Active Reward Modelling for Agent Alignment

Sam Clarke

Hertford College

University of Oxford

Submitted in partial completion of the

MSc in Computer Science

Trinity Term 2019

Abstract

To solve a task using reinforcement learning, we require a reward function which correctly specifies the task's objective. However, such reward functions are hard to construct for complex, real-world tasks. An alternative is to learn a model of that reward function from human feedback, an approach called *reward modelling*. This dissertation considers feedback in the form of preferences about an agent's behaviour. Reward modelling should be efficient, to avoid putting too much burden on the human giving feedback. An approach to improve efficiency is active learning: acquiring preferences on the behaviour about which the reward model is the most uncertain. Previous work applied one instantiation of active learning to reward modelling, but the results were mixed [Christiano et al., 2017]. In this dissertation, we conduct a detailed study into whether active learning can be used to improve the efficiency of reward modelling. Specifically, we propose and test different hypotheses to explain the mixed results in previous work. We find that whether active learning improves on the random acquisition of preferences depends on two properties, which have strong dependencies on the environment and task. Accordingly, we suggest that future work in reward modelling should use environments and tasks which increasingly resemble its intended real-world applications.

Contents

I Background

1 Introduction **1**

2 Deep Neural Networks **4**

2.1 Model 4

2.2 Cost function 5

2.3 Optimization procedure 6

3 Reinforcement Learning **9**

3.1 Finite Markov Decision Processes 10

3.1.1 The Agent-Environment Interface 10

3.1.2 Goals and Rewards 11

3.1.3 Returns and Episodes 12

3.1.4 Policies and Value Functions 13

3.1.5 Optimal Policies and Optimal Value Functions 13

3.1.6 Bellman Equations 14

3.2 Reinforcement Learning Solution Methods 16

3.2.1 Deep Q-network 16

3.3 Reinforcement Learning from Unknown Reward Functions 18

3.3.1 Reward Learning from Trajectory Preferences in Deep RL . . 21

3.3.2 Reward Learning from Trajectory Preferences with Handcrafted Feature Transformations 25

4 Uncertainty in Deep Learning **27**

4.1 Bayesian Neural Networks 28

4.2 Active Learning 29

4.2.1 Max Entropy 30

4.2.2 Bayesian Active Learning by Disagreement 30

4.2.3 Variation Ratios 33

4.2.4 Mean STD 33

4.2.5 Approximating acquisition functions with Bayesian Neural Networks 34

II Innovation **36**

5 Method **37**

5.1 Applying acquisition functions to reward modelling 37

5.2 Active Reward Modelling 39

5.3 Possible failure modes of active reward modelling 40

5.3.1 Not retraining reward model from scratch 40

5.3.2 Acquisition size is too large 41

5.3.3 Choice of acquisition function 41

5.3.4 Choice of uncertainty estimate method 41

5.3.5 Uncertainty estimate quality 41

5.3.6 Learning the reward model is too easy 42

5.3.7 Too few trajectories generated by the agent are disproportionately informative 43

5.4 Implementation Details 43

6 Experiments and Results **45**

6.1 CartPole experiments 45

6.2 Hypothesis 1: Reward model retraining 47

6.3 Hypothesis 2: Acquisition size 48

6.4 Hypotheses 3 and 4: Uncertainty estimate method and acquisition functions 49
6.5 Hypothesis 5: Uncertainty estimate quality 51
6.6 Hypothesis 6: Learning the reward model is too easy 52
6.7 Gridworld experiments 54
6.8 Hypothesis 7: Too few trajectories generated by the agent are disproportionately informative 55

7 Conclusions **59**
7.1 Summary 59
7.2 Future Work 60
7.3 Relation to Material Studied on the MSc Course 61
7.4 Personal Development 61

References **63**

Part I

Background

Chapter 1

Introduction

Reinforcement Learning (RL) has made rapid progress in recent years at complex tasks such as playing Go [Silver et al., 2016] and StarCraft II [Vinyals et al., 2019]. In their current form, the success of these systems relies on the objectives of the task being well-defined. Specifically, they require that the objectives can be expressed as a *reward function*, which defines goals in terms of the reward associated with taking a given action when the world is in a given state. In the cases of Go and StarCraft, the game score function provides this.

