseline 32×24 CoordConv network results, shown in the previous section. In addition, these results are also consistently (across dataset sizes) worse than the 32×24 variants of V2 and tiling-based attention networks. Furthermore, increasing the encoding dimension does not provide a better attention encoding: performance decreases from $L = 1$ to $L = 5$.

The difference in performance may be due to a variety of reasons. Firstly, even for small

Scene 1						
Encoding dimension L/% training data	80%	40%	20%	15%	10%	5%
$L = 1$	**81.0±1.63**	**48.67±2.05**	**23.0±6.38**	**8.67±1.25**	**7.33±2.49**	**0.67±0.47**
$L = 2$	70.67±1.25	34.0±5.35	14.0±3.74	5.33±2.05	2.67±1.7	**0.67±0.47**
$L = 5$	48.67±5.25	21.0±2.45	5.0±1.63	2.67±0.94	3.33±0.94	0.33±0.47

Table 5.2: Success rates for a 32×24 CoordConv-based attention network using positional encodings, with different encoding dimensions L after 29 steps across 100 trajectories.

encoding dimensions, the number of channels in the input image is greatly increased. As an example, for $L = 2$ and $L = 5$ the input image to the network has 11 and 23 channels respectively. This may cause overfitting due to the increased number of parameters, which is particularly relevant in our case due to the small size of our dataset. More importantly, larger input variations may be seen at test time by the network due to the increased size of the input maps, exacerbating the effect of distribution drift. These issues do not appear in either of the papers described earlier, as both methods are trained with large datasets and are not evaluated in a trajectory-centric setting.

For these reasons, we will not explore further sinusoidal positional encodings in this thesis.

5.3.2 Alignment Loss

In our standard approaches we only employ an MSE loss (BC Loss). However, since we are predicting a velocity and a rotation component, we can also add an alignment component to our loss that encourages directional alignment between the ground truth and the predicted velocity and orientation vectors. We employ in several experiments the alignment loss from [5], defined as:

$$\mathcal{L}_{align} = \arccos\left(\frac{a_{E_t}^T \pi_\theta(s_{E_t})}{||a_{E_t}||||\pi_\theta(s_{E_t})||}\right) \tag{5.5}$$

where, as before, a_{E_t} is the expert action (from the dataset) and $\pi_\theta(s_{E_t})$ is the predicted action by our parameterised model. Including the directional alignment, the final loss function is of the form:

$$\mathcal{L} = \lambda_{MSE}\mathcal{L}_{BC} + \lambda_{align}\mathcal{L}_{align} \tag{5.6}$$

with λ_{MSE} and λ_{align} being hyperparameters.

Tables 5.3 and 5.4 show the success rates and the mean distance after 29 steps to the target point for a controller trained with various λ_{align} values, respectively. In particular, we train our best model so far, a CoordConv 32×24 hard attention network with the same setup as previous experiments, while only varying the mentioned loss hyperparameter and the amount of training data. For comparability between different training dataset sizes, we use the same 10% validation set and early stopping for every network, as before. The results provided were averaged across three training runs per hyperparameter value and amount of training data.

The outcome of the experiment shows that adding an alignment loss term to the overall loss can be valuable, most significantly when little data is available. As an example, in

Scene 1						
λ_{align}/% training data	80%	40%	20%	15%	10%	5%
0.25	84.67±3.86	71.33±4.92	47.33±7.32	26.0±2.45	19.67±2.05	1.67±1.25
0.1	87.0±2.94	**72.33±2.05**	**52.33±4.64**	**27.0±2.94**	**20.67±4.11**	**4.33±0.94**
0.05	87.33±3.77	70.0±1.41	48.33±8.73	26.33±5.73	16.33±3.3	2.0±0.82
0.01	86.67±2.36	65.67±2.62	40.33±8.18	23.0±2.94	18.0±2.94	3.0±1.41
0.0	**89.0±1.63**	66.0±2.16	45.0±4.08	20.67±4.5	13.33±4.78	4.0±0.82

Table 5.3: Success rates for different λ_{align} values after 29 steps across 100 trajectories for a CoordConv-based, 32×24 attention network.

terms of the success rate, a network trained with $\lambda_{align} = 0.1$ consistently outperforms a baseline ($\lambda_{align} = 0$) without this contribution for training dataset sizes of $5\% - 40\%$. Similarly, for these same training data percentages, we observe the success rate is consistently higher than the baseline for $\lambda_{align} > 0$. With respect to the mean final target distance, we also see improvements throughout the same ranges.

Nonetheless, for the remainder of the project we consider setups where $\lambda_{align} = 0$, as we plan to investigate the benefits of various attention models. By including the alignment contribution, we might obtain results that are not directly derived from the attention mechanism. This also removes the burden of tuning yet another hyperparameter, which can be very task-specific. However, we note that if any of the methods described in this thesis are employed, and not much data is available, an alignment loss can be considered and the λ_{align} hyperparameter optimised for improved performance.

Scene 1						
λ_{align}/% training data	80%	40%	20%	15%	10%	5%
0.25	3.14±0.55	6.39±1.05	10.02±2.03	16.21±0.7	19.27±1.91	37.2±8.94
0.1	3.22±0.74	**5.53±1.58**	**9.14±0.72**	15.11±1.13	17.58±2.95	31.07±6.34
0.05	3.14±0.83	6.53±1.68	9.88±3.04	**14.99±2.27**	17.91±2.65	34.63±5.34
0.01	3.61±0.45	7.72±1.38	11.24±1.18	18.5±1.53	**15.86±2.19**	**25.21±5.23**
0.0	**3.12±0.41**	7.12±1.13	11.15±2.85	16.11±2.66	18.22±3.09	27.59±0.93

Table 5.4: Mean distance to target (cm) for different λ_{align} values after 29 steps across 100 trajectories for a CoordConv-based, 32×24 attention network.

5.3.3 Regularisation

Networks that take as input the full, high-dimensional visual input may overfit more easily on small datasets than their lower-dimensional counterparts due to their higher relative capacity. A technique that can be used to address this problem is *regularisation*, or *weight decay*, where a penalty is imposed on the parameters of the model. The penalty is included into the optimisation loss as an additional term, and both the optimisation objective and regularisation penalty are optimised jointly.

For that reason, we add to our analysis a variant of the full input network where regularisation is employed. With this experiment, our goal is to ascertain whether regularisation is able to bridge the performance gap between these non-attentive models and the attentive variants presented.

Tables 5.5 and 5.6 show the results from an experiment in which we consider the full image network from the previous sections, and train the same model with different weight decay coefficients w_{decay}. Rather than using the Adam optimiser, as for the previous experiments, with $w_{decay} > 0$ we use the AdamW optimiser - Adam with improved decoupled weight decay, from [55]. The last row shows the results for a baseline full image network trained without regularisation.

Scene 1						
w_{decay}/% training data	80%	40%	20%	15%	10%	5%
0.1	34.67±3.09	**25.67±2.49**	17.0±4.32	4.33±1.25	**4.67±0.94**	**0.33±0.47**
0.05	31.67±1.25	20.67±2.87	14.67±2.62	5.33±1.7	3.33±0.94	**0.33±0.47**
0.01	37.0±6.68	21.33±5.44	17.67±4.11	**7.67±2.49**	3.0±2.16	0.0±0.0
0.001	**41.33±8.99**	20.33±2.49	**18.33±1.7**	6.0±1.41	4.0±0.82	0.0±0.0
0.0	37.0±2.83	22.0±2.16	16.33±1.25	7.33±0.47	4.33±1.25	**0.33±0.47**

Table 5.5: Success rates for different w_{decay} regularisation values after 29 steps across 100 trajectories for networks taking the full image as input.

Scene 1						
w_{decay}/% training data	80%	40%	20%	15%	10%	5%
0.1	15.7±3.51	24.6±1.71	28.09±4.62	**28.48±3.34**	34.55±6.98	43.31±4.26
0.05	15.95±3.79	24.48±6.4	**23.36±4.91**	35.75±4.13	**26.72±3.87**	46.27±5.54
0.01	15.33±3.62	**20.5±3.29**	26.1±4.96	31.68±4.59	33.74±4.47	42.11±2.47
0.001	**14.37±4.7**	23.44±5.3	24.18±4.96	34.24±4.64	37.6±5.43	**38.22±4.74**
0.0	18.87±4.58	25.08±6.56	25.72±2.15	35.05±2.71	29.8±4.63	40.57±4.65

Table 5.6: Mean distance to target (cm) for different w_{decay} regularisation values after 29 steps across 100 trajectories for networks taking the full image as input.

We observe that a non-zero coefficient for regularisation can improve performance both in terms of success rate and mean distance to target, when compared to a network with $w_{decay} = 0$. However, the increase in performance is minor; hard attention variants still perform much better than a full image network with weight decay for this setup. Moreover, the weight decay coefficient has to be tuned; even in these preliminary results, we can see that different weight decay values give very different performance across various training dataset sizes. For that reason, as well as the ones outlined in the previous section, we set w_{decay} to zero for the remainder of the project.

Chapter 6

Attention via Spatial Features

In the previous chapter, we analysed hard attention architectures and their performance, and we assumed a relevant location in an image was provided. In this chapter, we will remove this assumption, and we investigate whether spatial features themselves, rather than visual features, can be *learnt automatically* and utilised to train an accurate controller.

More specifically, we will explore the applicability of deep spatial autoencoders (beginning of Section 3.2.3) and their learnt representations to control tasks. We can consider the spatial features learnt by these autoencoders as a form of *unsupervised attention*: these models learn the locations - the *where* as opposed of the *what* - in an image that can better summarise it in an unsupervised manner, effectively attending to specific pixel locations per image.

We will concentrate on the following questions:

- Can an out-of-the-box deep spatial autoencoder provide useful features for our task? Could we introduce supervision in their training procedure to obtain more relevant features?

- By employing a spatial feature-based low dimensional representation instead of an image, can a controller perform better than its visual counterpart? In this line, are these controllers affected less by distribution drift due to this lower dimensional representation?

- Can spatial features be used as heuristic locations for the hancrafted attention windows employed in the hard attention models from Chapter 5?

6.1 Deep Spatial Autoencoders

Deep Spatial Autoencoders (DSAEs) have been employed in the robotics literature as a dimensionality reduction technique. They learn features that represent locations in space - a robot is more interested in the location of specific objects in its environment rather than semantic features regarding what such objects represent. Thus, we believe obtaining a spatial feature-based representation may be useful for the task at hand. In the following, we describe in more detail the optimisation procedure for these autoencoders and we investigate the applicability to our tasks.

6.1.1 Optimisation

A deep spatial autoencoder, as described in [6], is optimised with the following loss:

$$\mathcal{L} = \lambda_{MSE}\mathcal{L}_{recon} + \lambda_{g_{slow}}\mathcal{L}_{g_{slow}} \tag{6.1}$$

where

$$\mathcal{L}_{recon} = ||I_t^{down,gray} - D(E(I_t))||_2^2 \tag{6.2}$$

$$\mathcal{L}_{g_{slow}} = ||(f_{t+1}^s - f_t^s) - (f_t^s - f_{t-1}^s)||_2^2 \tag{6.3}$$

In the previous formulas, $\mathcal{L}_{recon}$ is the reconstruction loss, where $I_t^{down,gray}$ is the downscaled, grayscaled input image to the autoencoder. D, E are the decoder and encoder networks, respectively. In terms of the spatial features, $f_t^s = E(I_t)$ are the features obtained at time step t from the encoder using the spatial soft argmax layer (Figure 3.4).

Since each spatial feature in f_t^s is composed of an x and y coordinate, f_t^s contains $latent/2$ spatial features, where $latent$ is the latent dimension of the representation. Additionally, we keep the features normalised in the range $[-1, 1]$ based on the dimensions of the input image. The $\mathcal{L}_{g_{slow}}$ loss term ensures that the velocity of the spatial features varies smoothly across each demonstration - as we will see later, the contribution of this term to the performance of our autoencoder is non-negligible. Finally, the hyper-parameters λ_{MSE}, $\lambda_{g_{slow}}$ control the contribution of each term to the loss.

While we need the spatial features f_t^s to provide enough information to be able to reconstruct the original image, we also aim to obtain features from which the action a_t at that time step can be predicted easily. For that reason, in contrast to the original paper, we add another term to Equation 6.1, $\mathcal{L}_{target}$, defined as:

$$\mathcal{L}_{target} = ||MLP(f_t^s) - a_t||_2^2 \tag{6.4}$$

This terms encourages the spatial features to provide a useful representation such that a shallow MLP network can predict the action to take at time step t. Our final loss becomes:

$$\mathcal{L} = \lambda_{MSE}\mathcal{L}_{recon} + \lambda_{g_{slow}}\mathcal{L}_{g_{slow}} + \lambda_{target}\mathcal{L}_{target} \tag{6.5}$$

with an additional hyperparameter λ_{target}.

Note that adding this final term introduces supervision through the action prediction; it is no longer a purely unsupervised method. Nevertheless, we will still refer to them as autoencoders for convenience.

An alternative way of ensuring the representation is useful for the task can be to first train with Equation 6.1, and then fine-tune the autoencoder during supervised learning for action prediction. Nonetheless, we use Equation 6.5 and investigate the contribution of each term when finding good spatial features for our task.

In a similar spirit to the paper from Finn et al., we aim to reconstruct a downsampled, grayscaled version of the input image. In their experiments, they use 240×240 RGB images, which they downscale to 60×60 for reconstruction with a linear decoder. We employ the same 4x downscaling for our 128×96 demonstration images. In terms of the decoder,

we experimented with a decoder formed by a single linear layer, like in the paper, and with decoders formed with transposed convolution layers. We found that a linear decoder was sufficient.

In terms of the implementation, since we could not find any decent implementations of deep spatial autoencoders in PyTorch, we decided to implement our own, and made it available with the default parameters from the original paper on Github[1].

6.1.2 Task-Relevant Features

Revisiting the initial research questions, we want to determine whether a spatial autoencoder with our modified objective is able to provide useful features for our robotic control task. In this case, for these reach-the-target tasks, a useful feature is the location of the target object in pixel space.

To find out whether these features are useful, we train several autoencoders with different $\lambda_{MSE}, \lambda_{g_{slow}}, \lambda_{target}$ hyperparameter values. For this experiment, we set the latent dimension $latent = 32$, i.e. we use 16 spatial features.

We also define the architecture of the CNN employed by the DSAE throughout this thesis as a three-layered, batch-normalised network with $latent * 2$, $latent$ and $latent/2$ channels, respectively. We also set strides of 2, 1 and 1 and employ square kernels of sizes 7, 5 and 5, respectively.

We train on `Scene 1` with three distractor objects, changing only the position of each object between demonstrations. We use three different amounts of training data (80%, 40% and 20%), and always keep the same (separate) 10% splits of the dataset for validation and testing. As the optimiser, we use Adam with default parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and a learning rate of 10^{-3}. We train using early stopping on the validation set with a patience of 10.

Once trained, for each image in the test split, we compute its spatial features and measure the L2 distance between each feature and the projection of the center of the target object onto pixel space. We average these results across the demonstrations in the test set, which gives us $latent/2$ average distances, one per spatial feature. These represent how accurately each feature tracks the target object across the entire test set.

Since the aim in this task is to reach the target object, good, relevant spatial features should be consistently close to the target object in pixel space. Consequently, we choose the minimum distance out of all average distances as our metric, i.e. the distance associated to the feature that is most consistently closest to the target object. The training and evaluation process is repeated 3 times per combination of hyperparameter values and amount of training data; the resulting distances are averaged.

Surprisingly, we find that autoencoders trained with the original objective are not able to produce features that track the target object consistently both across images in a demonstration, and across demonstrations in a dataset. The results from the aforementioned experiment are shown in Table 6.1. We observe that:

[1] Available at https://github.com/gorosgobe/dsae-torch

- Predicting the action only ($\lambda_{target} = 1$) or only reconstructing the image prevents the autoencoder from tracking the target object successfully.

- Adding the g_{slow} term improves performance in both of the cases described in the previous point.

- Adding a small reconstruction error to an autoencoder that predicts the action trained with the g_{slow} term makes performance worse; in fact, we observe that with an MSE contribution of 0.01, 4 times more data is needed to achieve the same performance than when removing this reconstruction term completely.

Scene 1				
Method	$\lambda_{MSE}, \lambda_{g_{slow}}, \lambda_{target}$/% training data	80%	40%	20%
Target only	$\lambda_{MSE} = 0, \lambda_{g_{slow}} = 0, \lambda_{target} = 1.0$	11.68±1.59	11.99±1.72	13.15±1.91
Target and g_{slow}	$\lambda_{MSE} = 0, \lambda_{g_{slow}} = 1.0, \lambda_{target} = 1.0$	**4.53±0.40**	**5.04±0.17**	**7.03±0.59**
MSE only	$\lambda_{MSE} = 1.0, \lambda_{g_{slow}} = 0, \lambda_{target} = 0$	18.79±1.33	20.03±1.22	19.03±1.29
MSE and g_{slow}	$\lambda_{MSE} = 1.0, \lambda_{g_{slow}} = 1.0, \lambda_{target} = 0$	16.44±1.85	18.83±1.62	20.13±0.95
MSE, g_{slow} and target	$\lambda_{MSE} = 0.01, \lambda_{g_{slow}} = 1.0, \lambda_{target} = 1.0$	7.56±1.93	12.75±0.79	13.77±0.76

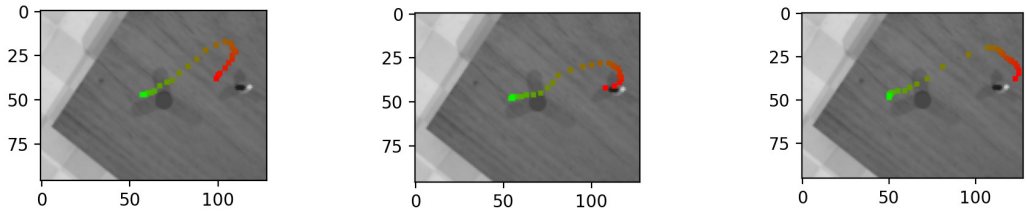
Table 6.1: Average minimum distance (L2 norm of pixel difference) to target pixel from all spatial features, calculated on a 10% held-out test set.

6.1.3 Visualisations

We visualise the spatial keypoints our spatial autoencoder learnt. Figure 6.2 shows the spatial features learnt by two autoencoders trained with the setup above. The features on the left column were learnt by an autoencoder with $\lambda_{MSE} = 1, \lambda_{g_{slow}} = 1$ without action prediction, while the right column ones are from one with $\lambda_{g_{slow}} = 1, \lambda_{target} = 1$, without the reconstruction term. Each feature is shown with a different colour, across images at different time steps in the same demonstration.

The demonstration shown is one the autoencoders were trained with; even for a *training demonstration*, employing the reconstruction error practically ignores the target object until it is big enough to fill the image. At early stages of the demonstration, the small scale of the target object means that most of the reconstruction error comes from the floor and the table. As an example, in Figure 6.2a), the autoencoder produces a red feature around pixel (20, 35) in the first image. This feature is used to model the change in intensity between the floor and the table. In contrast, by ignoring the reconstruction term, the autoencoder in Figure 6.2b) learns spatial features associated with the action, and we observe that it can consistently track the target object.

For the same demonstration, we also visualise the change in spatial location of features throughout this trajectory. In Figure 6.1, we show the three features with lowest pixel distance on the test set. We observe that they accurately track the target object as the camera rotates towards it. In addition, the effect of the g_{slow} term can be visualised: the change in location is smooth throughout the trajectory.



(a) (b) (c)

Figure 6.1: Three best spatial features based on test set distance to a target pixel on a demonstration, with action prediction and no image reconstruction. Features at the beginning of the trajectory are shown in red, and colour is interpolated towards green throughout the demonstration.

6.2 Spatial Keypoint-based Controller

We now address whether we can employ the lower-dimensional spatial features obtained from the above methods to train a more stable controller under distribution drift. To that end, we train a DSAE, and then use the autoencoder to provide spatial features to an MLP. The latter network outputs the action a_t. Note that in this case we keep the parameters of the DSAE fixed during training.

We train autoencoders with 64 and 128 spatial features (latent dimensions of 128 and 256, respectively) with the above described CNN architecture. We experiment with reconstruction term weights of 0.0001 and 0, while keeping fixed weights $\lambda_{g_{slow}} = 1$, $\lambda_{target} = 1$, as these gave us the best performance in the previous section. As for the MLP, we employ a network with two hidden layers of 64 neurons, and we train it with Adam, with a learning rate of 10^{-3} and default parameters. We use the same split as before with 80% of training data, and early stop with patience 10. Table 6.2 shows preliminary results on the same scene as the previous section, where performance is averaged across three runs.

We observe that spatial features from autoencoders trained with a non-zero reconstruction loss term perform worse than the ones with no reconstruction. Furthermore, autoencoders with 64 and 128 features perform similarly without reconstruction loss.

Comparing with the results in Chapter 5, we achieve 81.67% success rate, an increase of 45% more successful trajectories than a controller taking the full image as the input (37%). These results are also competitive with the attention variants described so far, with the advantage that no crop positions are required. We highlight these are preliminary results, with full results shown in Chapter 9.

Nonetheless, they show that attending to spatial positions in the image may be beneficial. For the remaining of the project, whenever we use a DSAE we will employ 64 features, unless otherwise specified, as this provides a good trade-off between performance, number of parameters and training speed.

Scene 1		
# features	Distance	Success rate
64, $\lambda_{MSE} = 0.0001$	**7.52±1.99**	74.33±2.49
64, $\lambda_{MSE} = 0$	7.73±0.28	**81.67±1.7**
128, $\lambda_{MSE} = 0.0001$	9.56±1.16	79.0±3.56
128, $\lambda_{MSE} = 0$	8.97±1.31	81.0±4.24

Table 6.2: Performance on 100 trajectories after 29 steps for an autoencoder-based controller.

