IMAGINATION-AUGMENTED DEEP REINFORCEMENT LEARNING FOR ROBOTIC APPLICATIONS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2022

Student id: 10379350

Department of Computer Science

Contents

Word Count: 24396

List of Abbreviations

A3C asynchronous advantage actor-critic. 44

AI artificial intelligence. 8, 15, 16, 17, 18, 44

BMDN Bayesian mixture density network. 8, 69, 70, 71, 74, 75, 101

BNN Bayesian neural network. 25, 29, 30, 47

CNN convolutional neural network. 25, 28, 29, 41

DDPG deep deterministic policy gradient. 42, 44

DL deep learning. 16, 17, 19, 25, 26, 41, 46, 63, 69

DQN deep Q-network. 8, 41, 42, 43, 44, 45, 56, 61, 87, 90, 91, 93, 94, 97, 99, 100, 102

DRL deep reinforcement learning. 3, 8, 19, 25, 40, 41, 43, 44, 47, 63, 80

ELBO evidence lower bound. 31

HRI human-robot interaction. 3, 24, 43, 44, 45, 47, 52, 53, 61

IK inverse kinematics. 7, 8, 73, 75, 79

KL Kullback-Liebler. 30, 31, 33, 85, 86

LSTM long short-term memory. 44, 45, 103

MC-dropout Monte Carlo dropout. 46, 47, 72, 78, 89, 101, 102

MCMC Markov chain Monte Carlo. 30

MCTS Monte Carlo tree search. 43

MDN mixture density network. 3, 24, 25, 33, 34, 45, 48, 52, 58, 59, 61, 62, 63, 64, 69, 72, 73, 75, 78, 87, 88, 89, 100, 101, 102, 103

MDP Markov decision process. 35

ML machine learning. 16, 17, 25

MLP multi-layer perceptron. 16

MPC model-predictive control. 38

MSE mean squared error. 56, 87

MVE model-based value expansion. 46

NN neural network. 16, 17, 25, 26, 27, 28, 29, 30, 31, 33, 41, 46, 47, 72, 73, 78

PPO proximal policy optimization. 44, 45

RL reinforcement learning. 3, 8, 19, 20, 21, 22, 23, 24, 25, 34, 35, 36, 37, 38, 39, 40, 41, 43, 44, 45, 47, 49, 50, 52, 54, 62, 63, 80, 81, 83, 99, 100, 102, 103

TD temporal differencing. 46

VAE variational autoencoder. 8, 25, 31, 45, 50, 51, 55, 57, 59, 61, 85, 86, 96, 100

List of Tables

List of Figures

Abstract

IMAGINATION-AUGMENTED DEEP REINFORCEMENT LEARNING FOR ROBOTIC APPLICATIONS
Mohammad Thabet
A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy, 2022

Deep reinforcement learning (RL) has recently emerged as a powerful technique that allows agents to solve complex sequential tasks. Its application in the field of robotics however has been held back by the impracticality of the enormous amount of interaction data it requires. Collecting data with a physical robot is usually prohibitively costly, prompting the need for more sample-efficient reinforcement learning algorithms. This thesis aims to develop an architecture that drastically improves the sample efficiency of RL and leads to faster learning with less data. The architecture incorporates a mechanism that mimics imagination in humans into model-free learning, allowing agents to simulate scenarios internally to lessen the need for actual interaction with the environment. In this model-assisted setting, an agent learns a stochastic environment model on-line from experience simultaneously with a policy. A variational autoencoder (VAE) is used to compress visual input into abstract representations in the latent space, and a mixture density network (MDN) is used to learn a forward model in this latent space. The agent then uses the learned model to generate imaginary rollouts to augment real data, which is then used to train a controller in an RL context. Uncertainty in the model predictions is estimated using Monte-Carlo dropout to limit the use of imaginary data, preventing the agent from using erroneous model predictions in learning. The thesis presents experiments involving human-robot interaction scenarios in an RL setting to verify the viability of the approach. The first experiment serves as

a proof of concept and involved an agent learning a pick-and-place task based on gestures by a human. The second experiment was designed to demonstrate the advantages of the approach and involved a robot learning to solve a puzzle based on gestures. Results show that the proposed imagination-augmented agents perform significantly better than baseline agents when data is scarce, proving the efficacy of the approach in increasing sample efficiency.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First and foremost, I would like to wholeheartedly thank my supervisor, Professor Angelo Cangelosi, for all the tremendous help he has been. His unending support, in matters academic or otherwise, has been crucial for the success of this undertaking. I will forever be grateful to him for the opportunities he has provided me. I would also like to thank my second supervisor, Professor Jon Shapiro, for his insightful input that helped shape my work.

I would like to thank Doctor Massimiliano Patacchiola for collaborating with me on the work relating to reinforcement learning. I have learned a lot from his excellent blog on reinforcement learning when I first started, and he has always been willing to help and provide his input to my work. I would also like to extend my gratitude to Professor Gavin Brown, head of the machine learning and optimization (MLO) group at the University of Manchester, for his invaluable input on my work on uncertainty estimation. He taught me to be a lot more precise in my writing, which also helped me think more clearly.

Most of the funding for the PhD came from the Horizon 2020 Marie Skłodowska-Curie Innovative Training Networks scheme as part of the SECURE project. I am grateful to have been the recipient of such a generous grant that allowed me to solely focus on research.

Last, but certainly not least, I would like to thank my mother, to whom I dedicate this thesis. She taught me the most important things of all: kindness, perseverance, responsibility, and conscientiousness. Any success I have had or will ever have is because she is my mother.

Chapter 1

Introduction

During the twentieth century, public imagination was rife with dreams of intelligent robots carrying out all sorts of manual labor. From robot factory workers tirelessly churning out commodities to robot butlers pampering their owners, optimism about practical robotics was abound. Rapid advances in technology and understanding of computing and related sciences seemed to herald a new age of technological wonder full of automatons. In 1956, the Dartmouth Summer Research Project on Artificial Intelligence was held in what is widely regarded as the birth moment of artificial intelligence (AI) as a field of study [Buc05]. The nascent field promised to accelerate the development of truly thinking machines, and by the 1960's leading experts such as Marvin Minsky and Herbert Simon were prophesizing that machines will be able to do anything that humans can do within a couple of decades [Cre93]. And yet, several decades later, general-purpose intelligent robots seem further away than it did back then.

These unrealistically high expectations about AI and robotics seem to be an instance of the Dunning-Kruger effect [Dun12] taking hold of the public en masse. The scientific community was just starting to understand computing and neuroscience, and the prevalent understanding back then was that high-level conscious processes such as logical reasoning was almost all that there is to intelligence. Therefore, it seemed logical that we can approximate human intelligence by programming logic into computers, which seemed rather straightforward to do. Of course, things turned out to be a lot more complicated than that.

By the 1980's, researchers started to realize that they had grossly underestimated the challenges of AI. It became apparent that higher-level reasoning was quite easy for computers, and in fact it was sensorimotor skills what were extremely difficult.

This ran in direct opposition to previously held beliefs, and the incongruity came to be known as *Moravec's paradox* [Mor88]. Indeed, without the ability to efficiently represent sensory information and carry out motor actions, there would be nothing for higher-level logical processes to manipulate. Furthermore, researchers came to understand that the embodiment of an agent is integral to its cognitive capacity, and that higher level reasoning cannot be achieved in disembodied isolation (embodied cognition). All these realizations conspired to spell the end of interest in the top-down symbolic approach to AI, and focus began to shift towards bottom-up approaches instead.

The interest in bottom-up AI began to gain traction in the 1990's, aided by the exponential increase in computational power. Machine learning (machine learning (ML)) broke out of AI as a separate field, focusing on applying statistical and probabilistic methods to learn from data [Lan11]. For robotics, that meant trying to build robots that can learn simple sensorimotor skills from raw sensory data instead of symbolically programming them to do certain actions.

A particular class of ML models suitable for bottom-up learning is the artificial neural network (NN). NNs are inspired by how brains work; they consist of a network of simple computational units, called *neurons*, connected to each other by *weights*. Modern NNs have their origins in the 1950's when Rosenblatt introduced *perceptrons* [Ros58] which consisted of a single layer of neurons for binary classification. Soon after, perceptrons were hailed as the way to achieving true artificial intelligence [Ola96]. However, a sobering moment finally came when it was discovered that basic perceptrons are unable to learn certain classes of tasks such as the exclusive-or (XOR) binary function [MP69]. Furthermore, even though it was pointed out that multi-layer perceptron (MLP) can learn such tasks, there was no known training algorithm for MLPs, and there was not enough computational power available at the time to effectively handle training of such models. As a result, research on NNs stagnated until it was revived in the early 1980's when Hopfield networks were introduced that could effectively serve as associative memories [Hop82]. The field then gained further momentum when the *backpropagation* algorithm that could effectively train MLPs was rediscovered [RHW86].

Success of NNs and the interest in bottom-up AI led to the development of deep learning (DL) as a distinct field of machine learning in the 2000's. DL models are NNs with many layers of neurons, and are usually trained on raw data. Layers can learn progressively more abstract internal representations of the data through training

and thus require no engineered features to learn. This makes the learning *end-to-end*, meaning that the model learns directly from one end, raw data, all the way to the other end, required outputs, without any additional processing.

The 2010's saw DL dominating the field of AI so much that the terms AI and DL are often (erroneously) used interchangeably. This was sparked by the incredible success of DL models in image classification, natural language processing, and speech recognition. Algorithmic and architectural innovations, as well as the ever increasing computational power and the seemingly endless stream of data on the internet, allowed DL models to outperform other ML techniques in these areas. The victory in the popular ImageNet image classification contest in 2012 by Krizhevsky et al. [KSH12] is often regarded as what sparked what was dubbed *the deep learning revolution*.

The history of AI is summarized in Figure 1.1. The figure shows the frequencies of AI-related terms as found in printed sources between 1950 and 2019 in Google's English text corpora. Frequency plots are juxtaposed with historical events that are widely believed to have had the most impact on interest in the field. It is interesting to observe how these events have reflected on the popularity of specific terms in the literature. We notice that prior to the Dartmouth workshop in 1956, AI has been virtually non-existent as a field. A rise in interest in perceptrons soon followed after Rosenblatt's work in 1958, only to completely die out after Minsky and Papert show their severe limitations in 1969. An inflection point in AI is found at around 1980 when the first commercially successful expert systems and Lisp machines start appearing. The introduction of Hopfield networks ignites interest in neural networks, which is further intensified after the rediscovery of backpropagation. The Lisp machine market collapse of 1987 is quickly followed by a sharp decline in the interest in AI. Neural networks keep gaining momentum however, but it soon starts to slowly fade due to lack of breakthroughs. The introduction of support vector machines (SVMs) with the kernel trick in 1992 [BGV92] and their immense success kept interest in machine learning afloat, but severely affected the perceived effectiveness of NNs as a practical alternative. This changed in 2012 however, when huge successes such as the AlexNet Imagenet victory kickstarted the deep learning revolution. It is interesting to note that around this time, the frequencies of the terms AI and NN start to become almost identical, perhaps reflecting the fact that DL made NNs dominate so much that they became almost synonymous with AI.

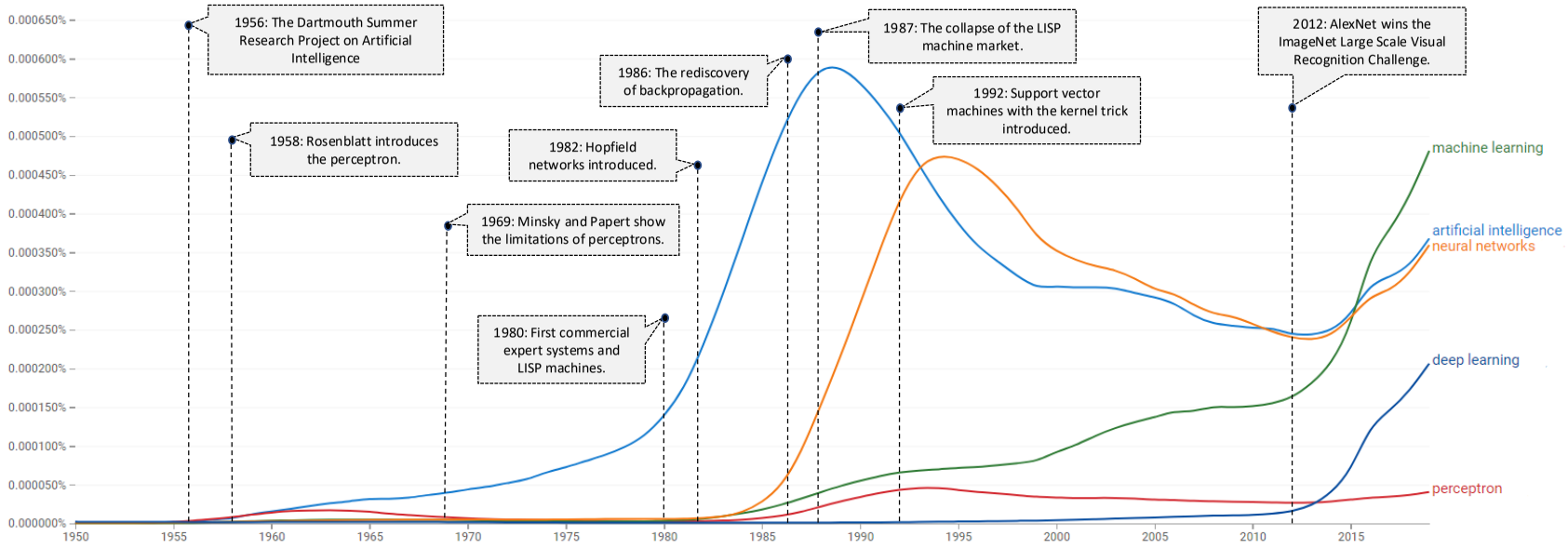


Figure 1.1: Frequencies of specific AI-related terms as found in Google's English text corpora between 1950 and 2019. Frequency data was obtained using the Google Ngram Viewer. Events believed to have significantly influenced the shape of the curves are highlighted.

1.1 Reinforcement Learning and Robotics

reinforcement learning (RL) is a branch of machine learning concerned with agents learning optimal policies through interaction with the environment. It can be viewed as an algorithmic formulation of how animals learn by trial and error. In RL, an agent performs an action on the environment which changes its state in response. The agent then receives a reward signal associated with the new environment state. This reward signal is then used by the agent to discern bad actions from good ones. The goal of the agent is to accumulate the most reward through the entire episode of interaction with the environment. Naturally, since RL mimics animal learning, it was extensively used in robotics research to teach robots to perform tasks [KBP13].

The deep learning revolution of the past decade led to the development of deep reinforcement learning (DRL), which incorporates the end-to-end paradigm of DL into RL. DRL systems comprise a neural network to process data and select actions, and they often learn policies from raw data such as a camera feed. DRL has found huge success teaching agents to play video games from raw pixels [MKS+13]. Consequently, there has been renewed interest in applying this variant of RL to robotics. The problem however is that DRL requires copious amounts of data to work. This is easy when dealing with computer games since you can cheaply run large amounts of simulations in little time. However, for physical systems this is very expensive. Robots will take a lot of time to perform millions of physical actions, and wear and tear will pose a significant problem.

To address this issue, there has been a lot of research on making DRL algorithms more *sample-efficient* (i.e. to make them achieve comparable performance but with significantly less data). Approaches to this end include transfer learning, model-assisted learning, and the development of entirely new RL algorithms as well as an array of algorithmic tweaks to existing ones. Out of these approaches, model-assisted learning will be the focus in this thesis.

1.2 Imagination and Forward Models

Traditional RL is either model-free or model-based. In model-free RL the agent learns the value of states and actions directly without understanding how the environment works, and selects the actions with the highest expected value. This is akin to a human instinctively choosing actions without any planning. For example, skilled drivers do

not need to think about what would happen every time they press the brake pedal; they instinctively know that breaking is the optimal action when the car in front brakes. Model-based RL agents on the other hand first learn a model of the environment, and then use that model to derive a policy. One way to do this is to use the model to simulate scenarios that can be used to learn the action-value function just as in model-free methods. This is akin to using imagination in humans.

So which is better in general, model-free of model-based RL? An important clue to answering this question, like many questions in artificial intelligence, lies in nature's own answer. Consider for example how people learn chess. Learning the rules of the game (i.e. the environment model) is much simpler than learning an optimal strategy (i.e. optimal action values). Beginners know very little about the true value of moves, and they tend to think every move through and simulate the possible scenarios for each before they choose one. But as players grow more experienced, they begin to learn the values of their actions directly, and experts often instinctively know what move is best without having to simulate outcomes at all. This suggests that humans use a learned environment model to facilitate learning a value function, and as they become more skilled they increasingly rely on the value function to make decisions without using the environment model explicitly.

However, not all tasks can benefit from a learned environment model equally. Tasks that focus on real-time motor control rely much more heavily on value functions since there is no time to use an environment model to simulate scenarios. Furthermore, low-level motor control learning problems by nature are not amenable to the same learning strategies used in high-level decision-making problems such as chess. Nobody can learn to swim by just imagining themselves in water. The only effective way to learn here is by physical trial and error to learn a policy directly through an action value function.

Nevertheless, RL algorithms do not have to belong to either of the these dichotomous paradigms exclusively. Hybrid approaches that combine model-free and model-based learning are possible. These approaches have the potential to draw on the strengths of both extremes, while mitigating their respective shortcomings. Much like the chess playing example, an RL agent can simultaneously learn a value function and a model of the environment. The environment model can be used for simulating imaginary experience, which can be used in conjunction with real experience to learn value functions. Such algorithms are sometimes referred to in the literature as model-assisted RL [LR14, KB17], or simply imagination [KBP13, HS18].

When using imagination, it is important to be able to distinguish between correct and incorrect predictions. If an agent relies on a bad model of its environment, it might generate erroneous imaginary experience that hinders learning a good policy. In such a situation, the agent is better off relying solely on actual experience, or directing its efforts to learn more about the environment to improve its model. Agents therefore must be able to tell when their predictions are likely to be wrong. In other words, they must be able to estimate their uncertainty.

1.3 Agents That Know They Don't Know

One of the hallmarks of truly intelligent agents is that when faced with a problem unfamiliar to them, they know that they don't know the answer. Intelligent agents such as humans associate a degree of uncertainty with every inference they make, ranging from full uncertainty when the problem is entirely novel, to full certainty when it is entirely familiar. Traditional machine learning systems by contrast fail to incorporate this estimate of uncertainty in the model outputs. The output of these systems is often taken blindly to be correct when it might not be the case, which can have disastrous consequences in safety-critical systems such as self-driving cars.

For classifier systems, it is tempting to think of class probabilities as measures of uncertainty. However, as Kendall et al. point out [KG17], this is not necessarily the case. The problem is even more pronounced in regression problems, where traditional models output a point estimate of the regrissand and there is no way to tell how confident this estimate is.

When dealing with uncertainty, it is important to distinguish between two types: *epistemic* uncertainty and *aleatoric* uncertainty. Epistemic uncertainty refers to the ignorance of the model, and can be explained away with more data. Aleatoric uncertainty refers to the irreducible noise inherent in the data, and cannot be reduced by acquiring more data. In other words, epistemic uncertainty tells you how much your model is unreliable, while aleatoric uncertainty tells you how much your data is unreliable.

Intelligent agents must therefore be able to estimate their epistemic uncertainty about the world, and use this knowledge to guide decision-making. For example, an RL agent might try to learn an environment model to predict the outcome of its actions, allowing it to plan and learn more efficiently. If it knows that its model is incomplete and is lacking some data from a certain region, then it can direct its efforts to exploring that region of the environment to fill the gaps in its knowledge. Indeed, the ability of an

agent to capture its own epistemic uncertainty can be very useful for artificial curiosity and active learning.

1.4 Research Objectives

The main goal of this thesis is to develop an architecture that incorporates imagination into model-free RL to improve sample efficiency. For such improvement to be possible, an environment model has to be learned efficiently on-line from visual data, simultaneously with the model-free path. A model learned in this manner can be used to generate imaginary rollouts that augment the data already being collected by the agent, thus leading to faster learning. To maximize the benefit from the model, uncertainty estimates should be used to prevent erroneous model predictions from contaminating the data and destabilizing learning.

The objectives of the thesis can be broken down into research questions as follows:

1. How can we implement a mechanism for visual imagination in RL agents? This question is addressed in Chapter 3.

2. How can we efficiently estimate the uncertainty for neural networks with multimodal predictive distributions? This question is addressed in Chapter 4.

3. How can we leverage imagination in order to improve the sample efficiency of model-free RL? This question is addressed in Chapter 5.

In Chapter 6, these research questions are revisited in a new light after the details of the work has been presented in the thesis.

1.5 Contribution to Knowledge

The main scientific contributions of the thesis can be summarised as follows:

1. In Chapter 3, an architecture is developed that allows learning stochastic environment models on-line from visual data simultaneously with a value function. The architecture enables learning an environment model that can be used to in a closed loop to generate imaginary rollouts for multiple timesteps into the future without suffering from compounding errors[1].

[1]The problem of compounding errors happens when a small error in an initial prediction quickly compounds into larger and large errors in subsequent predictions due to feedback.

2. In Chapter 4, a method is developed for uncertainty estimation in Bayesian mixture density networks, providing for the first time a mathematical treatment of the problem.

3. In Chapter 5, an architecture for imagination-augmented RL is developed, which can significantly improve performance when data is scarce. The architecture puts together all the components and ideas developed throughout the thesis.

Section 2.4.5 revisits the contributions of the thesis and situates them within the relevant literature.

Most of the original research in this thesis has been published in the following peer-reviewed papers:

- Mohammad Thabet, Massimiliano Patacchiola, and Angelo Cangelosi. *"Sample-efficient Deep Reinforcement Learning with Imaginary Rollouts for Human-Robot Interaction"*. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2019). IEEE, 2019.

- Mohammad Thabet, Massimiliano Patacchiola, and Angelo Cangelosi. *"Toward Imagination-assisted Deep Reinforcement Learning for Human-robot Interaction"*. International PhD Conference on Safe and Social Robotics (SSR-2018), 2018.

1.6 Thesis Structure

This section provides an overview of the structure of the thesis and briefly describes the topics discussed in each chapter. The thesis is structured as follows:

Chapter 2 explains the methods used in the thesis with some detail, as well as a brief review of the relevant literature. Machine learning techniques such as deep learning and reinforcement learning are discussed, enabling the reader to better understand the rest of the thesis. In particular, the topics of mixture density networks, variational autoencoders, and deep Q-networks are all essential to understanding the architectures presented later in Chapters 3 and 5. Moreover, the topic of Bayesian neural networks is required to understand the concepts relating to uncertainty estimation discussed in Chapter 4.

Chapter 3 presents an architecture for learning stochastic environment models online and using them for model-based RL. The chapter describes the architecture and

its components in detail, which will be also used in the architecture presented in Chapter 5. It also includes the details of an experiment involving a simulated human-robot interaction (HRI) task, in which the agent has to learn to perform the correct actions by generating imaginary rollouts using a learned environment model. In this model-based setting, the experiment aims to verify the efficacy of the model by training a controller on imaginary rollout on testing it on the real environment. The concepts developed in this chapter will be used heavily in Chapter 5.

Chapter 4 discusses uncertainty estimation in deep learning models, which allows agents be aware of their own uncertainty when they are presented with novel inputs they had not been trained on. It also includes experiments to test uncertainty estimation in mixture density networks and how it can be useful for decision-making. Uncertainty estimation in MDNs allows rejecting uncertain predictions generated by environment models to improve the overall quality of imaginary data.

Chapter 5 describes an architecture for imagination-assisted RL. The architecture integrates all the components and ideas developed in the rest of the thesis and to allow agents to leverage imagination to accelerate learning. It achieves this by combining model-free learning and model-based imagination with uncertainty estimation. An experiment is conducted to validate the architecture in which a robot learns to solve a puzzle based on gestures from a human. The performance of the imagination-augmented agent is compared to baseline DQN to assess the performance gain the architecture provides.

Finally, Chapter 6 provides conclusions to the thesis as well as some discussion and directions for future work. The chapter revisits the research questions of the thesis in light of the findings of the previous chapters to draw conclusions.

Chapter 2

Background

This chapter serves as a concise introduction to the technical methods employed in the rest of the thesis, as well as a brief review of recent and relevant literature. It is intended for those with at least a rudimentary understanding of the topics of machine learning, neural networks, and reinforcement learning. The material for deep learning is largely based on [GBCB16], and reinforcement learning is on [SB18]. Interested readers looking for a deeper understanding of these topics are encouraged to refer to these books.

The chapter is organized as follows. Section 2.1 introduces basic concepts of NNs and DL, including topics such as convolutional neural networks, Bayesian neural networks, mixture density networks, and variational autoencoders. Section 2.2 introduces basic concepts of reinforcement learning, including perhaps the most popular RL algorithm: Q-learning. It also introduces Dyna-Q, a basic architecture for model-assisted RL. Section 2.3 introduces DRL as the intersection between DL and RL, and provides a summary of two of the most important DRL algorithms: deep Q-networks, and deep deterministic policy gradient. Finally, Section 2.4 presents a brief review of recent and relevant research in the methods employed in this thesis

2.1 Deep Learning

DL is a family of ML algorithms employing artificial NNs comprising many successive layers of neurons, hence the adjective ”deep”. The key difference between DL and other ML algorithms is the ability of DL models to automatically learn representations of the raw input data. In traditional ML algorithms (e.g. linear regression), a feature engineering step is performed to extract useful features from data before learning can

begin. In contrast, features are automatically learned in DL models, where each successive layer of neurons learns increasingly abstract features of the input. For example, when a DL model is trained to perform image classification, the first layer will learn to detect edges, the second will detect more abstract features such as basic shapes and contours, while the third might detect even more abstract features like object parts and so on. This ability of of DL models to perform automatic *representation learning* makes it suitable for *end-to-end* learning, and is the main reason DL models are the state of the art in so many ML tasks today.

Neurons in NNs are simple computational units that perform some operation on its input to produce an output. A neuron can have multiple scalar inputs, but only one scalar output. Each scalar input is multiplied by a scalar weight and the products are summed together to produce the total input to the neuron. The neuron then performs some operation on that sum to produce the output. Mathematically, artificial neurons arranged in a layer are modeled by the following equation:

$$y_m = \phi\left(\sum_{n=1}^{i=N} w_{mn} x_n\right), \tag{2.1}$$

where y_m is the output of the m-th neuron in a layer, x_n is the n-th input of N total inputs, w_{mn} is the weight of the connection between the n-th input and the m-th neuron, and ϕ is the activation function. The relationship between the input and output of a single neuron is illustrated in Figure 2.1.

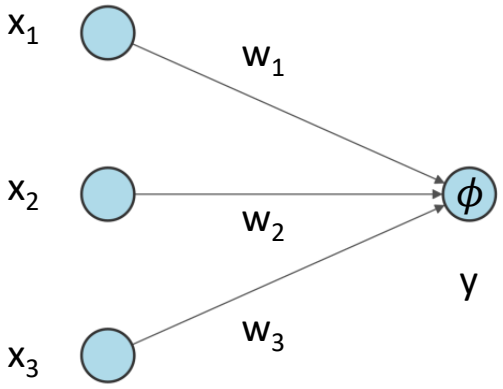


Figure 2.1: A single artificial neuron with three inputs.

Activation functions play an important role in NNs. The simplest activation function is the linear function whose output is exactly equal to its input. However, as will

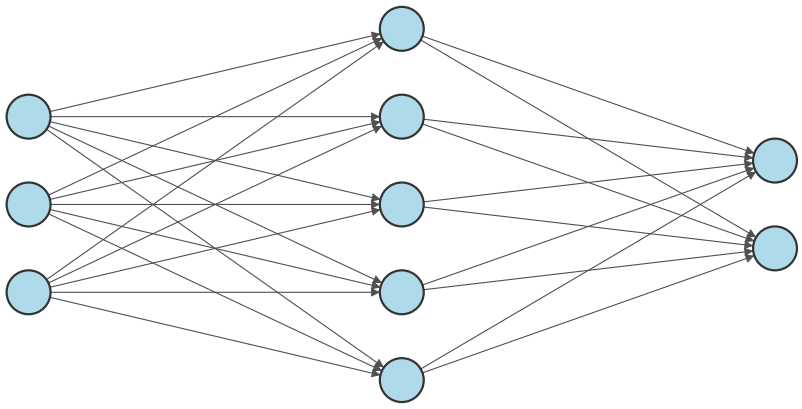
be explained in the next section, nonlinear activation functions are crucial for any non-trivial function approximation. Examples of non-linear activation functions include the sigmoid, the hyperbolic tangent (tanh), or the rectified linear unit (ReLU).