However, real-world tasks lack such obvious reward functions. Therefore, if we aspire to use RL to assist us in the real world, we need some other method of specifying our intentions. Put another way, we need to solve the *agent alignment problem* [Leike et al., 2018, p. 1]:

> *How can we create agents that behave in accordance with the user's intentions?*

So far, systems have been developed which allow users to communicate their intentions by giving demonstrations and providing feedback (in the form of preferences or scalar rewards) on the agent's behaviour. For example, the *inverse reinforcement learning* approach [Ng and Russell, 2000, Ziebart et al., 2008] uses demonstrations by the user to infer their reward function. One can then use a standard RL algorithm to train an agent on the inferred reward function, to follow the user's intentions as

expressed in the demonstrations. Alternatively, the *RL from preferences* approach [Christiano et al., 2017] queries the user for preferences about current agent behaviour, which it uses to learn the user's reward function and then refine the agent's behaviour accordingly, again by standard RL techniques.

These systems have been prototyped in simple domains, such as Atari games [Bellemare et al., 2013] and simulated robotics tasks [Todorov et al., 2012]. However, the technology is far from mature. For example, the RL from preferences method requires on the order of thousands of user queries to learn to play simple Atari games, and so may struggle to scale to much more complex tasks. Indeed, an important property of any such system is to minimise the burden placed on the user. Clearly, there is no point in trying to automate some task if the automation procedure is even more demanding for the human than just performing the task themselves. In other words, we desire systems trained on user feedback to make sample efficient requests.

For this purpose, the RL from preferences method developed in [Christiano et al., 2017] attempts to request only the most informative preferences from the user, a technique known as active learning (AL). However, the results of applying this technique were mixed; on some tasks and environments, it showed improvement over requesting preferences uniformly at random, whilst for others, it made no significant difference, or even impaired performance. The authors conjectured that their crude AL method was at fault, but did not provide a detailed diagnosis or attempt to fix it. This component of the system was omitted the subsequent work by [Ibarz et al., 2018].

Our contribution is to conduct a detailed study into whether AL can be used to improve the efficiency of RL from user preferences, which we call *active reward modelling*. Specifically, we propose and test different hypotheses to explain the mixed results in [Christiano et al., 2017]. Consistent with their results, we find that whether AL improves on the random acquisition of preferences depends on two properties, which have strong dependencies on the environment and task. These properties are: the number of preferences required to learn a good reward model,

and the frequency with which the agent generates informative trajectories. This environment-dependency is also consistent with current state of RL in general: it is common for RL algorithmic innovations tested on certain combinations of tasks and environments to fail to replicate when the task and environment are changed [Henderson et al., 2018].

The dissertation is organised as follows. Part I lays out the relevant background material. All this material rests on an understanding of deep neural networks, which are explained in chapter 1. Chapter 2 outlines the basics of reinforcement learning. Chapter 3 concerns techniques for equipping deep neural networks with the ability to give estimates of the uncertainty in their output, which is used in active learning. These chapters provide only short overviews of entire research fields, but should be self-contained and give sufficient detail to allow the reader to assess our work. Part II is concerned with our contribution. Chapter 5 explains our hypotheses for lack of success of active reward modelling in the previous work [Christiano et al., 2017] and outlines the basic active reward modelling training protocol with which we test our hypotheses. Chapter 6 details the experiments we performed to test our hypotheses, and the conclusions we drew from them. Chapter 7 summarises our contribution and avenues for future work, and gives a discussion of personal development and the relation of this dissertation to the material studied on the MSc course.

Chapter 2

Deep Neural Networks

The goal of a deep neural network (NN) is to approximate some function $f^* : \mathbf{X} \mapsto \mathbf{Y}$ [Goodfellow et al., 2016]. For example, in image classification, $\mathbf{X}$ may be image pixels, and $\mathbf{Y}$ some image categories, for example, bird, plane or superhero. The neural network specifies a mapping $y = f(\mathbf{x}; \theta)$ that depends on some parameters θ. The task is then to learn the θ that give the best approximation to the true function f^*. A deep learning algorithm specifies how to do this.

As in machine learning in general, there are three key components to such an algorithm: model (or function approximator), cost function and optimization procedure.