6.3 Heuristic methods

Can we employ spatial features as proxies for hard attention locations? In Chapter 5 we saw that hard attention networks with hand-crafted relevant image positions can perform very well on the task at hand. In the previous section, we saw that by including an action prediction term to the loss of DSAEs, we could obtain spatial features that were able to consistently track the target object. Naturally, we investigate whether these two approaches can be combined so hard attention locations are determined automatically.

6.3.1 Heuristic Locations

We propose an approach where we choose a single spatial feature $f_i^s \in \mathbb{R}^2$ out of those provided by the autoencoder, where $i \in \{1, 2, ..., latent/2\}$ is the index of the feature. Importantly, once we have chosen an index i, we use the feature f_i^s consistently across demonstrations and at test time. In order to choose this heuristic feature, we consider the index of this feature as a hyperparameter, and optimise with respect to the validation loss in a grid-search fashion.

In essence, we fix a certain window size $W_c \times H_c$, and for feature index i, we crop around location f_i^s. Once we obtain these crops, we train a CoordConv hard attention network with the same architecture and training process as in the experiments in Chapter 5, on 80% of the training data. Again, early stopping is used, and the best validation loss obtained on a 10% validation dataset is used as the score for such index. We repeat this process for all feature indices. Finally, once each feature has an associated validation loss, we choose the feature with lowest validation loss. Throughout this process, we keep the best attention network and index so far stored, in order to avoid retraining once the optimisation is complete.

The first two rows of Table 6.3 show a preliminary evaluation of this method on `Scene 1`, in the same way as before. We report the performance across three runs, for two autoencoders with different reconstruction weight. As in previous sections, we re-validate that a non-zero reconstruction error hurts the success rate of our controllers; for the same small, 32×24 crop size, following a feature from an autoencoder without the reconstruction terms provides better performance. We also highlight that, while this is an automatic way of determining which feature/location to attend to, performance is not as good as our hard attention controller with hand-crafted crop locations. However, the preliminary results remain promising, as they show higher success rates than full image networks.

This approach is inherently heuristic - there are no guarantees that the selected feature will be able to track the target object consistently. In addition, we outline three important

Scene 1			
λ_{MSE}	Distance after 29 steps	Success rate	Crop size
$\lambda_{MSE} = 0.0001$	14.71±5.59	45.67±20.27	32×24
$\lambda_{MSE} = 0$	13.04±2.07	56.33±0.47	32×24
$\lambda_{MSE} = 0.0001$ (optimised)	**11.09±0.71**	**63.67±1.25**	48×37, 53×39, 76×43
$\lambda_{MSE} = 0$ (optimised)	13.39±0.67	55.0±4.55	31×39, 24×35, 24×32

Table 6.3: Performance on 100 trajectories after 29 steps for a controller that chooses the position to attend to from the features of an autoencoder. The feature chosen is kept fixed across the entire dataset/testing trajectories. Bayesian-optimised versions (`optimised`) are also shown for comparison, where the size of the crop is learnt.

limitations of this method:

- Throughout optimisation, the crop size has to be fixed, and the size of this window may not be optimal. Choosing too small or too large a crop will negatively impact the quality of the validation loss signal we obtain during optimisation. This could lead to choosing a feature that does not provide consistently relevant crops.

- Moreover, the same crop size is applied uniformly across the dataset, but this may also not be optimal. Intuitively, a smaller crop at the beginning of the trajectory may be beneficial, while bigger window sizes could be required once the target is close to the camera. This may only apply to our task, however - other tasks may have other requirements, preventing the generalisation of hand-crafted approaches.

- We choose a single feature to attend to. However, it might be the case that a task requires attending simultaneously to two or more objects. Performing the optimisation we outline above for n features, attending to k positions will require $\binom{n}{k}$ evaluations of our score function. Each of these involves training an entire network, which is an expensive process. Even considering small datasets such as ours, where we are able to cache the dataset in memory to avoid loading from disk, there is a lengthy optimisation process. This renders our approach computationally infeasible when considering multiple attention locations.

6.3.2 Bayesian Optimisation

We now introduce Bayesian optimisation, with the aim to apply this technique to improve upon some of the shortcomings of the aforementioned heuristic method.

Bayesian optimisation is a *black-box* global optimisation method. A black-box function is defined as a function that we can only evaluate, i.e. it has no known analytical form, and we cannot compute its derivatives. Objectives optimised via black-box optimisation techniques are generally hard or expensive to evaluate - we aim to optimise these with as few evaluations of the function as possible.

In Bayesian optimisation, two main components can be distinguished: a surrogate function and an acquisition function. The former models the function f that we are trying to optimise. It is generally modelled as a Gaussian Process, which expresses the Bayesian posterior probability distribution $p(f(x)|f(x_1), f(x_2), ..., f(x_k))$ over the function f, where $f(x_i), i \in \{1, 2, ..., k\}$ are previous evaluations of the function f at points $x_1, x_2, ..., x_k$. Put more simply, it allows us to express a belief over the form of the function f given some

evaluations of such function.

The latter, the acquisition function, decides which x to sample next (after x_k for some k) based on some criterion, with the aim to minimise the total number of evaluations required to achieve a value close to the optimum. While for the surrogate function Gaussian Processes are often employed, a variety of acquisition functions exist.

Since the usage of Bayesian optimisation is not a key feature of our approaches, we do not provide a lengthy description of Bayesian optimisation methods. Notwithstanding that, we point the reader to [56], an excellent tutorial on Bayesian optimisation, should they wish to know more.

6.3.3 Window Size Optimisation

We address the first limitation described in the previous approach: crop size must be fixed. In the same way we treated the feature index as a hyperparameter, we do the same for the crop size. To optimise the window size, we employ Bayesian optimisation for efficiency, rather than employing more traditional approaches such as the already mentioned grid search or random search.

We model the width W_{opt} and the height H_{opt} of the crop as variables that can be optimised independently - the optimal crop may not conform to the same ratio as the original image (as the crops we have explored up until now). More specifically, we optimise $W_{off}, H_{off} \in [0, 1]$, from which W_{opt} and H_{opt} can be derived as follows:

$$W_{opt} = W_{min} + (W_{max} - W_{min}) * W_{off} \tag{6.6}$$

$$H_{opt} = H_{min} + (H_{max} - H_{min}) * H_{off} \tag{6.7}$$

Here, W_{min} and H_{min} are the minimum width and height image resolutions that our CoordConv attention network can receive as input. For the architecture in Figure 5.3, the one we utilise in our experiments, $W_{min} = H_{min} = 24$. The maximum width and height are set to the dimensions of the input image - in our case, $W_{max} = 128, H_{max} = 96$. Note that the width and height obtained are rounded to obtain integer dimensions for the crop.

Bayesian optimisation can struggle with optimisation of discrete variables [57]. To avoid searching over the feature index, on top of width and height of the crop size, we use the previous approach to obtain a good initial feature. We then fix this feature, and optimise the crop size for that specific feature.

Table 6.3 shows (last two rows) the results for features provided by autoencoders with different values of the reconstruction hyperparameter λ_{MSE}. Under the crop size column, we show the three crop sizes found by a Bayesian optimisation procedure during three replications for each autoencoder. The feature index is fixed, and we obtain it from the optimisation experiments in the above section. We employ the autoencoders whose performance is reported in the first two rows in the table. We perform 64 optimisation trials during each replication, and average the performance of the final controller over these three replications.

Interestingly, we observe a major improvement when considering autoencoders with non-zero reconstruction loss. As we observed in the previous section, these tend to not be able

to accurately track the target object. As a side-effect of the Bayesian optimisation process, the error in the target position is compensated via a larger crop. In contrast, the autoencoder without reconstruction can track the target object quite accurately. Therefore, the final crop is of similar dimensions as the original 32×24 window used to determine the best feature. As a consequence, at test time, whenever a controller is affected by distribution drift, the controller with a larger crop size is more likely to recover, as the target object might still be visible within the window.

Implementation

Since Bayesian optimisation is non-trivial to implement, we use the Adaptive Experimentation platform, also known as Ax, in order to perform this optimisation. Developed by Facebook, the platform supports and provides management and deployment utilities for adaptive experiments, a subset of which involve Bayesian optimisation.

It boasts a simple API that enables usage of Bayesian optimisation techniques with very few lines of code. Although lower-level APIs are also made available by the team behind Ax, we use the highest-level one for convenience[2]. This allows us to avoid choosing the hyperparameters for the optimisation - the API chooses the acquisition function and other parameters based on our optimisation problem.

In our case, the optimisation is performed using 5 initial evaluations with randomly sampled candidates, followed by Gaussian Process regression using the expected improvement (EI) acquisition function [56, Page 7]. Other parameters are kept as the defaults provided by the platform.

[2]See `https://ax.dev/tutorials/gpei_hartmann_loop.html` for an example usage.

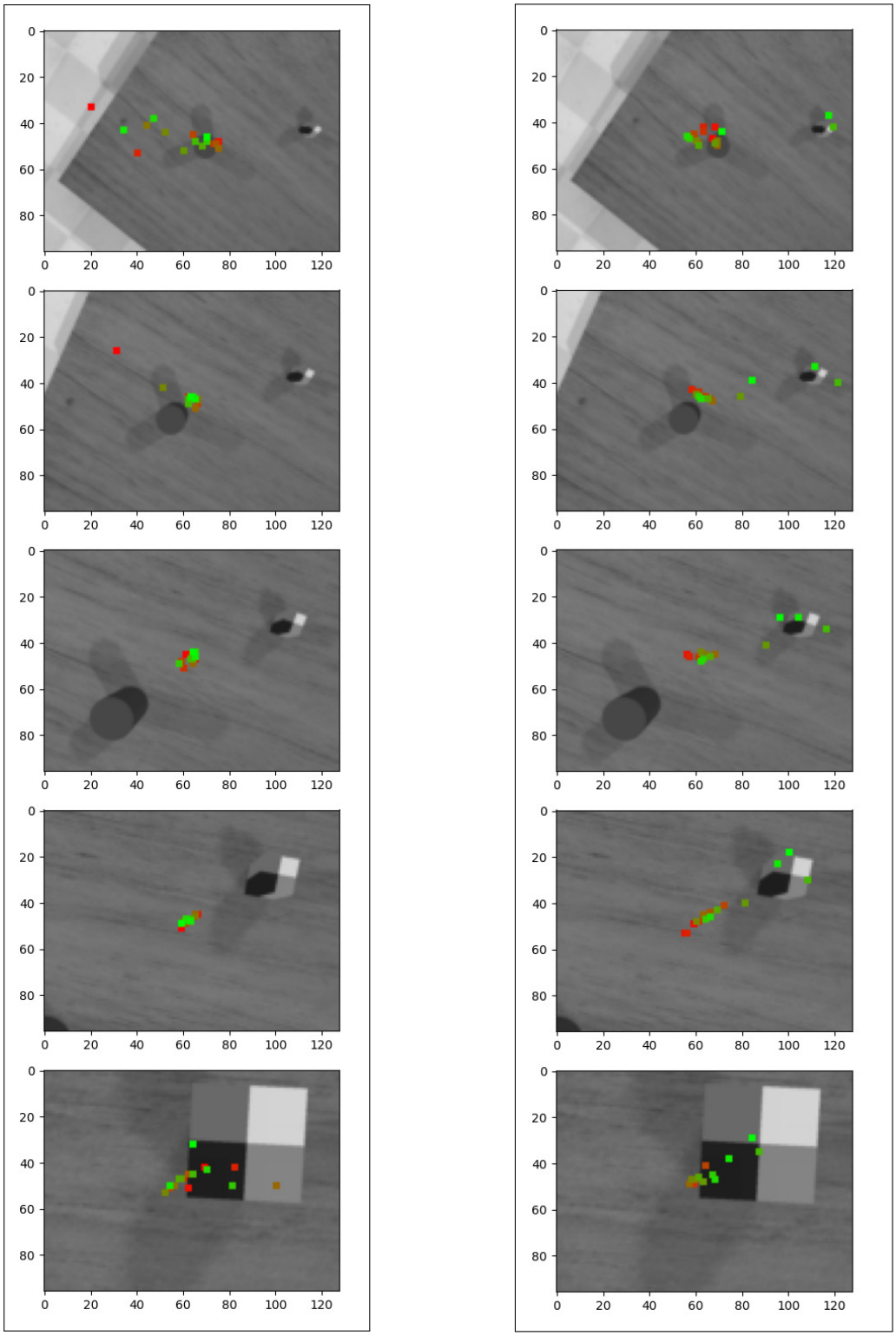


Figure 6.2: Spatial features learnt by an autoencoder with (a) $\lambda_{MSE} = 1, \lambda_{g_{slow}} = 1$ and (b) $\lambda_{g_{slow}} = 1, \lambda_{target} = 1$, with the remaining hyperparameters set to zero.

Chapter 7

Recurrent Attention

We now shift our focus and explore methods that learn soft attention automatically. As described in Section 3.2, soft attention has been successfully employed in a variety of settings. Typically, soft attention involves computation of some sort of *weighting mask*, which can then be applied to the input or feature maps in the network. We note that there may be multiple ways in which this application may be defined. Learning soft attention allows a network to weight more important features more prominently with an explicit operation. In our setting, the mask can be represented as a single-channel map with the same height and width dimensions as the input image/feature maps it is applied on.

Recurrent neural networks (RNNs) have featured alongside soft attention in multiple fields. By combining soft attention with recurrent neural networks, temporal attention may be learnt, where the model learns to attend to possibly different features across time. Temporal attention, on top of spatial attention, could be useful for our setting; a controller may find different objects/areas of a scene relevant for a task at different time steps throughout a trajectory.

One of the main disadvantages of soft attention mechanisms is that the full input image must be processed. Hard attention models, in contrast, employ foveated (cropped) input representations, reducing the forward pass computation cost of the network. In addition, by processing the full image, a soft attention model may be more prone to the distribution drift problem. At the same time, the soft attention framework allows for end-to-end differentiable attentive models, a property difficult (albeit not impossible[1]) to obtain in a hard attention-based model, where usually slow and data inefficient reinforcement learning optimisation techniques are used.

In this chapter, we aim to investigate soft attention mechanisms applied to robotic control tasks.

7.1 Weighted Context Vectors

Our first approach is inspired by the Deep Attention Recurrent Q-Network (DARQN), introduced by [58]. A recurrent neural network, an LSTM, is employed alongside soft attention to train a reinforcement learning agent on an Atari benchmark. In the paper, the authors show that the agent learns to focus on different areas of the screen depending on

[1] See Sections 8.2.1 and 10.1.2

the state of the game. As an example, in the Breakout game their agent learns to attend to the area of the input image where the ball is.

We apply some of their ideas to our setting. Given an input image $I_t \in \mathbb{R}^{3\times H\times W}$ at time step t, we apply a CoordConv layer, producing a $\mathbb{R}^{5\times H'\times W'}$ feature map as described in Section 5.1.2. This operation is followed by several standard convolution layers, which produce an output feature map of size $C \times H' \times W'$ containing visual features $v_t = \{v_t^1, ..., v_t^{H'\times W'}\}$. Note each $v_t^i \in \mathbb{R}^C$.

Our goal thus is to attend over visual features v_t using soft attention. Thus, we compute a soft attention mask $\alpha_t = \{\alpha_t^1, ...\alpha_t^{H'\times W'}\}$, where $\alpha_t^i \in \mathbb{R}$. The mask is computed in the following way:

$$s_t^i = g(v_t^i, h_{t-1}) \tag{7.1}$$

$$a_t^i = Softmax([s_t^1, ..., s_t^{H'\times W'}]^T)_i \tag{7.2}$$

for all $i \in [1, H' \times W']$. Function g is defined in a similar way to [58], but we substitute the tanh non-linearity by a ReLU function, as this is known to alleviate vanishing gradients, namely:

$$g(v_t^i, h_{t-1}) = Linear_{out}(ReLU(Linear_v(v_t^i) + Linear_h(h_{t-1}))) \tag{7.3}$$

In the above formula, "Linear" denotes a standard linear layer of the form $Linear_a(x) = W_a x + b_a$, where both W_a and b_a are learnable parameters. In terms of their dimensions, $W_{out} \in \mathbb{R}^{1\times p}$, $W_v \in \mathbb{R}^{p\times C}$ and $W_h \in \mathbb{R}^{p\times d_{hidden}}$, with p being the dimensionality of an intermediate subspace and d_{hidden} being the hidden size of the LSTM. We experimented with setting $p = d_{hidden}/\beta$ with $\beta = \{2, 4, 8\}$, but found that this intermediate, lower dimensional projection hurt performance, and so decided to keep $p = d_{hidden}$. For convenience, we also set the output number of channels of the last convolutional layer to d_{hidden}, i.e $C = d_{hidden}$.

Then, we obtain a context vector $z_t \in \mathbb{R}^{d_{hidden}}$ by applying a weighted average of features v_t according to the soft mask α_t, by computing:

$$z_t = \sum_{i=1}^{H'\times W'} \alpha_t^i v_t^i \tag{7.4}$$

This context vector acts as a summary of the important features in the last layer. Finally, in order to obtain the prediction a_t for the current time step and the next hidden state h_t, we compute:

$$h_t = LSTM(z_t, h_{t-1}) \tag{7.5}$$

$$a_t = MLP(z_t) \tag{7.6}$$

Initially, we followed the approach in [58] and computed a_t using $a_t = MLP(h_t)$ rather than Equation 7.6. However, we found that employing the hidden state as a representation from which both attention and the action could be predicted was too limiting. By decoupling the action prediction from the attention recurrence, we obtained much better performance.

We show a diagram of the described architecture in Figure 7.1. In the diagram, $\otimes$ denotes the weighted average from Equation 7.4.

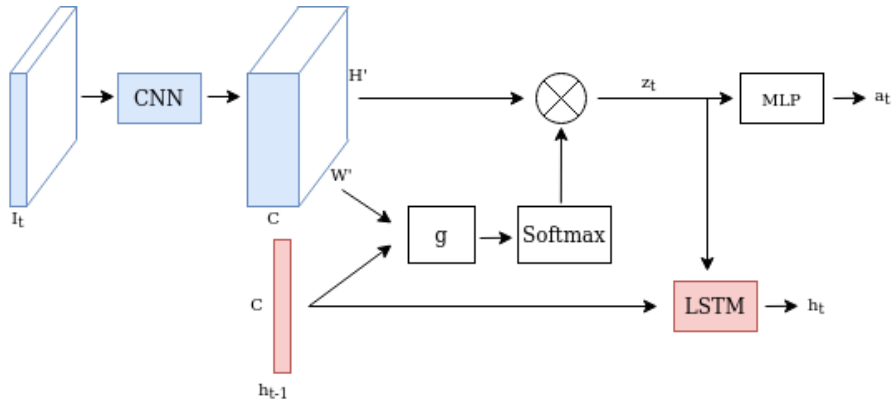


Figure 7.1: Architecture of our proposed recurrent networks with soft attention.

7.1.1 Loss and Entropy Penalty

We considered a standard MSE loss for this setting, as we have been doing so far. Nevertheless, we experimented with an entropy penalty term $\mathcal{L}_{\mathcal{H}}$ on the attention weights. By minimising the entropy of this distribution, we would encourage a more focused and discrete attention distribution. Our intuition was to prevent the attention distribution from degenerating into a uniform distribution.

The loss $\mathcal{L}^t$ including the entropy term for a sample at time step t is as follows:

$$\mathcal{L}^t_{\mathcal{H}} = -\sum_i \alpha^i_t \log \alpha^i_t \tag{7.7}$$

$$\mathcal{L}^t = \lambda_{MSE}\mathcal{L}^t_{BC} + \lambda_{\mathcal{H}}\mathcal{L}^t_{\mathcal{H}} \tag{7.8}$$

where $\lambda_{\mathcal{H}}$ is the hyperparameter tuning the contribution of the entropy term.

We performed a Bayesian optimisation search on the entropy hyperparameter, and found that models without this entropy term performed better: in practice, the model learnt to focus the attention on the target object without assistance from such penalty. For that reason, we set $\lambda_{\mathcal{H}} = 0$ for the subsequent experiments.

7.2 Masking Visual Features

Rather than applying a weighted average over the visual features, as seen in Equation 7.4, we also experimented with an alternative approach where we simply multiply the feature maps and the mask element-wise. In this setting, we modify the attention mask computation as follows:

$$\alpha^i_t = Sigmoid([s^1_t, ..., s^{H' \times W'}_t]^T)_i \tag{7.9}$$

where the sigmoid function is applied element-wise. Once we obtain α_t, we compute:

$$\tilde{v}_t = \alpha_t \odot v_t \tag{7.10}$$

$$h_t = LSTM(\tilde{v}_t, h_{t-1}) \tag{7.11}$$

$$a_t = MLP(\tilde{v}_t) \tag{7.12}$$

Here, $\odot$ denotes element-wise mutiplication and $\tilde{v}_t$ the masked features. We highlight that the input to the $LSTM$ and MLP networks is much higher dimensional than the previous approach - they process the masked visual features, which are of size $\mathbb{R}^{d_{hidden} \times H' \times W'}$. With reference to Figure 7.1, the softmax activation would be substituted by a sigmoid function and $\otimes$ would denote element-wise multiplication.

7.3 Evaluation

7.3.1 Recurrent Baselines

We need to ensure, for a fair comparison, that simply adding recurrence does not allow recurrent models to perform better than non-recurrent ones on the task at hand. Therefore, we develop several recurrent baselines based on the baselines introduced in Chapter 5. Namely, we introduce:

- Recurrent Relative Baseline Network: In a similar spirit to the original relative baseline, we use the relative position and orientation of the target object with respect to the camera as input. Recurrence is added by feeding the input to an LSTM. The outputted hidden state is concatenated with the input, and fed into the relative baseline network to obtain a prediction.