2.1.1 Feedforward Neural Networks

Feedforward NNs are the simplest architecture of NNs. In this type of networks, connections are exclusively made from one layer to the next with no intra-layer connections. Data flows one-way from the input layer to hidden layers to the output layer, where the input to each layer is the output of the previous one. Figure 2.5 shows an example of a feedforward network. For such a network with one hidden layer, the overall transfer function of feedforward networks can be composed from that of single neurons as follows:

$$Y = \phi_2\big(W_2\phi_1(W_1X)\big), \tag{2.2}$$

where Y is the vector of outputs, X is the vector of inputs, W_1 and W_2 are the matrices of the weight associated with the hidden layer and the output layer respectively, and ϕ_1 and ϕ_2 are the activation functions of hidden and the output layer respectively.



Input Layer Hidden Layer Output Layer

Figure 2.2: A feedforward neural network with 3 input units, one hidden layer with 5 units, and two output units.

To train NNs, one must first define a loss function that measures how much the output of the model differs from desired output in training data. Common loss functions include the squared error loss and the crossentropy loss. Training involves finding

the derivatives of the loss function with respect to individual weights. The derivatives are found using the backpropagation algorithm [RHW86], which makes use of the chain rule to propagate the derivative of the error backwards through the network. Once derivatives are found, an optimizer is used to find the optimal values for the weights. Optimizers typically employ a variant of the stochastic gradient decent algorithm [Rud16], which uses minibatches of the training data to approximate the loss function and then iteratively nudges the weights in the direction of the gradient.

Feedforward NNs are immensely useful as function approximators. The universal approximation theorem [HSW89] states that a feedforward NN with at least one hidden layer with a nonlinear activation function can approximate any function arbitrarily well provided that the network has enough hidden units. However, this does not guarantee that any large-enough network will be able to learn a specific function. This is because of two reasons. First, the optimizer might not be able to find the right parameters of the network, perhaps because of an inadequate optimization algorithm, numerical instability, or insufficient compute resources. For instance, if the learning rate used with the optimizer is inadequate, training might not converge. Second, there might be many functions that fit the data, and the training algorithm might find the parameters for the wrong function. The problem of overfitting is a typical example of this, where the learned function does indeed fit the data too well, but fail to generalize to unseen data.

2.1.2 Convolutional Neural Networks

While feedforward NNs can in principle be used to process images, in practice they are impractical for doing so. This is due to the large dimensionality of the images, which in turn requires very large models that are difficult to optimize. For example, even a very small 48×48 image means the input space is 2304-dimensional. A convolutional neural network (CNN) alleviates this problem by exploiting the spacial structure of images. Images usually have a strong inherent spacial structure. For example, to detect a line, we only need to look at a small region of the image; pixels far away are unlikely to provide useful information.

CNNs exploit spacial structure by employing the convolution operation. For a two-dimensional input such as an image, a kernel is convolved with the input to produce the output. This kernel can be thought of as a window that slides on the input, producing a feature every time the window slides. The two-dimensional array of features produced from convolving with a kernel is called a feature map. The kernel does not change

when it slides on the input, which in effect means that the weights are shared for each point in a feature map. Convolution layers are composed by convolving with multiple kernels to produce multiple feature maps. Another way CNNs exploit spacial structure is by using pooling layers. Similar to convolutional layers, pooling layers are also constructed by sliding a window on the input, but instead of convolving with a kernel they simply perform a predefined pooling operation. Examples of pooling operations include maximum pooling, which outputs the maximum value in the window, and average pooling, which outputs the average value within the window. Pooling layers effectively perform downsampling on the input feature map and makes the model more robust to translations, and doing so efficiently since they have no trainable weights. Typical CNNs are composed of alternations of convolutional layers and pooling layers, with one or more regular feedforward (dense) layers at the end. Figure 2.3 shows an example CNN.

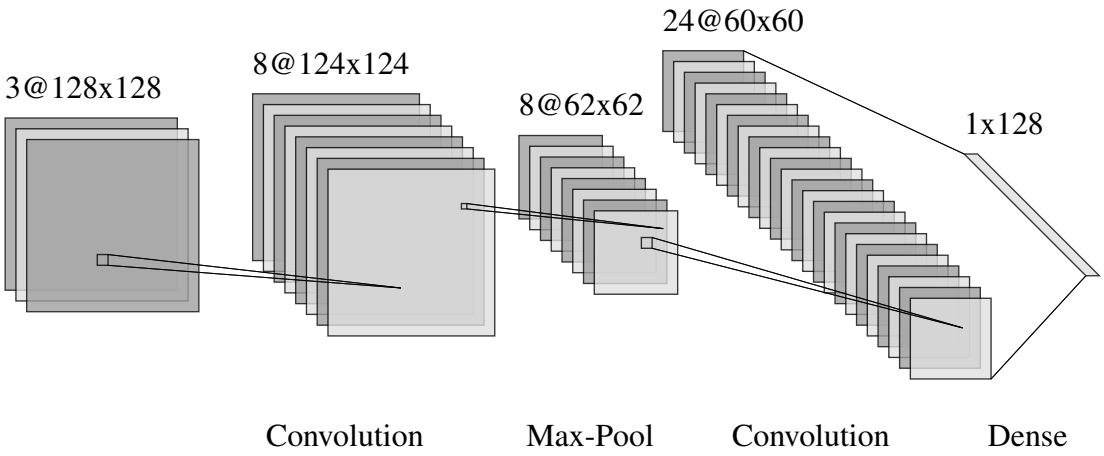


Figure 2.3: An example convolutional neural network. The input is a 128×128 image with 3 channels. The first convolution layers produces 8 feature maps that get downsamples by the max-pool layer and then convolved again to produce 24 feature maps. The output layer is fully connected to the previous layer.

2.1.3 Bayesian Neural Networks

Traditional NNs are trained to find a point estimate of the output by finding a point estimate of the optimal weights. Conversely, a Bayesian neural network (BNN) is trained to find the distribution of the output by finding the distribution of the weights. The advantages of this are twofold. First, finding distributions allows us to evaluate

the uncertainty in predictions. Second, BNNs naturally incorporate regularization, reducing overfitting when data is scarce. Indeed, regularization methods in NNs such as L1 or L2 regularization and dropout can be interpreted as approximations of Bayesian inference on the weights. More generally, traditional NNs can be thought of as special cases of BNNs in which the weights have a deterministic (degenerate) distribution [1].

Let w be the weights of a NN and $\mathcal{D} = \{(x_i, y_i) : i = \{1, \ldots, n\}\}$ be a dataset of n inputs $x_i \in X$ and corresponding targets $y_i \in Y$. The posterior distribution of w can be found using Bayes theorem:

$$p(w|\mathcal{D}) = \frac{p(\mathcal{D}|w)p(w)}{\int p(\mathcal{D}|w)d(w)} \tag{2.3}$$

Here, $p(\mathcal{D}|w)$ is the likelihood of the data, $p(w)$ is the prior distribution of the weights and the denominator $\int p(\mathcal{D}|w)d(w) = p(\mathcal{D})$ is the evidence, where the integral is taken over all possible values of w. For predictive models, we are generally more interested in the predictive distribution $p(Y|X,w)$ rather than the joint data distribution $p(\mathcal{D}|w) = p(X,Y|w)$. Bayes theorem can be written using the predictive distribution as:

$$p(w|\mathcal{D}) = \frac{p(Y|X,w)p(w)}{\int p(Y|X,w)d(w)}. \tag{2.4}$$

The problem with performing inference using Bayes theorem to find the posterior is that the evidence integral is intractable. However, even though we can't find the exact posterior, we can approximate it since it is proportional to the product of the likelihood and the prior, i.e. $p(w|\mathcal{D}) \propto p(Y|X,w)p(w)$. One technique that makes use of this fact is Markov chain Monte Carlo (MCMC), which allows sampling from an unknown distribution given that it is proportional to a known function. The downside of MCMC is that it is rather computationally costly and not guaranteed to converge [CC96]. Another more practical approach is to use variational inference. Variational inference works by approximating the posterior with a variational distribution $q(w) \approx p(w|\mathcal{D})$, and then minimizing the Kullback-Liebler (KL) divergence between p and q as follows:

[1] Equivalent to a Gaussian distribution in the limit when the variance approaches zero.

$$
\begin{aligned}
D_{\mathrm{KL}}\big(q(w)\,||\,p(w|\mathcal{D})\big) &= \int q(w)\log\frac{q(w)}{p(w|\mathcal{D})}dw \\
&= \mathbb{E}_q\big[\log\frac{q(w)}{p(w|\mathcal{D})}\big] \\
&= D_{\mathrm{KL}}\big(q(w)\,||\,p(w)\big) - \mathbb{E}_q\big[\log p(\mathcal{D}|w)\big] + \log p(\mathcal{D}) \qquad (2.5)
\end{aligned}
$$

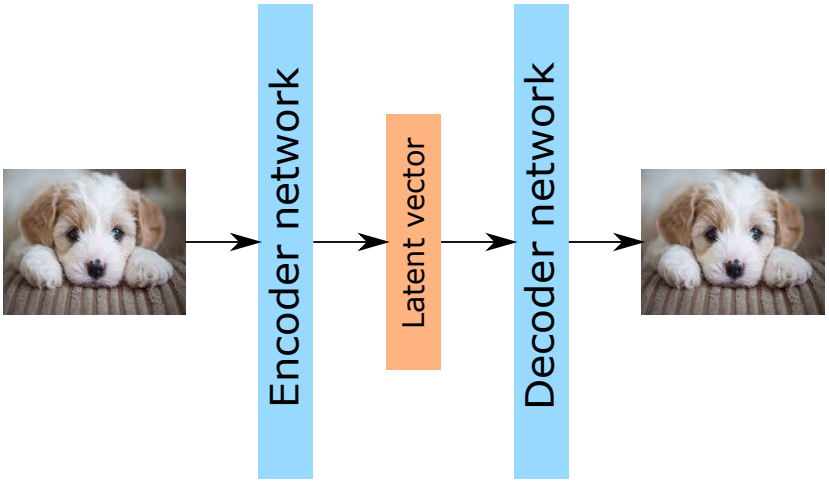
Here, the first quantity is the divergence of q from the prior, the second quantity is the likelihood of the data, and the third is the evidence. The negative of the first two quantities is called the evidence lower bound (ELBO). Since the evidence (which was intractable) does not depend on w and thus plays no part in optimization, minimizing the KL divergence is equivalent to maximizing the ELBO. The problem is thus transformed from one of inference into one of optimization, solving the intractability problem.

To perform variational inference, one must select the form of the variational distribution. Usually, a factorized Gaussian is used for simplicity. However, even for such a simple distribution, variational inference is computationally costly since it effectively doubles the number of weights (each weight now has mean and variance parameters). Thus, approximate methods such as MC-dropout are frequently used, as previously discussed in Section 2.4.4.

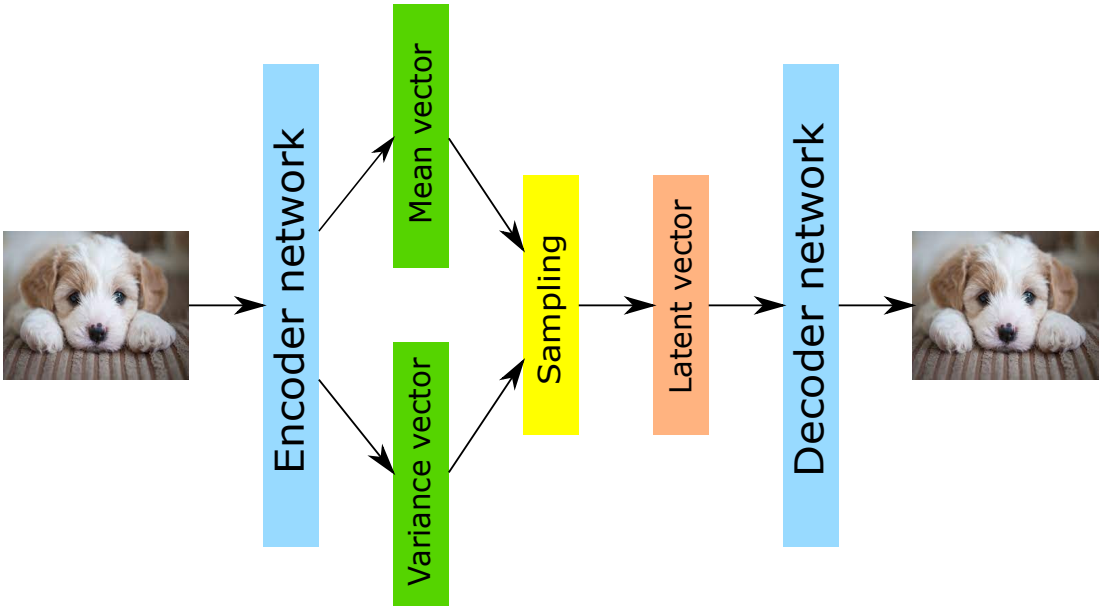
2.1.4 Variational Autoencoders

Autoencoders are NNs that can learn a compact representation of their inputs. They are architecturally composed of two NNs in series. The first network is the encoder that takes in inputs and encodes them into low-dimensional representations in the latent space. The second network is the decoder that takes the encoded representations and reconstructs the original input from it. Both networks are trained end-to-end to reconstruct the inputs. Autoencoders are mainly useful for dimensionality reduction.

VAEs [KW14] are generative models that can be used to both generate synthetic data, and to encode existing data into low-dimensional representations. Like traditional autoencoders, they consist of an encoding network and a decoding one. The key difference is that they encode a data point x into a probability distribution over the latent vector z (Figure 2.4). Variational inference is used to learn an approximate factorized Gaussian posterior distribution $q(z|x) = \mathcal{N}(\mu(x), \Sigma(x)\mathbf{I})$ which is assumed to have a unit Gaussian prior $p(z) = \mathcal{N}(\mathbf{0}, \mathbf{I})$. Similar to Equation 2.5, this can be done



(a) Traditional autoencoder.



(b) Variational autoencoder.

Figure 2.4: The traditional autoencoder and the variational autoencoder architectures. While the traditional autoencoder (a) encodes the input as a latent-space vector, the variational autoencoder encodes it as a probability distribution over the latent-space vector.

by minimizing the KL divergence between $q(z|x)$ and the true posterior $p(z|x)$:

$$D_{\mathrm{KL}}(q(z|x)||p(z|x)) = -\mathbb{E}_q\big[\log p(x|z)\big] + \beta D_{\mathrm{KL}}\big(q(z|x)||p(z)\big) + \log p(x) \tag{2.6}$$

Here, the first term is the reconstruction loss which incentivizes faithful reconstruction. The second term penalizes divergence of the learned variational distribution from the prior, and acts to limit the capacity of the latent information channel. The third term is the evidence, which plays no part in optimization. The multiplier β is added to control the latent channel capacity [HMP+17]. The expectation $\mathbb{E}_q[(x|z)]$ can be approximated by sampling a vector $z = \mu(x) + \Sigma^{1/2}(x) \odot \varepsilon$ with $\varepsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and decoding it with the decoder network. Sampling with ε this way is known as the reparameterization trick, and is done in order to be able propagate the errors backwards through the network. For a factorized Gaussian q, minimizing Equation 2.6 is equivalent to minimizing:

$$\mathcal{L}(\theta, \phi) = -\log p_\phi(x|z,\phi) + \beta \sum_{j=1}^{J} (1 + log\, \sigma_j^2(x,\theta) - \mu_j^2(x,\theta) - \sigma_j^2(x,\theta)), \tag{2.7}$$

where the encoder and decoder networks are parameterized with θ and ϕ respectively, J is the dimensionality of the latent space, and σ_j are the diagonal elements of $\Sigma(x;\theta)$. The encoder and decoder networks are trained back to back to minimize the loss given by Equation 2.7. Note that if $p_\phi(x|z)$ is Bernoulli, the reconstruction loss is equivalent to the cross-entropy between the actual x and the predicted $\hat{x}$.

2.1.5 Mixture Density Networks

Traditional NNs assume that the distribution of targets conditioned on inputs is unimodal. This is sufficient for single-valued functions that map each input to exactly one output. What happens then when we have a multi-valued function that maps each input to multiple correct outputs? For example, suppose we want to model the inverse of the function $x = y^2$ using a dataset of (x_i, y_i) points. The target function is thus $y = \pm\sqrt{x}$, where each input x maps into two valid outputs y. If we train a traditional NN on the data, it would learn to model the output as the average of the two valid outputs, which is itself not a valid output. In other words, it would try to find the unimodal target distribution that best covers the actual bimodal distribution. This effectively means it would model the function $y = \frac{\sqrt{x}-\sqrt{x}}{2} = 0$, which is the incorrect function. Therefore, for such functions, we need to be able to model the multimodal distribution of targets.

MDNs [Bis94] are neural networks that model the predictive distribution of targets

as a mixture of Guassians. Since the predictive distribution is allowed to be multi-modal, this is useful for modeling multi-valued functions such as many inverse functions or stochastic processes. MDNs model the distribution of target data y conditioned on input data x as the Gaussian mixture:

$$p(y|x) = \sum_{i=1}^{m} \alpha_i(x)\, \mathcal{N}(y; \mu_i(x), \sigma_i(x)^2), \tag{2.8}$$

where m is the number of components, α_i are the mixture coefficients subject to $\sum_{i=1}^{m} \alpha_i = 1$, and $\mathcal{N}(\cdot; \mu, \sigma^2)$ is a Gaussian kernel with mean μ and variance σ^2. MDNs have a similar structure to feedforward networks, except that they have three parallel output layers for three vectors: one for the means, one the variances, and one for the mixture coefficients. The network weights w are optimized by minimizing the negative log-likelihood of the data:

$$\mathcal{L}(w) = -\log \sum_{i=1}^{m} \alpha_i(x; w)\, \mathcal{N}(y; \mu_i(x; w), \sigma_i(x; w)^2) \tag{2.9}$$

To predict an output for a given input, we sample from the resulting mixture distribution by first sampling from categorical distribution defined by α_i to select a component Gaussian, and then sampling from the latter.

2.2 Reinforcement Learning

Reinforcement learning is concerned with agents acting in an environment to maximize some perceived cumulative reward. An agent explores the space of possible strategies in an environment and receives feedback for individual actions, and its goal is to find the optimal policy that result in the greatest possible cumulative reward. Tasks can be *episodic* if they have a clear end and can be restarted. In such a setting the goal of the agent is to maximize the cumulative reward per episode. If the task is ongoing with no clear end, the the agent's goal is to maximize the reward over its lifetime. In either case, maximizing the cumulative reward means taking actions that might not necessarily provide the most immediate reward, but are believed to lead to states that yield more reward in the future.

The problem of exploration-exploitation trade-off plays a central role in RL. It is concerned with the dilemma of whether to take actions which are known to be good (exploitation), or explore those whose value are unknown (exploration). If an agent

only exploits known good actions, it might miss out on other possible better actions just because it doesn't know their value. On the other hand, if the agent only explores, it will suffer from poor performance since it will not capitalize on known good actions. The solution thus involves maintaining some sort of balance between exploration and exploitation. Finding such good balance is an important problem in RL.

In RL, a task is modelled as a Markov decision process (MDP) where at any time step t an agent influences its environment state $s_t \in S$ with action $a_t \in A$. The environment then transitions into a new state $s_{t+1} \in S$ and provides the agent with a reward signal $r_t = r(s_t, a_t)$. The agent chooses its actions according to some policy $\pi(s)$ which can be deterministic or probabilistic. For deterministic policies, the agent always selects the same action for a given state such that $a = \pi(s)$. In the probabilistic case, the agent randomly samples from the distribution over actions for a given state such that $a \sim \pi(s,a) = p(s|a)$. Likewise, the environment dynamics T can be deterministic such that $s_{t+1} = T(s_t, a_t)$, or probabilistic such that $s_{t+1} \sim T(s_{t+1}, a_t, s_t) = p(s_{t+1}|s_t, a_t)$. This process repeats until the environment reaches a terminal state, concluding an episode of interaction. The goal of the agent is to learn an optimal policy π^* and use it to maximize the expected return, which is the discounted sum of rewards, $R_t = \mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t]$, where $\gamma \in [0,1]$ is the discount factor and T is the timestep a terminal state is reached.

Figure 2.5: In reinforcement learning, an agent influence the environment with an action. The environment then transitions to a new state and provides a reward.

In order to solve RL problems, we must first define a notion of value for states and actions, which allows us to choose from the space of actions available. The state-value function $V_\pi(s)$ of a state s under policy π is defined as the expected return from starting in s and following π thereafter: $V_\pi(s) = \mathbb{E}[R|s,\pi]$. Similarly, the action-value function $Q_\pi(s,a)$ of an action a taken in state s under policy π is defined as the expected return

starting from s and taking action a, then following π thereafter: $Q_\pi(s,a) = \mathbb{E}[R|s,a,\pi]$. The action-value function is often referred to as simply the *Q-function*, a name which we shall adopt in the rest of the thesis.

Of central importance in RL is the *Bellamn equation* which allows us to define value functions recursively in terms of the value of succeeding states and actions. For state value functions, the Bellman equation is given by:

$$V_\pi(s_t) = \mathbb{E}_{a_t}[r_t(s_t,a_t)] + \gamma \mathbb{E}_{s_{t+1}}\left[V_\pi(s_{t+1})|s_t\right], \tag{2.10}$$

where $\mathbb{E}_{a_t}[\cdot]$ denotes the expectation over all possible actions at time t, and $\mathbb{E}_{s_{t+1}}[\cdot]$ denotes the expectation over all possible next states.

Intuitively, the Bellman equation states that the value of any state is the sum of the expected reward from that state and the expected value of the next state. This allows us to estimate the value of earlier states from estimates of preceding states without needing to finish an episode to get the return first. The bellman equation can be written in terms of the Q-function as:

$$Q_\pi(s_t,a_t) = r_t(s_t,a_t) + \gamma \mathbb{E}_{s_{t+1}}\left[\max_{a_{t+1}} Q_\pi(s_{t+1},a_{t+1})|s_t,a_t\right] \tag{2.11}$$

An optimal policy π^*, which an agent strives to find, is the one under which the value of states are highest, i.e. $V_{\pi^*}(s) \geq V_\pi(s) \forall s \in S$. This results in an optimal state-value function $V^*(s)$ defined as $V^*(s) = \max_\pi V_\pi(s)$. Similarly, the optimal action-value function is $Q^*(s,a) = \max_\pi Q_\pi(s,a)$. We can rewrite Equations 2.10 and 2.11 in terms of optimal value functions to get the *Bellman optimality equation*:

$$V^*(s_t) = \mathbb{E}_{a_t}[r_t(s_t,a_t)] + \gamma \mathbb{E}_{s_{t+1}}\left[V^*(s_{t+1})|s_t\right] \tag{2.12}$$

$$Q^*(s_t,a_t) = r_t(s_t,a_t) + \gamma \mathbb{E}_{s_{t+1}}\left[\max_{a_{t+1}} Q^*(s_{t+1},a_{t+1})|s_t,a_t\right] \tag{2.13}$$

Once $Q^*(s,a)$ is found, it is straightforward to derive an optimal policy by greedily choosing the action that maximizes $Q^*(s,a)$, i.e. $\pi^*(s) = \arg\max_a Q^*(s,a)$. The Bellman optimality equations can be used for *temporal difference* learning, where estimates of the values of states and actions can be used to update the estimates of preceding states and state-actions.

2.2.1 Value Function Approaches vs. Policy Search

RL algorithms can be classified into two types: *Value-function approaches* aim to first find optimal value functions and then derive optimal policies from them, while *policy search* methods aim to find the optimal policy directly.

Value function approaches are conceptually simple. The tools for estimating optimal value functions are well-studied and rather straightforward for discrete or low-dimensional state and action spaces, and as discussed in the previous section, deriving an optimal policy is trivial once the optimal action-value function $Q^*(s,a)$ is found. For high-dimensional state and action spaces, value function approaches require function approximation. However, this introduces stability problems since a small change in the policy may cause a large change in the value function, which in turn causes a large change in the policy, thus destabilizing the learning. Furthermore, value function approaches require total coverage of the state-space, which may prove difficult or time-consuming to do.

Policy search methods are concerned with optimizing the parameters of the policy directly. For a stochastic policy $\pi(a|s;\theta)$ parameterized by parameter vector θ, the gradient of the policy is found using the policy gradient theorem as [SMS+99]:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi\big[Q_\pi(s,a)\nabla_\theta \log \pi(a|s;\theta)\big], \tag{2.14}$$

while the policy gradient for a deterministic policy $\mu_\theta(s)$ is given by [SLH+14]:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\mu}\big[\nabla_a Q_\mu(s,a)\nabla_\theta \mu_\theta(s)|_{a=\mu_\theta(s)}\big], \tag{2.15}$$

where ρ^μ is the state distribution under policy $\mu(s)$.

As such, policy gradient approaches are only concerned with the value function in the vicinity of the policy, not globally as in value function approaches. This usually leads to faster and more stable optimization. Furthermore, this method works well with continuous space and action spaces. However, since value functions are not evaluated globally, policy search usually find only locally optimal policies.

A variety of RL algorithms, called *Actor-critic* methods, aim to incorporate the advantages of both value function approaches and policy search. These methods maintain both a value function and a policy separately and explicitly. The value function (the critic) is not employed for action selection. Rather, it just evaluates the performance of the actor (the policy), and suggests the direction in which the policy needs

to be changed in order to increase performance. This way, actor-critic methods feature the stability of policy search with the reduced variance of updates of value function methods.

2.2.2 Model-free vs. Model-based Methods

Another important distinction between RL algorithms is whether they are *model-based* or *model-free*. Model-based methods require a model of the dynamics of the environment. This model can be either provided beforehand, or learned from interaction with the environment. In simple environments, such as games like chess or Go, the model is readily available as the rules of the game. In most real-world tasks however, the agent has no prior knowledge of the environment, and therefore the model has to be learned. Once the model is provided or learned, dynamic programming, Monte-Carlo, or even TD methods can be employed to learn optimal value functions or perform policy search. In fact, learned models can be used just to simulate experience, and then learn policies using model-free algorithms trained on this simulated experience. Learned models can also be used for planning, allowing methods such as model-predictive control (MPC) to be used. in MPC, the model is used to generate a plan by simulating trajectories and greedily selecting an action at every time step. The agent then takes the first action in the plan, observes the result in the real environment, and then plans again from there.

The performance of model-based algorithms depends heavily on the accuracy of the learned model, assuming it is feasible to learn in the first place. Even slight errors can compound when planning, leading to suboptimal asymptotic performance. In contrast, model-free algorithms do not require an environment model. These methods learn a value function or policy directly from experience. They are therefore most suitable if the environment dynamics are too complicated to learn efficiently. Moreover, they often have better asymptotic performance than their model-based counterpart, since they do not depend on imperfect learned models. However, they often suffer from worse sample efficiency since they depend on sampling from the environment and have no access to planning. Overall, model-free algorithms are more versatile and general, and therefore have been historically preferred for applications that care little about sample efficiency. In case sample efficiency is important, such as in robotic applications, model-based algorithms have an important advantage. Figure 2.6 shows the relationship between collecting experience, learning a model, planning, and model-free (direct) RL.

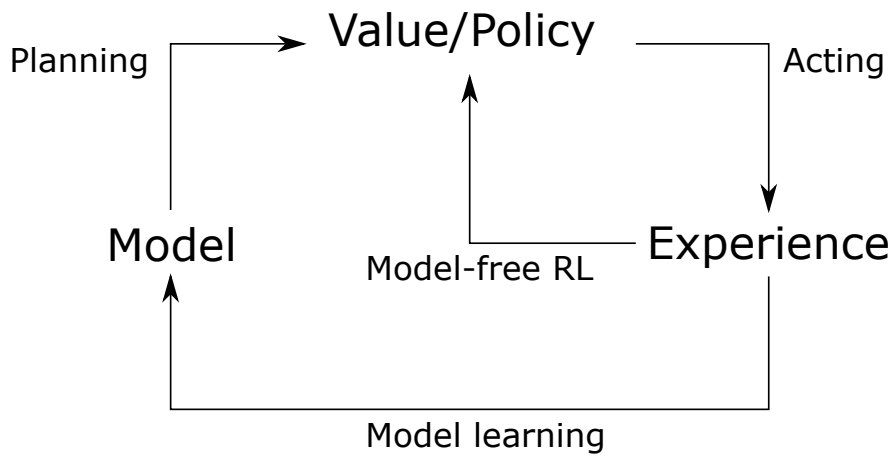


Figure 2.6: The relationship between acting, model learning, and planning.