2.1 Model

The model, of course, is a deep NN. This simplest form of deep NN is a *deep feedforward network*[1]. They are called feedforward because when the model is evaluated on input $\mathbf{x}$, computation flows without ever feeding back into itself in a loop. They are called networks because it is natural to think of these models as being composed of many different functions, each of which is termed a *layer*. Starting from the *input layer*, the output of each layer flows into the next, through each of the *hidden layers* until the *output layer* is reaches and the function returns some value.

[1] They are also referred to as feedforward neural networks or multilayer perceptrons (MLPs)

Deep feedforward networks are deep inasmuch as they have many hidden layers [Goodfellow et al., 2016, p. 164].

Example 1. *Consider a very simple feedforward NN* $f : \mathbb{R}^2 \mapsto \mathbb{R}$ *with one hidden layer.* f *is composed of three functions:* $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$*, where:*

$$f^{(1)}(\mathbf{x}) = \mathbf{z} = \mathbf{W^1 x} \quad \textit{for some } 2 \times 2 \textit{ matrix } \mathbf{W^1}$$
$$f^{(2)}(\mathbf{z}) = \mathbf{a} = \text{ReLu}(\mathbf{a}) := \max\{0, \mathbf{a}\} \quad \textit{where } \max(.) \textit{ is applied pointwise}$$
$$f^{(3)}(\mathbf{a}) = \mathbf{W^2 a} \quad \textit{for some } 1 \times 2 \textit{ matrix } \mathbf{W^2}$$

2.2 Cost function

A *cost function* (or *loss function*) is some function that quantifies the performance of our model i.e. how close it is to the true function f^* we are trying to approximate. Most modern NNs use the *negative log-likelihood* cost function[2] [Goodfellow et al., 2016, p. 138], which is simply the negative logarithm of the *likelihood function*. Given a parametric family of probability distributions $p_{\text{model}}(\mathbf{x}; \theta)$ over the same space, a likelihood function gives the probability of a set of observations for different settings of the model parameters θ. More formally, consider a set of m i.i.d. examples $\mathbb{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ drawn from a true but unknown data generating distribution. Then the likelihood function $L(\theta)$ is defined by:

$$\begin{aligned} L(\theta) &:= p_{\text{model}}(\mathbb{X}; \theta) \\ &= \Pi_{i=1}^{m} p_{\text{model}}(\mathbf{x}_i; \theta) \end{aligned}$$

Taking the logarithm of this function improves numerical stability, and taking its negative value is a convention because optimization in machine learning typically means *minimizing* some function. Finally, we can divide by m, which shows how we can interpret this function cost function as an expectation with respect to the empirical distribution $\hat{p}_{\text{data}}$ defined by the training data $\mathbb{X}$. Notice that minimising

[2] This is also referred to as the *cross-entropy loss* function.

this modified function will give the same result and maximising the likelihood. So, we define the negative log likelihood cost function $\text{NLL}(\theta)$ as

$$\begin{aligned}\text{NLL}(\theta) :=& -\frac{1}{m}\log L(\theta) \\ =& -\frac{1}{m}\sum_{i=1}^{m}\log p_{\text{model}}(\mathbf{x}_i;\theta) \\ =& -\mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}}\left[\log p_{\text{model}}(\mathbf{x};\theta)\right] \end{aligned} \tag{2.1}$$

and seek the parameters θ_{ML} which minimise this function:

$$\theta_{\text{ML}} :=\arg\min_{\theta}\text{NLL}(\theta)$$

In this thesis, we will use $\text{NLL}(\theta)$ cost function. Our model p_{model} is some neural network. Importantly, the NN output must define a probability distribution, which is implemented by making its final layer a softmax or Gaussian probability density function, for categorical and real-valued data, respectively. Furthermore, we actually use a slight generalisation of this procedure to estimate a conditional probability $p_{\text{model}}(\mathbf{y} \mid \mathbf{x};\theta)$ where $\mathbf{x} \in \mathbf{X}$ are some inputs and $\mathbf{y} \in \mathbf{Y}$ some outputs (also called *targets*) of the function $f^* : \mathbf{X} \mapsto \mathbf{Y}$ we are trying to approximate. This setting is called *supervised learning*, because the training data is a set of supervised examples i.e. pairs of inputs and correctly labelled outputs. Finally, note that in the real-valued case, using a Gaussian density function on the output of a NN is equivalent to not passing the output values through the density function, and instead simply minimising $\frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)$, the mean squared error between the inputs and targets over the training data $\langle(\mathbf{x}_i, y_i)\rangle_{i=1}^{m}$ where $\hat{y}_i = f(\mathbf{x}_i)$, our model's prediction for input $\mathbf{x}_i$ [Goodfellow et al., 2016, p. 132].