- Recurrent Full Image Network + Coord: We feed the visual features obtained from the last convolutional layer on the full image to an LSTM. The outputted hidden state is then concatenated with the visual features themselves, before processing by the MLP. We also include a variant that performs an initial CoordConv operation on the full input.

- Recurrent Attention Network Coord: We supply the visual features with the attended coordinate maps of the cropped input to an LSTM. As in the previous case, we concatenate them with the outputted hidden state.

A recurrent theme in the above baselines is the concatenation of the hidden state and the input. We perform this operation, rather than only processing the hidden state of the LSTM, so baselines can choose to ignore the features from the LSTM. This may be useful for the networks if temporal features do not help them learn to predict the actions.

7.3.2 Preliminary Results

We train the recurrent networks with a batch size of 16 demonstrations, Adam with learning rate of 10^{-4} and 80% training data. Otherwise, the same parameters from the experiments in Chapters 5 and 6 are used, e.g. data split. To ensure comparability, we use the same CNN architecture as the attention networks in Chapter 5 to extract visual features.

We also vary the dimension d_{hidden} of the hidden state, conducting a few experiments and choosing one that gives the best validation performance. We find $d_{hidden} = 512$ works well for our `Soft Context` attention network, and choose $d_{hidden} = 85$ for the `Soft Mask` version so both networks have a similar capacity. We show results in Table 7.1.

A result we found surprising was the complete failure from the relative recurrent network in this task. The network does not achieve a single successful trajectory in three independent training replications, with 100 test trajectories per replication. Moreover, the

Scene 1				
Network	Distance	Success rate	d_{hidden}	# parameters
Recurrent Baseline	29.03±5.71	0.0±0.0	16	8K
Recurrent Full Image (128 × 96)	13.59±1.6	29.67±5.25	64	860K
Recurrent Full Image + CoordConv (128 × 96)	20.84±3.32	28.33±1.25	64	863K
Recurrent CoordConv Attention (32 × 24)	**5.28±1.39**	66.0±7.48	16	132K
Recurrent Soft Mask Attention	21.75±1.89	38.67±6.94	85	3.691M
Recurrent Soft Context Attention	10.95±2.44	**71.67±2.36**	512	3.694M

Table 7.1: Performance on 100 trajectories after 29 steps for recurrent controllers.

obtained validation loss during training was much lower than the rest of the networks, which ruled out a potential bug in our training procedure. Attempting to reduce training time to ensure overfitting was not occurring yielded similar results.

We suspect this result is due to our demonstration data. For this network, the input takes the form of a relative position and orientation per image in a demonstration. During data generation, the camera moves towards the target in a straight line, while rotating towards the pose of the target object. This allows for varied *images* within a demonstration, but the generated relative positions and orientations in the same trajectory are very highly correlated, if not almost identical. As a result, a recurrent relative baseline can learn to predict the action based on an initial position and orientation. Then, for the remaining time steps in the demonstration, the network can simply use the hidden state to remember to predict a very similar action. Thus, at test time, a small error in the initial action prediction has an unexpectedly large effect on the subsequent actions. The relative baseline network does not suffer from this as it does not exploit recurrence, and is trained with images independently from the demonstration they belong to.

With respect to the remaining baselines, we highlight that adding recurrence does not help full image networks perform better than the same network without recurrence. Out of all the baselines, the CoordConv variant performs the best, although performance can be unstable across replications, as we can see from a large standard deviation of 7.48 trajectories.

Analysing our attention model proposals and comparing with some of the results in Chapter 5, we see that a network using a mask directly on the input can outperform full image networks. However, the improvement is minor when compared to both hard attention and recurrent CoordConv-based networks.

We observe a big difference in performance between the `Soft Context` variant and the `Soft Mask` one. The key difference between these methods is that the former enforces a lower dimensional representation on the visual features via the context vector. This could force the network to pick the most relevant features. In contrast, the `Soft Mask` variant keeps all the features and simply learns to weight them. As we will see in the visualisation section below, at test time this translates into attention weights that are unable to ignore distractor objects.

Our `Soft Context` approach obtains the best success rate (71.67%) out of all networks in this section, although performance is still suboptimal with respect to hard attention networks. Nonetheless, in a similar way to the methods in Chapter 6, no relevant pixel positions/crops need to be specified for this method.

An additional benefit of this method with respect to spatial feature-based methods is *reduced training time*. All spatial methods require training times at least an order of magnitude longer - in this sense, for this setup we trade training time for performance. Note we will evaluate training times, capacity and performance trade-offs across various tasks in Chapter 9; these results should be interpreted as preliminary.

7.3.3 Visualisations

As we introduced in the Related Work section (Chapter 3), soft attention can aid interpretability of our models. In our case, it is easy to visualise which part of the image our model concentrates on, by visualising the attention weights. To do so, after an input image is fed to the network, we obtain the attention weights produced from the visual features in the last layer. To produce the visualisation, we upsample the attention weights to the original input image size (128×96), and overlay the generated attention map over the input image. Brighter areas in our visualisations, therefore, correspond to higher attention probabilities.

We visualise the attention over a test trajectory, using a `Soft Context` network trained as defined earlier. The images produced can be seen in Figure 7.2. It can be seen that very early in the trajectory, the network is able to focus on the target object, as we expected. Interestingly, at the beginning of the test trajectory, the network concentrates on the side of the floor, until it disappears from the field of view of the camera. Indeed, it is sufficient (for our dataset) to look at that area of the image to estimate the rotation component, but only at the start. This is the case because we apply a limited rotation to the camera around the z axis to generate new viewpoints - the part of the image the camera sees the floor in can be used to determine the rotation. Once the camera only sees the table and the objects on its surface, the velocity and rotation components can only be estimated accurately from the target object.

We also visualise the attention weights learnt for our `Soft Mask` method. In this case, since we are using sigmoid activations, we choose to visualise the region of the image with upsampled weights larger than 0.5. A visualisation of two test trajectories is shown in Figures 7.3 and 7.4, where the former is an unsuccessful trial and the latter a successful one. In both cases, even in the successful trajectory we observe that the network is unable to ignore (i.e. assign lower weights) to the distractor objects. Moreover, although slightly higher weights are placed on the area around the target object, the model fails to ignore other parts of the image, often applying a high weight to task-irrelevant parts of the table.

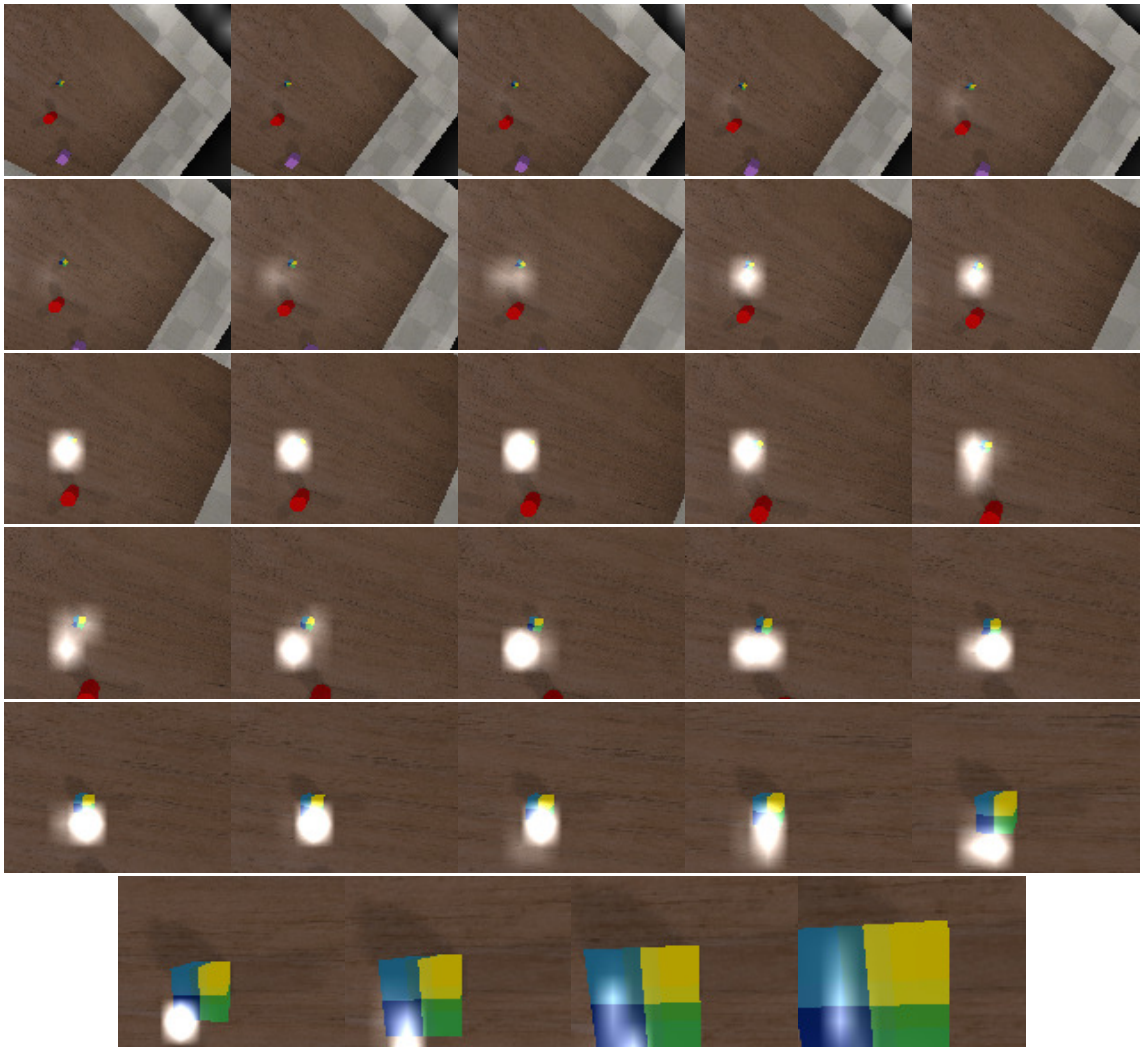


Figure 7.2: Test trajectory by a context vector attention network. The trajectory is shown across time from top to bottom, left to right. The network focuses on the regions with larger attention weights, shown in white.

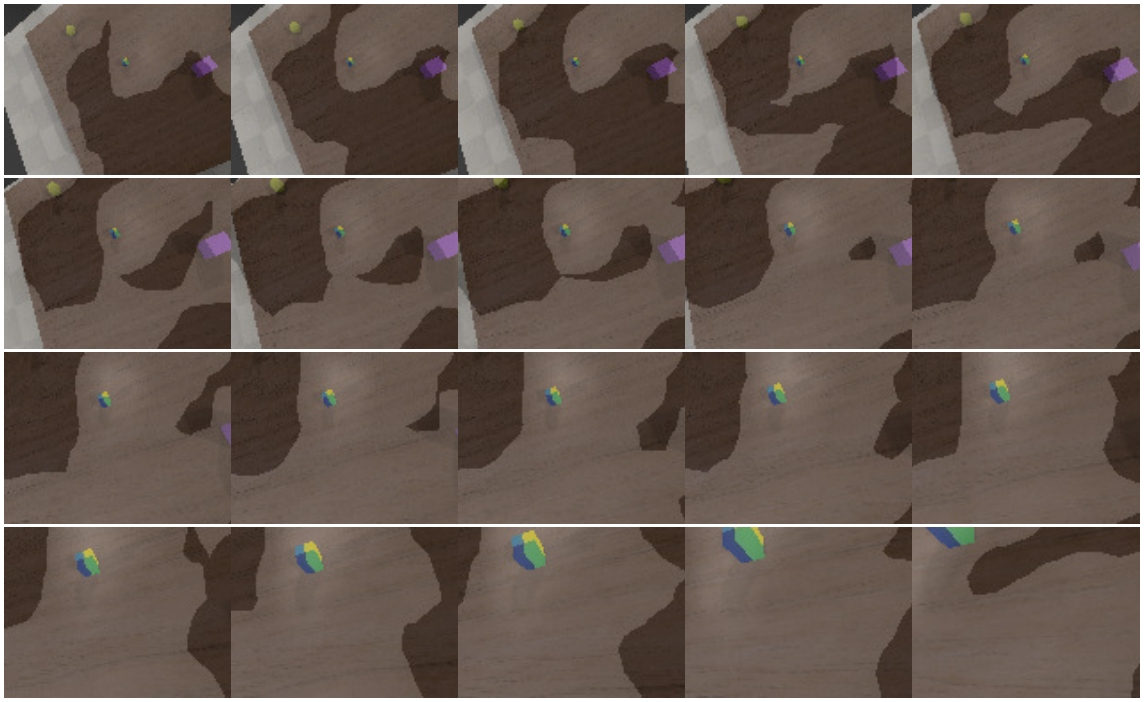


Figure 7.3: Unsuccessful test trajectory by a soft mask attention network. The trajectory is shown across time from top to bottom, left to right.

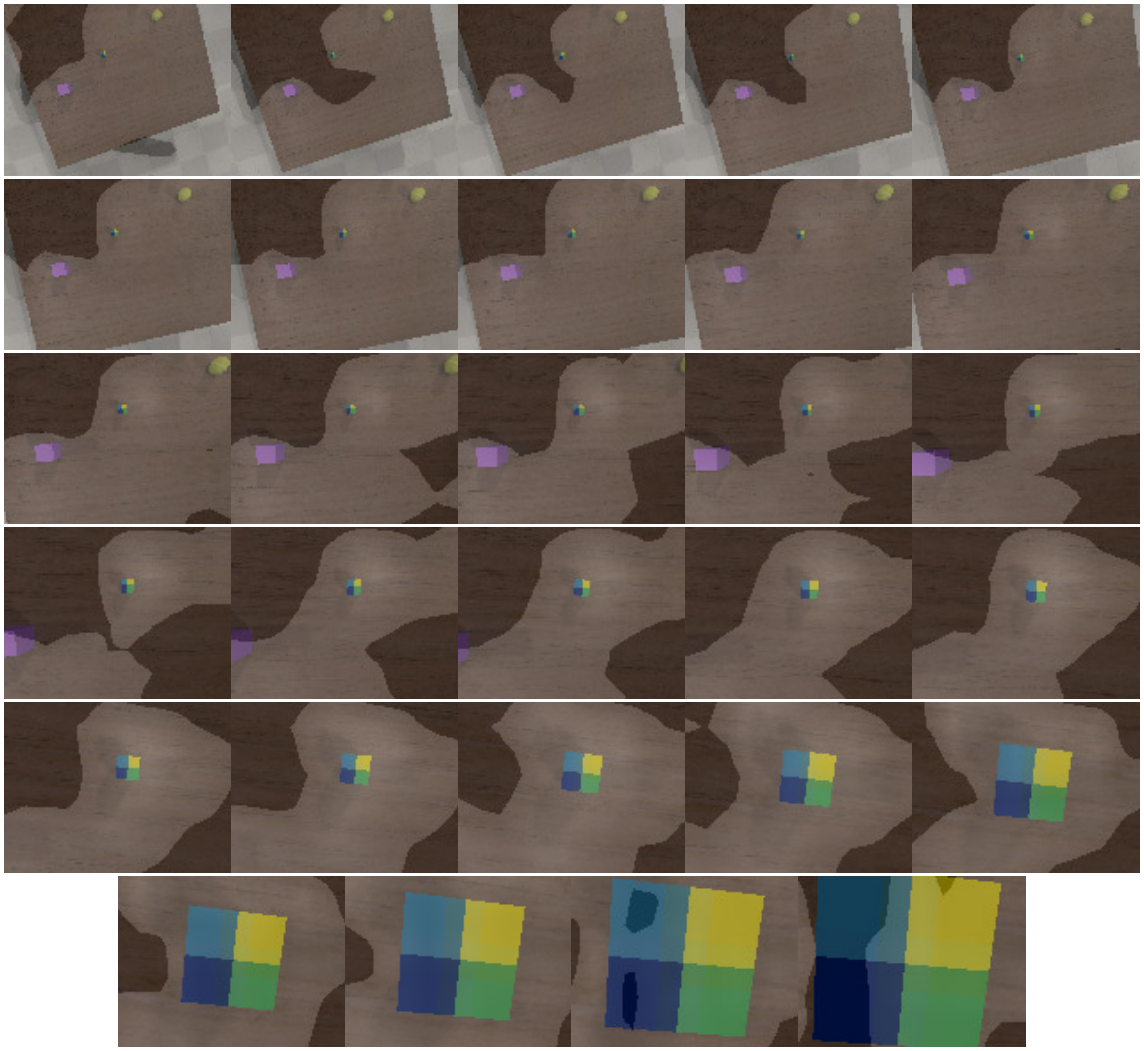


Figure 7.4: Successful test trajectory by a soft mask attention network. The trajectory is shown across time from top to bottom, left to right.

Chapter 8

Meta-Learning Attention

In Chapter 5, we developed a variety of hard attention methods that leveraged predefined, handcrafted crop locations to outperform full image networks. These locations are not available at test time in the real world, and are obtained from the simulator. To address this limitation, we looked at several heuristic methods to determine relevant image locations for hard attention networks in Chapter 6. Can we find these positions without relying on heuristic approaches?

In this chapter, we attempt to learn the relevant locations of an image automatically by leveraging a validation signal, *without requiring any privileged data at test time*. In that sense, we aim to investigate whether we can we find the relevant parts of an image that, when trained upon, allow our model to generalise more easily. Therefore, the methods presented in this chapter can be considered as *meta-learning* methods. Specifically, we propose two different approaches, one based on reinforcement learning and another one based on spatial transformer networks.

8.1 RL Bounding Box Controller

In this section, we explore whether we can use deep reinforcement learning on the demonstration dataset to learn a suitable hard attention cropping policy.

8.1.1 Environment and State/Action Spaces

Our core idea is to develop and train an agent that can learn how to crop images from the training dataset, with the hope it can generalise to unseen test trajectories and maximise the performance of an underlying hard attention controller.

For that reason, we do not perform reinforcement learning on a simulated environment, but rather on the demonstration dataset itself. We divide the demonstration dataset into training, validation and test splits, and we propose the following environment setup:

- Throughout an episode, demonstrations are sampled from the training demonstration dataset. At every step, the agent receives an image from the current demonstration, crops it for a specific window size $W_c \times H_c$ and receives a reward of 0 from the environment. Once the demonstration is over, a new demonstration not seen so far during the episode is sampled. The first image of the demonstration is then provided to the agent, and this demonstration becomes the current demonstration.

- The steps mentioned above happen all within the same episode. An episode is over once all demonstrations have been sampled from the training demonstration dataset. At that point, we have cropped images $I^i_{cr,train}$ corresponding to each image I^i_{train} in the training split. Then, we evaluate the agent on the validation set to obtain cropped images $I^j_{cr,val}$ from each validation image I^j_{val}.

- With both datasets of cropped images, we train a CoordConv-based hard attention network (Section 5.1.2). We use the validation dataset for early stopping, and once the training is complete, we return to the agent the negative validation loss obtained by the network. This reward completes the episode.

As for the state of the environment the agent sees at any given step, we include the current image I_t alongside the center position of the crop applied to the previous image in the demonstration, i.e. an observation is of the form $o_t = [I_t, x^C_{t-1}, y^C_{t-1}]$. If I_t is an image at the beginning of a demonstration, we initialise x^C_{t-1}, y^C_{t-1} to the center of the image.

The agent, rather than predicting the location of the crop at every time step, predicts the offset applied to the previous crop position to obtain the current crop. This translates into an action of the form $a_t = [dx^C_t, dy^C_t]$, where dx^C_t and dy^C_t are the offsets to be applied to x^C_{t-1}, y^C_{t-1} to obtain x^C_t, y^C_t. Note that for the agent, we keep both observations and actions normalised in the range $[-1, 1]$. However, when applying the action on the environment, we rescale to the size of the images in the demonstration dataset. In addition, if applying the offset to a crop makes some part of the window remain outside of the image, we shift the crop so it is fully contained within the image.

With this structure, we would need to use a CNN in the policy network. Moreover, memory constraints would limit the number of images we would be able to store at once in a replay buffer. To improve training speed, reduce the observations to a lower-dimensional representation and increase the size of the replay buffer, we also consider using DSAE features of the images rather than the images themselves. For this case, the state becomes $[f^s_t, x^C_{t-1}, y^C_{t-1}]$.

8.1.2 Implementation

Implementation-wise, we used Stable-Baselines [59], an open source collection of implementations of reinforcement learning algorithms, such as SAC or PPO, written in TensorFlow. Born as a fork of the popular OpenAI Baselines project [60], Stable-Baselines provides better documentation and a uniform interface for their RL algorithms.

In order to define an RL agent and task, an environment must be set up. To create one, a class that inherits from the `gym.Env` environment interface has to be written, which must define the following:

- A `reset(self)` method. This method is called at the beginning of an episode, and must return the first observation of the episode.

- A `step(self, action)` method. This method is called at every step in the environment within an episode initialised via `reset()`. It returns the observation obtained after the action is applied to the environment, followed by a reward, whether the episode is done and an optional dictionary for extra information. If the episode is done, the RL algorithm will call the `reset()` method afterwards.

- Two fields: `self.observation_space` and `self.action_space` that define the observation space and action spaces of the agent via the `gym.spaces` module.

- A `render(self, mode)` method to render the environment, if necessary.

Once the environment is created, an RL agent can be implemented quickly, as Stable-Baselines provides default network implementations in TensorFlow. For the case where our state was composed of spatial features, this allowed us to create every network required by the RL algorithm with a single line of code. For the image-based state case, however, we had to implement our own custom CNN-based networks directly in TensorFlow.

More specifically, in the spatial feature setting we employed the default `"MlpPolicy"` architecture. In terms of the CNN, we developed an architecture based on the attention encodings of Chapter 5:

- Initially, we concatenate the image I_t with CoordConv coordinate maps.