2.2.3 Q-learning

One of the most known and widely used model-free RL algorithms is Q-learning [WD92]. This algorithm uses TD learning to approximate the optimal action-value function (Q-function) and derive an optimal policy from it. In Q-learning, the estimate $Q(s_t, a_t)$ is updated as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{2.16}$$

where α is the step-size parameter (learning rate). Intuitively, the update moves the Q-function estimate in the direction of the TD error by a distance controlled by the step-size. The algorithm is trained on data composed of tuples of the form $(s_t, a_t, r_{t+1}, s_{t+1})$ which the agent collects as it is interacting with the environment. At test time, the agent acts greedily with respect to the Q-function (full exploitation). While learning however, the agent follows a different exploratory policy. One common such policy is the ε-greedy policy, in which the optimal action is chosen with probability $1 - \varepsilon$ and a random action with probability ε. The parameter ε is usually decreased over time to approach zero asymptotically as the policy converges to the optimal policy. Since Q-learning uses a behavior policy different from the target policy, it is thus an *off-policy* algorithm. This is contrasted with *on-policy* algorithms such as SARSA [SB18], in which the behavior policy is the same as the target policy.

2.2.4 The Dyna-Q Architecture

Model-based and model-free approaches can be combined to incorporate the advantages of both. Perhaps the earliest and most well-known example is the Dyna-Q architecture [Sut90], shown in Figure 2.7. Real experience obtained from observations are used to learn the Q-function and an environment model simultaneously. For each time step, the model is instantiated with a random state-action pair encountered before and a successor state is predicted, producing simulated experience. This simulated experience is used to update the Q-function as in Q-learning. This process of applying Q-learning on model-generated experience is called *Q-planning*. Generating simulated transitions and Q-planning can be take place as many times each time step as time allows.

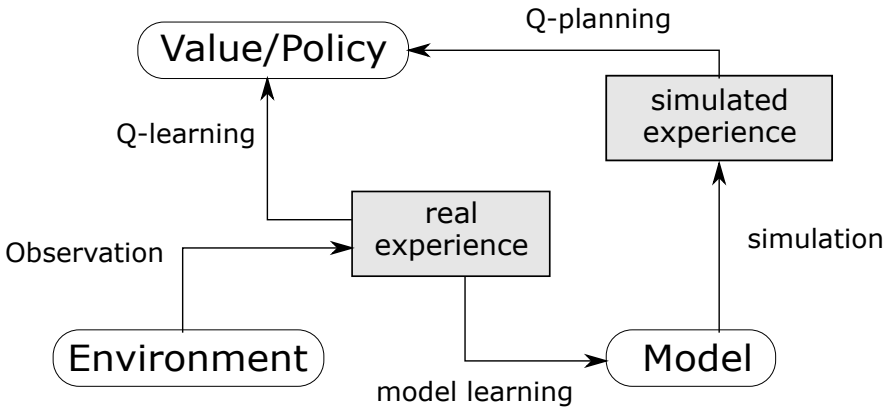


Figure 2.7: The Dyna-Q architecture. Real experience is used to learn the value function and the environment model simultaneously. The model generates simulated experience which is also used to learn the value function.

2.3 Deep Reinforcement learning

Reinforcement learning often requires function approximation. Value functions and models for large and continuous state spaces cannot be learned in tabular form, and therefore have to be approximated. Owing to their universal approximation property, neural networks naturally present a solution for function approximation in RL.

DRL algorithms employ deep networks to approximate value functions, dynamics models, or policies in solving RL problems. This is especially useful in tasks with high-dimensional state spaces such as vision-related problems. RL agents that rely on visual input are of immense importance in many applications such as robotics and

autonomous driving. The huge success of DL in tackling vision tasks naturally leads to the interest in employing these methods in the context of RL.

In the next two sections, two of the most widely used DRL algorithms will be introduced: deep Q-networks and deep deterministic policy gradient. The former is a model-free algorithm suitable for discrete action spaces, while the latter is an actor-critic algorithm suitable for continuous action spaces.

2.3.1 Deep Q-Networks

DQNs [MKS+13] are deep NNs that approximate the Q-function. As in Q-learning, transitions are stored as tuples of the form $(s_t, a_t, r_{t+1}, s_{t+1})$ in memory, and used to train the DQN. A DQN with weight vector w is trained to minimize the loss function given by the squared Bellman error:

$$\mathcal{L}(w) = \left[r_{t+1} + \gamma \max_{a_{t+1}} Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t)\right]^2 \tag{2.17}$$

Given the gradients of the loss function with respect to w, the network can be trained using some variation of stochastic gradient descent. Batches used for updates are randomly samples from memory, in what is termed *experience replay*. Random sampling ensures the transitions in each batch are uncorrelated, which helps stabilizes learning. Furthermore, a separate target network is usually maintained in addition to the Q-network. The target network is a copy of the DQN, but gets updated with the weights of the DQN less often. The purpose of this network is to provide targets for the update to the DQN, instead of getting targets from the same DQN. This prevents the problem of chasing a moving target for the DQN and enhances stability.

Figure 2.17 shows the architecture of a typical DQN a task in which states are given by images and the action-space is discrete. The state does through a CNN to extract features. The output of the CNN then goes through a fully-connected (dense) network which outputs an n-dimensional vector in which each element correspond to the Q-function of the input and a certain action in the action-space. In some tasks where velocities of objects are important features, a single image is not sufficient as the state. In such cases, multiple successive frames can be used together as the state, and fed together to the DQN.

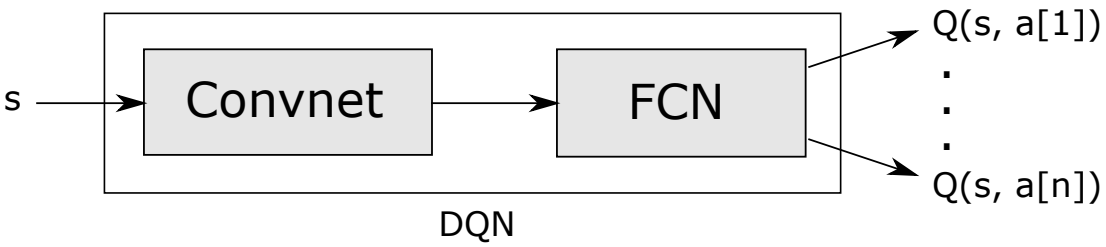


Figure 2.8: A typical DQN for a task in which states are images and actions are discrete. The state s goes through a convolutional network and then a fully-connected network to produce n outputs for n actions.

2.3.2 Deep Deterministic Policy Gradient

DQN algorithms are limited in that they only work for tasks with discrete action spaces. This is not a problem for high-level tasks in which the agent has to choose from a discrete set of decisions. However, it makes them unsuitable for low-level tasks that require continuous control such as robot manipulation.

DDPG [LHP+15] is a popular actor-critic algorithm capable of learning continuous control tasks. In DDPG, a critic network learns the Q-function, and an actor network learns the policy. The critic gets updated as in DQN, while the actor gets updated according to the gradients given by Equation 2.15. Separate target networks are maintained for both the critic and the actor. However, instead of periodically copying the weights of the networks to the target, "soft updates" are performed where only a fraction of the weights are transferred as follows:

$$\theta^{\mu}_{\text{targ}} \leftarrow \tau\theta^{\mu}_{\text{targ}} + (1-\tau)\theta^{\mu} \tag{2.18}$$

$$\theta^{Q}_{\text{targ}} \leftarrow \tau\theta^{Q}_{\text{targ}} + (1-\tau)\theta^{Q} \tag{2.19}$$

where θ^{μ} and θ^{Q} are the weights of the actor and critic respectively, and $\theta^{\mu}_{\text{targ}}$ and θ^{Q}_{targ} are the weights of the corresponding target networks.

The advantage of using a deterministic policy instead of a stochastic policy such as in regular policy gradient algorithms, is that the former is much more efficient to compute. In stochastic policy gradient, state and action spaces have to be integrated over to compute expectations. In contrast, only the state space need to be integrated over in deterministic policy gradient. Consequently, stochastic policy gradient may require more samples especially if the action space is high-dimensional.

2.4 Literature Review

This Section presents a brief review of recent and relevant research in the methods employed in this thesis. The literature is far too expansive to exhaustively cover here, so the chapter is limited to some selected research that is most relevant and impactful.

The rest of the chapter is organized as follows. Section 2.4.1 provides a review of DRL techniques and recent advances in the field. Section 2.4.2 reviews work on RL in HRI tasks. Section 2.4.3 is dedicated to some of the recent work on model-assisted RL. Section 2.4.4 discusses recent work on uncertainty estimation in DRL models. Finally, Section 2.4.5 situates the contributions of the thesis within the literature.

2.4.1 Deep Reinforcement Learning

The field of DRL is widely considered to begin when researchers in Google Deepmind published the DQN algorithm [MKS+13, MKS+15]. DQN consists of a deep neural network that gets trained on targets given by the Bellman equation used in the Q-learning algorithm [WD92]. In their work, Mnih et al. trained a DQN to play Atari games at a superhuman level of performance. The input to the network was raw pixels from the frames of the video games, in contrast to the simple or engineered state representations such as positions and velocities used in traditional RL. The output of the network was commands that would usually be sent to the game via a joystick. The DQN algorithm incorporated many algorithmic innovations that allowed the difficult problem of network convergence in RL to be solved, such as random sampling from experience replay and a separate target network.

Since the debut of DQN, the field has seen many milestones being reached by various DRL algorithms. In 2016, DeepMind released AlphaGo [SHM+16], an algorithm that beat the world champion in the game of Go, which is considered one of the most challenging of classical games due to its huge search space. AlphaGo combined the Monte Carlo tree search (MCTS) algorithm and deep neural networks to learn policies and value functions, with the policy network being pretrained on human expert play in a supervised fashion. The algorithm was improved with the release of AlphaZero in 2017 [SHS+18], which could master chess and Shogi in addition to Go. In 2019, DeepMind released AlphaStar, an agent that achieved grandmaster status in competitive play of the popular real-time strategy game Starcraft II, and beat world champions in the game [VBC+19]. A similar breakthrough was achieved by OpenAI's Five which beat world champions in Dota 2, a popular esports game. The achievement

of AlphaStar and Five are especially notable since these games have long time horizons and immensely complex continuous state-action spaces, which are key challenges for more general AI systems.

DRL in robotics has focused mainly on continuous control tasks [LFDA16, LHP+15, GHLL17, PHL+17], using algorithms suitable for continuous action spaces such as DDPG [LHP+15], asynchronous advantage actor-critic (A3C) [MBM+16], and proximal policy optimization (PPO) [SWD+17]. Since collecting the huge amounts of data required for training DRL algorithms is expensive on actual robots, researchers have tried to alleviate this problem mainly in two ways. The first is to train cheaply in simulators and then transfer skills to physical robots, possibly with some fine-tuning on the robot [CSM+16, RVR+17, TDH+15]. The second is to improve the sample efficiency of DRL algorithms [LA14, AWR+17, VHS+17, TZL+16, BHT+18]. One of the main ways to improve sample efficiency is to incorporate model-based RL techniques, which is the focus of section 2.4.3.

2.4.2 Reinforcement Learning for HRI

Deep reinforcement learning is increasingly being employed successfully for robots in continuous control tasks [LFDA16, LHP+15, GHLL17, PHL+17], and in physical HRI scenarios in which positions and forces have to be controlled [GBM+16]. However, its application to high-level tasks and in social HRI has been relatively limited and only recently started to gather momentum. Qureshi et al. [QNYI16] used a multimodal DQN to teach a humanoid robot basic social skills such as successfully shaking hands with strangers. Input to their system consisted of depth and greyscale images. Interaction data were collected using the robot over a period of 14 days, where they have separated the data collection and training phases and alternated between them for practical reasons. The system was later modified to use intrinsic rewards resulting from the difference between actual results and those of a learned predictive model and produced better results. [QNYI18]. Clark-Turner et al. used audio-visual signals to teach a DQN agent appropriate social intervention skills from demonstrations [CTB18].

Cruz et al. used audio-visual feedback from a human teacher to train a robot to correctly move objects as instructed using SARSA algorithm [CPTW16]. Lathuilere et al. used audio-visual sensory information to train an long short-term memory (LSTM)-based Q-function approximator for gaze control in social settings to maximize the number of persons speaking and in the field of view of the robot [LMMH19]. Gao et al. used visual information to teach an agent appropriate social approaching behavior

using the PPO algorithm where they used an autoencoder to compress the visual signal into state representations [GYF+19]. Hussain et al. used speech features to teach a DQN agent appropriate backchanneling such as laughing during dialogue [HESY19]. In [Cua20], a DQN is trained on visual information augmented with speech to learn to play Tic-Tac-Toe interactively with a human. A recurrent DQN is used in [GCY+20] to proactively assist a human in a packaging task by inferring the type of box they are carrying. The DQN is trained on motion data sequences from a motion sensor suit with the goal of minimizing the time it takes the robot to act correctly. A VAE was used to compress the motion sequences into low-dimensional representations, which can then be fed to the DQN for inference.

2.4.3 Model-assisted Deep Reinforcement Learning

There has been much interest in the literature about combining model-free and model-based approaches to reinforcement learning. Such approaches are sometimes called model-assisted RL (e.g. [LR14, KB17]) or model-based acceleration (e.g. [GLSL16]), to distinguish them from pure model-based and model-free approaches. When the learned model is used to simulate trajectories (i.e. imagine scenarios), it is sometimes called imagination (e.g. [RWR+17]) or mental rehearsal (e.g. [KBP13]). Ha and Schmidhuber [HS18] built models for various video game environments using a combination of an MDN and an LSTM, which they call MDN-RNN. In their approach, they first compress visual data into low-dimensional representations via a VAE, and then train the MDN-RNN to predict future state vectors, which are used by the controller as additional information to select optimal actions. However, they pretrained the environment models on data collected by a random agent playing video games, whereas in our work a model for an HRI task is learned online. .

The use of learned models to create synthetic training data has also been explored. Kalweit et al. [KB17] used learned models to create imaginary rollouts to be used in conjunction with real rollouts. In their approach, they limit model usage based on an uncertainty estimate in the Q-function approximator, which they obtain with bootstrapping. They were able to achieve significantly faster learning on simulated continuous robot control tasks. However, they relied on well-defined, low-dimensional state representations such as joint states and velocities, as opposed to raw visual data as in our approach.

Racaniere et al. [RWR+17] used a learned model to generate multiple imaginary rollouts, which they compress and aggregate to provide context for a controller that

they train on classic video games. The advantage of this approach is that the controller can leverage important information contained in sub-sequences of imagined rollouts, and is more robust to erroneous model predictions.

Model rollouts can be used to improve targets for temporal differencing (TD) algorithms as well. Feinburg et al. [FWS+18] used a model rollout to compute improved targets over many steps, in what they termed model-based value expansion (MVE). More recently, Buckman et al. [BHT+18] proposed an extension to MVE, in which they used an ensemble of models to generate multiple rollouts of various lengths, interpolating between them and favoring those with lower uncertainty.

2.4.4 Uncertainty Estimation in Deep Learning

Kendall et al. define two types of model uncertainties: epistemic uncertinaty and aleatoric uncertainty [KG17]. Epistemic uncertainty is defined as the uncertainty in model parameters due to the ignorance of the model. This kind of uncertainty can be reduced be acquiring more data. On the other hand, aleatoric uncertainty is the uncertainty due to inherent noise in the data, and cannot be reduced by data. In DL, epistemic uncertainty is captured by the distribution of model weights, while aleatoric uncertainty is modeled by the output distribution of the model. Estimating the epistemic uncertainty thus requires either learning a distribution of the model weights, or the ability to indirectly sample from such distribution. Learning the weights distribution can be done with Bayesian inference. However, exact Bayesian inference for non-trivial networks is intractable. Consequently, approximate inference methods have become popular for their simplicity and low computational cost [BCKW15, GG16, HLA15, Gra11]. Another approach to estimating epistemic uncertainty is through model ensembles [LPB17, KCD+18, Osb16]. Each model in the ensemble can be considered as a sample from the distribution of weights, and the uncertainty can be estimated from the variance in the outputs of the individual models.

One popular method to estimate uncertainty in NNs is through Monte Carlo dropout (MC-dropout). Gal et al. [GG16] have shown that training a network with dropout is equivalent to approximate Bayesian inference to obtain an approximation of the posterior distribution over the weights. Samples from the approximate posterior can then be obtained by applying dropout at test time with different binary mask vectors. Thus, by feeding the network the same input multiple times but with different dropout masks, different outputs can be obtained for the same input, and the variance in the output due to the different weight samples can be used to estimate the epistemic uncertainty.

MC-dropout has been used to estimate the uncertainty in computer vision applications [KG17], reinforcement learning for collision avoidance [KVP+17], and in medical applications [LAA+17] among others.

ALakshminarayanan et al. [LPB17] used deep ensembles to estimate the uncertainty, where each network modeled the density of the target data as a Gaussian distribution. The output of the ensemble is given by a uniformly-weighted mixture of Gaussians, and the uncertainty is given by its variance. Instead of the variance, the entropy of the predictive distribution can be also used to quantify the uncertainty. Depeweg et al [DHLDVU17] used BNNs with latent variables to estimate the uncertainty for bimodal function approximation. They used the reduction in the entropy of the posterior distribution of weights as a criteria for active learning, where the agent collects datapoints that result in the largest expected reduction in the entropy.

Model uncertainty has naturally found use in RL applications. Examples include artificial curiosity [SGS11, HCD+16] and risk-sensitive RL [GF15, DHLDVU17]. Chua et al. [CCML18] used NN ensembles to estimate the epistemic uncertainty in the dynamics model in order to prevent overfitting in the low-data regime at the beginning of training. Kahn et al. [KVP+17] used both MC-dropout and model ensembles to estimate the uncertainty of a collision prediction network in order to make a robot move cautiously when it is uncertain about the environment. Recently, Lee et al. [LLSA20] proposed a framework for ensemble learning of Q-functions that utilizes uncertainty estimates from the ensemble in two way. First, the uncertainty estimates are used to reweigh the Bellman backup so as to mitigate the propagation of errors from the target Q-function. Second, the uncertainty estimates are used to encourage exploration of novel states where the ensemble uncertainty is high.

2.4.5 Contribution

Research on high-level or social HRI so far have largely neglected the sample complexity problem in RL. For example, in [QNYI16], data was collected over a period of 14 days in a public area where there is a lot of visitors to interact with the robot. This can be infeasible in many cases. The work in this thesis focuses on improving sample efficiency to allow DRL to be more practical for HRI scenarios.

Furthermore, other similar work on model-based acceleration in RL has either used simple state representations [KB17], offline learning of the dynamics model, or separate and sequential learning of the dynamics and the controller [HS18]. To my knowledge, this is the first work to focus on fully online and simultaneous learning of both

the dynamics model and the controller from raw visual data. In addition, the thesis shows how the dynamics model can be learned with a feedforward architecture and simulated accurately multiple timesteps into the future with the compounding error problem taking over.

With regard to uncertainty estimation, research discussed so far have focused mainly on unimodal data. In contrast, we show how to obtain decomposed uncertainty estimates for highly multimodal data with heteroscedastic noise, which is the most general case. This thesis is also the first work to mathematically derive expressions for uncertainty estimates for MDNs. Besides epistemic and aleatoric uncertainties, the thesis introduces a third kind: modal uncertainty. This kind of uncertainty represents the stochasticity inherent in multimodal distributions. This is important because the modal uncertainty is not undesirable or something to be reduced, unlike the other two kinds of uncertainty. Therefore, this allows us to better analyze the behavior of models and to know whether the data is noisy or scarce in certain regimes of the input space.

Chapter 3

Imagination-based Deep RL

The ability of humans to use their imagination allows them to learn efficiently without needing to interact with their environment as often. Once they have a model of how the environment works, they can simulate scenarios in their mind and derive optimal policies from them. The ability to implement a similar mechanism for RL agents can be tremendously useful for increasing the sample efficiency. This is especially true for robotic applications, since physically interacting with the environment for prolonged periods can be prohibitively impractical.

This chapter introduces an architecture for learning RL-based tasks through imagination. The agent first learns a model of its environment, and uses this model to generate synthetic experience. Entire imaginary rollouts can be created by the model, spanning multiple timesteps into the future. This imaginary experience can then be used to train a controller as if it was real experience collected from the physical environment. The key difference between this imagination-based agents and regular model-based ones is that the former use a generative model to ”imagine” new valid states and transitions between them. In contrast, regular model-based agents can only predict transitions for states they actually encounter.

The chapter is organized as follows. Section 3.1 presents an architecture to generate imaginary rollouts with a learned environment model and provides details for all the components. Section 3.2 presents an experiment to validate the architecture and show how the components can be trained and used. Finally, Section 3.3 concludes the chapter with a summary and discussion.

3.1 Architecture

The purpose of the architecture presented here is to allow an agent to generate imaginary rollouts that can be used to train an RL algorithm. The architecture consists of three main components: the vision encoder module that produces abstract representations of input images, the environment model which generates imaginary rollouts (simulated experience), and the controller that learns to map states into actions. For the purpose of generating imaginary rollouts, the encoder and the environment model are assumed to have been already trained. The controller is allowed to be any function, depending on the nature of actions we wish to generate our rollouts with. It might be random if we wish to generate random trajectories, or it may be trained if we wish to generate optimal trajectories.

Figure 3.1 shows an overview of the architecture. The flow of data in the architecture starts with the agent observing the environment through a camera. The raw image obtained is the environment state s, which get passed to the encoder that transforms it into an encoded state vector z. The controller receives the state vector z and selects action a. The environment model receives a and z as input, and produces a predicted next state z' and a predicted reward r'. The predicted z' in turn gets fed to the controller which selects another action a'. The encoded old and new states (z and z') are concatenated with the action a and the reward r' to form a transition tuple (z, a, r', z'), which is then stored in the imaginary memory. The loop continues when z' gets fed back to the environment model along with a' to produce the next predicted state. The environment model keeps running in closed loop in this manner until a terminal state is predicted, concluding the imaginary rollout. To determine when to terminate a rollout, a trained classifier is used to classify the predicted states into terminal and non-terminal ones.

In the following sections, each of the components of the architecture will be examined in detail.

3.1.1 Vision Encoder

The vision encoder comprises the encoder part of a VAE, and is responsible for mapping the high-dimensional input images into low-dimensional state representations. The encoder thus effectively serves as the perception module of the agent. It takes in an image representing the state s, and outputs the distribution of the encoded state vector $p(z|x)$. The latent state vector can then be sampled from this distribution as $z \sim p(z|x)$. In practice however, the mean of the distribution is taken to be the state

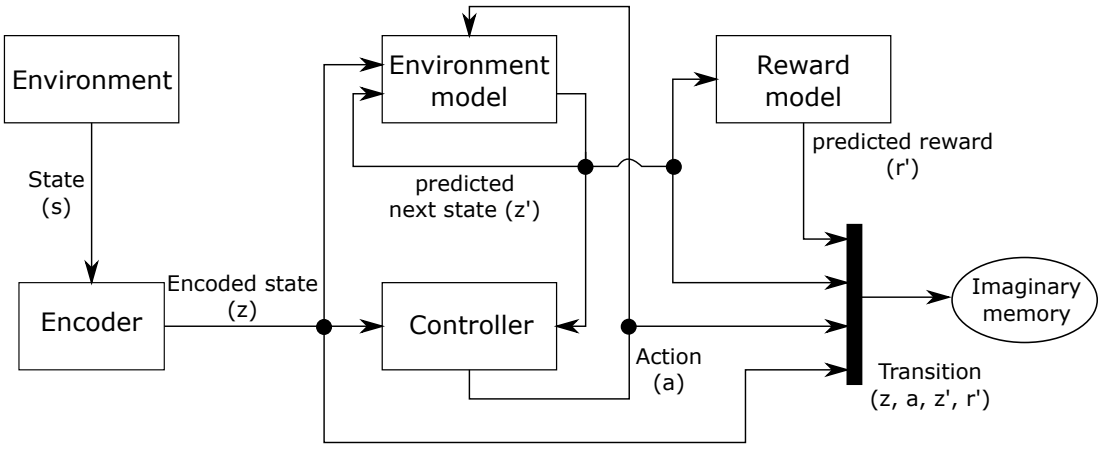


Figure 3.1: Overview of the architecture for generating imaginary rollouts with a learned environment model. The initial environment state s is encoded by the encoder into a state vector z, which get passed to the controller to select action a. The environment model is seeded by the initial z and a, and predicts the next state z' and reward r'. The controller then selects an new action for z' and the loop continues until a predicted z' is classified as terminal. The transitions generated by the model are saved in the imaginary memory for training controllers.

vector. The main advantage of compressing states with the encoder is the significant reduction in computational and memory costs when training the controller and the environment model. The encoder gets trained in an offline manner before the actual learning of the task. To train it, a dataset of images relevant to the task must be collected beforehand. However, the same encoder can be used for multiple tasks, so long as they share the same visual domain.

The choice of a VAE instead of a regular auto-encoder has two main advantages. First, while the regular auto-encoder maps an input image into a point in the latent space, the VAE maps it into a continuous region in the latent space defined by the parameters of a factorized Gaussian distribution. Consequently, points in that region basically represent the same state since they can all be produced from the same distribution. Thus, any prediction of the next state lying within that region is considered correct, giving predictions a lot more margin for acceptable error. This makes the environment model more robust and ensures that its output is meaningful and can be mapped back into realistic images. Furthermore, it prevents the problem of compounding errors when predicting successive next states from a given observed initial state. If the environment model is to be used to predict multiple timesteps into the future, this advantage becomes essential.

The second advantage is that the VAE is also a generative model, which means that

it can be used to generate novel data similar to a certain datapoint. This allows the environment model to generate entirely new imaginary experience, in which not only the transition are novel, but also the states themselves in the transitions. Consequently, the synthetic experience becomes much richer, further improving the ability of the agent to learn. This sort of data augmentation is the main difference between this approach and regular model-based RL.

3.1.2 Environment Model

The environment model learns a forward model of the environment dynamics. It is responsible for generating synthetic transitions, and predicts the future state z' and reward r' based on current states z and input action a. It can also predict multiple timesteps into the future by running it in closed loop where its output is fed back into its input along with the selected action. To generate an entire imaginary rollout, it is first seeded with an initial state vector z, which can be the current state or any state sampled from the real memory. The model is then left to run in closed loop for a predefined time horizon or until the imaginary episode terminates. For each state prediction, the model also predicts whether that state is terminal or not, so that it knows when to terminate the rollout.

The environment model is composed of three component models: an MDN that learns the transition dynamics, a reward predictor called the r-network, and a terminal state classifier called the d-network. The MDN learns the conditional probability distribution of the next state $p(z'|z,a)$. The next state vector is obtained by sampling from the predictive distribution, i.e. $z' \sim p(z'|z,a)$. The advantage of using an MDN is that it allows learning a model of stochastic environments, in which an action taken in any given state can lead to multiple next states. This is especially useful for multi-agent environments such as HRI tasks, in which the response of other agents to actions taken by the robot cannot be expected with certainty. Furthermore, modelling the next state probabilistically is much more robust to errors in prediction, allowing the environment model to run in closed loop.

The r-network learns to predict the reward for each state as a function of the state vector as $r = f(z)$. The choice to make the reward dependent on just the state instead of the state-action pair is made for simplicity and ease of training. The d-network learns to classify states into terminal and non-terminal. Both the r- and d-networks are implemented as feed-forward neural networks. When the environment model is run in closed loop to generate rollouts, the r-network predicts the reward for each predicted

next state. The rollout is terminated whenever the d-network classifies the predicted next state as terminal.

3.1.3 Controller

The controller is responsible for selecting the appropriate action in a given state, predicted or observed. The actions the controller selects control the evolution of the imaginary trajectories. The controller can be random if random imaginary trajectories are to be generated. It can also be trained beforehand if we wish to generate optimal imaginary trajectories instead.