2.3 Optimization procedure

The third component of a deep NN algorithm is the *optimization procedure*. This specifies how we use the cost function to update the NN parameters θ. Since NNs

of interest are typically non-convex functions (due to non-linear layers such $f^{(2)}$ in Example 1), Equation 2.1 cannot be optimized in closed form. Thus, we require some numerical optimization procedure [Goodfellow et al., 2016, p. 151]; the most common is some variation of *gradient descent*.

Gradient descent works by computing $\nabla_\theta L(\theta)$, the partial derivate of some cost function $L(\theta)$ with respect to the model parameters θ. Since the partial derivate of a function points in the direction of steepest ascent, selecting a new set of parameters $\theta' = \theta - \epsilon \nabla_\theta L(\theta)$ will mean that $L(\theta')$ is less than $L(\theta)$ for some small enough ϵ [Goodfellow et al., 2016, p. 83]. We can iteratively perform this procedure and eventually reach a *local minimum* i.e. a point where $L(\theta)$ is lower than at all neighbouring points. ϵ is called the *learning rate*; it is typically chosen by trying several small values and choosing that which results in the lowest final $L(\theta)$.

If $L(\theta)$ is non-convex then there is no guarantee that this point will be a *global minimum* i.e. a point where $L(\theta)$ takes its lowest possible value. Much machine learning research is concerned with modifications to this procedure, and how to set initial parameter values in order to avoid getting stuck in local minimum that lead to poor performance. As it turns out, finding local minima that are low enough often leads to very good performance.

One important modification is called *stochastic gradient descent* (SGD). Notice that minimising Equation 2.1 via gradient descent requires computing

$$\nabla_\theta \text{NLL}(\theta) = \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} -\log p_{\text{model}}(\mathbf{x}_i; \theta)$$

on each iteration, which has computational complexity $O(m)$ [Goodfellow et al., 2016, p. 149]. For large training sets, this is infeasible.

SGD is based on the simple idea that this gradient is an expectation over the training data which we can approximate using a small sample [Goodfellow et al., 2016, p. 149]:

$$\nabla_\theta \text{NLL}(\theta) \approx \frac{1}{m'} \nabla_\theta \sum_{i=1}^{m'} -\log p_{\text{model}}(\mathbf{x}_i; \theta) \tag{2.2}$$

for some $\mathbb{B} = \{\mathbf{x}_1, \ldots, \mathbf{x}_{m'}\} \subset \mathbb{X}$ called a *minibatch*, sampled uniformly at random from $\mathbb{X}$.

In this thesis, we use a further variant of SGD called *RMSProp* [Tieleman and Hinton, 2012]. This maintains individual learning rates for each model parameter and adapts them individually. Specifically, on each learning update, parameter θ_i is rescaled according to the inverse square root of an exponentially weighted moving average of the historical squared values of the partial derivate of the cost function with respect to θ_i. This results in larger learning updates for parameters with small partial derivatives, and smaller learning updates for parameters with larger partial derivates. The intention is to converge to a minimum more rapidly than SGD. Algorithm 1 gives the precise procedure [Goodfellow et al., 2016, p. 304]. Note that the operations on

Algorithm 1 RMSProp.

Require: Global learning rate ϵ, decay rate ρ, initial parameters θ
1: Initialise $\mathbf{r} = 0$ to accumulate squared gradients for each parameter
2: **repeat**
3: Sample minibatch $\mathbb{B} = \{\mathbf{x}_1, \ldots, \mathbf{x}_{m'}\}$
4: Compute gradient $\mathbf{g} \leftarrow \frac{1}{m} \nabla_\theta \sum_i \text{NLL}(\mathbf{x}_i; \theta)$
5: Accumulate squared gradient $\mathbf{r} \leftarrow \rho\mathbf{r} + (1 - \rho)g \odot \mathbf{g}$
6: Update parameters: $\theta \leftarrow \theta - \frac{\epsilon}{\sqrt{r+10^{-6}}} \odot \mathbf{g}$
7: **until** convergence

lines 4-6 are all applied element-wise (that is, to each parameter individually). The adding 10^{-6} to r on line 6 improves numerical stability.