- From x^C_{t-1}, y^C_{t-1} and a given crop size, we calculate the top left and bottom right normalised pixels corresponding to the cropped part of the image. In the same way as our tiling method, we tile both into separate feature maps.

- We feed the image with CoordConv maps into our network, and we concatenate the tiled feature maps to the output map of the first CNN layer.

- The final visual features are flattened and fed to an MLP. Its output is passed to other smaller MLPs, where each compute different values required by the RL algorithms, such as the action (policy network) or the value function. This design choice is applied throughout the Stable-Baselines project, including the default MLP provided.

8.1.3 Outcome

In terms of hyperparameters, we built a 3-layered CNN (each with stride 2) followed by an MLP with two hidden layers, and we varied the number of neurons and size of filters during experimentation. The default MLP provided by Stable-Baselines is a two hidden layer feedforward network with 64 neurons. ReLU was used as the activation function.

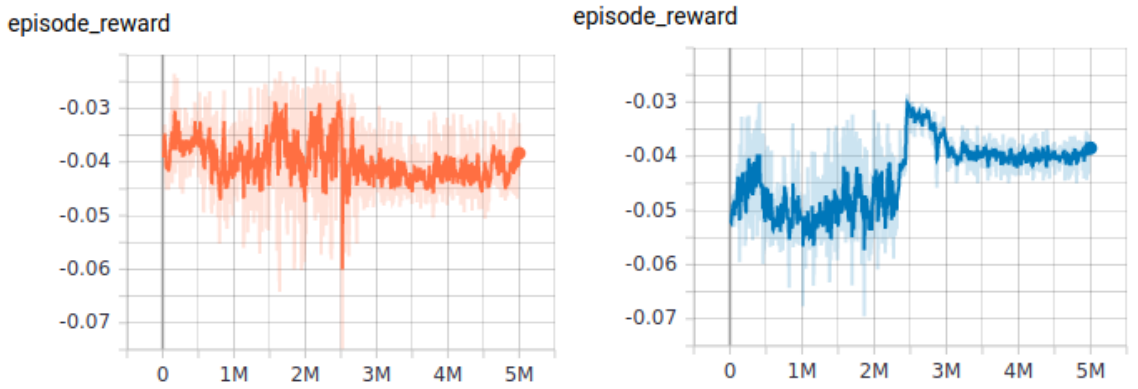
We ran both SAC and PPO (Proximal Policy Optimisation) on the same scene as all other experiments up until now, with cropping window sizes of 64×48 and 32×24. For each state, we found that:

- Image-based state: The training time was very high. Furthermore, even after trying multiple architectures and data splits, we found that neither of the algorithms was able to obtain a successful policy. In fact, the agent continuously attempted to crop the corners of the images. In addition, for SAC the size of the replay buffer was constrained due to the size of the images. We decided to switch to the spatial feature-based representation, in an attempt to fix these issues.

- Spatial feature-based state: With this representation, we were able to store more information into the replay buffer. We set its size to 10^6 transitions. However, we observed the same behaviour from the agent.

Furthermore, we noticed that the reward during training barely improved with respect to the initial reward, even for 5-day training runs of 20M time steps. We also compared the reward obtained with the negative validation loss obtained by the attention networks we trained in Chapter 5. We found that the attention networks obtained a much higher negative validation loss, with even full image networks performing better than the reinforcement learning agent.

As an example, we show the reward curves of two SAC agents in Figure 8.1. Each agent was trained for 5M time steps ($\sim$ 13 hours) with a 60%-30% split for training and validation (10% unused). The orange and blue agents employed 64×48 and 32×24-sized cropping windows, respectively.



(a) (b)

Figure 8.1: Episode reward (shown with exponential smoothing of 0.7) for two sample SAC RL agents controlling cropping windows of sizes (a) 64×48 and (b) 32×24. The shaded area around the main plot lines corresponds to the unsmoothed episode reward.

For comparison purposes, we look at an example where we obtain the validation loss from a CoordConv hard attention network trained with the same data split (60-30) as the RL agent. With regards to the window size, we use a 64×48 window and our predefined pixel location (see Chapter 5). Across three replications, this network obtains an average negative validation loss of -0.0047. In contrast, with RL after 3M steps of training rewards smaller than -0.03 are obtained. In the 32×24 case, we observe an even more acute difference: -0.0028 versus -0.035.

We also monitored the pixel distance between the crop produced by the agent and the target object during training. To do so, every N steps in the environment (normally we set $N = 25$K) a validation demonstration was sampled. On that demonstration, we computed the average distance across the demonstration between the center of the predicted crop and our expected pixel position. We would expect this distance to be (and remain) low if the agent manages to track the object successfully across demonstrations. We plot this distance for both of the previous example agents in Figure 8.2.

Both agents initially crop the corners of the images, with the smaller window agent not improving throughout training. The agent with the larger 64×48 window eventually learns to crop the center of the image. This policy works for big window sizes at the end of each demonstration, when the target object is usually in the center of the image. However, it

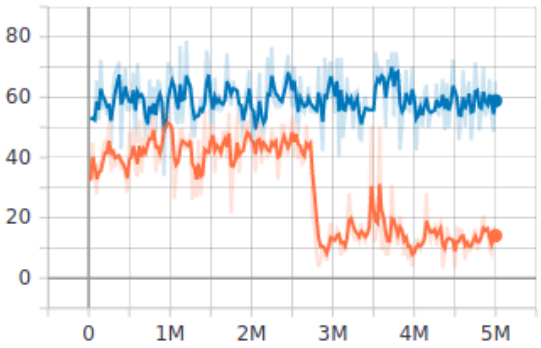


Figure 8.2: Average distance in pixel space between the center of the predicted crop and the pixel corresponding to the center of the target object, shown during training. Exponential smoothing of 0.7 is used in the plot; the shaded area around each plot line is the unsmoothed distance.

does not learn to generalise this policy to track the object, achieving average distances of $\sim$ 15 pixels from the target object. We highlight this is much larger than the distance obtained by the DSAE features ($\sim$ 4), which successfully managed to track the object.

Figures 8.3, 8.4 and 8.5 show sample validation demonstrations cropped by the example SAC agents. The first two correspond to an agent with a cropping window of 64×48, and show that initially the agent cropped the corners of the demonstrations, but was able to learn that cropping the center of the images was better. As opposed to this, the 32×24 variant failed to learn, cropping the corners of the image even after 5M training time steps.

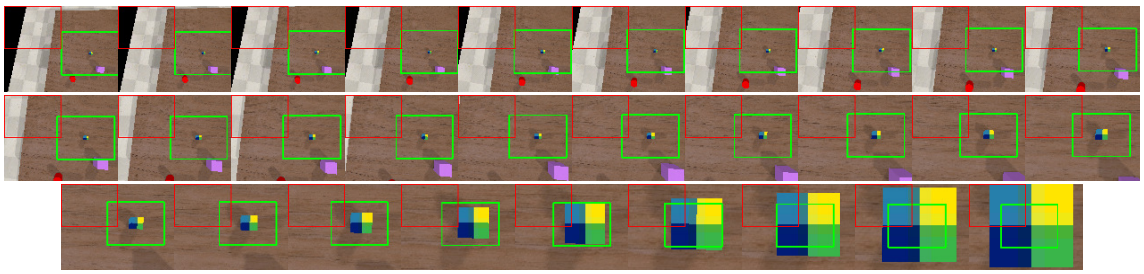


Figure 8.3: Crops produced by a SAC agent (in red) with size 64×48 after training for 2.5M steps. In green we show the handcrafted crop used to train CoordConv hard attention networks.

Overall, at least for the techniques described throughout this section, we found reinforcement learning failed at generalising across demonstrations. Even after spending an extensive period of time on it, lengthy training times and the inability to produce suitable cropping policies led us to abandon this approach.

8.2 MetaSTN

In this section, we introduce a meta-learning algorithm for hard attention based on Spatial Transformer Networks (STNs).

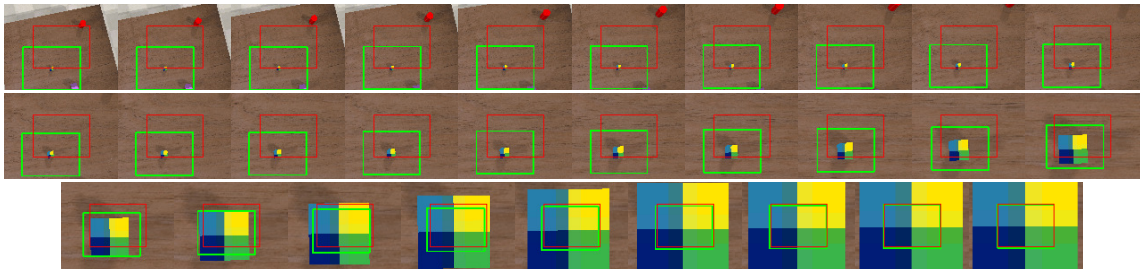


Figure 8.4: Crops produced by a SAC agent (in red) with size 64×48 after training for 5M steps. In green we show the handcrafted crop used to train CoordConv hard attention networks.

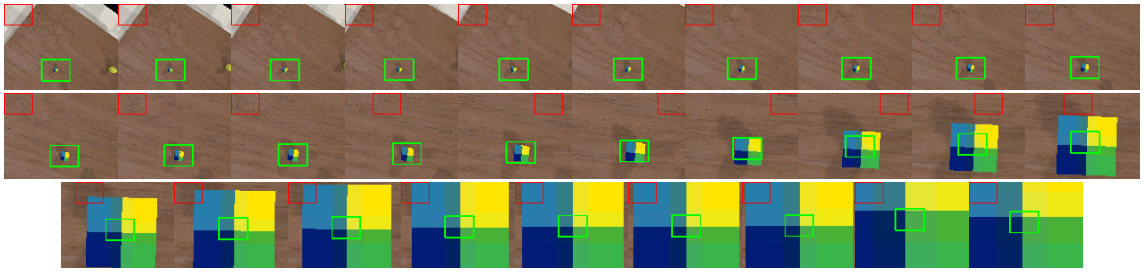


Figure 8.5: Crops produced by a SAC agent (in red) with size 32×24 after training for 5M steps. In green we show the handcrafted crop used to train CoordConv hard attention networks.

8.2.1 Spatial Transformer Networks

Spatial Transformers (ST) were introduced by Jaderberg et al. in 2015 through [61]. In their paper, the researchers developed a component that can be added at every level of existing network architectures, much like a CoordConv layer. By learning the parameters of a transformation, this component, the Spatial Transformer, can learn to transform its input so a downstream layer/network performs better at a certain task.

More specifically, a Spatial Transformer is composed of three main components: a localisation network, a grid generator and a sampler. The localisation network takes as input feature maps $U \in \mathbb{R}^{C \times H \times W}$, and predicts the parameters θ for a spatial transformation such that:

$$\begin{bmatrix} x_i^s \\ y_i^s \end{bmatrix} = A_\theta \begin{bmatrix} x_i^t \\ y_i^t \\ 1 \end{bmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix} \begin{bmatrix} x_i^t \\ y_i^t \\ 1 \end{bmatrix} \tag{8.1}$$

where (x_i^s, y_i^s) and (x_i^t, y_i^t) are the source and target coordinates of the input and output feature maps, respectively. With respect to this project, this transformation matrix can be restricted to an "attention" version, such that:

$$A_{s_x, s_y, t_x, t_y} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \end{bmatrix} \tag{8.2}$$

By learning the latter transformation, the localisation network essentially learns to control a cropping window of size $s_x \times s_y$, by moving it to position (t_x, t_y). Note that these

parameters are dependent on the input map - the network may learn to crop different images/feature maps at different resolutions and locations.

The last two components are essential in order to make this transformation differentiable. The grid generator $\mathcal{T}_\theta$ creates a grid G of a specifiable output size $C' \times H' \times W'$, and applies the transformation to it, obtaining $\mathcal{T}_\theta(G)$. Then, the grid sampler generates the output map $V \in \mathbb{R}^{C' \times H' \times W'}$.

To do so, since the coordinates in $\mathcal{T}_\theta(G)$ specify the sampling points on the input image, a sampling kernel is applied to obtain every value in V. By default, the bilinear kernel is used, for which the authors compute the subgradients in order to make the sampling differentiable. Note that during backpropagation the gradient flows from the output grid towards the sampled points in the input map.

Figure 8.6 depicts a sample Spatial Transformer.

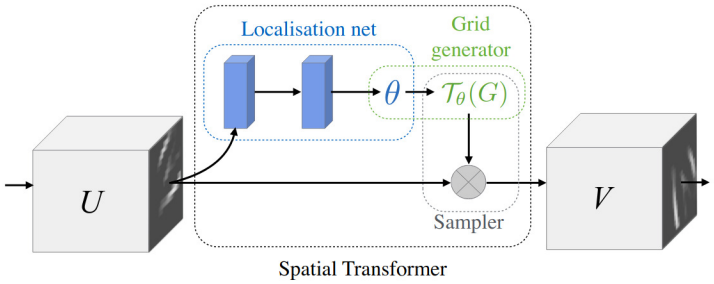


Figure 8.6: Diagram of the main components of a Spatial Transformer: the localisation network, grid generator and sampler [61].

8.2.2 Motivation: classification on CUB-200-2011 dataset

Our main motivation to attempt a meta-learning approach using Spatial Transformers comes from a particular experiment performed by the authors of the paper. In such experiment, they aimed to investigate whether multiple attention transformations in parallel (Equation 8.2) could be used effectively for a classification task. Particularly, they used the CUB-200-2011 bird classification dataset.

In a specific instance of the experiment, they set up a localisation network, predicting two attention transformations with $s_x = s_y$. The cropped output feature maps were then fed to two independent Inception networks, and classified. They found that, while only employing the classification loss signal, each Spatial Transformer learnt to attend consistently to different parts of the birds, such as their head or body.

In our case, we do not want to simply optimise with respect to the training loss, but also with respect to the validation performance, to ensure that we attend to parts of the image that allow the controller to generalise. To achieve so, we propose the MetaSTN algorithm.

8.2.3 MetaSTN algorithm

We present a meta-learning optimisation-based algorithm that learns a hard attention policy in a (sub)differentiable manner. We call this algorithm MetaSTN. We draw inspiration from the structure of the MAXL algorithm [62], a meta-learning algorithm that aims to improve the classification performance of a network by generating auxiliary labels during training.

In essence, we want to meta-learn parameters ϕ for a Spatial Transformer g_ϕ parameterised for attention, such that the generalisation capabilities of a controller $f_\theta(g_\phi(\cdot))$ are maximised. To do so, we divide our training demonstration dataset D into D_{train} and D_{val} splits. Then, we sample training and validation batches from each respectively, and update the meta-parameters as follows:

$$\theta^{'} = \theta - \alpha \nabla_\theta \mathcal{L}(f_\theta(g_\phi(x_{train})), y_{train}) \tag{8.3}$$

$$\phi \leftarrow \phi - \beta \nabla_\phi \mathcal{L}(f_{\theta'}(g_\phi(x_{val})), y_{val}) \tag{8.4}$$

where g_ϕ performs the attention crop on a batch of images, and α and β are step-size hyperparameters. Equation 8.3 finds the parameters $\theta^{'}$ that enable the controller to learn on the training data. Using $\theta^{'}$, Equation 8.4 updates the parameters of the Spatial Transformer using the validation data and a *double gradient*.

Note that the *meta-optimiser* (optimiser for meta-parameters) is SGD in this case - in principle, we could use any other optimiser for the parameter update in Equation 8.4. When updating the parameters θ, in contrast, we need to use SGD due to the form of Equation 8.3.

With this procedure, the ST learns to attend to the parts of the image that enable the controller to perform well on unseen data, provided the controller is trained with the training data. For that reason, the parameters θ of the controller are also updated using the data in D_{train}:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(f_\theta(g_\phi(x_{train})), y_{train}) \tag{8.5}$$

The full algorithm is shown in Algorithm 1. Notice we pre-train on the training data, rather than doing it simultaneously with the meta-update; we found in practice this gave better performance.

Moreover, we found that learning the parameters s_x, s_y alongside the translation parameters prevented the algorithm from learning suitable policies. We decided to manually set the scale of the attention crop, and simplified it further by setting $s_x = s_y$.

As a controller f_θ, we chose our CoordConv-based hard attention model. As a consequence, RGB images have to be concatenated with CoordConv maps before being inputted to our network. By cropping these coordinate maps through the spatial transformer, the controller obtains the cropped image concatenated with the coordinate maps associated with the cropped region. This allows us to simulate our CoordConv approach from Chapter 5, with the interest point being located by the localisation network.

Algorithm 1: MetaSTN

Input: Controller network f_θ
Input: STN network g_ϕ
Input: Demonstration dataset D, with D_{train} and D_{val} splits

Initialise parameters θ, ϕ
for *M epochs* **do**
  for *num training batches* **do**
    sample batch $(x_{train}, y_{train}) \sim D_{train}$
    $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(f_\theta(g_\phi(x_{train})), y_{train})$
  end
  for *num training batches* **do**
    sample batch $(x_{train}, y_{train}) \sim D_{train}$
    sample batch $(x_{val}, y_{val}) \sim D_{val}$
    $\theta' = \theta - \alpha \nabla_\theta \mathcal{L}(f_\theta(g_\phi(x_{train})), y_{train})$
    $\phi \leftarrow \phi - \beta \nabla_\phi \mathcal{L}(f_{\theta'}(g_\phi(x_{val})), y_{val})$
  end
end

Representation Learning for Localisation Network

We initially experimented with a scale of 0.5 (64×48) for the output grid and several convolutional architectures for the localisation network. We limited our search to architectures with 3 CNN layers and an MLP for regressing the transformation parameters.

We found that our algorithm consistently failed to crop around the target object, with crops often remaining static in the center of the image. Before training, as recommended by the paper, the output layer of the MLP was initialised to the identity transformation (with $t_x = 0, t_y = 0$). However, during training, the parameters of the localisation network were barely being updated. We suspected this could be due to an issue with the gradient flow, as STNs can be hard to train when downsampling is performed. As the original paper points out, the gradient flows only through the sampled points of the input feature map, and this process is heavily affected by the extreme locality of the bilinear sampling method.

To test whether this issue was due to our network failing to learn a suitable representation for the task, we switched the localisation network by a pretrained DSAE and an MLP. By using the spatial features obtained from the autoencoder, the MLP predicted the transformation parameters. With this initialisation, we found performance increased.

Linearised Multi-Sampling

As we mentioned in the previous section, bilinear sampling in STs performs badly under the presence of downsampling. A more recent paper presented an alternative sampling procedure called *Linearised Multi-Sampling* [63]. The authors show that this technique enables improved gradient flow in STNs, with a particularly beneficial effect when downsampling is applied.

We conducted a few experiments (see Section 8.2.4) and confirmed that indeed, for our case, this type of sampling performed better than bilinear sampling. With the exception of these experiments, we employ linearised sampling throughout the evaluation of the

MetaSTN algorithm.

Spatially Guided Initialisation

As a consequence of the gradient flow being restricted to sampled points, the initialisation of the transformation parameters (the location of the cropping window at the beginning of training) can have a non-negligible effect on the performance of the network.

In [64], the authors mention the same problem. Their setting considers videos in which an action is taking place. By applying a hard attention window with STs, they aimed to more accurately recognise which action was taking place. The researchers proposed addressing the initialisation problem by performing an optimisation procedure to initialise the regression layer of the localisation network. For their particular case, they compute the optical flow across a video, and initialise these parameters to the location in each frame with most motion. Then, the STN is trained normally.

In an approach similar to the authors', we initialise the crop location to a suitable location, using the heuristic locations (*spatially guided*) obtained from DSAEs in Section 6.3.1. The optimisation problem we solve before learning starts is:

$$\arg\min_{\psi} \mathcal{L}(loc_{\psi}(d(x_{train})), d(x_{train})_i) \tag{8.6}$$

where d is the DSAE used as in the localisation network, i is the index of the feature obtained from such DSAE following Section 6.3.1, and ψ are the parameters of the regression layer in the localisation network. In our implementation, we perform this guided initialisation for a fixed number of epochs, which we set to 10, and employ an MSE loss.

Scale Annealing

By using the heuristic feature for initialisation, we increase the training time. Furthermore, we cannot extend our approach easily to multiple attention locations, as we saw when we analysed the disadvantages of these heuristic methods in Section 6.3.1.

For that reason, we propose an alternative approach in which we anneal the scale throughout training. We initialise the scale to 1, and reduce it every epoch to $max(s_{target}, e^{-\gamma t})$, where s_{target} is the target scale, such as 0.5 for 64×48 crops, t is the current epoch and γ is an annealing rate. We set this rate to 2×10^{-2}. We also feed the current scale to the MLP in the localisation network. With this, we aim to guide the attention crop towards the relevant region of the image as training progresses, without relying on heuristic features.

8.2.4 Preliminary Results

For the preliminary experiments in this section, we use a 60%-30%-10% split of the data, where 60% is used to train the controller ($\mathcal{D}_{train}$), 30% for the meta-update (D_{val}) and the last 10% as a validation set for early stopping. We use SGD with momentum of 0.9 and weight decay of 10^{-4} for the meta-optimiser. We set learning rates of $\alpha = 10^{-1}$ and $\beta = 10^{-2}$ for the controller and localisation networks, respectively.

During training, if performance on the 10% set does not improve for 5 consecutive epochs, we halve the learning rate. We continue training until a patience of 10 is reached. For the scale annealing case, we only start recording the loss for early stopping once the scale has

reached the target scale.

Furthermore, at the end of the meta-learning procedure, we retrain the network normally, using both $\mathcal{D}_{train}$ and D_{val} as the training data.