3.2 Experiment

The purpose of the experiment detailed in this section is the following. First, to validate the method of generating complete and reliable imaginary rollouts. Second, to see if an environment model can be learned on-line. Third, to test the performance of controllers trained on imaginary data, and compare them to those trained on actual data. The experiment involves a simulated pick-and-place task, in which a human requests the agent to pick up objects and hand them over. The agent is expected to learn a model of the high-level dynamics of the task from visual data, and to correctly perform the actions requested by the human afterwards.

This section is organized as follows. Section 3.2.1 describes the experiment setup, providing details about the task to be learned, the environment built for it, as well as data collection. Section 3.2.2 provides implementation details of the components and the training procedure. Section 3.2.3 presents the results obtained from the experiment, while Section 3.2.4 presents conclusions drawn from the results.

3.2.1 Experiment Setup

The experiment is a simulated HRI task in which the agent learns to pick and place objects as instructed by its human partner. In the experiment, the human starts by pointing at any one of three objects placed on a table, which the agent picks up. The human can then either point at another object at random, at which point the agent has to place the object it currently holds back on the table and pick the new one, or they can request a handover. Figure 3.2 shows an example image from the point of view of the agent.

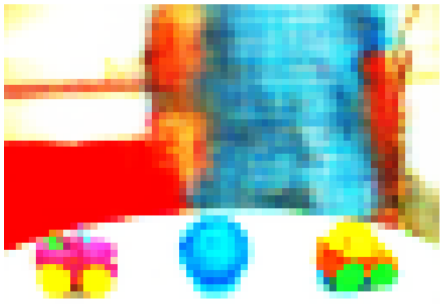


Figure 3.2: An example image from the point of view of the agent, showing the experiment setup. In this image, the human is pointing to the blue ball, requesting the robot to pick it up.

The task is formulated as an RL problem in which the agent can choose from 4 discrete actions at any given time. The first three actions are to pick/place objects 1, 2, or 3 respectively, while the fourth is to perform a handover. The agent gets a reward of +1 for correctly picking up an object, 0 for putting an object back, +5 for correctly handing over, or -5 for choosing an incorrect action. An episode terminates if either a handover is correctly performed, or the agent chooses an incorrect action. The choice of the exact magnitude of rewards here is rather arbitrary, and any choice of magnitudes should work, given that the signs (the values of the signum function) of rewards is preserved and the difference between the values is not too large.

To create the simulated environment, approximately 100 RGB images were collected for every possible configuration of object placements and hand gestures, for a total of about 1200 images. The images were collected using the iCub robot eye camera [MSV+08]. The images were then scaled down to a manageable 64×64 resolution. The simulated environment was implemented as a finite state machine, in which each state corresponds to a configuration of which object the agent has picked up, and what gesture the human makes. The state machine has a total of 12 states, covering all combinations of possible hand gestures and object positions. Table!3.1 shows all allowable states as implemented in the machine. Each state is associated with the set of images collected for the corresponding configuration of the task. In each run, the environment is initialized with a random initial state. It then receives a certain action as input, and transitions into the next state and outputs the reward. At each state, the environment also outputs an image randomly selected from the set of images associated with that state.

State	Left object	Middle object	Right object	Gesture
1	placed	placed	placed	left
2	placed	placed	placed	middle
3	placed	placed	placed	right
4	picked	placed	placed	middle
5	picked	placed	placed	right
6	picked	placed	placed	handover
7	placed	picked	placed	left
8	placed	picked	placed	right
9	placed	picked	placed	handover
10	placed	placed	picked	left
11	placed	placed	picked	middle
12	placed	placed	picked	handover

Table 3.1: Allowable states in the pick-and-place experiment. Each state can have all objects placed, or just one of them picked. Gestures can be to either point to any of the objects, or request a handover if an object is picked.

3.2.2 Implementation and Training

In this section, the implementation details and training methods used for the individual components are provided.

VAE

The architecture of the VAE used for the experiment is that used by Ha and Schmidhuber in [HS18], except that the latent space is 4-dimensional. Figure 3.3 shows the VAE architecture used. The VAE was trained on all the images collected for the task for 1000 epochs using the Adam optimizer with a learning rate of 0.0001. The objective of the training was to minimize the loss given by:

$$\mathcal{L}(\theta,\phi) = \sum_{n=1}^{N} \Big[-\log p_\phi(x_n|z_n,\phi) + \sum_{j=1}^{J} (1 + log\,\sigma_j^2(x_n,\theta) - \mu_j^2(x_n,\theta) - \sigma_j^2(x_n,\theta)) \Big], \tag{3.1}$$

where the encoder and decoder networks are parameterized with θ and ϕ respectively, N is the number of training examples, and $J = 4$ is the dimensionality of the latent space. See Section 2.1.4 for more details on training VAEs.

The vision encoder network used in the experiment is simply the encoder part of the VAE. The 4-dimensional mean of the distribution of the latent vector produced by

the encoder was taken to be the latent space vector.

Controller

The controller was implemented as a DQN and trained on the simulated environment in a regular Q-learning setting. The environment would provide an image representing the state, which gets encoded into a state vector by the encoder and passed to the DQN. The controller then outputs an action as input to the environment which responds by changing its state and providing a reward signal. Transitions collected in this way were stored in memory for and used for training via experience replay.

The DQN had 2 hidden layers (512 ReLU, 256 ReLU) and a linear output layer. It was updated once on a batch of 64 transitions each timestep. The DQN was trained to minimize the mean squared error (MSE) given by:

$$\mathcal{L}(y,\hat{y}) = \sum_{n=1}^{N} (y_n - \hat{y_n})^2, \tag{3.2}$$

where N is the number of training examples, $\hat{y}_n = Q(s_t, a_t)$ is the output of the DQN for the state (s_t) at time t, and $y = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$ is the bellman target. The Adam optimizer was used with a learning rate of 0.00025.

When selecting actions, the controller used an ε-greedy strategy with an exponentially decreasing exploration rate ε given by:

$$\varepsilon = \varepsilon_{min} + (\varepsilon_{\max} - \varepsilon_{\min}) e^{-\lambda t} \tag{3.3}$$

with $\varepsilon_{\min} = 0.001$, $\varepsilon_{\max} = 0.8$, $\lambda = 0.03$, and t is the time step. Starting with a large ε and decreasing it over time allows the controller to explore more in the beginning of learning, and less towards the end when it has sufficiently explored the state space.

For the experiment, two separate controllers were trained. The first was trained on the data obtained by interacting with the environment. The second controller was trained on data generated by the environment model, where the controller interacted with the model as if it was the actual environment. Both controllers were trained for 1000 episodes.

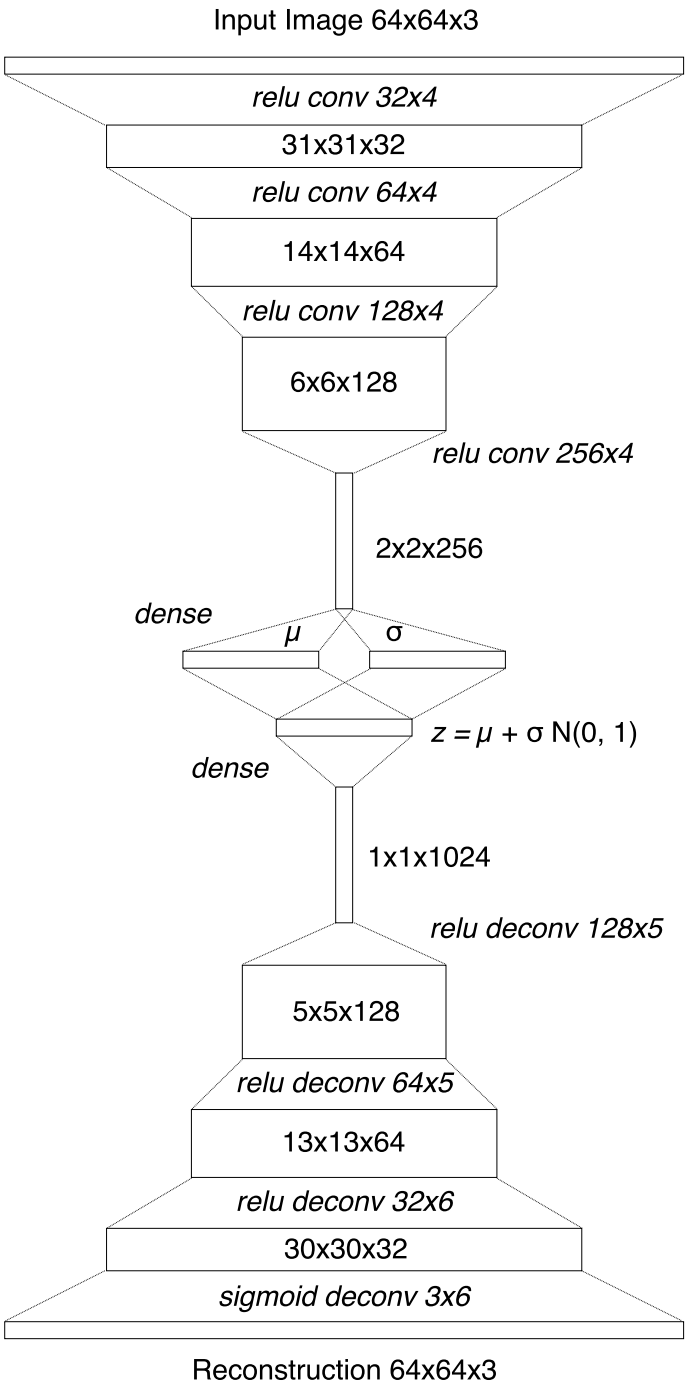


Figure 3.3: The VAE architecture used in the experiment. Source: [HS18]

Environment Model

The environment model is composed of three networks: an MDN for the dynamics model, the reward model, and the terminal state classifier. The MDN had 3 hidden layers of 256 ReLU units and 3 parallel output layers for the distribution parameters: one for the mixture coefficients with softmax activation, one for means and one for logarithm of variances both with linear activation. The reason for outputting the logarithm of the variance is to ensure the variance is always positive[1]. The input to the MDN consisted of the state vector concatenated with the one-hot encoded action vector.

The MDN was trained to minimize the negative log likelihood given by:

$$\mathcal{L}(y,\theta) = \sum_{n=1}^{N} \Big[-\log \sum_{k=1}^{K} \alpha_k(x_n;\theta)\, \mathcal{N}(y_n;\mu_k(x_n;\theta),\sigma_k(x_n;\theta)^2) \Big], \tag{3.4}$$

where θ is the vector of network weights, y is the vector of target next states, N is the number of training examples, $K = 24$ is the number of Gaussian components, α_k is the probability of component k, and μ_k and σ_k are the mean and variance vectors of the k-th component respectively. The network was trained with the Adam optimizer with a learning rate of 0.001.

The reward model (r-network) had a similar architecture to the controller, and was trained to minimize the logarithmic hyperbolic cosine loss given by:

$$\mathcal{L}(y,\hat{y}) = \sum_{n=1}^{N} \log\big(\cosh(\hat{y}_n - y_n)\big), \tag{3.5}$$

where y and $\hat{y}$ are the target and the predicted output respectively, and N is the number of training examples. The Adam optimizer was used with a learning rate of 0.001.

The terminal state classifier (d-network) shared the same input and hidden layers with the r-network, but had its own output layer with sigmoid activation. It was trained to minimize the binary cross-entropy loss given by:

$$\mathcal{L}(y,\theta) = \sum_{n=1}^{N} y_n \log p_\theta(y_n) + (1-y_n)\log\big(1 - p_\theta(y_n)\big), \tag{3.6}$$

where θ is the vector of network weights, y is the vector of target binary variables representing whether a state is terminal or not, $p_\theta(y)$ is the predicted probability of the the state being terminal, and N is the number of training examples.

[1]The inverse of the logarithmic function is always non-negative

During training, the MDN, the r-network and the d-network were all updated 4 times on batches of 64 transitions each timestep using the Adam optimizer with a learning rate of 0.001.

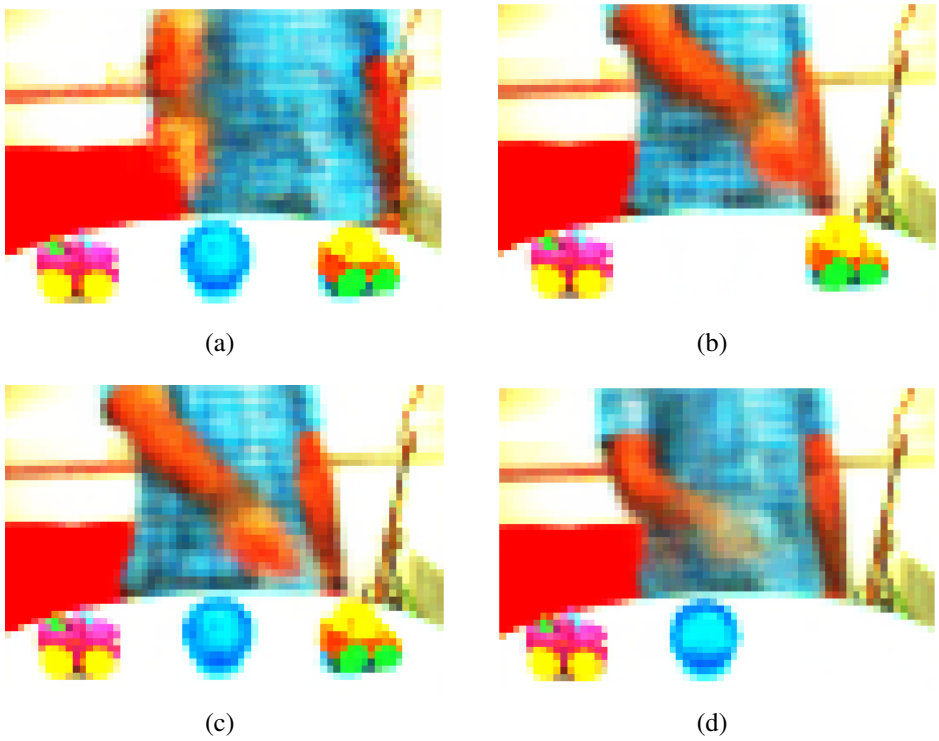
3.2.3 Results

The controller was allowed to interact with the environment for 10 runs of 1000 episodes each. It was noticed that the controller learned an optimal policy after about 500 episodes in each run. After each training run, a test run of 100 episodes was performed, in which a separate set of validation images were used. The controller reached an average success rate [2] of about 98% over all the test runs.

In each of the training runs, an environment model was also being trained, but wasn't being used to generate data. To test the learned environment models, another controller was trained on imaginary rollouts by simply replacing the actual environment with the learned model. This controller had an identical architecture to the original, and was trained in exactly the same manner, but on imaginary data. After training, the controller was tested in the actual environment for 10 runs of 100 episodes each. The controller achieved an average success rate of about 78%, compared to the 98% percent of the original.

Visual inspection of the quality of the imaginary data showed that the model can generate realistic rollouts. Figure 3.4 shows an imaginary rollout generated by the model, in which only the first image is obtained from the actual environment. After being seeded with the initial encoded image, the model ran in closed loop to generate the rollout. Actions for the rollout was selected using a trained controller. The imagined states were mapped back to images using the decoder part of the VAE. It is important here to note that, except for the first image that seeds the model, none of these images are real; they are entirely imagined by the model. They represent what the model thinks is going to happen next given a certain state-action pair. Furthermore, each imaginary rollout with the same initial state results is different, reflecting the stochastic nature of the environment. The trajectory is stochastic due to sampling from the different components of the MDN, and the images themselves are slightly different for the same configuration due to sampling from the encoder.

[2] The success rate here is defined as the ratio of episodes completed successfully to those that failed.



(a) (b)

(c) (d)

Figure 3.4: A sample imaginary rollout produced by the environment model. The images are visualizations of states imagined by the model. (a) the initial state in which the human requests the agent to pick the blue ball. (b) the model predicts that the ball will be picked up and the human will point to another object. (c)the model predicts the agent will put down the ball. (c) The model predicts the agent will pick the car on the right and the human will request a handover.

3.2.4 Conclusion

The experiment was designed to test three hypotheses. First, that it is possible to learn an environment model in latent space. Second, that this model can be learned incrementally as the agent is interacting with the environment. Third, that the imaginary data are valid enough that if we train a controller on them, it would perform well in the actual environment.

With regard to the first hypothesis, the results demonstrate that we can learn a model in the latent space obtained by transforming images with an encoder. The rollouts obtained from running the model in closed loops are realistic, and can be used to train a controller. The controller trained exclusively on imaginary rollout performed reasonably well when tested in the real environment. Furthermore, visual inspection of the imaginary rollouts showed realistic states and trajectories. The trajectories and the states were stochastic, reflecting the stochastic nature of the HRI task.

With regard to the second hypothesis, the results showed that the model can be learned incrementally while the agent is interacting with the environment, much like how a DQN is trained. It is less efficient than offline learning, but still leads to good models.

As for the third hypothesis, the results showed that the controller trained exclusively on imaginary data performed reasonably well in the actual environment. The results also show that it performed worse that the original controller, but this is to be expected since model-free learning usually leads to better performance than model-based learning.

3.3 Summary and Discussion

This chapter introduced methods for learning environment models online via experience replay, and for generating imaginary complete imaginary rollouts with the model. The method involves using a VAE to encode images into latent state vectors, and an MDN to model transition dynamics in this latent space. An experiment involving a simulated HRI task showed that a model can be learned in such manner, and a DQN can be trained on imaginary data generated by the model.

The use of probabilistic models makes the system robust enough to predict multiple time steps into the future, enabling it to generate entire rollouts from an initial state. The combined use of a VAE for the encoder and an MDN for the environment model mitigates the problem of compounding errors when predictions are made based on

earlier predictions. If point estimates were used instead, the model would only be able to predict just one or two time steps into the future before predictions become totally erroneous. Indeed, it was quite surprising (and thrilling) for me the first time I saw how good the imaginary rollouts were. Granted, the environment was very simple. Yet, to see the model actually visually imagine scenarios playing out was extremely exciting.

The experiment presented in this chapter served as a proof of concept, and did not demonstrate any advantage to using imaginary rollouts for training. It was a sort of sanity check to see if the components can work individually and together. In the next chapter, we will see how to estimate the epistemic uncertainty in MDNs, which allows us to discard uncertain imaginary rollouts to prevent them from adversely affecting training the controller. Chapter 5 further builds on the architecture presented in this chapter and presents an architecture that combines real and imaginary data to increase the sample efficiency in RL.

Chapter 4

Uncertainty Estimation in Bayesian MDNs

The ability to estimate the uncertainty in DL models is of great importance for DRL. Uncertainty estimation allows DL-based agents to know what data to seek to improve their performance, and prevents them from making erroneous decisions when presented with novel situations. In the architecture presented in chapter 5, the RL agent learns an MDN model of a stochastic environment and uses it to produce synthetic experience. It is thus important for the agent to estimate the epistemic uncertainty of its model, which allows it to reject experience generated by the model if it has high uncertainty, since that would mean it is most likely erroneous. This chapter explores uncertainty estimation in DL models for the most general case, in which the data is highly multimodal with heteroscedastic noise. It also details how these uncertainties can be computed for Bayesian mixture density networks, and then used for decision making.

Throughout this chapter the term *random function* is used to denote a function whose value is a random variable. Whereas the output of a regular function is deterministic given its inputs, the output of a random function is a random variable defined by a probability distribution conditioned on its inputs. Similarly, a *random function approximator* is some function that approximates a target random function. The problem of learning such an approximator will be treated as equivalent to that of multimodal density estimation.

The chapter is organized as follows. Section 4.1 introduces the Bayesian bias-variance decomposition, which allows us to decompose the mean squared error of

learning algorithms to obtain sources of uncertainty. In Section 4.2, this decomposition is applied to the problem of multi-modal density estimation. In Section 4.3, an alternative view of epistemic uncertainty as the degree of stationarity of the predictive distribution is discussed. Section 4.4 presents mathematical derivations of uncertainty estimates for Bayesian MDNs and gives concrete formulas that can be used directly. Section 4.5.1 presents a toy problem to show how uncertainties can be estimated for MDNs with highly multimodal predictive densities. Section 4.5.2 presents a simulated robot inverse kinematics problem that shows how uncertainty estimates can be used to guide decision-making. Finally, Section 4.6 concludes the chapter with a summary and discussion.

4.1 The Bayesian Bias-Variance Decomposition

The bias-variance decomposition is a method of analyzing the mean squared error of a learning algorithm. It decomposes the error into three terms: the bias, the variance, and the irreducible error [Bis06]. Let $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i) : i = \{1, \ldots, n\}\}$ be a dataset of n samples drawn form the distribution $p(\mathbf{x}, \mathbf{y})$ where $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{y} \in \mathbb{R}^k$. Assume $\mathcal{D}$ is generated by the process:

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) + \boldsymbol{\eta}, \tag{4.1}$$

where $\mathbf{f}(\cdot) : \mathbb{R}^d \to \mathbb{R}^k$ is some deterministic function and $\boldsymbol{\eta} \sim \mathcal{N}(\mathbf{0}, \sigma_\eta^2 \mathbf{I})$ is the measurement noise with a diagonal $k \times k$ covariance matrix whose diagonal entries form the vector $\sigma_\eta^2 = (\sigma_1^2, \ldots, \sigma_k^2)$. Thus, the true conditional density of the target data is given by $p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{f}(\mathbf{x}), \sigma_\eta^2 \mathbf{I})$. Assume that instead of simply learning a function $\hat{\mathbf{f}}(\mathbf{x})$ that approximates $\mathbf{f}(\mathbf{x})$, we wish to estimate the true density as $\hat{p}(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\hat{\mathbf{f}}(\mathbf{x}), \hat{\sigma}_\eta^2 \mathbf{I})$ and use it to generate predictions $\hat{\mathbf{y}}(\mathbf{x})$. Applying the well-known bias-variance decomposition to the expected squared error between $\hat{\mathbf{y}}$ and $\mathbf{y}$ for a certain value of $\mathbf{x}$ yields:

$$\begin{aligned}
\int (\mathbf{y}(x) - \hat{\mathbf{y}}(\mathbf{x}))^2 p(\mathcal{D}) d(\mathcal{D}) &= \mathbb{E}_{p(\mathcal{D})}[(\mathbf{y}(\mathbf{x}) - \hat{\mathbf{y}}(\mathbf{x}))^2] \\
&= \mathbb{E}_{p(\mathcal{D})}[(\mathbf{f}(\mathbf{x}) - \hat{\mathbf{y}}(\mathbf{x}))^2] + \sigma_\eta^2 \\
&= (\mathbf{f}(\mathbf{x}) - \mathbb{E}_{p(\mathcal{D})}[\hat{\mathbf{y}}(\mathbf{x})])^2 \\
&\quad + \mathbb{E}_{p(\mathcal{D})}[(\hat{\mathbf{y}}(\mathbf{x}) - \mathbb{E}_{p(\mathcal{D})}\hat{\mathbf{y}}(\mathbf{x}))^2] + \sigma_\eta^2, \\
&= (\mathbf{f}(\mathbf{x}) - \mathbb{E}_{p(\mathcal{D})}[\hat{\mathbf{y}}(\mathbf{x})])^2 + \mathbb{V}_{p(\mathcal{D})}[\hat{\mathbf{y}}(\mathbf{x})] + \sigma_\eta^2,
\end{aligned} \tag{4.2}$$

where the expectations are taken over all possible datasets $\mathcal{D}$ with samples drawn from the joint distribution $p(\mathbf{x},\mathbf{y})$, i.e. $p(\mathcal{D}) = \prod_{i=1}^{n} p(\mathbf{x}_i,\mathbf{y}_i)$. The first term in Equation 4.2 is the square of the *bias*, the second term is the *variance* of the approximator, and the third term is the *irreducible noise* in the data.

The bias-variance decomposition in Equation 4.2 assumes a *frequentist* stance, in which the weights of the approximator are fixed and the dataset is a random variable. In contrast, we will approach the decomposition from a *Bayesian* perspective, in which we consider the dataset fixed and instead assume the weights of the approximator to be random variables. Consider an approximator with weight vector $\mathbf{w}$ distributed according to some distribution function $q(\mathbf{w}|D)$. The variance in $\mathcal{D}$ in the frequentist approach thus corresponds to that in $\mathbf{w}$ in the Bayesian one. As such, the expectation over $\mathbf{w}$ can be substituted for the expectation over $\mathcal{D}$, and Equation 4.2 can be written as:

$$\begin{aligned}\int (\mathbf{y}(x)-\hat{\mathbf{y}}(\mathbf{x}))^2 q(\mathbf{w}|\mathcal{D})d(\mathbf{w}) &= \mathbb{E}_q[(\mathbf{y}(\mathbf{x})-\hat{\mathbf{y}}(\mathbf{x}))^2] \\ &= \mathbb{E}_q[(\mathbf{f}(\mathbf{x})-\hat{\mathbf{y}}(\mathbf{x}))^2]+\sigma_\eta^2. \\ &= (\mathbf{f}(\mathbf{x})-\mathbb{E}_q[\hat{\mathbf{y}}(\mathbf{x})])^2+\mathbb{V}_q[\hat{\mathbf{y}}(\mathbf{x})]+\sigma_\eta^2, \end{aligned} \tag{4.3}$$

Since the quantity $\mathbb{E}_q[(\mathbf{f}(\mathbf{x})-\hat{\mathbf{y}}(\mathbf{x}))^2]$ is a second moment (albeit a non-central one), we shall, perhaps with a slight abuse of notation, denote it by σ_e^2. Equation 4.3 can thus be written as:

$$\mathbb{E}_q[(\mathbf{y}(\mathbf{x})-\hat{\mathbf{y}}(\mathbf{x}))^2] = \sigma_e^2+\sigma_\eta^2. \tag{4.4}$$

From a Bayesian perspective, Equation 4.4 decomposes the total uncertainty in $\hat{\mathbf{y}}$ into two sources. The first term represents the ignorance of the model which is known as the *epistemic* uncertainty . The second is the irreducible noise inherent in the data, which is known as the *aleatoric* uncertainty.

4.2 Decomposition for Multi-modal Density Estimation

In some practical applications, $\mathbf{f}$ can be a random function that take on multiple values for the same input, irrespective of the measurement noise. For example, $\mathbf{f}$ can be a multi-valued function with a categorical distribution such as in robot inverse kinematics problems, where a certain endpoint pose (x, the input) can be achieved via multiple joint configurations (y, the multi-valued output). In this case, when combined with

Gaussian measurement noise, the conditional distribution of the target data $p(\mathbf{y}|\mathbf{x})$ is a Gaussian mixture. Another example is when there is incomplete information about the inputs of a deterministic process. For instance, suppose that $\mathbf{f}(\mathbf{x}_1, \mathbf{x}_2)$ is a deterministic process that we wish to model but only have access to one of its two input variables. In this case, it might be necessary to model it as a random process $\mathbf{f}(\mathbf{x}_1; t)$ with some index set T such that $t \in T$. In fact, the general case is when $\mathbf{f}(\mathbf{x}; t)$ is a random function, with deterministic functions being special cases in which $\mathbf{f}(\mathbf{x})$ has a deterministic distribution that does not depend on t, i.e. $\mathbf{f}(\mathbf{x}) = \mathbb{E}[\mathbf{f}(\mathbf{x}; t)]$ for all t.