Results for the same setup as previous chapters are shown in Table 8.1. We show results for scales of 0.5 and 0.25, with and without retraining. We find that both using pretrained

Scene 1			
Network	Scale	Sampling	Success rate
MetaSTN	0.5	Linearised	27.0±1.41/28.33±3.77
MetaSTN + DSAE features	0.5	Linearised	49.67±4.5/47.33±4.19
MetaSTN + DSAE features + Guided Init	0.5	Linearised	**68.67±6.24/66.0±4.97**
MetaSTN + DSAE features + Scale annealing	0.5	Linearised	29.0±8.64/44.0±2.16
MetaSTN	0.25	Linearised	11.67±1.25/12.33±2.62
MetaSTN + DSAE features	0.25	Linearised	21.67±4.92/32.0±2.83
MetaSTN + DSAE features + Guided Init	0.25	Bilinear	27.33±4.19/44.0±0.82
MetaSTN + DSAE features + Guided Init	0.25	Linearised	**35.0±15.56/55.33±17.75**
MetaSTN + DSAE features + Scale annealing	0.25	Bilinear	9.33±4.64/34.33±4.5
MetaSTN + DSAE features + Scale annealing	0.25	Linearised	15.0±3.27/32.67±4.92

Table 8.1: Performance on 100 trajectories after 29 steps for MetaSTN-based controllers. We show the success rate (a) just after meta learning and (b) retraining after the meta learning with an 80%-10%-10% split, in "(a)/(b)" format

spatial features and guided initialisation work much better than vanilla MetaSTN. Retraining seems to be beneficial too: performance after retraining is similar for a scale of 0.5, but improves for a smaller scale of 0.25. In addition, we notice that a smaller scale can lead to worse and unstable performance across replications, most likely due to the gradient flow problem described earlier.

We conducted experiments with guided initialisation and scale annealing, varying the type of sampling between bilinear and linearised. We observe that linearised sampling performs similarly or better than bilinear sampling for these cases. Moreover, although we aimed to address some of the disadvantages of guided initialisation with it, we find that scale annealing is unsuccessful at providing a better/similar initialisation.

We identify a potential problem with the guided initialisation version of the meta-learning algorithm: the localisation network could learn to predict the location of the crop always based on the spatial feature it was initialised with. If that were to happen, we would expect our meta-learning algorithm to perform similarly to the heuristic methods seen in Section 6.3.1. Since we employ a different data split for this experiment, we investigate whether this is an issue in the evaluation section (Chapter 9).

8.2.5 Visualisations

We visualise the attention window produced by the localisation network for D_{val} and validation demonstrations with a scale of 0.5. This can be computed as follows: we first obtain the trasformation matrix A_θ from the parameters predicted by the localisation network. Then, we find the source coordinates for target coordinates $[-1, -1]$, $[-1, 1]$, $[1, 1]$, $[1, -1]$ corresponding to the corners of the transformed image. Finally we use the source coordi-

nates to draw the crop on the input image.

In particular, we visualise the crops produced by vanilla MetaSTN in Figure 8.7 and MetaSTN with DSAE features and guided initialisation in Figure 8.8. We can see that vanilla MetaSTN does not find a suitable crop, with many remaining at the initialised location. For MetaSTN with guided initialisation, we observe that crops accurately track the target object.

We also show the instability in learning the attention crops at a smaller scale of 0.25. We show the same test demonstration in Figure 8.9 for two training replications of the same MetaSTN with guided initialisation. In the first case, we obtain good attention regions, whereas in the second one, crops do not track the target object properly.

Finally, we showcase our best meta-learning controller (DSAE and guided initialisation at scale 0.5) on a *test trajectory* in Figure 8.10.

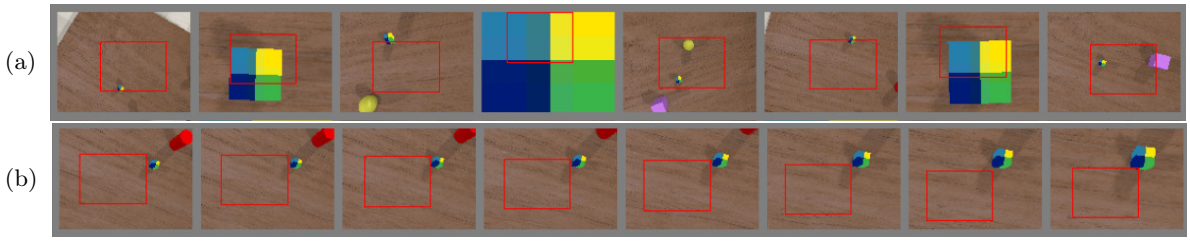


Figure 8.7: Attention crops at scale 0.5 for vanilla MetaSTN. We show (a) images from D_{val} only seen during the meta-update and (b) a validation test demonstration.

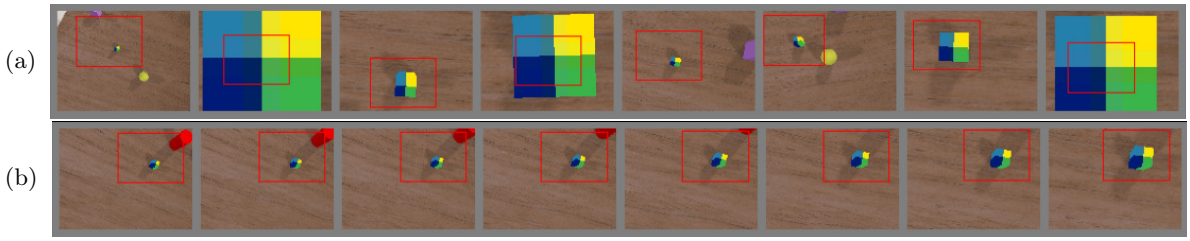


Figure 8.8: Attention crops at scale 0.5 for MetaSTN with DSAE features and guided initialisation. We show (a) images from D_{val} only seen during the meta-update and (b) a validation demonstration.

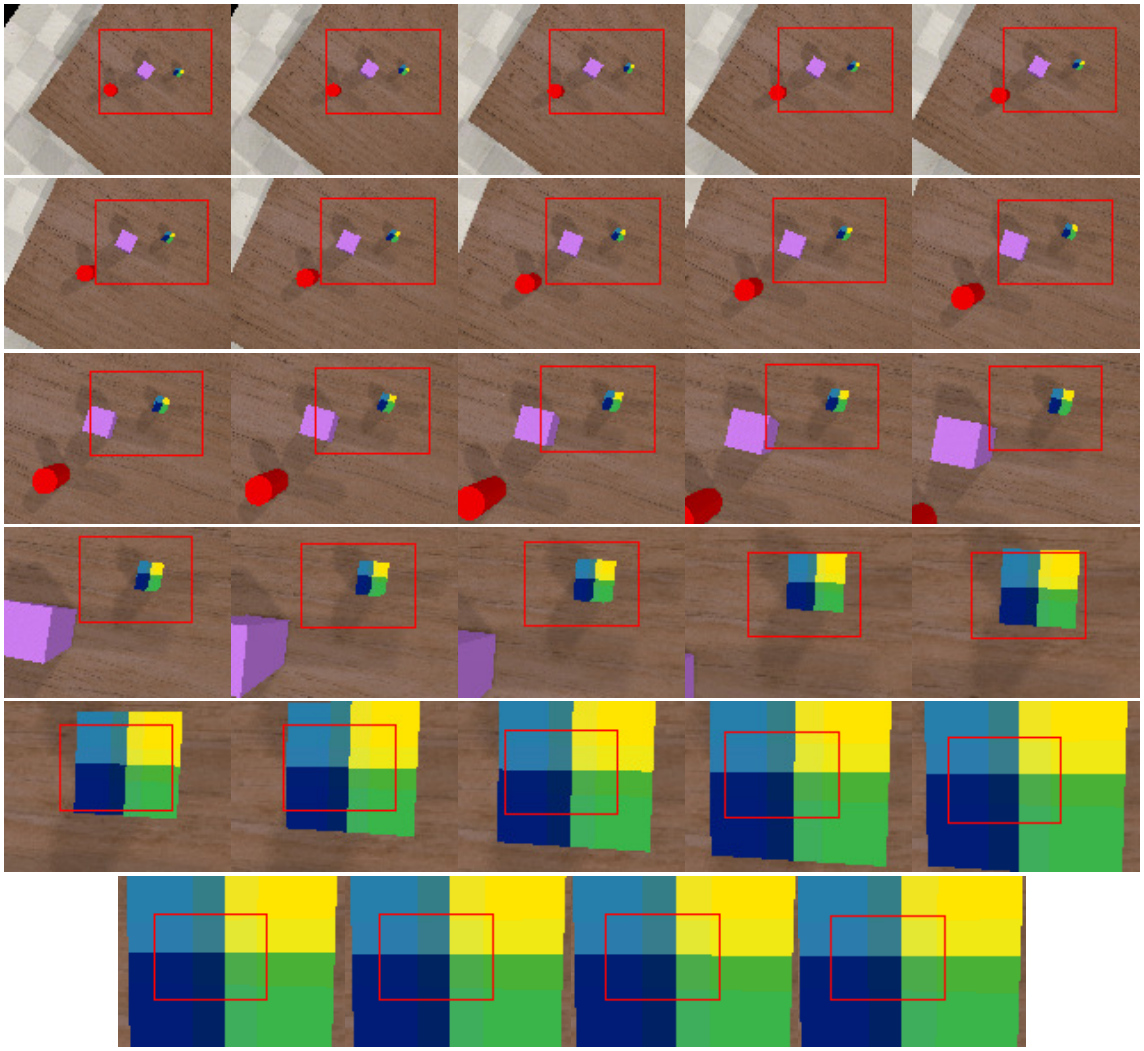


Figure 8.10: MetaSTN with DSAE features and guided initialisation on a successful test trajectory.

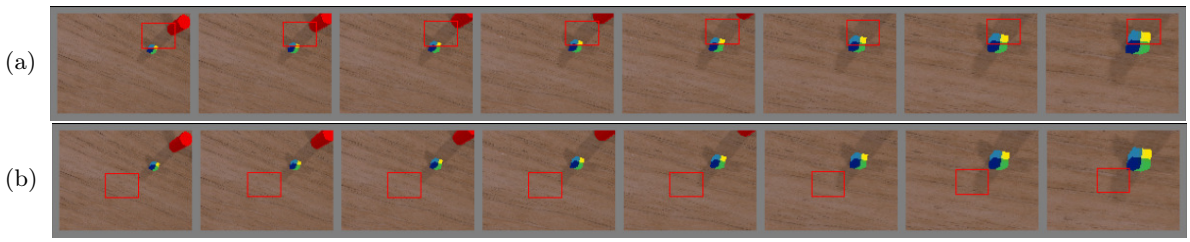


Figure 8.9: Training MetaSTN for small scales can be unstable: we show the same validation demonstration for (a) a replication that obtained good performance and (b) a replication that obtained bad performance. The attention crops were obtained at scale 0.25 for a MetaSTN with DSAE features, guided initialisation and no retrain. The network was trained with the same parameters/procedure in both cases.

Chapter 9

Evaluation

In this chapter, we evaluate the methods presented in this thesis across `Scenes 1-5`. In this way, we investigate the generalisation of our methods across multiple scenes. Unlike Chapters 5-8 where we evaluated the methods described in each chapter independently, we now seek to compare and obtain quantitative insights across the different proposed approaches and attention types.

To further this investigation, we analyse the performance of our methods in the same environments but with random distractors at test time. In addition, we explore the effect of training with random distractors when testing with and without such disturbances.

Then, we compare our best methods on a fine-grained robotic control task in a limited-sized dataset where a Sawyer robotic arm is directly operated. Finally, we discuss the main trade-offs of each method in terms of training/inference time and model capacity, and conclude by highlighting the strengths and limitations of each attentive model.

9.1 Training without Distractors

We evaluate our methods by training them on `Scenes 1-5`, as described in Section 4.3.1 with only the target object and no distractors during training time.

9.1.1 Results

We show the success rate in Table 9.1 for the methods in this thesis when trained without distractors and tested without distractors.

With regards to full image networks, we find that a full image network can perform relatively well in some scenes, namely `Scene 2`, `Scene 3` and `Scene 5`. In these cases, the full image network may learn to ignore irrelevant aspects of the image. Adding coordinate encodings or downsampling the input image do not provide a significant improvement, and in the case of downsampling, can perform much worse than the higher dimensional versions.

At the same time, we find that there is at least a hard attention variant that is better than the full image network in every scene, with `CoordConv` hard attention networks (at different window sizes) outperforming full image networks throughout all scenes. This implies that hard attention, when relevant parts of a scene are accurately located, can be very beneficial even when no randomness is introduced at test time. Moreover, it shows

that simple downscaling or spatial encodings on full images cannot counteract distribution drift as well as attending to relevant image locations.

Spatial feature-based controllers achieve the best performance on `Scene 3` and `Scene 4`. However, they fail to generalise as well in two other scenes. Heuristic methods do not necessarily perform really well, although we highlight that window size optimisation can compensate for error when choosing features to track. As an example, our heuristic method with a window size of 32×24 obtains 26.67% and 19.67% success rates in `Scene 3` and `Scene 5` respectively; with Bayesian window size optimisation 2.32x and 2.96x better performance is achieved, respectively.

With regards to recurrent attention networks, we find that `Soft Mask` versions perform quite badly, most likely due to the issues described in Section 7.3.2. Context vector-based networks perform better than their mask counterparts, although performance is not so consistent across scenes. This could be due to the fact that attention is computed based on full-sized inputs. Thus, the attention mechanism may be subject to the effect of distribution drift, leading to worse predictions from the downstream MLP predicting the action to take at a given time step.

As for recurrent baselines, we find that `Soft Context` networks perform better than them in all cases when no hard attention is considered. Nevertheless, the recurrent hard attention network only obtains a clearly better success rate in a single scene.

One of the potential issues we identified for the MetaSTN algorithm with DSAE features and guided initialisation was that it could rely only on the spatial features used for initialisation. We compare this method (using the better performing retrained version) with an attention window of 64×48 to the heuristic methods. We find that our meta-learning algorithm outperforms these heuristic variants in every scene, which rules out the described problem.

Additionally, we find that this same variant of MetaSTN performs extremely well in these settings, with its performance being very competitive to a CoordConv hard attention network with a 64×48 window. In `Scene 4`, our meta-learning algorithm even outperforms all hard attention variants, which is quite remarkable given that relevant image locations are learnt automatically using only a validation signal. Nonetheless, we do observe some instability in performance across replications, with high standard deviations in `Scenes 3-5`.

At a smaller scale of 0.25, with guided initialisation, DSAE and retraining, MetaSTN performs worse than the bigger scale version, although it does obtain a slightly higher success rate in `Scene 2`. The network achieves better performance than full image networks in 3 scenes, and surpasses the heuristic spatial localisation method for the same sized window in every scene.

For both scales, we confirm that annealing is not a successful initialisation strategy, as MetaSTN with DSAE features and no initialisation performs similarly across scenes and scales.

	Trained w/o dist., tested w/o dist.				
Method/Scenes	1	2	3	4	5
Relative Baseline	99.0±0.0	100.0±0.0	100.0±0.0	99.67±0.47	91.33±4.11
Full Image (128 × 96)	46.33±3.09	61.67±2.87	70.33±3.09	49.67±3.09	72.67±4.92
Full Image + Coord (128 × 96)	43.67±4.11	57.33±5.25	64.67±2.36	42.0±2.45	74.33±6.6
Full Image (64 × 48)	38.67±4.03	51.0±2.16	53.33±3.09	20.0±4.55	52.67±4.99
Full Image + Coord (64 × 48)	34.67±3.86	51.67±4.78	50.33±6.94	20.33±0.47	56.67±8.18
Full Image (32 × 24)	36.67±3.09	42.0±0.82	30.0±1.63	14.0±2.16	47.33±8.73
Full Image + Coord (32 × 24)	35.33±4.5	40.33±0.94	35.67±6.13	9.33±1.89	49.0±2.16
Attention V1 (64 × 48)	64.33±10.53	64.67±13.1	46.0±12.36	44.0±5.66	58.33±4.11
Attention V2 (64 × 48)	70.33±2.05	73.0±3.27	42.67±3.3	43.67±1.7	57.0±4.32
Attention Tile (64 × 48)	73.67±2.36	77.0±2.45	58.33±2.87	53.0±2.94	63.67±4.64
Attention Coord (64 × 48)	75.0±3.74	80.67±0.94	63.67±4.19	50.0±2.94	75.33±6.85
Attention V1 (32 × 24)	68.67±7.93	80.0±2.16	56.67±16.01	64.0±8.64	**82.0±2.16**
Attention V2 (32 × 24)	78.0±2.83	80.0±1.63	60.67±1.25	63.33±5.19	59.33±0.94
Attention Tile (32 × 24)	77.67±3.09	81.0±2.83	61.67±4.64	61.0±1.41	64.0±1.63
Attention Coord (32 × 24)	**80.67±5.56**	**86.67±1.89**	**71.0±0.82**	**68.0±0.82**	69.33±2.87
Spatial Keypoint Controller	**79.33±4.92**	57.0±4.55	**83.33±1.7**	**86.33±1.25**	**58.33±2.62**
Heuristic Spatial Loc. (64 × 48)	57.67±5.73	66.33±2.36	34.33±8.96	44.33±5.91	31.0±3.74
Heuristic Spatial Loc. (32 × 24)	55.0±3.56	48.67±1.89	26.67±1.7	51.67±4.92	19.67±2.49
Window Size Optimisation	54.67±7.41	**68.0±4.24**	63.0±12.75	48.67±1.7	58.33±4.78
Rec. Full Image	38.33±5.56	50.67±4.71	39.0±8.64	28.33±5.56	65.67±4.11
Rec. Full Image + Coord	40.67±8.18	46.67±5.25	33.0±2.16	25.0±3.56	55.33±9.39
Rec. Attention Coord (32 × 24)	**62.0±3.74**	56.33±8.73	**54.67±3.3**	42.0±6.98	50.67±7.36
Rec. Context Vectors	61.33±1.25	**70.67±2.05**	32.67±23.7	**50.0±5.89**	**72.33±7.32**
Rec. Mask	39.33±0.47	35.67±14.82	33.67±1.7	30.0±1.63	63.33±3.09
MetaSTN (64 × 48)	24.33±6.13	43.33±3.3	20.0±4.32	25.67±3.4	32.67±2.36
MetaSTN (64 × 48) R	32.67±5.91	48.0±4.55	33.67±1.25	30.0±0.82	55.33±1.89
+ DSAE (64 × 48)	35.0±0.82	28.33±8.96	23.67±2.49	26.33±5.73	43.33±0.94
+ DSAE (64 × 48) R	49.67±4.03	50.0±11.52	42.67±4.03	43.33±2.05	58.67±3.68
+ DSAE + Guide (64 × 48)	59.0±6.16	49.33±3.3	36.67±4.71	47.67±0.94	58.0±15.25
+ DSAE + Guide (64 × 48) R	**73.67±3.3**	79.0±3.74	**54.33±11.84**	**75.33±9.46**	**74.33±15.92**
+ DSAE + Scale (64 × 48)	32.33±5.44	35.67±7.76	18.67±2.49	20.33±4.5	31.67±1.7
+ DSAE + Scale (64 × 48) R	55.67±6.94	59.67±9.39	38.0±5.1	42.67±3.09	54.33±5.25
MetaSTN (32 × 24)	15.0±3.56	11.67±4.99	8.33±6.94	2.33±2.62	15.33±3.68
MetaSTN (32 × 24) R	28.33±7.59	31.67±11.59	34.0±8.83	10.67±6.24	42.67±1.7
+ DSAE (32 × 24)	26.33±8.99	3.33±1.25	7.0±0.82	11.0±3.74	18.67±4.99
+ DSAE (32 × 24) R	35.67±8.73	17.67±4.64	36.67±8.96	27.67±11.47	51.0±3.74
+ DSAE + Guide (32 × 24)	37.0±9.09	38.33±5.79	28.0±5.35	39.67±16.21	22.0±8.04
+ DSAE + Guide (32 × 24) R	66.67±0.47	**80.33±5.19**	44.33±2.87	73.0±22.11	44.67±1.7
+ DSAE + Scale (32 × 24)	22.0±4.97	8.67±6.13	5.0±1.63	8.67±1.25	15.0±7.79
+ DSAE + Scale (32 × 24) R	52.0±4.55	17.0±9.42	35.33±5.19	26.67±6.18	52.67±2.49

Table 9.1: Results for networks trained with **no distractors**, tested with **no distractors**. We show in bold the best method out of each of the hard, spatial, soft and meta attention approaches presented.

9.1.2 Random Distractors at Test Time

When injecting random distractors at test time, we would expect performance to drop across all models due to the shift between the training and test distributions. However, we would expect the degradation of performance to be less severe in attention variants when compared to full image networks. We show results in Table 9.2. Indeed, we observe that our hard attention variants outperform full image networks consistently across scenes, with the `CoordConv` variant being the most consistent.

Interestingly, we observe that spatial feature-based controllers can be quite sensitive to randomisation at test time when trained without such disturbances. In `Scene 3`, performance drops from 83.33% to 14%, although it must be noted that `Scene 3` has the highest number of distractors across all the scenes[1].

Methods dependent on DSAE features such as our MetaSTN methods also suffer from this issue - the success rate for MetaSTN with DSAE, guided initalisation and a 64×48 window (retrained) on `Scene 3` drops from 54.33% to 17.6%. Besides, when compared to full image networks and unlike in the previous experiment without distractors, this MetaSTN variant is outperformed by full image networks on `Scene 5`. Again, this is most likely due to the effect of test randomisation upon the underlying spatial feature encoder.

Recurrent networks also suffer from this variability, although not at the same level as spatial feature-based methods. Even though soft attention is used to find the most relevant part of the image, the attention computation itself relies on the full input image, which is perturbed due to the distractors, hence the impact on performance.

Nevertheless, context vector-based methods still outperform full image networks in every scene except for `Scene 5`. These results show that our best soft attention methods can work well when no randomisation is used during training time yet can be present at evaluation time.

[1] You may want to revisit Table 4.1, where we give the number of distractors per scene.

	Trained w/o dist., tested w/ dist.				
Method/Scenes	1	2	3	4	5
Relative Baseline	99.0±0.0	100.0±0.0	100.0±0.0	99.67±0.47	91.33±4.11
Full Image (128 × 96)	36.0±2.83	47.67±2.05	21.33±6.6	31.33±2.36	65.67±3.86
Full Image + Coord (128 × 96)	34.67±3.3	39.67±4.99	19.67±1.7	28.33±1.25	63.0±3.74
Full Image (64 × 48)	33.0±2.83	33.0±3.56	13.33±1.25	12.0±4.97	48.67±8.96
Full Image + Coord (64 × 48)	28.33±3.68	39.0±0.0	13.33±1.7	12.67±0.94	52.33±8.22
Full Image (32 × 24)	27.33±2.49	26.0±2.94	10.0±2.45	9.33±2.36	42.67±4.64
Full Image + Coord (32 × 24)	30.67±1.25	28.67±1.25	9.0±1.41	6.0±0.82	42.67±2.05
Attention V1 (64 × 48)	65.67±9.29	63.33±14.82	51.0±12.83	43.0±1.41	55.67±8.65
Attention V2 (64 × 48)	68.33±2.62	72.67±3.09	46.67±2.62	43.0±0.82	52.67±3.09
Attention Tile (64 × 48)	72.67±1.7	79.33±3.4	56.0±3.56	48.33±4.64	63.33±0.47
Attention Coord (64 × 48)	74.33±1.89	80.67±2.05	62.67±6.18	47.0±5.66	73.67±7.36
Attention V1 (32 × 24)	69.0±8.64	80.0±2.45	59.67±11.47	62.67±6.94	**79.67±2.05**
Attention V2 (32 × 24)	79.0±2.94	80.67±1.89	61.0±0.82	63.0±4.55	59.33±2.87
Attention Tile (32 × 24)	77.33±3.4	80.0±0.82	62.0±2.16	63.33±1.7	62.0±2.45
Attention Coord (32 × 24)	**80.0±4.55**	**85.33±1.7**	**72.0±3.27**	**67.33±0.47**	69.33±1.7
Spatial Keypoint Controller	**63.33±6.13**	40.33±2.05	14.0±2.45	**77.33±2.49**	43.67±3.3
Heuristic Spatial Loc. (64 × 48)	46.0±2.16	41.0±1.63	13.0±2.94	39.33±6.02	22.67±1.7
Heuristic Spatial Loc. (32 × 24)	41.33±5.56	29.33±4.11	10.33±2.05	49.0±2.83	17.67±4.64
Window Size Optimisation	43.67±4.5	**46.0±5.72**	**18.67±3.86**	36.0±2.94	**49.33±4.71**
Rec. Full Image	32.33±4.64	32.67±2.87	12.67±3.09	20.67±4.03	59.33±4.92
Rec. Full Image + Coord	34.33±9.39	37.33±6.13	11.67±0.47	16.33±2.05	49.67±6.13
Rec. Attention Coord (32 × 24)	**62.33±4.64**	53.33±6.6	**55.0±2.83**	**42.0±7.79**	49.0±4.9
Rec. Context Vectors	52.0±1.41	**55.0±2.16**	12.33±5.73	40.0±8.29	**67.67±6.02**
Rec. Mask	33.0±2.45	25.67±9.67	11.0±0.82	23.67±2.49	59.67±3.4
MetaSTN (64 × 48)	19.33±7.59	34.33±4.03	7.67±0.94	19.0±0.0	31.33±2.87
MetaSTN (64 × 48) R	26.67±6.65	36.0±4.97	14.67±3.09	23.33±2.05	47.33±2.49
+ DSAE (64 × 48)	25.67±2.05	23.33±8.06	12.33±1.7	22.33±5.56	37.0±4.24
+ DSAE (64 × 48) R	38.33±3.3	34.33±5.79	16.67±2.05	29.33±2.87	48.0±0.0
+ DSAE + Guide (64 × 48)	50.33±3.4	36.0±2.16	12.0±1.41	43.0±3.74	44.33±8.73
+ DSAE + Guide (64 × 48) R	**59.67±5.73**	**48.0±5.1**	17.67±4.19	65.33±10.87	48.67±3.86
+ DSAE + Scale (64 × 48)	26.67±4.19	30.33±0.47	11.33±2.62	17.67±2.62	29.67±0.94
+ DSAE + Scale (64 × 48) R	45.0±7.12	43.67±7.85	**18.33±1.7**	34.33±2.87	**49.67±4.19**
MetaSTN (32 × 24)	11.0±4.24	8.0±2.94	3.0±2.45	2.33±2.62	15.33±2.05
MetaSTN (32 × 24) R	23.0±5.35	27.0±10.23	13.33±2.05	6.67±2.36	37.33±4.78
+ DSAE (32 × 24)	20.0±7.35	7.0±2.16	6.33±1.25	9.33±2.62	17.67±4.03
+ DSAE (32 × 24) R	28.33±6.18	20.67±13.89	11.67±3.3	21.67±7.85	42.67±4.19
+ DSAE + Guide (32 × 24)	31.33±7.72	27.0±8.16	7.33±2.49	36.67±12.66	21.0±3.56
+ DSAE + Guide (32 × 24) R	53.0±3.56	49.0±3.74	15.67±1.25	**66.0±23.15**	35.33±0.47
+ DSAE + Scale (32 × 24)	17.33±6.8	5.33±3.77	4.0±0.82	5.33±2.05	13.33±4.92
+ DSAE + Scale (32 × 24) R	44.33±6.13	10.67±4.78	9.0±2.16	24.33±4.11	41.33±3.4

Table 9.2: Results for networks trained with **no distractors**, tested **with distractors**. We show in bold the best method out of each of the hard, spatial, soft and meta attention approaches presented.

9.2 Training with Distractors

To assess whether the above results are due to the lack of randomisation during training, we generate the same number of demonstrations for each of the scenes previously described containing randomly generated distractors. We then analyse whether including these when collecting demonstrations can lead to better performance when testing with and without these distractors.

9.2.1 Results

Table 9.3 shows our results for networks trained with random distractors and evaluated with random distractors too. In the previous setting, we hypothesised that full image networks were able to learn to concentrate on the target object in some scenes. In this case, in contrast, their performance drops even when the training and test distributions are generated in the same way. Hard attention networks still perform better than full image networks consistently, obtaining top results in `Scene 1` and `Scene 2`. Again, we validate that downsampling and coordinate feature maps do not help improve performance.

In the previous results, we observed that random distractions had a negative effect on the performance of spatial feature-based controllers. When trained with distractors, however, these controllers perform really well across `Scenes 1-4`, obtaining the best performance out of all methods in `Scene 3` and `Scene 4`. Although the success rate is only 56.67% in `Scene 5`, we find that this scene is also difficult for hard attention and recurrent variants, and this result is still better than the performance of the full image network.

Furthermore, we highlight that even when trained with random distractors, if too much randomness is present both during training and test time, networks may fail to learn an appropriate attention policy due to the limited size of the dataset. As an example, we show that even when trained with random distractors, `Soft Context` and `Soft Mask` recurrent networks completely fail on `Scene 3` (11 distractors).

Interestingly, the spatial feature-based controller is not affected by this. This phenomenon could be explained by the fact that the latter network employs a lower dimensional spatial feature representation for the input space. In contrast, the context vector we employ in this experiments has a dimensionality of 512, much bigger than the 128 features used in the autoencoder-based model. Further experiments including hyperparameter optimisation procedures could determine whether the performance difference is due to this factor.

Success rates for our MetaSTN networks with DSAE features, initialisation and retrained still perform better than the heuristic and recurrent attention methods. In scenes with a small to medium number of distractors, this method performs quite well. Impressively, our meta-learning algorithm obtains 81.0% and 84.33% success rates for 64×48 and 32×24 window sizes on `Scene 5`, respectively. These results make it the best method out of all the network variants, surpassing even hard attention networks.

Moreover, we find that for this particular scene, the results are consistent across replications. This suggests that this algorithm could benefit from demonstration collection with distractors, and that our proposed handcrafted attention locations may not necessarily be optimal and can be overcome with automatic methods.

	Trained w/ dist., tested w/ dist.				
Method/Scenes	1	2	3	4	5
Relative Baseline	99.0±0.82	99.33±0.94	100.0±0.0	100.0±0.0	99.0±0.0
Full Image (128 × 96)	29.67±1.25	48.33±2.62	31.33±3.09	29.67±1.89	54.33±1.25
Full Image + Coord (128 × 96)	34.33±2.05	47.67±1.25	27.0±5.72	22.67±3.3	56.67±7.59
Full Image (64 × 48)	32.0±3.56	29.67±2.49	20.33±3.86	15.33±1.7	40.0±5.72
Full Image + Coord (64 × 48)	27.0±3.74	27.67±4.64	15.33±2.62	11.33±0.94	45.0±6.53
Full Image (32 × 24)	24.33±4.03	25.33±3.68	9.33±1.89	6.67±2.05	29.0±0.82
Full Image + Coord (32 × 24)	19.0±1.41	24.33±3.3	10.0±2.45	6.0±1.63	33.33±6.6
Attention V1 (64 × 48)	75.0±2.16	72.0±8.29	58.67±2.62	42.0±7.87	61.33±3.77
Attention V2 (64 × 48)	62.0±3.74	65.33±1.25	46.33±0.47	40.67±2.05	44.67±3.3
Attention Tile (64 × 48)	65.67±3.09	74.0±0.0	49.0±4.24	37.67±1.7	53.33±6.94
Attention Coord (64 × 48)	74.67±4.5	75.33±1.25	**61.67±4.11**	42.33±3.3	68.67±10.53
Attention V1 (32 × 24)	70.67±4.99	66.0±3.74	48.0±3.56	52.33±15.17	67.0±8.98
Attention V2 (32 × 24)	83.33±1.25	73.0±2.16	61.0±4.08	54.67±1.7	54.0±2.16
Attention Tile (32 × 24)	**84.0±2.45**	**82.67±3.4**	58.67±3.4	56.0±7.26	55.0±1.41
Attention Coord (32 × 24)	83.33±6.8	80.0±3.56	54.33±13.3	**58.67±7.04**	**72.0±6.38**
Spatial Keypoint Controller	**72.67±2.49**	**81.33±2.05**	**75.67±0.47**	**85.67±2.36**	**56.67±3.68**
Heuristic Spatial Loc. (64 × 48)	47.0±4.97	43.67±4.11	25.0±9.63	26.33±3.86	23.33±1.25
Heuristic Spatial Loc. (32 × 24)	53.0±3.56	47.0±0.82	20.0±2.94	41.0±4.55	4.33±3.09
Window Size Optimisation	49.33±4.99	53.67±2.49	34.0±1.41	33.33±2.05	31.0±12.57
Rec. Full Image	28.0±2.16	28.33±5.73	18.0±6.16	14.67±2.49	48.0±11.34
Rec. Full Image + Coord	24.0±2.45	27.33±2.87	10.0±0.0	9.0±3.56	41.0±9.63
Rec. Attention Coord (32 × 24)	**59.33±4.03**	49.0±12.36	**52.33±6.85**	**44.0±5.66**	41.33±2.87
Rec. Context Vectors	47.33±6.65	**65.67±8.26**	18.0±13.37	43.67±1.25	42.33±5.79
Rec. Mask	33.67±9.03	41.67±4.03	20.0±2.94	14.33±1.7	**59.0±2.16**
MetaSTN (64 × 48)	26.33±11.67	23.0±6.16	10.0±0.82	13.33±3.4	32.0±5.1
MetaSTN (64 × 48) R	28.33±0.47	27.67±7.72	24.0±1.63	17.33±3.4	47.0±1.63
+ DSAE (64 × 48)	39.0±9.63	32.67±11.09	21.67±1.7	14.33±3.4	34.67±0.94
+ DSAE (64 × 48) R	50.33±3.86	43.0±4.32	**50.33±6.34**	26.33±4.11	44.33±3.4
+ DSAE + Guide (64 × 48)	51.67±2.87	41.67±3.77	31.67±3.3	37.33±12.26	66.67±4.03
+ DSAE + Guide (64 × 48) R	64.0±5.72	56.67±1.7	38.33±8.73	**53.67±21.08**	81.0±5.72
+ DSAE + Scale (64 × 48)	24.67±6.6	28.0±8.6	9.67±2.87	11.67±2.36	19.0±3.56
+ DSAE + Scale (64 × 48) R	47.0±6.16	45.33±13.67	50.33±6.85	22.33±4.03	52.33±4.03
MetaSTN (32 × 24)	7.33±3.3	11.0±1.63	1.33±0.94	4.67±2.05	10.67±3.86
MetaSTN (32 × 24) R	13.33±1.89	21.0±4.32	11.33±0.94	11.33±1.25	30.0±4.55
+ DSAE (32 × 24)	17.67±8.06	17.33±13.07	5.33±1.7	11.33±0.47	20.0±10.61
+ DSAE (32 × 24) R	37.67±2.36	45.33±13.3	33.0±17.15	27.0±2.45	44.33±5.91
+ DSAE + Guide (32 × 24)	41.67±1.7	35.33±1.25	19.0±4.32	18.33±13.42	66.33±4.03
+ DSAE + Guide (32 × 24) R	**67.33±3.86**	**63.33±13.02**	31.67±4.5	25.33±15.33	**84.33±6.02**
+ DSAE + Scale (32 × 24)	7.33±2.05	22.67±13.1	0.67±0.94	6.67±2.36	5.0±2.16
+ DSAE + Scale (32 × 24) R	23.67±14.38	58.67±17.56	26.0±10.68	15.67±2.05	36.0±3.56

Table 9.3: Results for networks trained **with distractors**, tested **with distractors**. We show in bold the best method out of each of the hard, spatial, soft and meta attention approaches presented.

9.2.2 Removing Distractors at Test Time

We now remove the random distractors at test time, and show results in Table 9.4.

Overall, we see performance improvements throughout, as expected since most sources of variability throughout a trajectory have been removed. Hard attention networks perform similarly or better, with CoordConv and V1 versions remaining very successful and outperforming full image networks in every scene.

Spatial feature-based controllers benefit from removing random distractors, improving upon the above results. In fact, they obtain the best results in `Scenes 2-4`, while remaining an automatic method that doesn't require handcrafted features, and performing consistently above full image variants.

Removing distractors does not affect significantly the performance of recurrent methods, which still fail to improve upon the results of full image networks in `Scene 3` and `Scene 5`. Indeed, we find that recurrent methods do not benefit greatly from the inclusion of random distractors during training.

Finally, we observe that, thanks to the learnt hard attention windows and the effective spatial features, meta-learning attention mechanisms obtain success rates comparable to when random distractors are used at test time. In addition, for a scale of 0.5, DSAE-based, guided models still outperform every full image network, and they remain the best method on `Scene 5` across all scales.

	Trained w/ dist., tested w/o dist.				
Method/Scenes	1	2	3	4	5
Relative Baseline	99.0±0.82	99.33±0.94	100.0±0.0	100.0±0.0	99.0±0.0
Full Image (128 × 96)	34.33±0.94	51.67±3.68	39.0±1.41	30.33±1.7	57.33±2.05
Full Image + Coord (128 × 96)	36.33±2.62	52.0±1.63	32.33±7.76	22.0±0.82	61.0±5.66
Full Image (64 × 48)	34.67±2.05	31.67±1.25	22.0±3.56	14.33±0.94	40.67±3.4
Full Image + Coord (64 × 48)	34.67±4.19	30.33±2.49	21.67±2.62	15.67±3.77	46.67±6.94
Full Image (32 × 24)	29.67±8.18	31.0±3.56	13.33±1.7	5.67±0.94	30.67±3.3
Full Image + Coord (32 × 24)	26.0±1.41	30.0±6.68	15.33±0.47	4.67±1.25	37.33±3.4
Attention V1 (64 × 48)	75.0±2.94	71.0±6.16	58.67±3.86	41.33±8.99	61.67±4.99
Attention V2 (64 × 48)	62.33±3.86	68.33±1.89	49.0±2.45	38.33±3.3	43.33±3.09
Attention Tile (64 × 48)	65.0±1.63	75.0±0.82	48.67±3.3	38.0±3.56	51.0±2.94
Attention Coord (64 × 48)	75.67±3.3	74.33±2.49	59.33±4.03	42.33±3.09	**72.33±9.03**
Attention V1 (32 × 24)	70.0±2.16	66.33±3.68	49.67±4.99	52.67±16.78	62.67±11.44
Attention V2 (32 × 24)	84.0±2.16	73.67±2.05	**62.33±5.31**	56.0±1.41	55.0±4.32
Attention Tile (32 × 24)	84.0±2.94	**82.0±4.97**	60.67±2.87	57.67±8.18	55.0±1.41
Attention Coord (32 × 24)	**84.33±6.24**	81.67±2.49	52.67±14.43	**58.0±6.68**	70.33±6.85
Spatial Keypoint Controller	**78.67±1.7**	**87.67±1.25**	**79.0±1.41**	**86.0±1.41**	**61.33±4.11**
Heuristic Spatial Loc. (64 × 48)	46.33±4.78	43.33±4.11	30.33±9.53	27.0±1.63	26.0±2.83
Heuristic Spatial Loc. (32 × 24)	55.67±3.68	47.67±2.49	22.67±1.7	38.33±3.86	4.67±3.68
Window Size Optimisation	53.33±5.79	51.67±1.25	41.33±1.7	32.0±8.29	31.0±14.45
Rec. Full Image	30.0±3.56	32.0±5.72	21.33±6.8	14.0±2.16	49.33±16.78
Rec. Full Image + Coord	28.0±4.55	27.67±3.68	14.67±0.47	10.67±4.78	41.0±10.03
Rec. Attention Coord (32 × 24)	**60.0±3.74**	50.0±12.08	**53.0±5.35**	**43.67±7.32**	43.0±3.56
Rec. Context Vectors	48.0±4.55	**69.67±7.59**	23.67±17.46	43.0±4.32	44.0±6.68
Rec. Mask	41.33±9.88	47.67±5.56	23.67±4.92	15.33±2.62	**59.0±4.32**
MetaSTN (64 × 48)	30.33±11.95	25.0±4.55	12.0±1.41	15.67±3.86	33.33±4.64
MetaSTN (64 × 48) R	29.0±4.32	27.0±4.08	26.67±2.05	20.33±2.05	47.0±7.07
+ DSAE (64 × 48)	44.33±7.36	31.67±7.72	25.67±0.47	18.67±4.64	34.33±2.05
+ DSAE (64 × 48) R	55.67±4.5	46.33±2.62	**54.33±7.72**	29.0±2.16	49.33±1.7
+ DSAE + Guide (64 × 48)	53.0±4.9	45.33±4.5	31.67±4.03	37.33±9.18	66.33±4.92
+ DSAE + Guide (64 × 48) R	66.0±4.32	59.67±2.62	42.0±7.35	**55.67±19.77**	82.0±6.16
+ DSAE + Scale (64 × 48)	27.0±8.52	30.67±10.08	14.0±2.45	14.33±8.26	24.0±3.27
+ DSAE + Scale (64 × 48) R	54.67±3.4	48.0±13.59	52.33±7.13	27.67±5.44	52.33±8.18
MetaSTN (32 × 24)	7.67±4.11	10.0±2.16	2.33±2.05	4.33±0.47	13.67±4.64
MetaSTN (32 × 24) R	17.33±3.3	21.67±3.68	17.0±2.83	14.33±1.7	30.67±0.94
+ DSAE (32 × 24)	19.33±6.18	19.67±14.38	6.0±1.63	11.67±1.7	19.0±9.27
+ DSAE (32 × 24) R	40.67±2.49	45.67±12.5	36.0±18.4	28.0±3.27	47.0±0.82
+ DSAE + Guide (32 × 24)	43.67±2.05	38.67±1.25	18.33±4.03	16.33±14.61	65.67±4.64
+ DSAE + Guide (32 × 24) R	**67.67±2.62**	**63.33±11.95**	34.33±3.4	26.0±16.27	**85.67±4.03**
+ DSAE + Scale (32 × 24)	8.67±1.25	23.33±14.82	2.0±1.41	6.0±2.94	8.33±4.71
+ DSAE + Scale (32 × 24) R	26.33±16.78	60.33±15.97	30.0±10.61	15.0±2.83	40.67±2.05

Table 9.4: Results for networks trained **with distractors**, tested with **no distractors**. We show in bold the best method out of each of the hard, spatial, soft and meta attention approaches presented.

9.3 Controlling a Robotic Arm

We extend our evaluation to a robotic control task where we operate a Sawyer arm directly, rather than controlling a camera. We generate a scene where the task is to control a disc and insert it into a peg. The peg remains static on the table, but the robotic arm may start from different configurations, with different camera viewpoints. Figure 9.1 shows the scene in simulation.

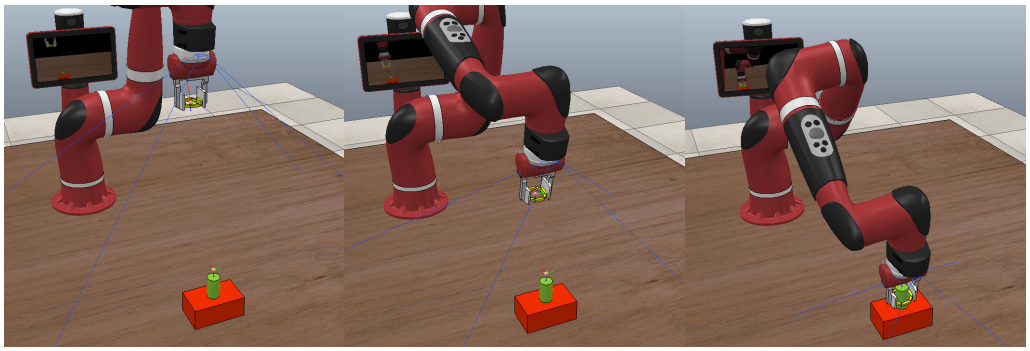


Figure 9.1: Scene for a robotic control task, where the aim of the robot is to insert the grasped yellow disc into the green peg. Images shown are at different stages of a sample trajectory.