We will now consider the more general case where $\mathbf{f}(\mathbf{x})$ is a random variable, and the measurement noise is input-dependent (i.e. heteroscedastic) such that $\eta = \eta(\mathbf{x}) \sim \mathcal{N}(\mathbf{0}, \sigma_\eta^2(\mathbf{x})\mathbf{I})$. In this case, the true conditional distribution $p(\mathbf{y}|\mathbf{x})$ can be arbitrarily complex. As before, we are interested in learning an approximate distribution $\hat{p}(\mathbf{y}|\mathbf{x}; \mathbf{w})$ with weights $\mathbf{w}$ and use it to make predictions $\hat{y}$. Let $\mathbf{w}$ be distributed according to some distribution function $q(\mathbf{w}|D)$. The following notation is adopted for expectations over the aforementioned distributions, keeping the dependence of $\mathbf{y}$ and $\hat{\mathbf{y}}$ on $\mathbf{x}$ implicit:

$$\mathbb{E}_p[\cdots] = \int [\cdots] p(\mathbf{y}|\mathbf{x}) d\mathbf{y}$$

$$\mathbb{E}_{\hat{p}}[\cdots] = \int [\cdots] \hat{p}(\hat{\mathbf{y}}|\mathbf{x}) d\hat{\mathbf{y}}$$

$$\mathbb{E}_q[\cdots] = \int [\cdots] q(\mathbf{w}|\mathcal{D}) dw,$$

Applying a similar decomposition as in Equation 4.3 yields (see Appendix A for derivation):

$$\begin{aligned}
\iiint (\mathbf{y} - \hat{\mathbf{y}})^2 p(\mathbf{y}|\mathbf{x}) \hat{p}(\hat{\mathbf{y}}|\mathbf{x}) q(\mathbf{w}|\mathcal{D}) d\mathbf{y} d\hat{\mathbf{y}} dw &= \mathbb{E}_{q,p,\hat{p}}\left[(\mathbf{y} - \hat{\mathbf{y}})^2\right] \\
&= \mathbb{E}_q\left[(\mathbb{E}_p[\mathbf{f}] - \mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}])^2\right] \\
&\quad + \sigma_\eta^2 + \mathbb{V}_p[\mathbf{f}] + \mathbb{E}_q\left[\mathbb{V}_{\hat{p}}[\hat{\mathbf{y}}]\right] \\
&= (\mathbb{E}_p[\mathbf{f}] - \mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}])^2 + \mathbb{V}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]\right] \\
&\quad + \sigma_\eta^2 + \mathbb{V}_p[\mathbf{f}] + \mathbb{E}_q\left[\mathbb{V}_{\hat{p}}[\hat{\mathbf{y}}]\right] \qquad (4.5)
\end{aligned}$$

Now define:

$$\begin{aligned}
\sigma_e^2 &= \mathbb{E}_q\left[(\mathbb{E}_p[\mathbf{f}] - \mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}])^2\right] \\
&= (\mathbb{E}_p[\mathbf{f}] - \mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}])^2 + \mathbb{V}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]\right] \qquad (4.6)
\end{aligned}$$

$$\sigma_m^2 = \mathbb{V}_p[\mathbf{f}] \qquad (4.7)$$

Then we have:

$$\mathbb{E}_{q,p,\hat{p}}\left[(\mathbf{y}-\hat{\mathbf{y}})^2\right] = \sigma_{\mathrm{e}}^2 + \sigma_{\eta}^2 + \sigma_{\mathrm{m}}^2 + \mathbb{E}_q\left[\mathbb{V}_{\hat{p}}[\hat{\mathbf{y}}]\right] \tag{4.8}$$

Compared to Equation 4.3, the decomposition in Equation 4.8 introduces two extra terms. σ_{m}^2 expresses the stochasticity inherent in f, which we refer to as the modal uncertainty , and $\mathbb{E}_q\left[\mathbb{V}_{\hat{p}}[\hat{\mathbf{y}}]\right]$ is the variance in the prediction $\hat{y}$ averaged over all possible weights of the predictive distribution $\hat{p}$, and approximates $\sigma_{\eta}^2 + \sigma_{\mathrm{m}}^2$. We call σ_{η}^2 the aleatoric uncertainty because it has to be modeled explicitly and cannot be calculated from other model weights, unlike its soft counterpart. Moreover, the modal uncertainty can in principle be reduced by changing our assumption of the process so that it includes more relevant information as input.

We are interested in estimating the uncertainty quantities given in Equation 4.8. We will assume that the noise σ_{η}^2 can be explicitly modeled from the data, which means that σ_{m}^2 can be estimated from $\mathbb{V}_{\hat{p}}[\hat{\mathbf{y}}]$ which approximates $\sigma_{\eta}^2 + \sigma_{\mathrm{m}}^2$. However, calculating σ_{e}^2 requires knowledge of the true distribution of $p(\mathbf{y}|\mathbf{x})$, which we want to estimate in the first place. How can we estimate the epistemic uncertainty given only our approximate model?

Let $\mathbf{w}^\star$ be the optimal values of the weights $\mathbf{w}$ such that $\hat{p}(\mathbf{w}^\star) = p^\star \approx p(\mathbf{y}|\mathbf{x})$ and let $\mathbf{y}^\star \sim p^\star(\mathbf{x})$. Since $\mathbb{E}_p[\mathbf{f}] = \mathbb{E}_p[\mathbf{y}] \approx \mathbb{E}_{p^\star}[\mathbf{y}^\star]$, we can rewrite Equation 4.6 as:

$$\begin{aligned}\sigma_{\mathrm{e}}^2 &= (\mathbb{E}_p[\mathbf{f}] - \mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}])^2 + \mathbb{V}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]\right] \\ &\approx (\mathbb{E}_{p^\star}[\mathbf{y}^\star] - \mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}])^2 + \mathbb{V}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]\right]\end{aligned} \tag{4.9}$$

When the model is well trained, i.e. $D_{\mathrm{KL}}(p^\star||\mathbb{E}_q[\hat{p}]) < \varepsilon$ for some small positive ε, we have $\mathbb{E}_{q,\hat{p}}[\hat{y}] \approx \mathbb{E}_{p^\star}[y^\star]$, and the model bias becomes negligible. Then from Equation 4.9 we have:

$$\hat{\sigma}_{\mathrm{e}}^2 = \mathbb{V}_q\left[\mathbb{E}_{\hat{p}}[\hat{y}]\right] \tag{4.10}$$

Since the squared bias is strictly non-negative, the variance $\mathbb{V}_q\left[\mathbb{E}_{\hat{p}}[\hat{y}]\right]$ provides a lower bound on the epistemic uncertainty estimate, which means that the estimate can never exceed the actual epistemic uncertainty even if the approximation $\mathbb{E}_{q,\hat{p}}[\hat{y}] \approx \mathbb{E}_{p^\star}[y^\star]$ does not hold.

Assuming that the noise σ_{η}^2 is explicitly modeled as $\hat{\sigma}_{\eta}^2$, the aleatoric uncertainty can be estimated as:

$$\hat{\sigma}_{\mathrm{a}}^2 = \mathbb{E}_q[\hat{\sigma}_{\eta}^2] \tag{4.11}$$

Lastly, the modal uncertainty can be estimated as:

$$\hat{\sigma}_{\text{m}}^2 = \mathbb{E}_q\big[\mathbb{V}_{\hat{p}}[\hat{y}]\big] - \hat{\sigma}_{\text{a}}^2 \tag{4.12}$$

It is important here to note that the approximations above hold only when the model is assumed to be well trained. This is generally a valid assumption since is only reasonable to estimate the predictive uncertainty for a model that does indeed approximate the target function.

4.3 Stationarity of the Estimator as Epistemic Uncertainty

If we consider the estimator as a stochastic process, then we can interpret the epistemic uncertainty as defined earlier as a measure of the stationarity of the process. To see how, assume that we draw T samples of the parameter vector $\mathbf{w}$ from the distribution $q(\mathbf{w}|\mathcal{D})$ to obtain the collection of weights $\{w_t\} = \{w_1, \ldots, w_T\}$. Then, for a given value of $\mathbf{x}$, the corresponding collection of random variables $\{\hat{\mathbf{y}}(\mathbf{x}; w_t)\} = \{\hat{\mathbf{y}}(\mathbf{x}; w_1), \ldots, \hat{\mathbf{y}}(\mathbf{x}, w_T)\}$ defines a stochastic process, in which each of these variables has the distribution $\hat{p}(\mathbf{y}, w_T|\mathbf{x})$. This distribution depends on w_T, and the process $\{\hat{\mathbf{y}}(\mathbf{x}; w_t)\}$ is thus non-stationary as long as the distribution $q(\mathbf{w}|\mathcal{D})$ is non-deterministic.

In light of this, the epistemic uncertainty approximation as defined in Equation 4.10 can be interpreted as the degree of stationarity for $\hat{\mathbf{y}}(\mathbf{x}; \mathbf{w})$. If the process is stationary, then $q(\mathbf{w}|\mathcal{D})$ is deterministic and the variance in $\hat{y}$ due to q vanishes, yielding null epistemic uncertainty estimate. Conversely, the more non-stationary the process is, the more variance q will induce on $\hat{\mathbf{y}}$, and the higher the epistemic uncertainty estimate will become.

This result seems intuitive, since the true function $\mathbf{f}$ is assumed to be stationary in the first place, in the sense that the distribution $p(\mathbf{y}|\mathbf{x})$ does not depend on time. Therefore, it stands to reason that for values of $\mathbf{x}$ where the estimator closely matches the true density, the estimate $\hat{\mathbf{y}}$ will be stationary as well. This suggests an alternative way to estimate the epistemic uncertainty, namely by directly quantifying the stationarity of $\hat{\mathbf{y}}$. One possible way this can be done is by defining a metric for the variability among the set of distributions $\{\hat{p}(\mathbf{y}; w_t|\mathbf{x})\}$. However, this is left for future work.

4.4 Estimating Uncertainties in Bayesian MDNs

We will now turn to a concrete example of estimating the uncertainty in a DL model with a multimodal predictive distribution. Consider a Bayesian mixture density network (BMDN) [Bis94] with a Gaussian prior distribution over its weights $\mathbf{w}$, and suppose the posterior distribution is $q(\mathbf{w})$. The conditional density of the target data as predicted by the BMDN is given by

$$p(\hat{\mathbf{y}}|\mathbf{x},\mathbf{w}) = \sum_{k=1}^{K} \alpha_k(\mathbf{x};\mathbf{w})\, \mathcal{N}\big(\hat{\mathbf{y}};\mu_k(\mathbf{x};\mathbf{w}),\sigma_k^2(\mathbf{x};\mathbf{w})\big) \tag{4.13}$$

where K is the number of components, α_k are the mixture coefficients subject to $\sum_{i=k}^{K}\alpha_k = 1$, and $\mathcal{N}(\cdot;\mu,\sigma^2)$ is a Gaussian kernel with mean μ and variance σ^2. From Equation 4.13 we can obtain the following statistics:

$$\mathbb{E}[\hat{\mathbf{y}}|\mathbf{x},k,\mathbf{w}] = \mu_k(\mathbf{x};\mathbf{w}) \tag{4.14}$$

$$\mathbb{E}[\hat{\mathbf{y}}|\mathbf{x},\mathbf{w}] = \sum_{k=1}^{K} \alpha_k(\mathbf{x};\mathbf{w})\mu_k(\mathbf{x};\mathbf{w}) \tag{4.15}$$

$$\mathbb{E}[\hat{\mathbf{y}}|x] = \mathbb{E}_q\left[\sum_{k=1}^{K} \alpha_k(\mathbf{x};\mathbf{w})\mu_k(\mathbf{x};\mathbf{w})\right] \tag{4.16}$$

$$\mathbb{V}[\hat{\mathbf{y}}|\mathbf{x},k,\mathbf{w}] = \sigma_k^2(\mathbf{x};\mathbf{w}) \tag{4.17}$$

$$\mathbb{E}\big[\mathbb{V}[\hat{\mathbf{y}}|\mathbf{x},k,\mathbf{w}]|\mathbf{x},\mathbf{w}\big] = \sum_{k=1}^{K} \alpha_k(\mathbf{x};\mathbf{w})\sigma_k^2(\mathbf{x};\mathbf{w}) \tag{4.18}$$

$$\mathbb{V}[\hat{\mathbf{y}}|\mathbf{x},\mathbf{w}] = \mathbb{E}\big[\mathbb{V}[\hat{\mathbf{y}}|\mathbf{x},k,\mathbf{w}]|\mathbf{x},\mathbf{w}\big] + \mathbb{V}\big[\mathbb{E}[\hat{\mathbf{y}}|\mathbf{x},k,\mathbf{w}]|\mathbf{x},\mathbf{w}\big] \tag{4.19}$$

where we have used the law of total variance in Equation 4.19. By applying the law of total variance again, the total predictive variance of the BMDN can be written as

$$\begin{aligned}\mathbb{V}[\hat{\mathbf{y}}|\mathbf{x}] &= \mathbb{V}_q\big[\mathbb{E}[\hat{\mathbf{y}}|\mathbf{x},\mathbf{w}]|\mathbf{x}\big] + \mathbb{E}_q\big[\mathbb{V}[\hat{\mathbf{y}}|\mathbf{x},\mathbf{w}]|\mathbf{x}\big] \\ &= \mathbb{V}_q\big[\mathbb{E}[\hat{\mathbf{y}}|\mathbf{x},\mathbf{w}]|\mathbf{x}\big] + \mathbb{E}_q\Big[\mathbb{E}\big[\mathbb{V}[\hat{\mathbf{y}}|\mathbf{x},k,\mathbf{w}]|\mathbf{x},\mathbf{w}\big]|\mathbf{x}\Big] \\ &\quad + \mathbb{E}_q\Big[\mathbb{V}\big[\mathbb{E}[\hat{\mathbf{y}}|\mathbf{x},k,\mathbf{w}]|\mathbf{x},\mathbf{w}\big]|\mathbf{x}\Big]\end{aligned} \tag{4.20}$$

Equation 4.20 decomposes the predictive variance of the BMDN in terms of the contributions of the variances of the three distributions governing $\hat{\mathbf{y}}$. The first term is

the contribution of the distribution of the weights defined by q, the second is that of the individual components Gaussians defined by μ_k and σ_k, and the third is that of the categorical distribution of the components defined by α_k. Now suppose that the model of $p(\mathbf{y}|\mathbf{x})$ in Equation A.1 is a BMDN. Making the dependence of μ_k and σ_k on $\mathbf{x}$ and $\mathbf{w}$ implicit, as well as the conditioning on $\mathbf{x}$, then from Equations 4.10, 4.11, and 4.12 we have:

$$\begin{aligned}
\hat{\sigma}_{\text{e}}^2 &= \mathbb{V}_q\big[\mathbb{E}[\hat{\mathbf{y}}|\mathbf{x},\mathbf{w}]\big] \\
&= \mathbb{V}_q\left[\sum_{k=1}^{K}\alpha_k\mu_k\right] \\
&= \mathbb{E}_q\left[\left(\sum_{k=1}^{K}\alpha_k\mu_k\right)^2\right] - \mathbb{E}_q\left[\sum_{k=1}^{K}\alpha_k\mu_k\right]^2 \qquad (4.21)
\end{aligned}$$

$$\begin{aligned}
\hat{\sigma}_{\text{a}}^2 &= \mathbb{E}_q\left[\mathbb{E}\big[\mathbb{V}[\hat{\mathbf{y}}|\mathbf{x},k,\mathbf{w}]|\mathbf{x},\mathbf{w}\big]\right] = \mathbb{E}_q\big[\mathbb{E}\big[\sigma_k^2\big]\big] \\
&= \mathbb{E}_q\left[\sum_{k=1}^{K}\alpha_k\sigma_k^2\right] \qquad (4.22)
\end{aligned}$$

$$\begin{aligned}
\hat{\sigma}_{\text{m}}^2 &= \mathbb{E}_q\big[\mathbb{V}[\hat{\mathbf{y}}|\mathbf{x},\mathbf{w}]\big] - \hat{\sigma}_{\text{a}}^2 \\
&= \mathbb{E}_q\left[\mathbb{V}\big[\mathbb{E}[\hat{\mathbf{y}}|\mathbf{x},k,\mathbf{w}]|\mathbf{x},\mathbf{w}\big]\right] = \mathbb{E}_q\left[\mathbb{V}\big[\mu_k|\mathbf{w}\big]\right] \\
&= \mathbb{E}_q\left[\sum_{k=1}^{K}\alpha_k\mu_k^2 - \left(\sum_{k=1}^{K}\alpha_k\mu_k\right)^2\right] \qquad (4.23)
\end{aligned}$$

where we have used Equation 4.19 and Equation 4.22 in Equation 4.23. Consequently, Equation 4.20 becomes

$$\mathbb{V}[\hat{\mathbf{y}}|\mathbf{x}] = \hat{\sigma}_{\text{e}}^2 + \hat{\sigma}_{\text{a}}^2 + \hat{\sigma}_{\text{m}}^2 \qquad (4.24)$$

which shows that the approximations of the three kinds of uncertainties can be obtained by the decomposition of the total predictive variance of the BMDN.

The expectations with respect to $q(\mathbf{w},D)$ can be approximated from samples of the weights using Monte Carlo integration. In this case, the uncertainties in Equations 4.21, 4.22 and 4.23 can be approximated as:

$$\hat{\sigma}_{\text{e}}^2 \approx \frac{1}{T}\sum_{t=1}^{T}\left(\sum_{k=1}^{K}\alpha_{k,t}\mu_{k,t}\right)^2 - \left(\frac{1}{T}\sum_{t=1}^{T}\sum_{k=1}^{K}\alpha_{k,t}\mu_{k,t}\right)^2 \qquad (4.25)$$

$$\hat{\sigma}_{\text{a}}^2 \approx \frac{1}{T}\sum_{t=1}^{T}\sum_{k=1}^{K}\alpha_{k,t}\sigma_{k,t}^2 \qquad (4.26)$$

$$\hat{\sigma}_{\mathrm{m}}^2 \approx \frac{1}{T}\sum_{t=1}^{T}\left[\sum_{k=1}^{K}\alpha_{k,t}\mu_{k,t}^2 - \left(\sum_{k=1}^{K}\alpha_{k,t}\mu_{k,t}\right)^2\right] \tag{4.27}$$

where T is the number of samples drawn from the distribution of weights.

4.5 Experiments

In this section, the findings in previous sections will be verified in two experiments. The first experiment illustrates the estimation of the different kinds of uncertainty for a BMDN. The synthetic dataset created for this experiment is one-dimensional to allow for clear visualization. The second experiment involves a simulated robot kinematics problem, in which the robot has to learn its inverse kinematics function and estimate the uncertainty in its prediction. The experiment is intended to show how epistemic uncertainty can be used to guide decision-making.

4.5.1 Toy Problem

In this experiment, the uncertainty estimation methods discussed so far will be evaluated on toy one-dimensional datasets that can be easily visualized. We will examine the effects of low-data regimes as well as high-noise regimes on the estimates of the different kinds of uncertainty. We will also examine uncertainty estimates when the network is both interpolating and extrapolating. For the experiment, three synthetic datasets were created. The first dataset serves as the base case and consists of 2000 (x,y) samples representing the input and output of the process given by:

$$y = f(x) + \eta,$$

where $\eta \sim \mathcal{N}(0,\sigma_\eta^2)$ with $\sigma_\eta = 1$ and f is randomly selected to be one of the following functions with equal probability:

$$\begin{aligned} f_1(x) &= 7\sin(0.75x) + 0.5x \\ f_2(x) &= 3\sin(0.5x) + 3x - 5 \\ f_3(x) &= 2\sin(0.3x) - 0.2x + 20 \end{aligned}$$

The input $x \in R$ was drawn from a uniform distribution over the domain $[-30,30]$.

The second dataset was similar to the first, except that the data was decimated by a factor of 50 in the region $[-10,10]$ of the domain of f to create a low-data region.

Similarly, the third dataset was similar to the first except that the noise was increased to $\sigma_\eta = 3$ in the same region to create a high-noise region.

Three identical MDNs were trained, one for each dataset. Each MDN had three hidden layers of 256 ReLU units with a dropout rate of 0.2 and L_2 regularization with weight decay of 0.02 in all hidden layers. Batch normalization was used before all layers. Each MDN had three parallel output layers, one for α_k with softmax activation, one for μ_k with and one for e^{σ_k} both with linear activation. The Adam optimizer was used with a learning rate of 0.001 to minimize the negative log likelihood given by:

$$\mathcal{L}(\mathbf{w}) = -\log \sum_{k=1}^{K} \alpha_k(x;\mathbf{w})\, \mathcal{N}\big(y;\mu_k(x;\mathbf{w}),\sigma_k^2(x;\mathbf{w})\big) \tag{4.28}$$

The MDNs had $K = 10$ components, and we used $T = 50$ MC-dropout samples to make predictions and calculate the uncertainties given by Equations 4.25, 4.26, and 4.27 for each prediction. To make a prediction for a given test point, the averages $\bar{\alpha}_k = \frac{1}{T}\sum_{t=1}^{T}\alpha_k$, $\bar{\mu}_k = \frac{1}{T}\sum_{t=1}^{T}\mu_k$, and $\bar{\sigma}_k^2 = \frac{1}{T}\sum_{t=1}^{T}\sigma_k^2$ are calculated first. Afterwards, a sample is drawn from the categorical distribution defined by $\bar{\alpha}_k$ to select a component Gaussian $\mathcal{N}(\bar{\mu}_k,\bar{\sigma}_k^2)$, and then a sample is drawn from the latter to obtain $\hat{y}$.

Figure 4.1 shows the predictions and the estimates of the different kinds of uncertainties for an MDN trained on the three datasets for 3000 epochs. The test input was 1000 points drawn from a uniform distribution over the interval $[-30, 40]$. In addition to the estimates, the actual values for the uncertainties calculated from Equations 4.6, 4.7, and the actual noise value are also shown. For all the datasets, the estimated epistemic uncertainty accurately reflected the actual epistemic uncertainty in the extrapolating region ($x > 30$), which increases the further away we go from the training data. Moreover, the aleatoric uncertainty estimate increases rapidly when extrapolating and start taking arbitrarily large values. In the low-data regime (the area shaded yellow in the middle column in the figure), the epistemic uncertainty is much higher than in the base case, while the aleatoric uncertainty is only slightly higher. In contrast, in the high-noise region the aleatoric uncertainty is higher while the epistemic uncertainty is the same. The modal uncertainty traces the distance between the means of the Gaussian components in all cases.

The results also show that epistemic uncertainty estimates are higher in the extrapolating case than in the interpolating one. This is expected since NNs are better at interpolating than extrapolating. The epistemic uncertainty is also proportional to the distance of the input from the training set. Furthermore, the networks experience high

aleatoric uncertainty when extrapolating (not just epistemic uncertainty), since they have no way of knowing what is the actual source of uncertainty there. For the same reason, the networks sometime interpret some of the uncertainty in low data regions as aleatoric uncertainty as in the bottom center plot in Figure 4.1, but the effect is much less pronounced.

4.5.2 Robot Inverse Kinematics

In this experiment we will see how uncertainty estimation can be used to guide decision-making using a simulated robot inverse kinematics (IK) problem. IK is a good example of a multi-valued mapping that cannot be learned by a regular NN. Consider an over-actuated three-link robot arm moving in a two-dimensional plane. For some given joint angles (q_1, q_2, q_3), the position of the end-effector (x_1, x_2) is given by the IK as follows:

$$\left.\begin{aligned} x_1 &= L_1 \cos(q_1) + L_2 \cos(q_1 + q_2) + L_3 \cos(q_1 + q_2 + q_3) \\ x_2 &= L_1 \sin(q_1) + L_2 \sin(q_1 + q_2) + L_3 \sin(q_1 + q_2 + q_3), \end{aligned}\right\} \tag{4.29}$$

where L_1, L_2, and L_3 are the length of the links. For this experiment we will consider the case where $L_1 = 0.5$, $L_2 = 0.3$, and $L_3 = 0.2$. We will also constrain the range of q_1 to the interval $[-2\pi/3, 2\pi/3]$ and both q_2 and q_3 to $[-3.6, 3.6]$.

Now suppose the robot has no knowledge of its IK function and has to learn it in order to know how to reach any point in its workspace. The robot collects data for training by randomly setting its joint positions a number of times and observing the corresponding end effector positions (motor babbling). This scenario was simulated by generating 2000 points from a uniform distribution over the range of the joints as the target values, and Equation 4.29 was used to calculate the corresponding input values. Gaussian noise with std. deviation of 0.01 was added to the joint positions before training. An MDN was then used with the same architecture as in the previous experiment to learn the mapping, with the difference being the number of inputs and output dimensions. The MDN was trained for 3000 epochs. To test the model, another 2000 test points were generated randomly over the joint space and used as input to the model. The error was then measured as the euclidean distance (error) between the coordinates predicted by the model and the corresponding true coordinates calculated from Equation 4.29. Predictions were made in the same way as in the previous experiment.

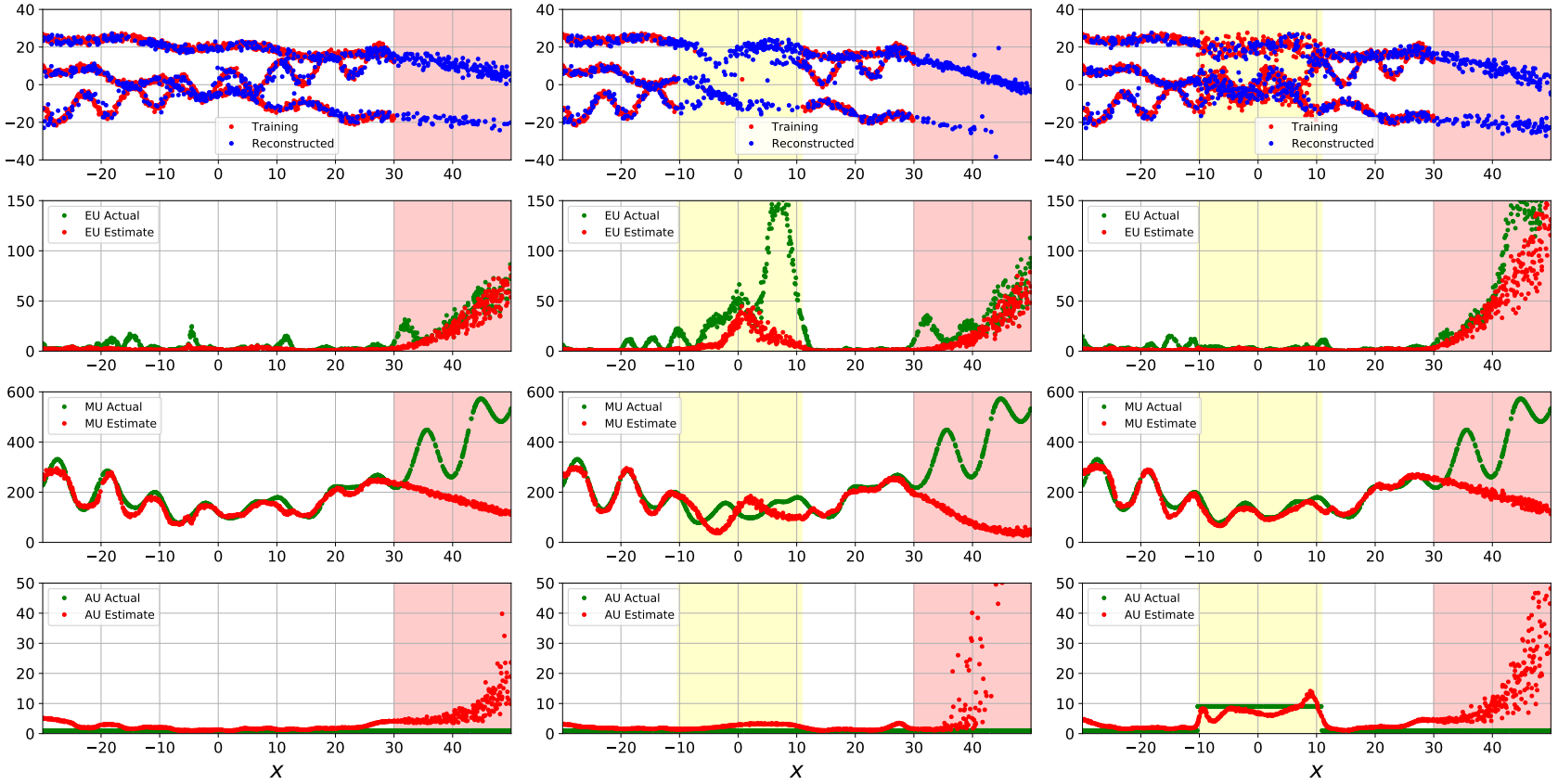


Figure 4.1: Estimation of uncertainties in the output of a BMDN. The left column is the base dataset. The region shaded in yellow is the low-data region in the middle column, and the high-noise region in the right column. The regions shaded in red are extrapolated out-of-distribution predictions. The second row shows the epistemic uncertainty (EU), the third shows the modal uncertainty (MU), and the bottom row shows the aleatroic uncertainty (AU).

Figure 4.2 shows the distribution of the training data over the workspace and the error in the test data, while Figure 4.3 shows the different uncertainty estimates for each output dimension. Even though the training points were sampled uniformly from the joint space, their distribution in Cartesian space in not uniform since there are points in the workspace that are more reachable than others (i.e. can be reached by more joint configurations). This leads to a low data region directly to the left of the base around $x_1 = -0.5$ and $x_2 = 0$, which in turn leads to high epistemic uncertainty in that region (Figure 4.3 bottom row). In general, it can be seen that high error is correlated with high epistemic uncertainty and aleatoric uncertainty. The modal uncertainty for each output dimension gives a measure of the distance in joint space between the number of different positions for the corresponding joint that can result in a certain end effector position. For example, the modal uncertainty is particularly high for q_1 in the low data region described before since it can only be reached by setting q_1 to one of the two extremes of its range. The modal uncertainty is also nil for all joints on the edges of the workspace since these point can only be reached through exactly one joint configuration, which is fully extending the arm.

To show how uncertainty estimation can be used in decision making, the trained BMDN (the robot) was presented with 20 points drawn from a uniform distribution over the workspace. The robot has to choose one point to try to reach, and gets a score equal to the error between the achieved position of the end-effector and that of the point. The goal of the robot is to choose a point that will minimize this score. Therefore, it has to make an informed decision on which point to attempt to reach. To this end, 10 identical MDNs were trained on 10 different randomly generated datasets, and the test was repeated 50 times for each MDN. The results were averaged over the 10 MDNs and over the 50 runs. The whole experiment was repeated 3 times, each time after the robot has collected a certain number of datapoints to train its IK. The amounts used for each experiment were 1000, 2000, and 3000 datapoints. This was done to highlight the effect of the abundance of data on the epistemic uncertainty.

Table 4.1 shows the cumulative score over the 50 runs averaged over the 10 MDNs for different uncertainty-based decision criteria. For each criteria, the robot chooses to reach for the point with the minimum value of the corresponding uncertainty. The results are shown for different amounts of training datapoints that the robot collected. The results show that choosing the point with the least epistemic uncertainty yields significantly better score than choosing at random, which supports the use of epistemic uncertainty as a heuristic for success. However, the epistemic uncertainty score is also

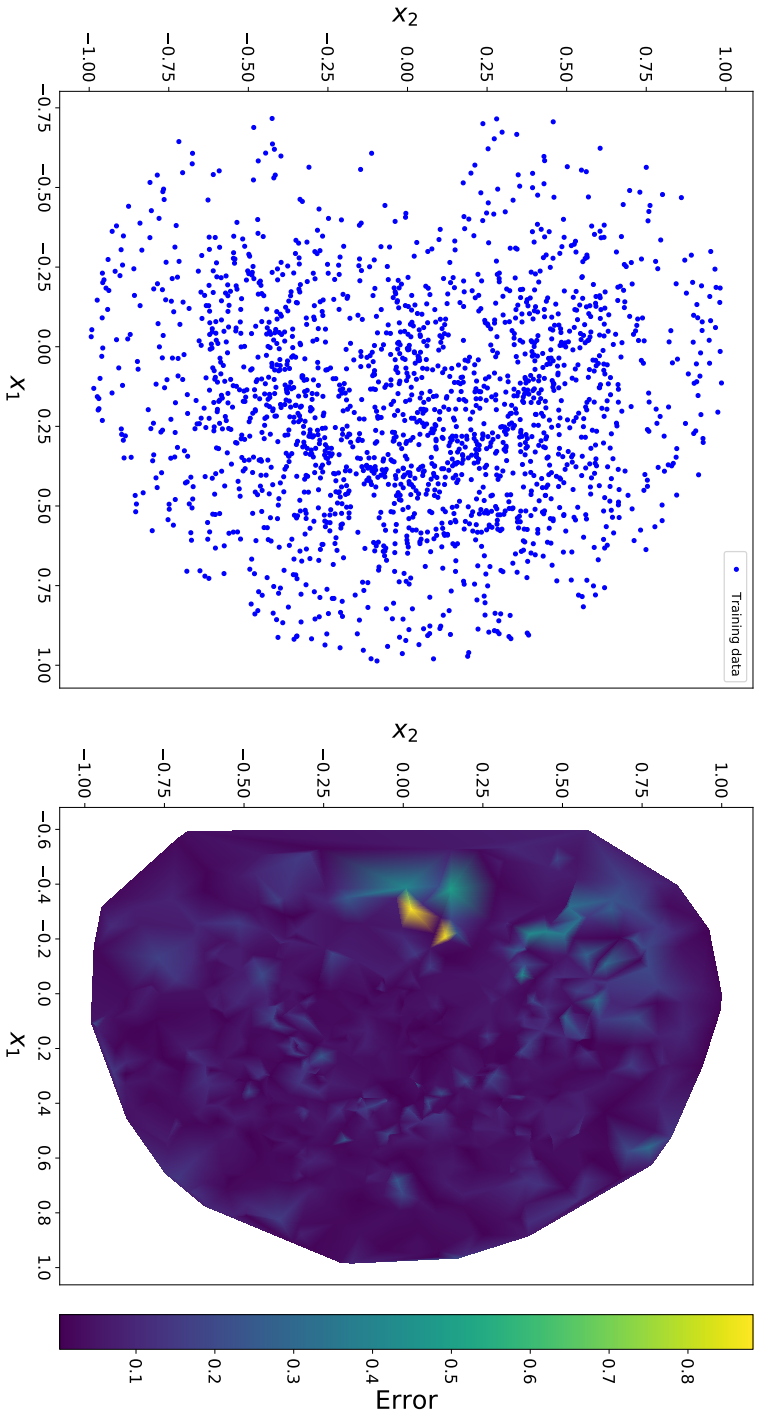


Figure 4.2: Left: Distribution of training datapoints. Right: distance between the test data and actual position achieved by the arm.

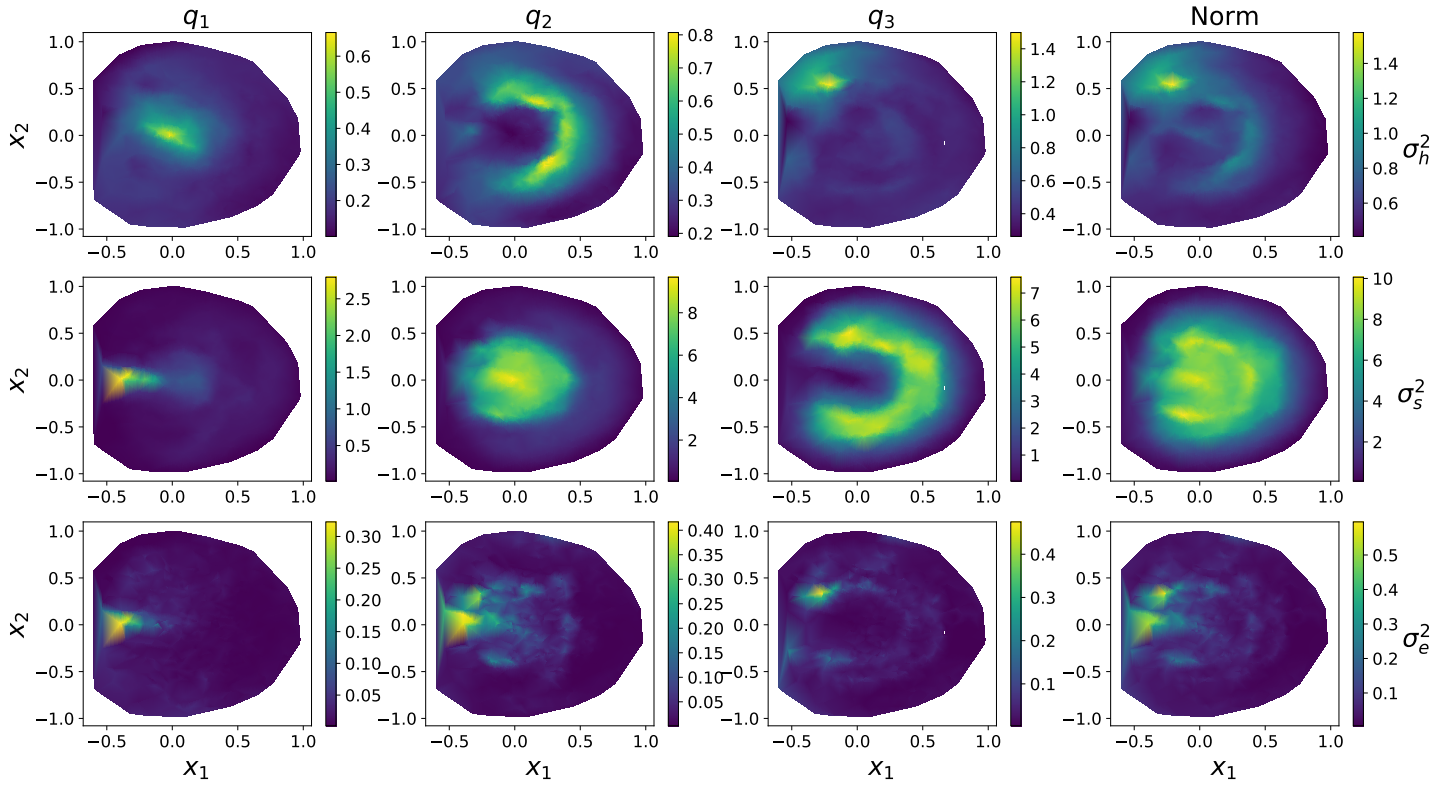


Figure 4.3: Different uncertainty estimates for all output dimensions for the inverse kinematics problem. The top row is the aleatoric uncertainty, the middle is the modal uncertainty, and the bottom is the epistemic uncertainty. The first three column from the left are the estimates for q_1, q_2 and q_3 respectively. The rightmost column is the L2 norm of the vector of estimates.

significantly worse than the minimum achievable on all points. This is expected since the epistemic uncertainty is just a heuristic and does not guarantee best results. Furthermore, in some cases the combination of epistemic uncertainty and aleatoric uncertainty can yield marginally better results than epistemic uncertainty alone. Unsurprisingly, the advantage of choosing based on uncertainty estimates diminishes as the amount of datapoints collected increases, since collecting lots of data decreases the variance in uncertainty across all points.

Data-points	epistemic	aleatoric	epistemic + aleatoric	min	max	rand
1000	3.89 (0.67)	3.88 (0.72)	3.98 (0.83)	0.92 (0.15)	29.24 (4.24)	6.35 (1.51)
2000	2.36 (0.45)	2.75 (0.85)	2.56 (0.52)	0.52 (0.06)	21.36 (3.8)	4.00 (0.94)
3000	2.37 (0.59)	2.37 (0.63)	2.35 (0.64)	0.45 (0.04)	19.12 (3.92)	3.92 (1.14)

Table 4.1: Cumulative results for the robot arm reaching experiment. The lower the score the better. Std. deviations are given in parentheses.

4.6 Summary and Discussion

In this chapter we have seen how uncertainty can be estimated for multimodal predictive density estimation with heteroscedastic noise, which is the most general case. We have seen how the error of the estimator can be decomposed in a Bayesian context into 3 components: epsitemic, aleatoric, and modal uncertainties. While epistemic and aleatoric uncertainties are present in all function approximators, modal uncertainty arises only in multimodal density estimation problems. It is thus important for such problems to isolate the different kinds of uncertainties, so that useful information can be extracted from the individual uncertainties. We have also seen how the three kinds of uncertainty can be estimated for MDNs, which is a type of NN used for multimodal predictive density estimation.

As a concrete realization of the concepts discussed in the chapter, a toy problem concerned with multimodal predictive density estimation was presented in which uncertainties was estimated using MC-dropout. The technique proved successful in estimating the different type of uncertainties in both high- and low-data regimes, in high-

and low-noise regimes, as well as in extrapolation and interpolation. To show how the epistemic uncertainty can be used to guide decision-making, an experiment involving a simulated robot learning its IK through motor babbling was conducted. The robot was tasked with minimizing the error when reaching for a point in its workspace. It used the epistemic uncertainty estimate to decide which point to reach for, yielding significantly lower error on average than choosing randomly.

Out of the three kinds of uncertainties discussed so far, perhaps the most important is epistemic uncertainty, since it is the only one we can do anything about. An agent can go look for more data to improve its model and reduce the epistemic uncertainty, but there is nothing it can do to reduce the aleatoric and modal uncertainties. This does not mean however that the latter two are useless. If the data is very noisy in some region, the agent can simply not trust its model in that region. It can know not to give much weight to its noisy predictions in that region. Likewise, if the modal uncertainty is high in some region, the agent should know to expect multiple different outcomes, and perhaps plan accordingly for all possible eventualities.

Chapter 5

An Architecture for Imagination-augmented DRL

In Chapter 3, we have seen how a stochastic environment model can be learned on-line, and how it can be used to generate imaginary rollouts. We have also seen that an agent trained solely on imaginary rollouts can perform reasonably well in the actual environment. However, training such an agent still suffers from the main drawback of model-based RL, namely sub-optimal asymptotic performance. The question this chapter tries to answer is: how can we utilize imagination to improve the sample-efficiency while achieving the same asymptotic performance of model-free RL?

This chapter describes an architecture that integrates imagination and model-free RL. In this approach, an agent encodes sensory information into low-dimensional representations, and learns a model of its environment on-line in latent space, while simultaneously learning an optimal policy. The model can be learned much faster than the policy, and therefore can be used to augment transitions collected from the real environment with synthetic transitions, improving the sample-efficiency of RL. The agent can simulate different scenarios internally and make use of this simulated experience just as if it was real experience. This approach requires no prior knowledge of the task; only the encoder needs to be pretrained on task-relevant images, and can generally be reused for multiple tasks.

The chapter is organized as follows. Section 5.1 describes the architecture and its inner workings. Section 5.2 presents an experimental evaluation of the architecture. Section 5.3 concludes the chapter with a summary and discussion.

5.1 Architecture

The architecture developed in this chapter is concerned with improving the sample efficiency of RL agents. The key idea is to incorporate model-free and model-based approaches to improve the sample efficiency while not sacrificing asymptotic performance. The basic concept of the architecture is similar to that of the Dyna-Q architecture discussed in Section 2.2.4. However, the architecture employs imagination-based learning instead of regular model-based learning. Imagination here refers to using generative models to not just predict transitions, but also to "imagine" new states.

The architecture consists of three main components: the vision encoder that encodes images into low-dimensional state representations, the environment model that predicts next states and rewards, and the controller that selects actions. The architecture also includes two separate memories: the real memory stores real experience generated by interaction with the environment, and the imaginary memory stores simulated transitions generated by the environment model.

Figure 5.1 shows an overview of the architecture. The agent starts by observing the environment through a camera at a certain timestep t. The raw image obtained is the environment state s_t, which get passed to the encoder that encodes it into a state vector z_t. The agent takes action a_t that influences the environment and changes its state. The agent then observes the new state s_{t+1}, encodes it into z_{t+1}, and observes the reward signal r_{t+1}. The encoded old and new states (z_t and z_{t+1}) are concatenated with the action a_t and the reward r_{t+1} to form a transition tuple $(z_t, a_t, r_{t+1}, z_{t+1})$, which is then stored in the real memory. Two sets of mini-batches of transitions are randomly sampled from the real memory and used to incrementally train the controller and the environment model each timestep. The environment model generates a number of imaginary rollouts starting from the current observes state z_t at each timestep. The generated imaginary transitions are stored in the imaginary memory. Mini-batches of imaginary transitions are randomly sampled from the imaginary memory to train the controller in conjunction with real transitions. Finally, the controller issues an action a_{t+1} that again influences the environment, starting the process for the new timestep $t+1$ and closing the loop. The details of the individual components and the process of generating imaginary rollouts are given in Section 3.1.

The architecture aims to boost the sample efficiency in RL by making the most use out of the data collected by the robot. The data is simultaneously used to train the controller and the environment model, which helps extract more useful information from it. It is assumed that the environment model can be learned faster than the optimal

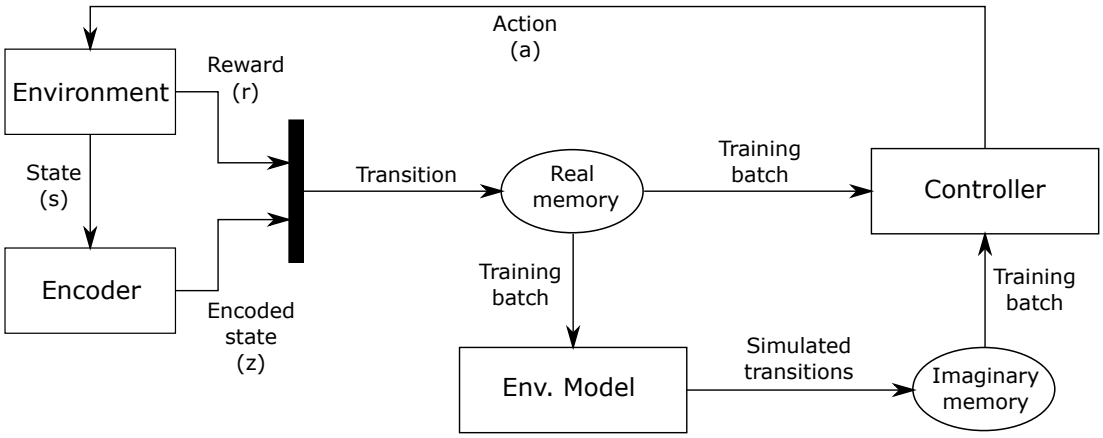


Figure 5.1: Overview of the proposed architecture. The controller influences the environment with an action, which produces state s and reward r. The encoder V encodes s into latent state vector z. The environment model can be trained on real transitions and then used to generate imaginary transitions. The controller can then be trained on both real and imaginary transitions.

policy, and therefore the controller would benefit from the data augmentation provided by the model.

5.2 Experiment

The experiment detailed in this section is designed to evaluate the architecture as a whole and to verify whether it improves sample efficiency. It also aims to evaluate the advantage of using uncertainty estimates of the environment model predictions as a limiting factor in using imaginary rollouts for training the controller. The experiment involves the Sawyer robot learning to solve a puzzle in which a human provides the goal state using gestures.

This section is organized as follows. Section 5.2.1 describes the experiment setup, providing details about the task to be learned, the environment built for it, and data collection. Section 5.2.2 provides implementation details of the architecture and the training procedure. Section 5.2.3 presents the results obtained and their analysis. Finally, Section 5.2.4 presents conclusions drawn from the results as regards the questions posed in the experiment.

5.2.1 Experiment Setup

To test the architecture, a task was designed in which a robot has to solve a puzzle based on pointing gestures made by a human. The robot sees three cubes with arrows painted on them, with each arrow pointing either up, down, left, or right. The human can point to any of the three cubes, but may not point to a different cube during the same episode. To successfully complete the task, the robot has to rotate the cubes so that only the arrow on the cube being pointed to is in the direction of the human, with the constraint that at no point should two arrows point to the same direction. Figure 5.2 shows the experiment in which the Sawyer robot acts as the agent. The task is similar to puzzle games typically used in studies about robot learning, such as the Towers of Hanoi puzzle [LSKD13].

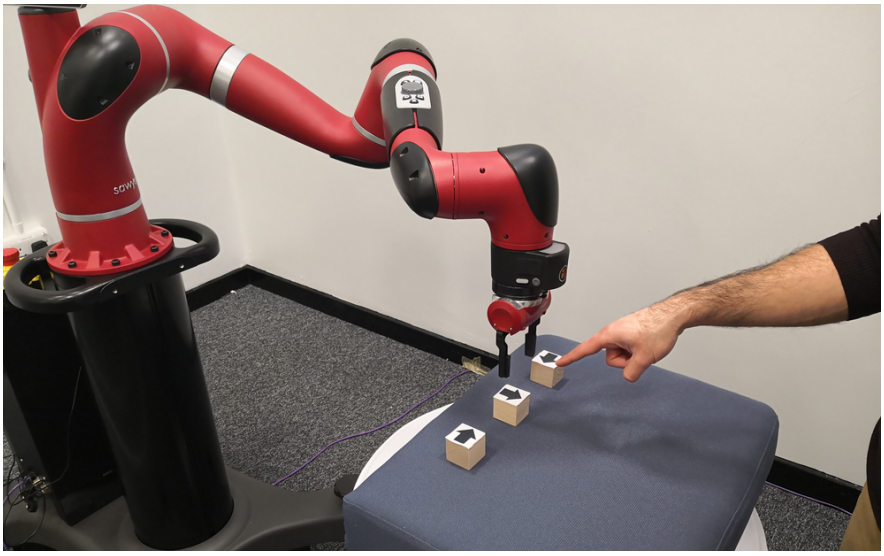
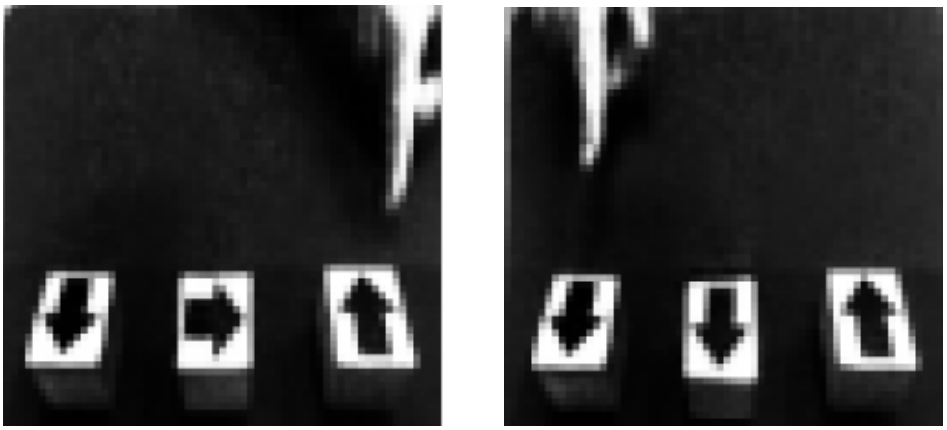


Figure 5.2: Experiments with the Sawyer robotic arm. The robot has to solve a puzzle by rotating the cubes to reach a goal state based on the pointing gesture by the human.

The task is formulated as an RL problem in which the agent can choose from 6 discrete actions at any given time. Three of the actions are for rotating any of the three cubes 90°clockwise, and the other three are for rotating counterclockwise. The robot gets a reward of +50 for reaching the goal state, -5 for reaching an illegal state, and -1 otherwise to penalize unnecessary actions and incentivize solving the task as efficiently as possible. An episode terminates if the robot reaches either a goal state or an illegal state, or after it performs 10 actions to prevent the robot from getting stuck in an episode. Figure 5.3 shows an example of goal and illegal terminal states.

To train the robot, a simulated environment was created for the agent to interact



(a) (b)

Figure 5.3: Examples of terminal states of the task. (a) is a goal state, while (b) is an illegal state.

with. The environment receives the selected action from the agent (the robot) as input, and outputs an image representing its state, along with a reward signal and a flag to mark terminal states. The environment is implemented as a finite state machine with 192 macrostates, where each macrostate is the combination of 3 microstates describing the directions of each of the three arrows, plus another microstate describing which box the hand is pointing to ($4^3 \times 3 = 192$). In the beginning of each episode, the environment is initialized with a random non-terminal state. Whenever the environment transitions to a new state, it outputs a random image from the set of images associated with that state, thus producing the observable state that the agent perceives.

To produce the images used for the environment, multiple image fragments were first collected for each of the possible microstates of the environment. Each of these fragments depicts a slight variation for the same microstate, for example slightly different box positions or hand positions. A pool of multiple image fragments is thus created for each possible microstate. To synthesize a complete image for a given macrostate, a random fragment is chosen for each of its constituent microstates. The fragments are then patched together to produce the final image, which then undergoes gamma correction to enhance contrast. For the experiments, 50 fragments were collected for each possible hand microstate, and 16 fragments for each possible arrow microstate, resulting in about 4×10^7 possible unique synthesized images. The images were taken with the Sawyer robotic arm camera (Fig. 5.2). For the experiments, 100,000 training images and 10,000 test images were synthesized.

5.2.2 Implementation and Training

The training procedure for the entire system can be summarized as follows:

1. Synthesize 100,000 training images and 10,000 test images.
2. Train the VAE on all training images.
3. Start collecting real experience and training the controller.
4. After some amount of episodes, start training the environment model on collected rollouts.
5. Use the environment model to generate imaginary rollouts as real experience is being collected.
6. Estimate the epistemic uncertainty for each transition generated by the model, and discard those with uncertainty above a certain value.
7. Continue training the controller using both real and synthetic rollouts.

The exact training procedure is given in Algorithm 1. In the following, the details of the training procedure for each component of the system are given.

Variational Autoencoder

To train the VAE, we split the grayscale training images along the horizontal axis into 3 strips, where each strip contains a single box. We then fed the strips into the VAE as 3 channels to help the VAE learn more task-relevant features. The architecture used for the VAE is the same as that used in Chapter 3 (Figure 3.3, except that the images are encoded into an 8-dimensional latent space. The VAE was trained on the 100,000 synthesized training images after scaling them down to a manageable 64×64 resolution for 1000 epochs. The VAE was trained to minimize the combined reconstruction and KL losses given by:

$$\mathcal{L}(\theta,\phi) = \sum_{n=1}^{N} \big[-\log p_\phi(x_n|z_n,\phi) + \beta \sum_{j=1}^{J} (1 + log\,\sigma_j^2(x_n,\theta) - \mu_j^2(x_n,\theta) - \sigma_j^2(x_n,\theta)) \big], \tag{5.1}$$

where the encoder and decoder networks are parameterized with θ and ϕ respectively, N is the number of training examples, $J =$ is the dimensionality of the latent space, and

Algorithm 1: Training procedure for agents.

Require: Pretrained encoder V
Initialize controller C and environment model E_θ
Initialize real memory M_R and imaginary memory M_I
for $e = 0$ **to** *num_episodes* **do**
    Observe initial state s_0
    $s_t = s_0$
    while s_t *is not terminal* **do**
        Use V to encode s_t into μ_t and σ_t
        Apply action $a_t = C(\mu_t)$
        Observe s_{t+1}, reward r_t, terminal signal d_{t+1}
        Encode s_{t+1} into μ_{t+1} and σ_{t+1}
        Save transition $(\mu_t, \sigma_t, a_t, \mu_{t+1}, \sigma_{t+1}, r_t, d_{t+1})$ in M_R
        for $i = 0$ **to** N_E **do**
            Train E_θ on minibatch from M_R
        for $i = 0$ **to** N_R **do**
            Train C on minibatch from M_R
        if $e \geq I_{start}$ **then**
            Use E_θ to generate I_B imaginary rollouts of depth I_D
            foreach *imaginary transition* T **do**
                Use MC-dropout to estimate the uncertainty σ_T
                if $\sigma_T >$ *threshold* **then**
                    Discard T and all subsequent transitions in the rollout
            Save imaginary transitions in M_I
            for $i = 0$ **to** N_I **do**
                Train C on minibatch from M_I
        $s_t = s_{t+1}$

β is the weighting factor of the KL divergence from the prior. See Section 2.1.4 for more details on training VAEs. In general, increasing β yields more efficient compression of the inputs and leads to learning independent and disentagled features, at the cost of worse reconstruction [HMP+17]. For this experiment, it was found that $\beta = 4$ produced the best results. The Adam optimizer was used with a learning rate of 0.0005 and a batch size of 2000.

The vision encoder network used in the experiment is simply the encoder part of the VAE. The 8-dimensional mean of the distribution of the latent vector produced by the encoder was taken to be the latent space vector.

Controller

The controller was implemented as a DQN and trained on the simulated environment in a regular Q-learning setting. The environment would provide an image representing the state, which gets encoded into a state vector by the encoder and passed to the DQN. The controller then outputs an action based on the Q-function.

The controller was implemented as a DQN consisting of 3 hidden layers (512 ReLU, 256 ReLU, 128 ReLU) and a linear output layer. It was updated once on a batch of 64 real transitions and once on a batch of 64 imaginary transitions each timestep. It was found that for such a relatively simple task, updating the controller more often led to worse performance. It was also found that using popular DQN extensions like a separate target network or prioritized experience replay did not significantly affect performance. The DQN was trained to minimize MSE given by:

$$\mathcal{L}(y,\hat{y}) = \sum_{n=1}^{N} (y_n - \hat{y_n})^2, \tag{5.2}$$

where N is the number of training examples, $\hat{y}_n = Q(s_t, a_t)$ is the output of the DQN for the state (s_t) at time t, and $y_n = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$ is the bellman target. The Adam optimizer was used with a learning rate of 0.001.