We generate two *limited-sized* datasets (with and without random distractors) for this task by applying a reduced position and orientation offset, much like in the generation process detailed in Section 4.3.1. In particular, the scene contains four distractor objects, and the datasets with and without random distractors each contain 39 demonstrations with 1280 and 1279 images, respectively. This equates to 64 seconds of demonstrations when obtained with a 20Hz controller. Even considering setup time in real life, collecting 39 demonstrations is feasible in the real world. We show some sample trajectories for each dataset in Appendix A.

We also show the dimensions of the peg and the disc in Figure 9.2. We take a test trajectory to be successful if the center point of the disc (shown in green in the middle of the disc in the figure) reaches the success area while remaining inserted into the peg. In addition, similarly to previous scenes, we take a fixed number of steps in the environment - in this case 33, as this is the average length of demonstrations in both datasets.

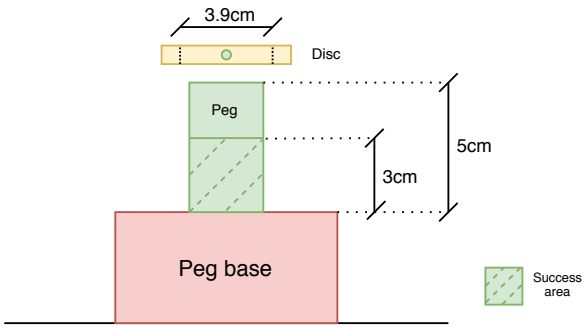


Figure 9.2: Peg and disc dimensions (not to scale).

When generating demonstrations, we operate the robot to reach a centered point 2cm above the peg base (within the success area) using inverse kinematics. For that reason, we use the projection of this point as the relevant pixel position for hard attention networks.

Results

We trained our best methods from the previous section on both datasets and evaluated them with and without random distractors, performing the same variability analysis. In the same line, we use the same architectures and hyperparameters as in our previous evaluation experiments. Results are shown in Table 9.5 averaged across three replications.

	Trained w/o dist.		Trained w/ dist.	
Method	Test w/o	Test w/	Test w/o	Test w/
Full Image (128 × 96)	73.67±3.09	67.67±5.56	70.0±4.55	64.33±5.31
Attention Coord (64 × 48)	83.67±2.49	**83.67±2.87**	81.0±7.79	79.67±6.94
Attention Coord (32 × 24)	83.0±4.32	82.33±4.99	94.0±4.32	**93.67±4.19**
Spatial Keypoint-based Controller	85.67±4.19	53.67±1.7	**96.0±1.41**	91.33±0.47
Recurrent Weighted Context Vectors	**86.0±5.35**	72.33±7.93	88.33±6.8	83.67±5.44
MetaSTN + DSAE + Guiding (64 × 48) R	76.67±11.12	62.67±4.92	82.0±3.74	82.0±4.55
MetaSTN + DSAE + Guiding (32 × 24) R	53.0±17.72	42.33±10.34	61.67±19.91	61.0±17.91

Table 9.5: Success rate for our best methods on a disc insertion robotic control task. We show results when trained with and without distractors and evaluated with and without this randomisation.

Our results confirm most of the conclusions we arrived to from the earlier experiments on `Scenes 1-5`. Firstly, we again observe the smallest scale MetaSTN variant fails to learn good cropping transformations, most likely due to the gradient flow and downsampling limitations of the method.

We confirm that spatial keypoint-based controllers are extremely sensitive to test time distractors when trained without them, as we previously mentioned, with the impact of these distractors reducing their success rate from 85.67% to 53.67%. Due to its inherent dependence on spatial features, the MetaSTN variant at scale 0.5 also fails to outperform the full image network in that setting.

Notwithstanding these, we find that **our automatic attentive methods surpass full image network performance in every variability scenario**. In particular, out of all the *automatic* methods, recurrent context vector-based soft attention networks perform most consistently.

We also revalidate the beneficial effect of randomised training for spatial keypoint-based controllers. As examples, when trained with randomised distractors they obtain an excellent success rate of 96% with no disturbances injected at test time. These results surpass even those of our best handcrafted hard attention methods, which score 81% and 94%. When tested with distractors, they obtain a success rate of 91.33% which translates into 27 more successful trajectories on average than a full-sized input model.

While *non-automatic* hard attention methods also outperform the baseline full image network, we find that our proposed methods can also surpass their performance. As examples,

spatial keypoint and recurrent methods outperform hard attention networks at both scales in the first setting in the table. Additionally, these two methods also obtain higher success rates than the 64×48 hard attention network when trained and tested with random distractors.

All in all, we find that our attentive variants perform much better than full-sized input networks, demonstrating the benefits of attention for distribution drift mitigation in robotic control tasks.

9.4 Discussion

In this section, we discuss the trade-offs of each approach in terms of parameter capacity and training/inference times. To conclude our evaluation, we consider the advantages and disadvantages of each of our presented attention mechanisms.

9.4.1 Capacity and Training/Inference Time

We show capacity, training and inference times for each of the methods presented in this thesis in Table 9.6.

All networks were trained on the Department of Computing *ray* cluster, whose 26 machines are fitted with NVIDIA GTX 1080 GPUs. We report training times for the training procedure of experiments shown in Sections 9.1 and 9.2, with dataset sizes of ~ 5250 images and ~ 150 demonstrations. Moreover, training times were averaged across several runs on multiple machines to iron out the impact of other users training their deep learning models simultaneously. For inference times, we report the average inference time per image on this GPU model, computed over three runs.

Note that we do not perform a thorough analysis of nor do we attempt to exhaustively optimise for training/inference time. Nonetheless, we do take advantage of the limited size of our datasets in the following ways:

- All hard attention methods in Chapter 5 leverage predefined crop locations. To accelerate training time, we cache the crops in system memory before starting the training procedure.
- When training DSAEs through our modified objective, we cache the dataset images in system memory.
- Spatial Keypoint-based controllers employ spatial features to regress actions. Before training, these features are pre-computed and cached in system memory. A similar technique is applied in heuristic methods.
- Images are also cached in system memory for all recurrent methods in Chapter 7 and MetaSTN methods in Section 8.2.

Training times are similar for full image, hard attention and recurrent methods. Note that for full image networks, we did not cache the training data. Since hard attention models "cheat" and require privileged data, we practically consider recurrent models to be the most effective if short training times are required.

Spatial feature-based and meta-learning methods require up to two orders of magnitude larger training times, which make them less suitable if quick experimentation is required or if they are trained as a component within a more complex controller. Nevertheless, they may obtain better performance as we saw in Sections 9.1, 9.2 and 9.3. Furthermore, excluding some heuristic methods, less than two hours are required to train these networks. This time is minimal compared to the training times of reinforcement learning agents, and results in a good trade-off between performance and training time.

Looking at the results in the table, we do not observe major differences in inference times between the various methods. All proposed controllers can run a forward pass through the network for a single image in much less than 50ms, showing that these networks can be incorporated into 20Hz controllers.

The number of parameters required by each method varies greatly, especially when spatial features are employed. Although increasing the number of parameters can sometimes improve performance, throughout this thesis we aimed to compare methods as fairly as possible.

As examples, we employed the same CNN and MLP architectures for all hard attention networks in Chapter 5. Both proposed recurrent attention networks have similar number of parameters with most of the capacity assigned to determining the attention probabilities and the same CNN architecture as the hard attention networks. In addition, in the interest of comparability the same MLP architecture as the hard attention variants is used to predict actions in the recurrent networks.

With respect to methods employing spatial features, we used 64-feature DSAEs trained with the same procedure for all networks, where most of the parameters in these networks are found (1.063M). Again, the same hard attention network architectures from Chapter 5 are used throughout heuristic approaches and MetaSTN variants.

9.4.2 Strengths and Limitations

We now briefly cover the main strengths and limitations of the presented approaches and recap some insights obtained throughout this evaluation.

Hard Attention Networks

The main limitation of hard attention models is that they rely on handcrafted positions of interest. These are specified by a human and/or obtained from a simulator, and thus would not be available at test time in the real world. Additionally, we require a fixed sized window, whose size is specified and not learnt automatically.

In spite of this, they are conceptually simple and performance of these methods consistently outperforms full image networks, with CoordConv-based methods performing best. Moreover, specifying the size of this window can be easily done based on the task at hand. In any case, we observe improvements in performance for a variety of window sizes.

Spatial Feature-based Attention

Strengths of spatial keypoint-based methods include that they obtain a relevant low-dimensional representation of the scene, and work really well in practice. However, training

times can be an issue, even with relatively small-sized datasets. Despite struggling when trained without randomisation and tested with variability in the environment, our experimental evaluation demonstrates that they can obtain the best performance out of all our automatic methods when trained in randomised environments. Empirically, this holds both with and without random distractors injected at test time.

Heuristic spatial methods learn hard attention policies heuristically, without requiring privileged information. Although Bayesian optimisation addresses the limitation of non-optimal window sizes across the dataset, performance can be quite variable. Training times are the longest out of all the attempted methods and these models empirically do not generalise as well as other methods.

Recurrent Attention

Soft attention methods learn attention that is not constrained by a specific window size, as opposed to hard attention variants. However, since recurrence is used, the effect of distribution drift may impact performance of the controller more prominently. Nevertheless, we find that our context vector method can perform more consistently than spatial methods when random disturbances are included at test time but not in the training data. Besides, these approaches lend themselves really well to visualisations, which can help better understand what the model is focusing on.

Meta-learning attention

Finally, MetaSTN learn hard attention window locations automatically, addressing the main limitation of hard attention models in Chapter 5. A disadvantage of these methods, however, is that performance can depend on the generalisation of the spatial feature encoder. Despite that, we observed that employing spatial features often leads to good generalisation, and therefore deem this limitation to be minor. Furthermore, the benefits of choosing smaller window sizes can be overshadowed by the training difficulty, with some instability during training replications. Notwithstanding this, MetaSTN methods with spatial features and guided initialisation outperform full image networks across most tasks, and remove the need to employ more costly hard attention optimisation techniques such as reinforcement learning.

Summary and Key Insights

Overall, out of all automatic methods we find that spatial feature-based methods such as spatial keypoint controllers are the best when variability is included in the training data. Therefore, they are most suitable to tasks where obtaining varied demonstrations with random disturbances is feasible.

MetaSTN approaches also show promising results for large enough scales, such as 0.5. However, for both MetaSTN and spatial feature-based methods, the increased training times may render them impractical for some applications. In addition, if demonstrations are collected without random distractors and test environments are likely to experience variability, these two methods should be avoided due to empirically-observed subpar performance.

Context vector-based soft attention networks may be used in such cases, as their training

times are comparatively shorter, although this may result in less successful controllers. In addition, for settings where interpretability of the model decisions is required, these soft attention networks may be preferred.

Chapter	Method	Training time (min)	Inference time (ms)	# parameters
Chapter 5	Relative Baseline	2	1	5K
	Full Image (128×96)	7	6	266K
	Full Image + Coord (128×96)	8	5	269K
	Full Image (64×48)	7	6	147K
	Full Image + Coord (64×48)	7	5	151K
	Full Image (32×24)	7	7	125K
	Full Image + Coord (32×24)	7	3	128K
	Attention V1 (64×48)	4	9	147K
	Attention V2 (64×48)	2	10	146K
	Attention Tile (64×48)	2	8	148K
	Attention Coord (64×48)	3	3	151K
	Attention V1 (32×24)	2	9	125K
	Attention V2 (32×24)	2	10	125K
	Attention Tile (32×24)	2	9	126K
	Attention Coord (32×24)	2	4	128K
Chapter 6	DSAE (64 f.)	37	6	1.063M
	Spatial Keypoint-based Controller	40	9	1.076M
	Heuristic Spatial Location (64×48)	79	9	1.214M
	Heuristic Spatial Location (32×24)	79	10	1.191M
	Window Size Optimisation	147	(variable)	(variable)
Chapter 7	Recurrent Full Image	3	7	860K
	Recurrent Full Image + Coord	3	9	863K
	Recurrent Attention Coord (32×24)	2	11	132K
	Recurrent Weighted Context Vectors	7	5	3.694M
	Recurrent Mask	4	5	3.691M
Chapter 8	MetaSTN (64×48)	11	9	279K
	MetaSTN + DSAE (64×48)	66	10	1.226M
	MetaSTN + DSAE + Guiding (64×48)	110	7	1.226M
	MetaSTN + DSAE + Scale annealing (64×48)	67	10	1.226M
	MetaSTN (32×24)	8	9	256K
	MetaSTN + DSAE (32×24)	65	8	1.204M
	MetaSTN + DSAE + Guiding (32×24)	99	8	1.204M
	MetaSTN + DSAE + Scale annealing (32×24)	74	8	1.204M

Table 9.6: Average training/inference time and number of parameters for methods and components presented in this thesis. Training and inference times are reported rounded to the nearest minute and millisecond, respectively. Wherever methods are composed of various components, the reported training time includes the time required to train every component from scratch.

Chapter 10

Conclusion

Throughout this thesis, we have considered a variety of attention methods. Our main aim was to ascertain whether distribution drift in *imitation learning* can be mitigated via the use of attention, as we hypothesised. During this investigation, we proposed a number of techniques to incorporate attention into behavioural cloning controllers trained with limited-sized datasets.

We started by proposing and analysing hard attention models based on predefined locations of interest. We found that, by leveraging information not available at test time in the real world, a controller can learn with up to 8x less data and obtain much better performance on control tasks than non-attentive networks. Moreover, we showed the performance superiority of CoordConv-based hard attention encodings among multiple variants.

Then, we explored three different paradigms to learn these locations of interest *automatically*. Firstly, we devised a modification to the training procedure of deep spatial autoencoders to obtain a successful spatial keypoint based controller. We proposed a heuristic method based on spatial features to determine hard attention locations, and addressed some of its shortcomings with a Bayesian optimisation mechanism.

Secondly, we developed two soft attention methods that exploited recurrence to include a temporal component into their attention mechanisms, with the additional benefit of obtaining insightful visualisations as a side-effect of their design.

Thirdly, we explored meta-learning techniques by developing two algorithms for automatic hard attention window localisation. We showed that reinforcement learning techniques may not be able to solve this task, and proposed MetaSTN, a novel optimisation-based meta-learning approach based on spatial transformer networks. We highlighted the difficulty of training spatial transformers, and proposed feature and window initialisation techniques to compensate for their gradient flow limitation.

To conclude, we evaluated and compared these methods across an extensive variety of robotic control tasks with different degrees of variability in the environment. Most importantly, we showed that attentive models outperformed full-sized input networks across every evaluated task, scene and variability scenario. We then discussed the inherent trade-offs, strengths and weaknesses of each proposed attention technique to motivate the choice of any such method.

Overall, through this thesis we demonstrated that visual attention can be a powerful component in neural network controllers for robotic tasks, and showed that attention can reduce the performance impact of environment-induced distribution drift on behavioural cloning-based models.

10.1 Future Work

We now propose a number of extensions to the work undertaken in this thesis.

10.1.1 Combining Hard and Soft Attention

In Chapters 5 and 7 we explored different imitation learning models with hard and soft attention, respectively. Nevertheless, this thesis does not address whether applying a soft attention mechanism on a lower-dimensional hard attention window can be beneficial. In the same line, hard attention window locations may be inferable from soft attention masks.

10.1.2 Gumbel-Softmax

Our recurrent soft attention methods employ a softmax activation to compute attention probabilities, from which a context vector is obtained, and element-wise sigmoid activations to compute a soft binary mask. In either of these, due to the *soft* aspect of the attention mechanisms employed, irrelevant data on the image may not be ignored during computation of the context vector/soft mask. Indeed, this can be considered a compromise between ease of training through differentiability and accuracy.

Nonetheless, alternatives beyond spatial transformers and reinforcement learning to compute hard attention masks do exist. Introduced in [65], the Gumbel-Softmax continuous distribution is a distribution that can approximate samples from the categorical distribution. In [66], Jang et al. introduce a way of replacing categorical samples (i.e. a hard mask) with Gumbel-Softmax samples, and provide a gradient estimator for the parameters of this distribution via a reparameterisation trick. Effectively, this allows learning of hard attention masks in a differentiable manner. Rather than applying a softmax activation, we could apply such trick to produce a discrete mask, much like our soft mask approach.

Unfortunately, we learnt about this technique late in the project, and even though we implemented an initial version of this trick in our recurrent variants, time constraints prevented us from employing it and evaluating its effect successfully. An interesting line of research would therefore be to assess whether learning this hard attention mask via the Gumbel-Softmax trick can obtain better performance, with a particular emphasis on robotic tasks. While this trick has been utilised successfully in a variety of papers such as [67], [68] or [69], to the best of our knowledge it is yet to be investigated within the robotics domain.

10.1.3 Effect of Number of Demonstrations

We provided an initial analysis of the effect of the amount of training data on the performance of hard attention models. However, due to time constraints, it was infeasible to perform such analysis for the remaining methods across all our scenes (with and without distractors). Given that demonstration collection in the real world can be a difficult and cumbersome process, methods whose performance degrades gracefully as the number of

demonstration decreases may be preferred. This choice could thus be motivated by such analysis.

10.1.4 Multiple Attention Locations

Some of the techniques discussed throughout this thesis are limited to attending to single objects/parts in an image, such as MetaSTN with guided initialisation or the heuristic methods from Section 6.3.1. Further work could attempt to address these limitations, or even explore how to determine the number of attention locations automatically.

10.1.5 MetaSTN Extensions

In line with the previous point, we proposed two initialisation variants for MetaSTN, with different degrees of success. Other initialisation methods may be more suitable, and could perhaps yield better performance. In particular, we observed performance of our best MetaSTN variants depended heavily on the generalisation of the underlying DSAE model. Additional work could be undertaken in order to improve spatial feature based representations. Furthermore, as we have seen, Spatial Transformers are hard to train - other alternatives, such as saliency-based sampling [70] could be considered, evaluated and compared against the presented approaches.

Appendix A

Sample Generated Dataset Demonstrations

A.1 Scene 1

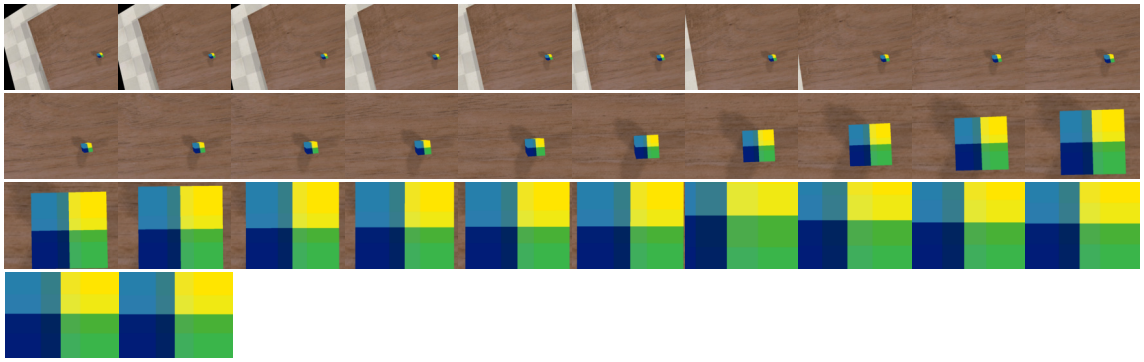


Figure A.1: Demonstration from `Scene 1` without distractors.

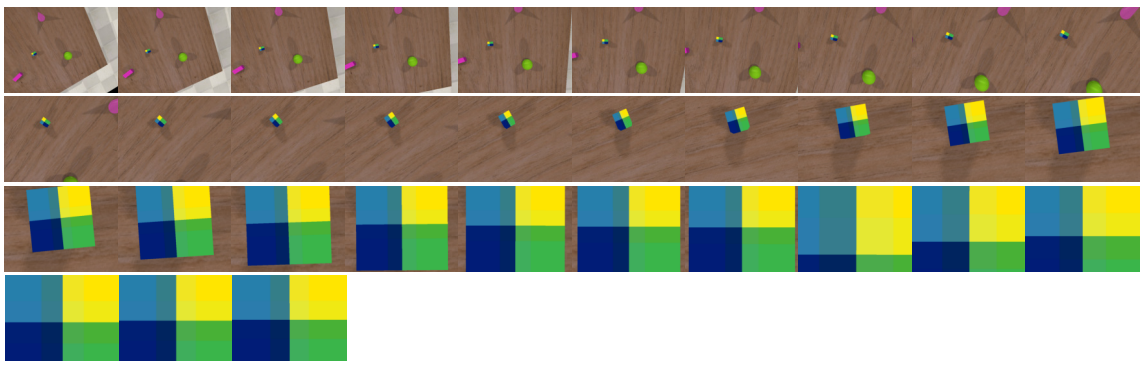


Figure A.2: Demonstration from `Scene 1` with random distractors.

A.2 Scene 2

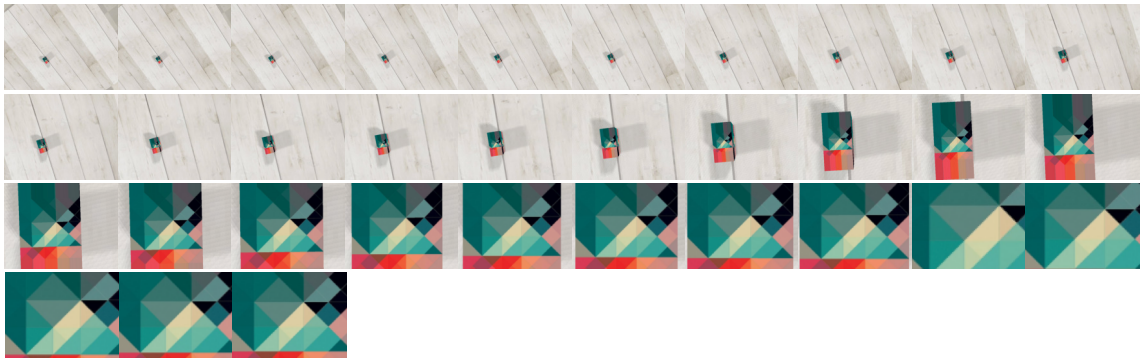


Figure A.3: Demonstration from `Scene 2` without distractors.