When selecting actions, the controller used an ε-greedy strategy with an exponentially decreasing exploration rate ε given by:

$$\varepsilon = \varepsilon_{min} + (\varepsilon_{\max} - \varepsilon_{\min}) e^{-\lambda t} \tag{5.3}$$

with $\varepsilon_{\min} = 0.001$, $\varepsilon_{\max} = 0.8$, $\lambda = 0.03$, and t is the time step.

Environment Model

The environment model is composed of three networks: an MDN for the dynamics model, the reward model, and the terminal state classifier. The MDN had 3 hidden layers of 256 ReLU units with 50% dropout, and 3 parallel output layers for the distribution parameters: one for the mixture coefficients with softmax activation, one for means and one for the logarithm of variances both with linear activation. The input to the MDN consisted of the state vector concatenated with the one-hot encoded action vector.

The MDN was trained to minimize the negative log likelihood given by:

$$\mathcal{L}(y,w) = \sum_{n=1}^{N} \Big[-\log \sum_{k=1}^{K} \alpha_k(x_n;w)\, \mathcal{N}(y_n;\mu_k(x_n;w),\sigma_k(x_n;w)^2) \Big], \tag{5.4}$$

where w is the vector of network weights, y is the vector of target next states, N is the number of training examples, $K = 5$ is the number of Gaussian components, α_k is the probability of component k, and μ_k and σ_k are the mean and variance vectors of the k-th component respectively. The network was trained with the Adam optimizer with a learning rate of 0.001.

When collecting transitions, the parameters μ and σ produced by the vision encoder for each frame were stored. Latent space vectors were then sampled from $\mathcal{N}(\mu,\sigma)$ when constructing a training batch. This ensures that the state vectors in each batch were unique, and slightly different from any previous one obtained. This form of data augmentation was found to greatly improve the generalization and performance of the model.

The reward model (r-network) had 3 hidden layers of 512 ReLU units each with 50% dropout, and a linear output layer. It was trained to minimize the logarithmic hyperbolic cosine loss given by:

$$\mathcal{L}(y,\hat{y}) = \sum_{n=1}^{N} \log\big(\cosh(\hat{y}_n - y_n)\big), \tag{5.5}$$

where y and $\hat{y}$ are the target and the predicted output respectively, and N is the number of training examples. The Adam optimizer was used with a learning rate of 0.001.

The terminal state classifier (d-network) had 2 hidden layers of 256 ReLU units each with 50% dropout, and a sigmoid output layer. It was trained to minimize the binary cross-entropy loss given by:

$$\mathcal{L}(y,w) = \sum_{n=1}^{N} y_n \log p_w(y_n) + (1-y_n)\log\big(1-p_w(y_n)\big), \tag{5.6}$$

where w is the vector of network weights, y is the target binary variable representing whether a state is terminal or not, $p_w(y)$ is the predicted probability of the the state being terminal, and N is the number of training examples.

During training, the MDN, the r-network and the d-network were all updated 16 times on batches of 512 transitions each timestep using the Adam optimizer with a

learning rate of 0.001. For extra regularization, a constraint was imposed on the network weights for all the networks such that the norm of the weights in each layer does not exceed 3.

Uncertainty Estimation

To prevent erroneous imaginary data from being used to train the controller, the epistemic uncertainty was estimated for the data generated by the environment model. Before getting stored in the imaginary memory, MC-dropout was applied to all transitions in the imaginary rollouts generated by the environment model. Transitions produced with uncertainty above a certain threshold were discarded, along with all subsequent transitions in their respective rollout. Thus, only transitions produced with some certainty get stored in memory.

The process of estimating the epistemic uncertainty with MC-dropout proceeded as follows. First, each state-action input to the environment model was duplicated T times. Afterwards, all the duplicated inputs were fed to the model with dropout at test time enabled, producing T samples of predictions. Even though the input is the same, each prediction is different since a different dropout mask is applied for each duplicate. In other words, each prediction is obtained using a different sample from the distribution of the MDN weights. Finally, the epistemic uncertainty $\hat{\sigma}_{\mathrm{a}}^2$ can be estimated from the outputs as:

$$\hat{\sigma}_{\mathrm{e}}^2 \approx \frac{1}{T}\sum_{t=1}^{T}\left(\sum_{k=1}^{K}\alpha_{k,t}\mu_{k,t}\right)^2 - \left(\frac{1}{T}\sum_{t=1}^{T}\sum_{k=1}^{K}\alpha_{k,t}\mu_{k,t}\right)^2 \tag{5.7}$$

where $\alpha_{k,t}$ is the weight of component k for sample t, and $\mu_{k,t}$ and $\sigma_{k,t}$ are the mean and variance vectors of the k-th component respectively.

Parameters

When training the agents, the depth of imaginary rollouts I_D was set to 10, and the breadth I_B to 3. The size of the real memory was 50,000 transitions, and that of the imaginary memory was 3,000. It was found that training the controller only on recently generated transitions leads to better performance, since more recent imaginary data are more accurate as the environment model gets better. This was achieved by both limiting the imaginary memory size, and generating multiple rollouts simultaneously. Furthermore, it was found that setting the update rate of the controller on both

real and imaginary transitions (N_R and N_I in Algorithm 1) to more than 1 can lead to stability issues. Another parameter that had to be tuned was the number of episodes to wait before staring to generate imaginary rollouts (I_{start} in Algorithm 1), since the environment model will produce erroneous predictions early on in the training. It was found that waiting for about 1000 episodes provides best results. The uncertainty threshold for discarding imaginary transitions was set to 0.3, as this value provided the most advantage in the experiment. In general, hyperparameters were chosen based on a trial-and-error approach and grid search.

5.2.3 Results

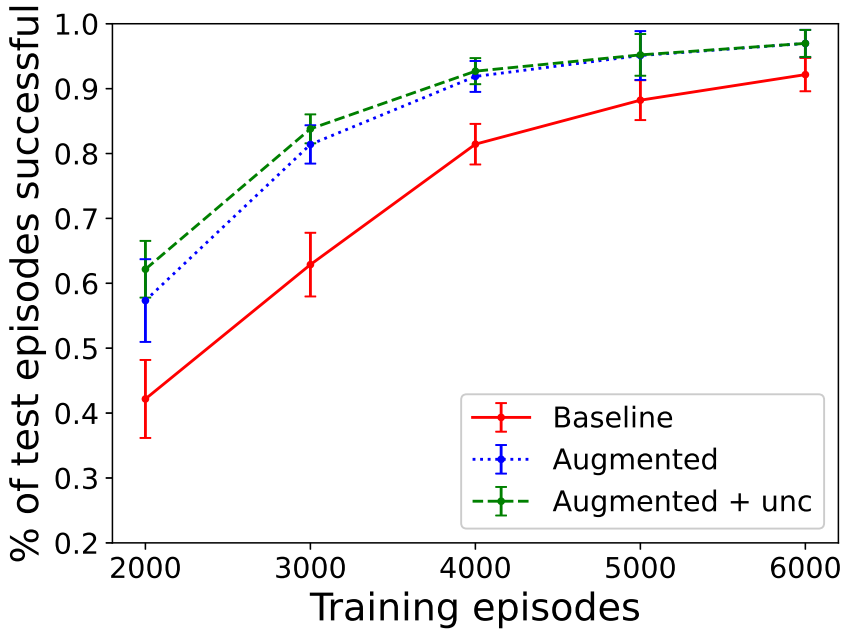
The purpose of the experiment is to test two hypotheses. First, that augmenting the data with imaginary rollouts leads to faster learning, and second, that limiting the use of imaginary data based on uncertainty estimates leads to better performance.

To test the first hypothesis, the performance of agents augmented with imaginary transitions is compared with a baseline DQN trained only on real transitions. To aid comparison, all hyperparameters and architectural choices were the same for augmented agents and the baseline DQN. For a given number of training episodes, 5 agents were trained from scratch and then tested on the simulated environment for 1000 episodes. The percentage of successfully completed episodes of all 5 agents in all test runs was then averaged to produce the final result. The experiment was repeated using agents trained for 2000, 3000, 4000, 5000, and 6000 episodes.

To test the second hypothesis, the experiment was repeated but without limiting the use of imaginary data based on uncertainty. In other words, the experiment was repeated without step 6 in the training procedure described in Section 5.2.2. The performance of agents with and without uncertainty estimation was compared.

Figure 5.4 shows the results of the experiment on the arrow puzzle task. Agents trained using the architecture performed significantly better than the baseline DQN when trained for a small amount of episodes. For agents trained for just 2000 episodes, augmented agents with uncertainty estimation successfully completed 44.4% more episodes than the baseline, compared to just 35.9% more episodes without uncertainty estimation. The performance gain then starts to decline the more episodes the agent is trained. This is to be expected since at higher episodes, the agent has collected enough real transitions and no longer needs the extra data generated by the environment model. Moreover, the gain due to limiting training on uncertain model predictions declines similarly as well, since the epistemic uncertainty of the environment model

decreases with more data. These results are illustrated in Figure 5.5, which shows the performance gain for augmented agents with and without uncertainty estimation compared to the baseline. The performance gain was calculated as $(P_{\text{aug}} - P_{\text{base}})/P_{\text{base}}$, where P_{aug} and P_{base} are the percentages of test episodes completed successfully for augmented and baseline agents, respectively. Table 5.1 shows the exact results for the experiment, while Table 5.2 shows the performance gain.



(a)

Figure 5.4: Test results for various numbers of training episodes for the task. Error bars represent standard deviations.

Episodes	Base DQN	Augmented	Aug. + unc.
2000	42.18 (6.01)	57.34 (6.37)	62.17 (4.37)
3000	62.88 (4.91)	81.4 (2.95)	83.8 (2.23)
4000	81.44 (3.13)	91.88 (2.38)	92.69 (2.01)
5000	88.22 (3.07)	95.1 (3.76)	95.2 (3.21)
6000	92.16 (2.57)	96.96 (2.1)	96.97 (2.08)

Table 5.1: Mean percentage of successful test episodes for various numbers of training episodes for the arrow puzzle task. Std. deviations are given in parenthesis. For reference, a random agent scored 3.72%.

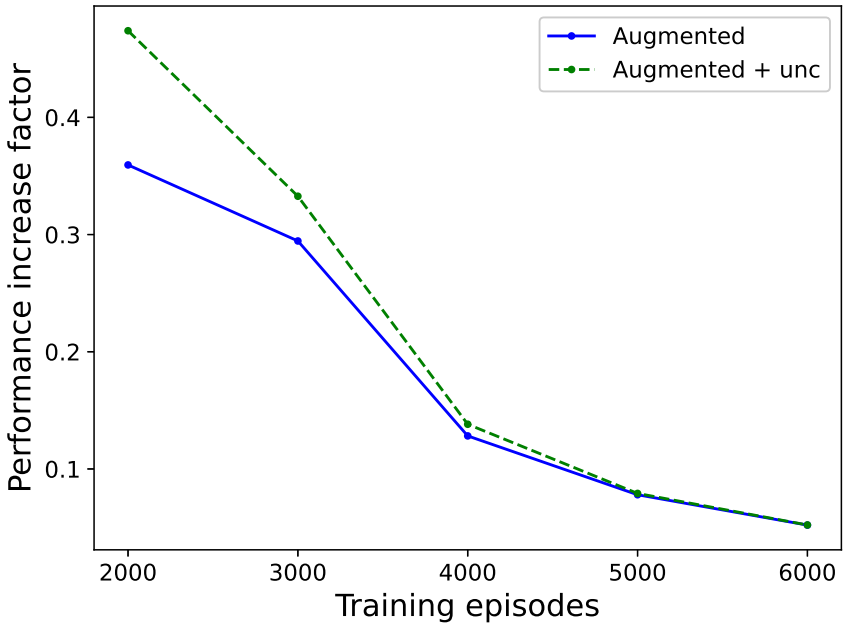


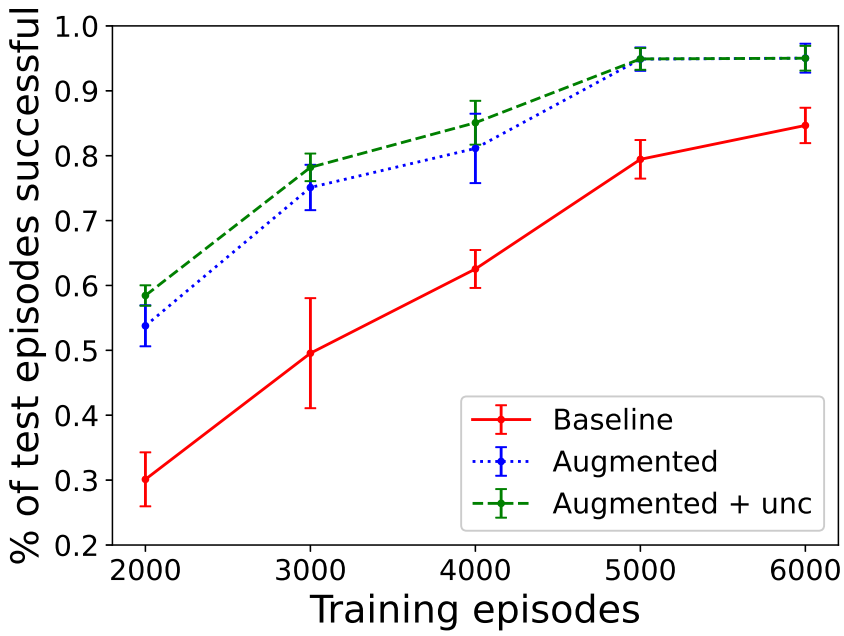
Figure 5.5: Performance gain for augmented agents in the arrow puzzle task. Agents were tested after training for different numbers of episodes given in the horizontal axis.

Episodes	Augmented	Aug. + unc.
2000	35.94	47.39
3000	29.45	33.27
4000	12.81	13.81
5000	7.79	7.91
6000	5.2	5.22

Table 5.2: Average percentage performance gain for augmented agents with and without uncertainty estimation.

Relationship Between Task and Environment Complexity

The gain in performance from using synthetic data is attributed to the relative simplicity of the environment dynamics with respect to the task itself. It is interesting to see the effect of this difference in complexity on the performance gain. To this end, another experiment was conducted using a slightly more difficult variation of the task. The difficulty of the task was increased while keeping the dynamics the same by additionally requiring the goal state not to have any arrows pointing towards the agent. All the architectural and parameter choices were the same for this new variation of the task, and the same training and testing procedures were used. Figure 5.6 shows the results for the difficult variation of the task, while Figure 5.7 shows the performance gain of augmented agents with and without uncertainty. Augmented agents showed even greater performance gain compared to the baseline DQN, with up to 94% increase in performance at 2000 training episodes. This shows that the performance increase due to using synthetic transitions is proportional to the difference in complexity between the task itself and the environment dynamics. The exact results are listed in Table 5.3, while Table 5.4 lists the performance gain with and without uncertainty estimation.



(a)

Figure 5.6: Test results for various numbers of training episodes for the difficult variation of the task. Error bars represent standard deviations.

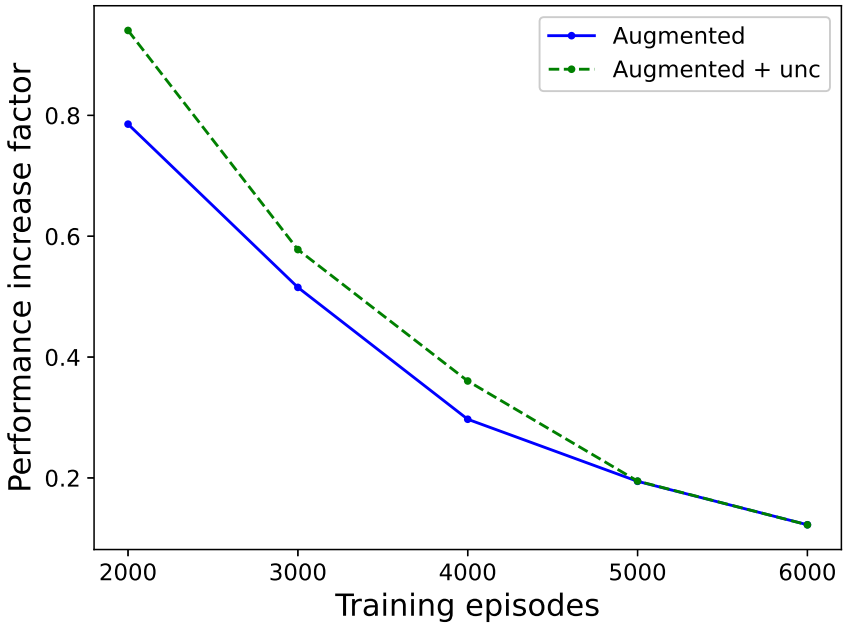


Figure 5.7: Performance gain for augmented agents in the difficult variation of the arrow puzzle task. Agents were tested after training for different numbers of episodes given in the horizontal axis.

Episodes	Base DQN	Augmented	Aug. + unc.
2000	30.12 (4.16)	53.78 (3.16)	58.45 (1.57)
3000	49.56 (8.48)	75.1 (3.5)	78.2 (2.13)
4000	62.54 (2.92)	81.12 (5.35)	85.08 (3.38)
5000	79.44 (2.97)	94.88 (1.8)	94.91 (1.67)
6000	84.66 (2.73)	95.03 (2.23)	95.02 (1.91)

Table 5.3: Mean percentage of successful test episodes for various numbers of training episodes for the difficult variation of the task. Std. deviations are given in parenthesis. For reference, a random agent scored 3.39%.

Episodes	Augmented	Aug. + unc.
2000	78.55	94.05
3000	51.53	57.78
4000	29.7	36.04
5000	19.43	19.47
6000	12.24	12.23

Table 5.4: Average percentage performance gain for augmented agents with and without uncertainty estimation for the difficult variation of the task.

Generating Plans

One of the advantages of learning an environment model is that it allows a trained agent to produce entire plans given only the initial state[1]. This can be achieved by initializing the environment model with the initial state, and then generating an imaginary rollout in which the controller always chooses the optimal action for each state. To demonstrate this, controller and an environment model were deployed on the Sawyer robotic arm (Fig 5.2), where both networks had been previously trained for 6000 episodes using the training method described in Section 5.2.2. Afterwards, tests were conducted to evaluate the planning capabilities of the system. Each test began by setting the cubes to a random state, with the experimenter pointing to a random cube. Then, the robot observed the configuration with the camera, and was asked to produce a plan consisting of a trajectory of actions to solve the task in its original form. The robot can execute the plan by selecting successively from a set of pre-programmed point-to-point movements to rotate the boxes. Out of 20 test runs, the robot successfully solved the task 17 times. The correct generated plans varied in length from 1 to 5, depending on the initial state. Moreover, the generated plans for all successful runs were optimal, containing only the fewest possible actions required to solve the task. Fig 5.8 shows an example of an imaginary rollout according to an optimal plan of length 5 as generated by the agent.

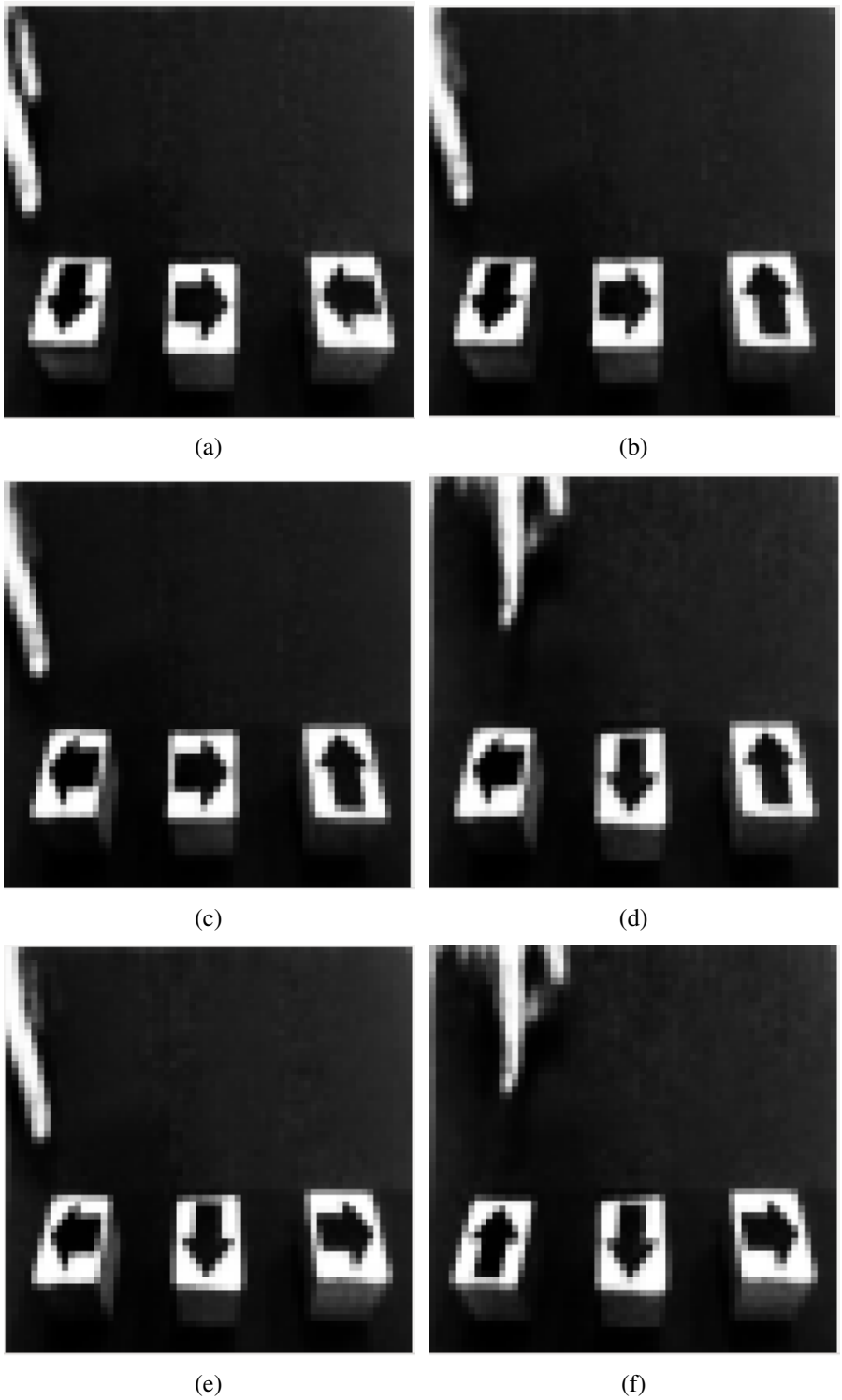
Model Generalization

One of the interesting results we noticed is that the model showed some generalization capabilities to transitions it had not experienced before. Since episodes always terminated after encountering a terminal state, the model never experienced any transitions from this kind of state. To test model generalization, we deliberately set the model state to a random terminal state 20 times, and then asked it to predict the next state for a random action each time. A model trained for 5000 episodes was able to correctly predict the next state 75 % of the time. Fig 5.9 shows an example of model prediction for unseen transitions.

5.2.4 Conclusion

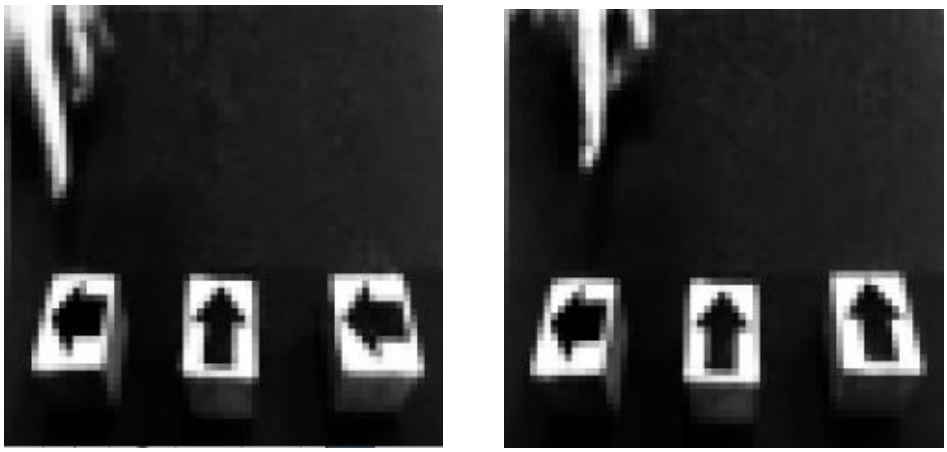
The experiment presented in this chapter was designed to test two hypotheses. The first hypothesis is that faster learning can be achieved by training an agent on imaginary

[1] This is only possible for environments with deterministic underlying dynamics



(a) (b) (c) (d) (e) (f)

Figure 5.8: An example of an imaginary rollout of length 5. (a) is the initial state as observed by the robot. (b) through (f) are imagined next states after successively applying actions in the optimal plan. The visualizations of the model predictions were obtained by mapping the latent space vectors to images via the decoder part of the VAE.



(a) (b)

Figure 5.9: An example of model prediction for unseen transitions. The action selected here is to rotate the rightmost cube clockwise. (a) is the state before the action, and (b) is the state after.

rollouts in conjunction with real experience. The second hypothesis is that limiting the use of imaginary data based on uncertainty estimates of model predictions leads to better performance.

Results show that a significant gain in performance was achieved by imagination-augmented agents compared to base DQN agents. This gain was highest when the agents were trained for fewer episodes, and vanished as the number of training episodes increase. This shows that using imaginary rollouts leads to faster learning when data is scarce. Moreover, the performance gain increased when the difficulty of the task increased while keeping the environment dynamics the same. This suggests that the gain is proportional to the difference in complexity between the task and the environment dynamics. However, as more and more real data becomes available to the agent, the benefit from data augmentation becomes insignificant. Results also showed that estimating the epistemic uncertainty of the environment model and discarding prediction above a certain uncertainty threshold improved performance. Uncertainty estimation ensures that the agent's memory is not contaminated by erroneous transitions, and removes a significant source of noise in the data.

5.3 Summary and Discussion

In this chapter, an architecture was presented that allows an agent to use imaginary rollouts and use them in conjunction with real experience to improve its policy. This

allows the agent to significantly reduce the amount of interactions it needs to make with the actual environment. This is especially useful for tasks involving real robots in which collecting real data can be prohibitively expensive. An algorithm was developed to train both the agent and the environment model simultaneously, and to train a controller on synthetic data in conjunction with real data. The architecture was validated through an experiment involving a high-level robotic task in which an agent has to simultaneously interpret a human gesture and solve a puzzle based on it. Results show that agents augmented with synthetic data outperform baseline methods especially in situations where only limited interaction data with the environment are available.

One of the main challenges in learning a model online is avoiding overfitting on the small subset of data that are made available early in the training. A model can easily get stuck in a local minimum if it gets trained excessively on initial data, and fail to converge later to an acceptable loss value in a reasonable amount of time as more data are made available[2]. This problem was alleviated through three things. First, the model capacity was limited by deliberately choosing smaller model sizes. Second, a probabilistic approach was adopted to encoding latent space representations and modeling environment dynamics. Third, high dropout rates were employed in the models. It was also found that selecting an unnecessarily large latent space dimensionality leads to worse models.

Probabilistic models are also much more robust, which is essential when using the dynamics model in closed loop to generate rollouts. Traditional models based on point estimates will produce some error in prediction, which will quickly compound resulting in completely erroneous predictions sometimes as early as the second pass. This of course makes using imaginary rollouts detrimental to learning.

The ability to learn stochastic models can be useful even for environments whose underlying dynamics are deterministic. An environment with deterministic underlying dynamics can have stochastic observable dynamics, since each latent state of the environment can produce multiple observable states. For example, the task we used for the experiments has deterministic underlying dynamics, since the configuration of the arrows will always change in the same way in response to a certain action. However, the observable state will change stochastically. The positions of the boxes or the hand may differ for the same configuration. The agent has no knowledge of the underlying dynamics since it only has access to observable states. Therefore, it needs to be able to

[2]When trained online, high-capacity models often exhibited a behaviour reminiscent of the Dunning-Kruger effect. They would achieve a very low loss value early in the training, which would quickly rise as more data are acquired, before eventually settling at a value in between.

model the observable dynamics stochastically in order to produces realistic imaginary rollouts.