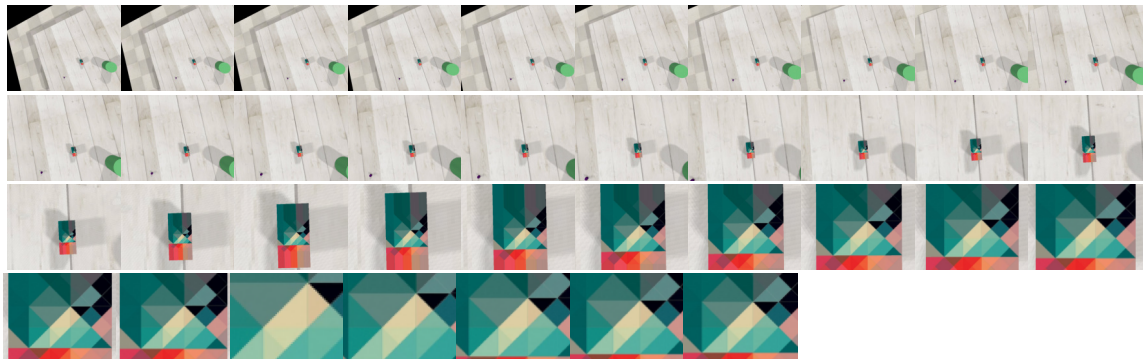


Figure A.4: Demonstration from `Scene 2` with random distractors.

A.3 Scene 3

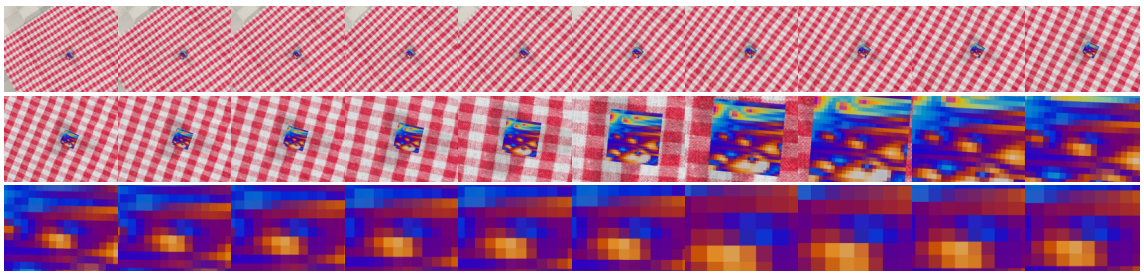


Figure A.5: Demonstration from `Scene 3` without distractors.

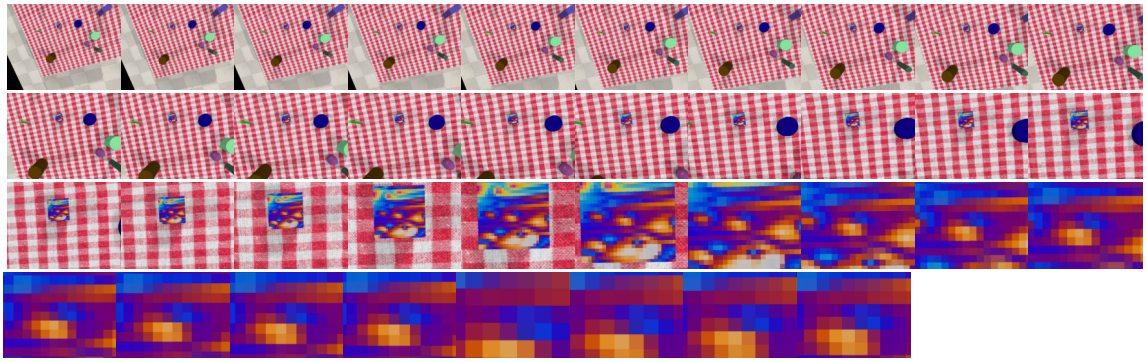


Figure A.6: Demonstration from `Scene 3` with random distractors.

A.4 Scene 4

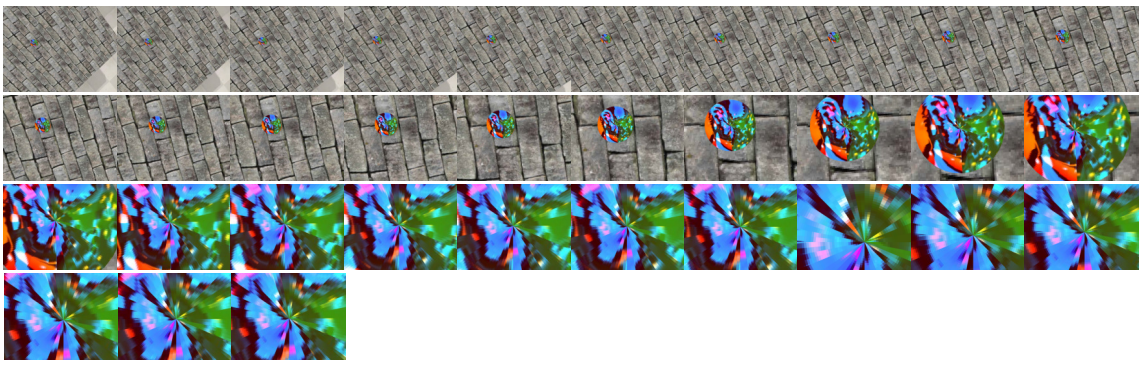


Figure A.7: Demonstration from `Scene 4` without distractors.

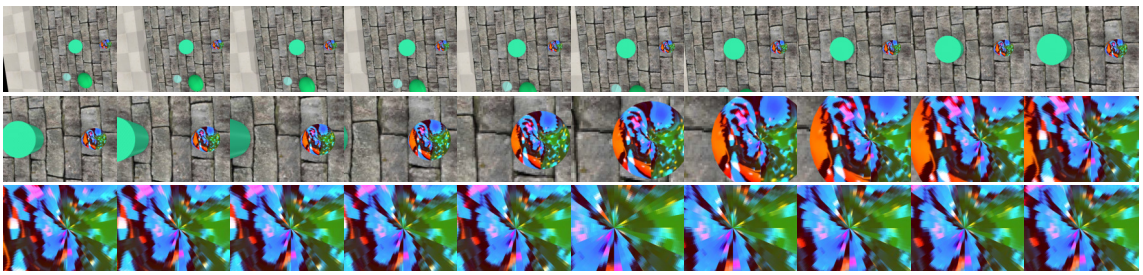


Figure A.8: Demonstration from `Scene 4` with random distractors.

A.5 Scene 5

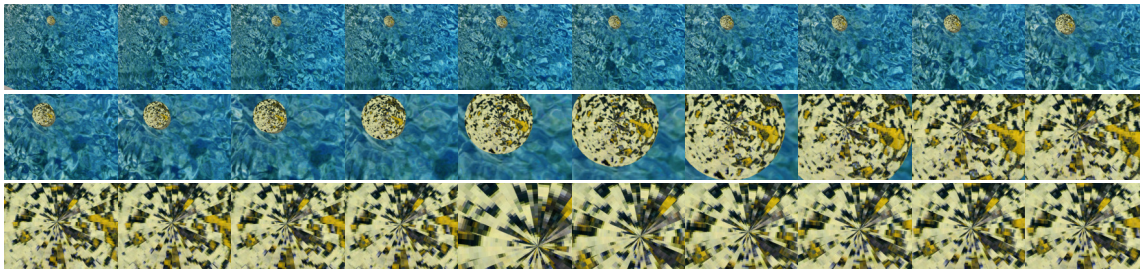


Figure A.9: Demonstration from `Scene 5` without distractors.

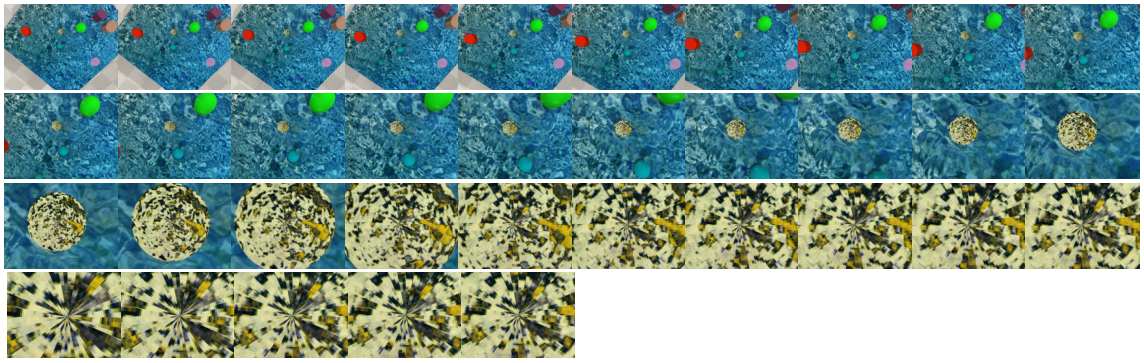


Figure A.10: Demonstration from `Scene 5` with random distractors.

A.6 Disc Insertion Task

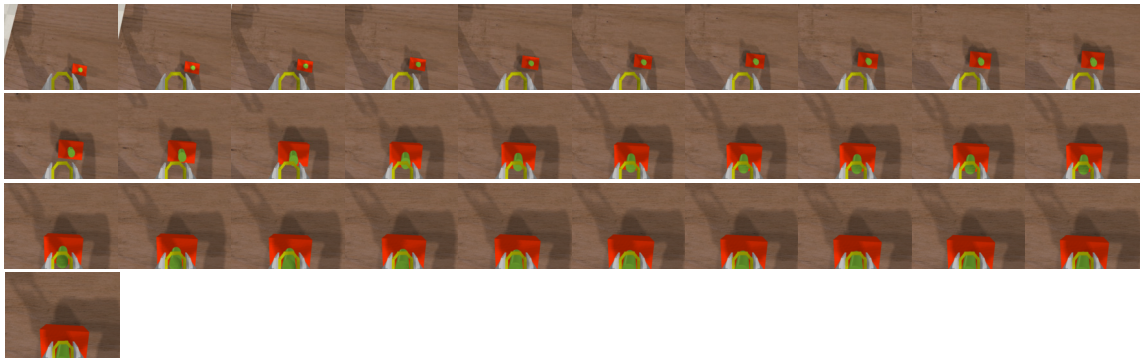


Figure A.11: Demonstration from the disc insertion task without distractors.

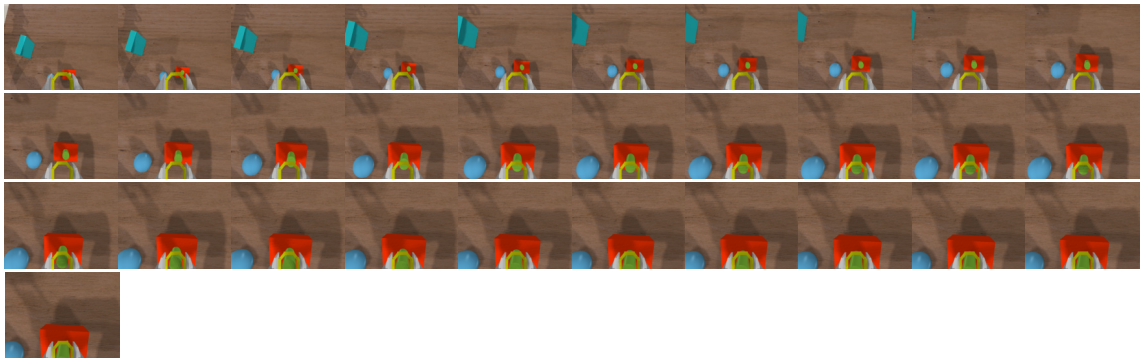


Figure A.12: Demonstration from the disc insertion task with random distractors.

Bibliography

[1] Jonathan Tilley. Automation, robotics, and the factory of the future. *McKinsey Company*. https://www.mckinsey.com/business-functions/operations/our-insights/automation-robotics-and-the-factory-of-the-future. Accessed: 2020-01-16.

[2] History. *iRobot*. https://www.irobot.com/about-irobot/company-information/history. Accessed: 2020-01-16.

[3] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations. *arXiv e-prints*, page arXiv:1709.10087, Sep 2017.

[4] Coline Devin, Pieter Abbeel, Trevor Darrell, and Sergey Levine. Deep Object-Centric Representations for Generalizable Robot Learning. *arXiv e-prints*, page arXiv:1708.04225, Aug 2017.

[5] Tianhao Zhang, Zoe McCarthy, Owen Jow, Dennis Lee, Xi Chen, Ken Goldberg, and Pieter Abbeel. Deep Imitation Learning for Complex Manipulation Tasks from Virtual Reality Teleoperation. *arXiv e-prints*, page arXiv:1710.04615, Oct 2017.

[6] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Deep Spatial Autoencoders for Visuomotor Learning. *arXiv e-prints*, page arXiv:1509.06113, Sep 2015.

[7] Mel Vecerik, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin Riedmiller. Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards. *arXiv e-prints*, page arXiv:1707.08817, Jul 2017.

[8] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep Q-learning from Demonstrations. *arXiv e-prints*, page arXiv:1704.03732, Apr 2017.

[9] Yuke Zhu, Ziyu Wang, Josh Merel, Andrei Rusu, Tom Erez, Serkan Cabi, Saran Tunyasuvunakool, János Kramár, Raia Hadsell, Nando de Freitas, and Nicolas Heess. Reinforcement and Imitation Learning for Diverse Visuomotor Skills. *arXiv e-prints*, page arXiv:1802.09564, Feb 2018.

[10] Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Overcoming Exploration in Reinforcement Learning with Demonstrations. *arXiv e-prints*, page arXiv:1709.10089, Sep 2017.

[11] Vinicius G. Goecks, Gregory M. Gremillion, Vernon J. Lawhern, John Valasek, and Nicholas R. Waytowich. Integrating Behavior Cloning and Reinforcement Learning for Improved Performance in Sparse Reward Environments. *arXiv e-prints*, page arXiv:1910.04281, Oct 2019.

[12] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *arXiv e-prints*, page arXiv:1502.03044, Feb 2015.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, Dec 2014.

[15] Missinglink.ai. *Convolution operation*. Missinglink, 2019. https://missinglink.ai/wp-content/uploads/2019/03/Frame-2.png. Accessed: 2020-06-14.

[16] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, page arXiv:1502.03167, Feb 2015.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[18] 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, January 2016.

[19] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

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

[21] 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, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[22] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[23] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989.

[24] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[25] Edward Johns. CO424 Reinforcement Learning: Part 2, Lecture 3: Policy Gradients, Imperial College London, November 2019.

[26] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21(4):682–697, 2008.

[27] Sergey Levine. Actor Critic Algorithms, CS 294-112: Deep Reinforcement Learning, UC Berkeley, Fall 2017.

[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv e-prints*, page arXiv:1707.06347, Jul 2017.

[29] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv e-prints*, page arXiv:1801.01290, Jan 2018.

[30] Brian D. Ziebart. *Modeling Purposeful Adaptive Behavior with the Principle of Maximum Causal Entropy*. PhD thesis, Carnegie Mellon University, USA, 2010.

[31] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. Soft Actor-Critic Algorithms and Applications. *arXiv e-prints*, page arXiv:1812.05905, Dec 2018.

[32] OpenAI. Soft Actor-Critic. https://spinningup.openai.com/en/latest/algorithms/sac.html. Accessed: 2020-01-03.

[33] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[34] Timothy Hospedales, Antreas Antoniou, Paul Micaelli, and Amos Storkey. Meta-Learning in Neural Networks: A Survey. *arXiv e-prints*, page arXiv:2004.05439, April 2020.

[35] Jake Snell, Kevin Swersky, and Richard S. Zemel. Prototypical Networks for Few-shot Learning. *arXiv e-prints*, page arXiv:1703.05175, March 2017.

[36] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. *arXiv e-prints*, page arXiv:1703.03400, March 2017.

[37] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. *arXiv e-prints*, page arXiv:1011.0686, Nov 2010.

[38] Timothy P. Lillicrap, Jonathan J. Hunt, Alexand er Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv e-prints*, page arXiv:1509.02971, Sep 2015.

[39] Maurizio Corbetta and Gordon L Shulman. Control of goal-directed and stimulus-driven attention in the brain. *Nature reviews neuroscience*, 3(3):201–215, 2002.

[40] Quanzeng You, Hailin Jin, Zhaowen Wang, Chen Fang, and Jiebo Luo. Image Captioning with Semantic Attention. *arXiv e-prints*, page arXiv:1603.03925, Mar 2016.

[41] Zichao Yang, Xiaodong He, Jianfeng Gao, Li Deng, and Alex Smola. Stacked Attention Networks for Image Question Answering. *arXiv e-prints*, page arXiv:1511.02274, Nov 2015.

[42] D. Yu, J. Fu, T. Mei, and Y. Rui. Multi-level attention networks for visual question answering. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4187–4195, July 2017.

[43] Sajad Mousavi, Michael Schukat, Enda Howley, Ali Borji, and Nasser Mozayani. Learning to predict where to look in interactive environments using deep recurrent q-learning. *arXiv e-prints*, page arXiv:1612.05753, Dec 2016.

[44] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent models of visual attention. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2204–2212. Curran Associates, Inc., 2014.

[45] Pooya Abolghasemi, Amir Mazaheri, Mubarak Shah, and Ladislau Bölöni. Pay attention! - Robustifying a Deep Visuomotor Policy through Task-Focused Attention. *arXiv e-prints*, page arXiv:1809.10093, Sep 2018.

[46] E. Rohmer, S. P. N. Singh, and M. Freese. Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013. www.coppeliarobotics.com.

[47] Stephen James, Marc Freese, and Andrew J. Davison. Pyrep: Bringing v-rep to deep robot learning. *arXiv preprint arXiv:1906.11176*, 2019.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv e-prints*, page arXiv:1912.01703, December 2019.

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

[50] François Chollet et al. Keras. https://keras.io, 2015.

[51] Chelsea Finn and Sergey Levine. Deep Visual Foresight for Planning Robot Motion. *arXiv e-prints*, page arXiv:1610.00696, October 2016.

[52] Rosanne Liu, Joel Lehman, Piero Molino, Felipe Petroski Such, Eric Frank, Alex Sergeev, and Jason Yosinski. An Intriguing Failing of Convolutional Neural Networks and the CoordConv Solution. *arXiv e-prints*, page arXiv:1807.03247, July 2018.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[54] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020.

[55] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *arXiv e-prints*, page arXiv:1711.05101, November 2017.

[56] Peter I. Frazier. A Tutorial on Bayesian Optimization. *arXiv e-prints*, page arXiv:1807.02811, July 2018.

[57] Eduardo C. Garrido-Merchán and Daniel Hernández-Lobato. Dealing with Integer-valued Variables in Bayesian Optimization with Gaussian Processes. *arXiv e-prints*, page arXiv:1706.03673, June 2017.

[58] Ivan Sorokin, Alexey Seleznev, Mikhail Pavlov, Aleksandr Fedorov, and Anastasiia Ignateva. Deep Attention Recurrent Q-Network. *arXiv e-prints*, page arXiv:1512.01693, December 2015.

[59] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[60] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

[61] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial Transformer Networks. *arXiv e-prints*, page arXiv:1506.02025, June 2015.

[62] Shikun Liu, Andrew Davison, and Edward Johns. Self-supervised generalisation with meta auxiliary learning. In *Advances in Neural Information Processing Systems*, pages 1677–1687, 2019.

[63] Wei Jiang, Weiwei Sun, Andrea Tagliasacchi, Eduard Trulls, and Kwang Moo Yi. Linearized Multi-Sampling for Differentiable Image Transformation. *arXiv e-prints*, page arXiv:1901.07124, January 2019.

[64] Dichao Liu., Yu Wang., and Jien Kato. Supervised spatial transformer networks for attention learning in fine-grained action recognition. In *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 4 VISAPP: VISAPP,*, pages 311–318. INSTICC, SciTePress, 2019.

[65] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. *arXiv e-prints*, page arXiv:1611.00712, November 2016.

[66] Eric Jang, Shixiang Gu, and Ben Poole. Categorical Reparameterization with Gumbel-Softmax. *arXiv e-prints*, page arXiv:1611.01144, November 2016.

[67] Chen Shen, Guo-Jun Qi, Rongxin Jiang, Zhongming Jin, Hongwei Yong, Yaowu Chen, and Xian-Sheng Hua. Sharp Attention Network via Adaptive Sampling for Person Re-identification. *arXiv e-prints*, page arXiv:1805.02336, May 2018.

[68] Shiyang Yan, Jeremy S. Smith, Wenjin Lu, and Bailing Zhang. Hierarchical Multi-scale Attention Networks for Action Recognition. *arXiv e-prints*, page arXiv:1708.07590, August 2017.

[69] Caglar Gulcehre, Sarath Chandar, and Yoshua Bengio. Memory Augmented Neural Networks with Wormhole Connections. *arXiv e-prints*, page arXiv:1701.08718, January 2017.

[70] Adrià Recasens, Petr Kellnhofer, Simon Stent, Wojciech Matusik, and Antonio Torralba. Learning to Zoom: a Saliency-Based Sampling Layer for Neural Networks. *arXiv e-prints*, page arXiv:1809.03355, September 2018.