The generalization capabilities of the dynamics model can in principle be used to facilitate learning other similar tasks. The two variations of the task we used for the experiments share the exact same dynamics; they are only different in the definition of the reward functions. Indeed, for any given dynamics, an arbitrarily large family of tasks can be defined by specifying different reward functions. If learning the reward function can be separated from learning the dynamics, and assuming that the former is easier to learn than the latter, then learning new tasks in the same family will become much faster once the agent learns a dynamics model. However, this is left for future work.

The architecture presented is general enough to work with any model-free RL algorithm, not necessarily DQN. DQN was chosen here as it seems the most suitable algorithm for the task. Any algorithm that can make use of rollouts can be used just as well.

Even though the sample efficiency was drastically improved by using the architecture, the thousands of transitions that were collected for training were still too much to collect physically. The result was that a simulated environment had to be created to train the robot. The models performed reasonably well however when tested in the real world, suggesting that the simulation was faithful to the real world. Hopefully in future work the sample efficiency can be improved further to allow training in the real world directly.

Chapter 6

Conclusion

The main goal of the thesis was to develop an architecture that improves the sample efficiency of RL algorithms by incorporating a mechanism for imagination. The emphasis was on robotic applications, where collecting real experience can be prohibitively costly. Now that the concepts and experiments of the thesis were presented in detail, it is time to revisit the contributions to knowledge more thoroughly. The results of the experiments will be assessed and conclusions drawn in light of the research questions posed in Section 1.4.

The first research question was about how to implement a mechanism for visual imagination in RL agents. In chapter 3, such a mechanism was implemented by combining three components. The first of these was the vision encoder, which transforms images into abstract low-dimensional state representations. The encoder was implemented as the encoder part of a VAE, which was pre-trained on task-relevant images. The second component was the controller, which selects an action for each state. The controller was implemented as a DQN, but could be kept relatively simple since much of the complexity required to process high-dimensional input images was relegated to the encoder. The third component was the environment model, which predicts the next state given the current state and the selected action. The environment model was implemented as an MDN, which could effectively model stochastic environments in which a state can lead to multiple next states given the same action.

Chapter 3 also presented an experiment that served as a proof of concept for the architecture. In the experiment, a simulated environment was created for the agent in which a human requests the agent to pick up objects or hand them over using gestures. The purpose of the experiment was to verify that an environment model can be learned on-line to produce realistic imaginary rollouts. To this end, a controller was trained

entirely on imaginary data and tested on the actual environment (which was still simulated, but not internally by the agent). The controller was found to perform relatively well, with about 80% of the performance of a controller trained on real data. Furthermore, visual inspection of the imaginary rollouts showed extremely realistic stochastic rollouts.

These results suggest that the architecture can indeed endow agents with an imagination mechanism. Given an initial state, the agent could imagine entire scenarios into the future, and simulate outcomes for different actions. The agent does not imagine images, but low-dimensional abstract state vectors that capture the essence of a scene.

The second research question was about how to efficiently estimate the uncertainty for neural networks with multi-modal predictive distributions. In the context of the architecture presented in Chapter 3, the importance of this question lies in the need to estimate the uncertainty of the environment model in order to prevent the controller from being trained on erroneous imaginary data. Predictions which have high epistemic uncertainty are likely to be erroneous, and therefore need to be discarded so as not to contaminate memory. Chapter 4 tackled this question, showing mathematically how to estimate the uncertainty of random function approximators. The mean squared error of predictions was decomposed in a Bayesian context into three components: epistemic, aleatoric, and modal uncertainties. Formulas were derived for the concrete case of BMDNs that can calculate estimates of the three kinds of uncertainties from samples drawn from the distribution of weights.

Chapter 4 also presented experiments to verify the findings. The first experiment involved a synthetic dataset with multimodal targets. An MDN was trained on the dataset, and MC-dropout was used to approximate the process of drawing samples from the weight distribution. Results showed that reasonably accurate uncertainty estimates can be obtained using this technique. In particular, predictions in low-data regimes in the dataset was associated with high epistemic uncertainty. The second experiment was concerned with the use of epistemic uncertainty estimates to guide decision making. The experiment was a simulation in which a robotic arm had to learn its inverse kinematics function from data collected through motor babbling. The robot was then presented with multiple points in its workspace, and it had to decide which point to try to reach so as to minimize the error between the final position of its end-effector and the point. In multiple test runs, results showed that choosing points with the lowest epistemic uncertainty led to significantly lower error than choosing randomly.

The results suggest that this method of uncertainty estimation is effective in obtaining decomposed uncertainty estimates for MDNs. Furthermore, they show that epistemic uncertainty estimates in particular can be used as a heuristic to guide decision making, as they reflect the quality of the model's predictions.

The third research question was about how to leverage imagination to improve the sample efficiency of model-free RL. In chapter 5, an architecture was developed to incorporate imagination into model-free RL. The overall architecture used the same components from Chapter 3, but arranged differently to allow the controller to be trained simultaneously on both real and imaginary data. Data collected by the agent through interacting with the environment was saved in the real memory after being encoded, and used to simultaneously train the controller and the environment model. After a certain amount of episodes, the model was used to generate imaginary rollouts. the epistemic uncertainty of the transitions in the imaginary rollouts was estimated, and those above a certain threshold were discarded, while the rest were saved in the imaginary memory. The controller would then get trained simultaneously on real and imaginary data.

Chapter 5 also presented an experiment to evaluate the architecture and its performance compared to baseline DQN. The experiment involved a task in which an agent had to solve a puzzle based on pointing gestures made by a human. The controller was trained simultaneously on both real and imaginary data, after uncertainty estimation was applied on the latter using MC-dropout. Results show that augmenting the agent with imagination led to significantly better performance when the data is scarce. The advantage vanished however when enough data has been collected from the real environment. Furthermore, discarding imaginary data with high epistemic uncertainty was found to improve the performance, especially during the early phase of the training when the environment model has not yet collected enough data.

These findings suggest that the imagination-augmented learning architecture developed in Chapter 5 can indeed drastically improve the sample efficiency of RL. Moreover, the experiment demonstrated one way uncertainty estimation can be used to further improve performance.

Future Work

There are many ways the work presented so far can be extended to address the limitations of the architectures presented in chapters 3 and 5. Such extensions have the

potential to make the approach more sample-efficient, more adaptable, or applicable to more classes of tasks. In the following, some of the future research directions will be discussed.

Non-Markovian Environments

The environments used in the experiments were assumed to be Markovian[1], meaning that the state of the environment is entirely contained within the image. This is not generally the case, as many tasks require memory of past observations[2]. In such cases, the environment model or the encoder can include some sort of recurrence mechanism so that information contained in the sequence of images are extracted. One way to do this is to combine the MDN of the environment model with an LSTM, which is what Ha et al. did in [HS18]. Alternatively, the recurrence mechanism can be implemented in the encoder instead of the environment model, such as in variational recurrent autoencoders [FvAK15], which can encode sequences into a latent vector representation.

Artificial Curiosity

One way to improve the sample efficiency in RL is to collect higher quality samples that contain more information about the task or the environment. Instead of having the agent explore its environment randomly, a mechanism for artificial curiosity can drive it. Under such mechanism, an intrinsic reward signal would motivate the agent to actively seek novel experiences that rapidly improves its model or policy. For example, Pathak et al. [PAED17] used a formulation for artificial curiosity as the error in the agent's predictions of its environment state. Perhaps a better formulation for curiosity would be one based on the epistemic uncertainty in the agent's model of the environment, as this does not require actually taking certain actions to measure the error. We have already seen in Chapter 4 how the epistemic uncertainty can be used as a heuristic for the prediction error. Bechtle et al. [BLR+20] used a similar concept in an model-based RL context in which agents explored areas with high uncertainty in its environment model. However, they made no distinction between the different kinds of

[1] A system is said to be Markovian if the current state depends solely on the previous state, independently from any earlier states.

[2] The Markovian property of a system actually depends on the state representation. For example, a mass-spring system is non-Markovian if the state representation is taken to be the vector of positions, but is Markovian if the velocities are also included.

uncertainty. It seems reasonable that a similar approach based on the epistemic uncertainty specifically would lead to better performance, especially in noisy or stochastic environments.

Programming by Demonstration and Teacher Feedback

To improve sample-efficiency even further, programming by demonstration techniques [BCDS08] can be used as a way to bootstrap learning. Instead of starting from scratch, an agent can start learning an environment model and a policy from human demonstrations, allowing it to require less samples to collect itself. While the agent is interacting with the environment, a human teacher can also provide feedback to allow the agent to learn a reward model faster.

Bibliography

[AWR+17] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.

[BCDS08] Aude Billard, Sylvain Calinon, Ruediger Dillmann, and Stefan Schaal. Robot programming by demonstration. *Springer handbook of robotics*, pages 1371–1394, 2008.

[BCKW15] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*, 2015.

[BGV92] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152, 1992.

[BHT+18] Jacob Buckman, Danijar Hafner, George Tucker, Eugene Brevdo, and Honglak Lee. Sample-efficient reinforcement learning with stochastic ensemble value expansion. In *Advances in Neural Information Processing Systems*, pages 8234–8244, 2018.

[Bis94] Christopher M Bishop. Mixture density networks. 1994.

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[BLR+20] Sarah Bechtle, Yixin Lin, Akshara Rai, Ludovic Righetti, and Franziska Meier. Curious ilqr: Resolving uncertainty in model-based

rl. In Leslie Pack Kaelbling, Danica Kragic, and Komei Sugiura, editors, *Proceedings of the Conference on Robot Learning*, volume 100 of *Proceedings of Machine Learning Research*, pages 162–171. PMLR, 30 Oct–01 Nov 2020.

[Buc05] Bruce G Buchanan. A (very) brief history of artificial intelligence. *Ai Magazine*, 26(4):53–53, 2005.

[CC96] Mary Kathryn Cowles and Bradley P Carlin. Markov chain monte carlo convergence diagnostics: a comparative review. *Journal of the American Statistical Association*, 91(434):883–904, 1996.

[CCML18] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. *arXiv preprint arXiv:1805.12114*, 2018.

[CPTW16] Francisco Cruz, German I Parisi, Johannes Twiefel, and Stefan Wermter. Multi-modal integration of dynamic audiovisual patterns for an interactive reinforcement learning scenario. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 759–766. IEEE, 2016.

[Cre93] Daniel Crevier. *AI: the tumultuous history of the search for artificial intelligence*. Basic Books, Inc., 1993.

[CSM+16] Paul Christiano, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba. Transfer from simulation to real world through learning deep inverse dynamics model. *arXiv preprint arXiv:1610.03518*, 2016.

[CTB18] Madison Clark-Turner and Momotaz Begum. Deep reinforcement learning of abstract reasoning from demonstrations. In *Proceedings of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*, pages 160–168, 2018.

[Cua20] Heriberto Cuayáhuitl. A data-efficient deep learning approach for deployable multimodal social robots. *Neurocomputing*, 396:587–598, 2020.

[DHLDVU17] Stefan Depeweg, José Miguel Hernández-Lobato, Finale Doshi-Velez, and Steffen Udluft. Decomposition of uncertainty in bayesian deep learning for efficient and risk-sensitive learning. *arXiv preprint arXiv:1710.07283*, 2017.

[Dun12] David Dunning. *Self-insight: Roadblocks and detours on the path to knowing thyself*. Psychology Press, 2012.

[FvAK15] Otto Fabius, Joost R van Amersfoort, and Diederik P Kingma. Variational recurrent auto-encoders. In *ICLR (Workshop)*, 2015.

[FWS+18] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I Jordan, Joseph E Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning. *arXiv preprint arXiv:1803.00101*, 2018.

[GBCB16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[GBM+16] Ali Ghadirzadeh, Judith Bütepage, Atsuto Maki, Danica Kragic, and Mårten Björkman. A sensorimotor reinforcement learning framework for physical human-robot interaction. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2682–2688. IEEE, 2016.

[GCY+20] Ali Ghadirzadeh, Xi Chen, Wenjie Yin, Zhengrong Yi, Marten Bjorkman, and Danica Kragic. Human-centered collaborative robots with deep reinforcement learning. *IEEE Robotics and Automation Letters*, 2020.

[GF15] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[GG16] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[GHLL17] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3389–3396. IEEE, 2017.

[GLSL16] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838. PMLR, 2016.

[Gra11] Alex Graves. Practical variational inference for neural networks. In *Advances in neural information processing systems*, pages 2348–2356. Citeseer, 2011.

[GYF+19] Yuan Gao, Fangkai Yang, Martin Frisk, Daniel Hemandez, Christopher Peters, and Ginevra Castellano. Learning socially appropriate robot approaching behavior toward groups using deep reinforcement learning. In *2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, pages 1–8. IEEE, 2019.

[HCD+16] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1109–1117, 2016.

[HESY19] Nusrah Hussain, Engin Erzin, T Metin Sezgin, and Yucel Yemez. Speech driven backchannel generation using deep q-network for enhancing engagement in human-robot interaction. *arXiv preprint arXiv:1908.01618*, 2019.

[HLA15] Jose Miguel Hernandez-Lobato and Ryan Adams. Probabilistic backpropagation for scalable learning of bayesian neural networks. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1861–1869, Lille, France, 07–09 Jul 2015. PMLR.

[HMP+17] Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier

Glorot, Matthew Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representations*, 2017.

[Hop82] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[HS18] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*, pages 2455–2467, 2018.

[HSW89] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[KB17] Gabriel Kalweit and Joschka Boedecker. Uncertainty-driven imagination for continuous deep reinforcement learning. In *Conference on Robot Learning*, pages 195–206, 2017.

[KBP13] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[KCD+18] Thanard Kurutach, Ignasi Clavera, Yan Duan, Aviv Tamar, and Pieter Abbeel. Model-ensemble trust-region policy optimization. *arXiv preprint arXiv:1802.10592*, 2018.

[KG17] Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? In *Advances in neural information processing systems*, pages 5574–5584, 2017.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[KVP+17] Gregory Kahn, Adam Villaflor, Vitchyr Pong, Pieter Abbeel, and Sergey Levine. Uncertainty-aware reinforcement learning for collision avoidance. *arXiv preprint arXiv:1702.01182*, 2017.

[KW14] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. In *International Conference on Learning Representations*, 2014.

[LA14] Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *NIPS*, volume 27, pages 1071–1079. Citeseer, 2014.

[LAA+17] Christian Leibig, Vaneeda Allken, Murat Seçkin Ayhan, Philipp Berens, and Siegfried Wahl. Leveraging uncertainty information from deep neural networks for disease detection. *Scientific reports*, 7(1):1–14, 2017.

[Lan11] Pat Langley. The changing science of machine learning, 2011.

[LFDA16] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[LHP+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[LLSA20] Kimin Lee, Michael Laskin, Aravind Srinivas, and Pieter Abbeel. Sunrise: A simple unified framework for ensemble learning in deep reinforcement learning. *arXiv preprint arXiv:2007.04938*, 2020.

[LMMH19] Stéphane Lathuilière, Benoît Massé, Pablo Mesejo, and Radu Horaud. Neural network based reinforcement learning for audio–visual gaze control in human–robot interaction. *Pattern Recognition Letters*, 118:61–71, 2019.

[LPB17] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in neural information processing systems*, pages 6402–6413, 2017.

[LR14] Thomas Lampe and Martin Riedmiller. Approximate model-assisted neural fitted q-iteration. In *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 2698–2704. IEEE, 2014.

[LSKD13] Kyuhwa Lee, Yanyu Su, Tae-Kyun Kim, and Yiannis Demiris. A syntactic approach to robot imitation learning using probabilistic activity grammars. *Robotics and Autonomous Systems*, 61(12):1323–1334, 2013.

[MBM+16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[MKS+13] 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.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[Mor88] Hans Moravec. *Mind children: The future of robot and human intelligence*. Harvard University Press, 1988.

[MP69] M.L. Minsky and S. Papert. *Perceptrons; an Introduction to Computational Geometry*. MIT Press, 1969.

[MSV+08] Giorgio Metta, Giulio Sandini, David Vernon, Lorenzo Natale, and Francesco Nori. The icub humanoid robot: an open platform for research in embodied cognition. In *Proceedings of the 8th workshop on performance metrics for intelligent systems*, pages 50–56, 2008.

[Ola96] Mikel Olazaran. A sociological study of the official history of the perceptrons controversy. *Social Studies of Science*, 26(3):611–659, 1996.

[Osb16] Ian Osband. Risk versus uncertainty in deep learning: Bayes, bootstrap and the dangers of dropout. In *NIPS Workshop on Bayesian Deep Learning*, volume 192, 2016.

[PAED17] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2778–2787. PMLR, 06–11 Aug 2017.

[PHL+17] Ivaylo Popov, Nicolas Heess, Timothy Lillicrap, Roland Hafner, Gabriel Barth-Maron, Matej Vecerik, Thomas Lampe, Yuval Tassa, Tom Erez, and Martin Riedmiller. Data-efficient deep reinforcement learning for dexterous manipulation. *arXiv preprint arXiv:1704.03073*, 2017.

[QNYI16] Ahmed Hussain Qureshi, Yutaka Nakamura, Yuichiro Yoshikawa, and Hiroshi Ishiguro. Robot gains social intelligence through multimodal deep reinforcement learning. In *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)*, pages 745–751. IEEE, 2016.

[QNYI18] Ahmed Hussain Qureshi, Yutaka Nakamura, Yuichiro Yoshikawa, and Hiroshi Ishiguro. Intrinsically motivated reinforcement learning for human–robot interaction in the real-world. *Neural Networks*, 107:23–33, 2018.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[RVR+17] Andrei A Rusu, Matej Večerík, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. In *Conference on Robot Learning*, pages 262–270. PMLR, 2017.

[RWR+17] Sébastien Racanière, Théophane Weber, David Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. In *Advances in neural information processing systems*, pages 5690–5701, 2017.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SGS11] Yi Sun, Faustino Gomez, and Jürgen Schmidhuber. Planning to be surprised: Optimal bayesian exploration in dynamic environments. In *International Conference on Artificial General Intelligence*, pages 41–51. Springer, 2011.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[SLH+14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014.

[SMS+99] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with

function approximation. In *NIPs*, volume 99, pages 1057–1063. Citeseer, 1999.

[Sut90] Richard S Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine learning proceedings 1990*, pages 216–224. Elsevier, 1990.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[TDH+15] Eric Tzeng, Coline Devin, Judy Hoffman, Chelsea Finn, Xingchao Peng, Sergey Levine, Kate Saenko, and Trevor Darrell. Towards adapting deep visuomotor representations from simulated to real environments. *arXiv preprint arXiv:1511.07111*, 2(3), 2015.

[TZL+16] Lei Tai, Jingwei Zhang, Ming Liu, Joschka Boedecker, and Wolfram Burgard. A survey of deep network solutions for learning control in robotics: From reinforcement to imitation. *arXiv preprint arXiv:1612.07139*, 2016.

[VBC+19] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[VHS+17] 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 preprint arXiv:1707.08817*, 2017.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Appendix A

Decomposition of MSE for Multimodal Predictive Density Estimation

Let $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i) : i = \{1, \ldots, n\}\}$ be a dataset of n samples drawn form the distribution $p(\mathbf{x}, \mathbf{y})$ where $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{y} \in \mathbb{R}^k$. Assume $\mathcal{D}$ is generated by the process:

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) + \eta, \tag{A.1}$$

where $\mathbf{f}(\cdot) : \mathbb{R}^d \to \mathbb{R}^k$ is some random function and $\eta \sim \mathcal{N}(\mathbf{0}, \sigma_\eta^2(\mathbf{x})\mathbf{I})$ is the measurement noise with a diagonal $k \times k$ covariance matrix whose diagonal entries form the vector $\sigma_\eta^2(\mathbf{x}) = (\sigma_1^2(\mathbf{x}), \ldots, \sigma_k^2(\mathbf{x}))$. Let the true conditional density of the target data be $p(\mathbf{y}|\mathbf{x})$, and let its approximation be $\hat{p}(\mathbf{y}|\mathbf{x})$ which we use to draw predictions $\hat{\mathbf{y}}(\mathbf{x})$. Applying the well-known bias-variance decomposition to the expected squared error between $\hat{\mathbf{y}}$ and $\mathbf{y}$ for a certain value of $\mathbf{x}$ yields [Bis06]:

$$\begin{aligned}
&\mathbb{E}_{p,q,\hat{p}}[(\mathbf{y} - \hat{\mathbf{y}})^2] \\
&= \mathbb{E}_{p,q,\hat{p}}[(\mathbf{y} - \mathbf{f} + \mathbf{f} - \hat{\mathbf{y}})^2] \\
&= \mathbb{E}_{p,q,\hat{p}}[(\mathbf{y} - \mathbf{f})^2 + (\mathbf{f} - \hat{\mathbf{y}})^2 + 2(\mathbf{y} - \mathbf{f})(\mathbf{f} - \hat{\mathbf{y}})] \\
&= \mathbb{E}_p[(\mathbf{y} - \mathbf{f})^2] + \mathbb{E}_{p,q,\hat{p}}[(\mathbf{f} - \hat{\mathbf{y}})^2] \\
&\quad + \mathbb{E}_{p,q,\hat{p}}[(\mathbf{y} - \mathbf{f})(\mathbf{f} - \hat{\mathbf{y}})] \\
&= \mathbb{E}[\eta^2] + \mathbb{E}_{p,q,\hat{p}}[(\mathbf{f} - \hat{\mathbf{y}})^2] \\
&\quad + 2\mathbb{E}[\eta]\mathbb{E}_{p,q,\hat{p}}[(\mathbf{f} - \hat{\mathbf{y}})] \\
&= \sigma_\eta^2 + \mathbb{E}_{p,q,\hat{p}}[(\mathbf{f} - \hat{\mathbf{y}})^2]
\end{aligned} \tag{A.2}$$

The second term in Equation A.2 can be written as:

$$\begin{aligned}
&\mathbb{E}_{p,q,\hat{p}}[(\mathbf{f}-\hat{\mathbf{y}})^2] \\
&= \mathbb{E}_{p,q,\hat{p}}\left[(\mathbf{f}-\mathbb{E}_p[\mathbf{f}]+\mathbb{E}_p[\mathbf{f}]-\hat{\mathbf{y}})^2\right] \\
&= \mathbb{E}_p\left[(\mathbf{f}-\mathbb{E}_p[\mathbf{f}])^2\right]+\mathbb{E}_{q,\hat{p}}\left[(\mathbb{E}_p[\mathbf{f}]-\hat{\mathbf{y}})^2\right] \\
&\quad +2\mathbb{E}_{p,q,\hat{p}}[(\mathbf{f}-\mathbb{E}_p[\mathbf{f}])(\mathbb{E}_p[\mathbf{f}]-\hat{\mathbf{y}})] \qquad \text{(A.3)}
\end{aligned}$$

Rewriting the second term in Equation A.3 in a similar fashion yields:

$$\begin{aligned}
&\mathbb{E}_{q,\hat{p}}\left[(\mathbb{E}_p[\mathbf{f}]-\hat{\mathbf{y}})^2\right] \\
&= \mathbb{E}_{q,\hat{p}}\left[(\mathbb{E}_p[\mathbf{f}]-\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]+\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]-\hat{\mathbf{y}})^2\right] \\
&= \mathbb{E}_q\left[(\mathbb{E}_p[\mathbf{f}]-\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}])^2\right]+\mathbb{E}_{q,\hat{p}}\left[(\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]-\hat{\mathbf{y}})^2\right] \\
&\quad +2\mathbb{E}_{p,q,\hat{p}}[(\mathbb{E}[\mathbf{f}]-\mathbb{E}[\hat{\mathbf{y}}])(\mathbb{E}[\hat{\mathbf{y}}]-\hat{\mathbf{y}})] \\
&= \mathbb{E}_q\left[(\mathbb{E}_p[\mathbf{f}]-\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}])^2\right]+\mathbb{E}_{q,\hat{p}}\left[(\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]-\hat{\mathbf{y}})^2\right] \\
&\quad +2(\mathbb{E}_p[\mathbf{f}]\mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}]-\mathbb{E}_p[\mathbf{f}]\mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}] \\
&\quad -\mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}]^2+\mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}]^2) \\
&= \mathbb{E}_q\left[(\mathbb{E}_p[\mathbf{f}]-\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}])^2\right]+\mathbb{E}_{q,\hat{p}}\left[(\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]-\hat{\mathbf{y}})^2\right] \qquad \text{(A.4)}
\end{aligned}$$

Expanding the last term in Equation A.3 yields:

$$\begin{aligned}
&2\mathbb{E}_{p,q,\hat{p}}\left[(\mathbf{f}-\mathbb{E}_p[\mathbf{f}])(\mathbb{E}_p[\mathbf{f}]-\hat{\mathbf{y}})\right] \\
&= 2\mathbb{E}_{p,q,\hat{p}}\left[\mathbf{f}\mathbb{E}_p[\mathbf{f}]-\mathbf{f}\hat{\mathbf{y}}-\mathbb{E}_p[\mathbf{f}]^2-\hat{\mathbf{y}}\mathbb{E}_p[\mathbf{f}])\right] \\
&= 2(\mathbb{E}_p[\mathbf{f}]^2-\mathbb{E}_p[\mathbf{f}]\mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}] \\
&\quad -\mathbb{E}_p[\mathbf{f}]^2+\mathbb{E}_p[\mathbf{f}]\mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}]) \\
&= 0, \qquad \text{(A.5)}
\end{aligned}$$

where we have used the fact that $\hat{y}$ and f are independent given x. Using Equations A.3, A.4 and A.5 in A.2, we have:

$$\begin{aligned}
&\mathbb{E}_{p,q,\hat{p}}\left[(\mathbf{y}-\hat{\mathbf{y}})^2\right] \\
&= \mathbb{E}_q\left[(\mathbb{E}_p[\mathbf{f}]-\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}])^2\right]+\mathbb{E}_p[(y-\mathbf{f})^2] \\
&\quad +\mathbb{E}_p\left[(\mathbf{f}-\mathbb{E}_p[\mathbf{f}])^2\right]+\mathbb{E}_{q,\hat{p}}\left[(\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]-\hat{\mathbf{y}})^2\right] \\
&= \mathbb{E}_q\left[(\mathbb{E}_p[\mathbf{f}]-\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}])^2\right] \\
&\quad +\sigma_\eta^2+\mathbb{V}_p[\mathbf{f}]+\mathbb{E}_q\left[\mathbb{V}_{\hat{p}}[\hat{\mathbf{y}}]\right]. \qquad \text{(A.6)}
\end{aligned}$$

The first term in Equation A.6 can be expanded as:

$$\begin{aligned}
\sigma_{\mathrm{e}}^2 &= \mathbb{E}_q\left[(\mathbb{E}_p[\mathbf{f}] - \mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}])^2\right] \\
&= \mathbb{E}_p[\mathbf{f}]^2 + \mathbb{E}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]^2\right] \\
&\quad - 2\mathbb{E}_p[\mathbf{f}]\mathbb{E}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]\right] \\
&= (\mathbb{E}_p[\mathbf{f}] - \mathbb{E}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]\right])^2 + \mathbb{E}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]^2\right] \\
&\quad - \mathbb{E}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]\right]^2 \\
&= (\mathbb{E}_p[\mathbf{f}] - \mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}])^2 + \mathbb{V}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]\right], \qquad \text{(A.7)}
\end{aligned}$$

which finally yields:

$$\begin{aligned}
&\mathbb{E}_{p,q,\hat{p}}\left[(\mathbf{y} - \hat{\mathbf{y}})^2\right] \\
&= \mathbb{E}_q\left[(\mathbb{E}_p[\mathbf{f}] - \mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}])^2\right] \\
&\quad + \sigma_{\eta}^2 + \mathbb{V}_p[\mathbf{f}] + \mathbb{E}_q\left[\mathbb{V}_{\hat{p}}[\hat{\mathbf{y}}]\right] \\
&= (\mathbb{E}_p[\mathbf{f}] - \mathbb{E}_{q,\hat{p}}[\hat{\mathbf{y}}])^2 + \mathbb{V}_q\left[\mathbb{E}_{\hat{p}}[\hat{\mathbf{y}}]\right] \\
&\quad + \sigma_{\eta}^2 + \mathbb{V}_p[\mathbf{f}] + \mathbb{E}_q\left[\mathbb{V}_{\hat{p}}[\hat{\mathbf{y}}]\right] \qquad \text{(A.8)}
\end{aligned}$$