This concludes our overview of deep neural networks. We now cover the basics of Reinforcement Learning.

Chapter 3

Reinforcement Learning

Reinforcement learning (RL) refers simultaneously to a problem, methods for solving that problem, and the field that studies the problem and its solution methods. The problem of RL is to learn what to do—how to map situations to actions—so as to maximise some numerical reward signal [Sutton and Barto, 2018, pp. 1-2]. More specifically, the RL setting consists of an *agent* that interacts with an *environment* by taking sequential *actions*. Each action taken affects the environment, which responds by changing *state*. The agent then receives information about this new state, and a corresponding numerical *reward*. The agent's goal is to learn a *policy* that maximises this reward over time. A policy is a specification of what action to take in each state.

Our aim in this chapter is to introduce the necessary material for the reader to understand the *RL from preferences* algorithm. To this end, we first show that the RL Problem can be formalised as the optimal control of Finite Markov decision processes. We then explain the idea of an RL solution method, and describe one such method, Deep Q-Learning, that is called as a subroutine in the *RL from preferences* algorithm. Next, we outline some approaches to adapting RL to settings without reward functions. Finally, we explain in detail one such approach, *RL from preferences*, which is the reward modelling algorithm that we use in this dissertation.

3.1 Finite Markov Decision Processes

Finite Markov Decision Processes (finite MDPs) are a way of mathematically formalising the RL problem: they capture the most important aspects of the problem faced by an agent interacting with its environment to achieve a goal. We introduce the elements of this formalism: the agent-environment interface, goals and rewards, returns and episodes. We then explain three further important concepts in light of this formalism: polices, value functions and the Bellman equations.

3.1.1 The Agent-Environment Interface

MDPs consist firstly of the continual interaction between an agent selecting actions, and an environment responding by changing state, and presenting the new state to the agent, along with an associated scalar reward. Recall that the agent seeks to maximise this reward over time through its choice of actions.

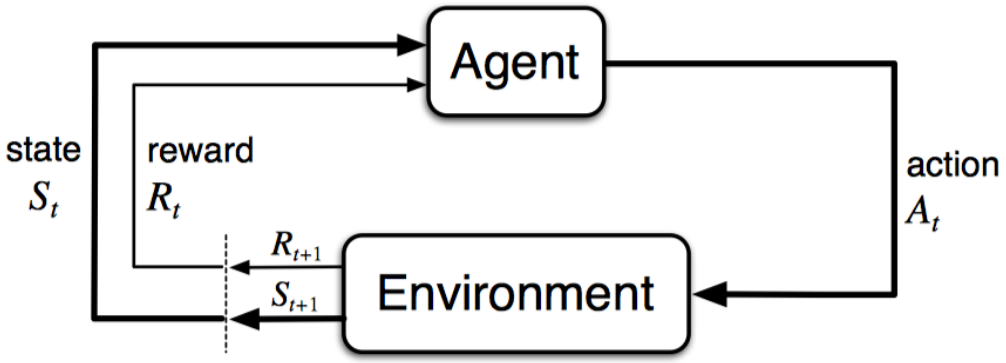


Figure 3.1: A schematic of the agent-environment interface.

More formally, consider a sequence of discrete time steps, $t = 1, 2, 3, \ldots$. At each time step t, the agent receives some representation of the environment's *state*, $S_t \in \mathcal{S}$, and chooses an *action*, $A_t \in \mathcal{A}$. On the next time step, the agent receives reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, and finds itself in a new state, S_{t+1}. A schematic of this interaction is show in 3.1. These interactions repeat over time, giving rise to a

trajectory, τ:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

A *finite* MDP is one where the sets of states, actions and rewards are finite. In this case, the random variables S_t and R_t have well-defined discrete probability distributions which depend only on the preceding state and action. This allows us to define the *dynamics* of the MDP, a probability mass function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$, as follows. For any particular values $s' \in \mathcal{S}$ and $r \in \mathcal{R}$ of the random variables S_t and R_t, there is a probability of these values occurring at time t, given any values of the previous state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$:

$$p(s', r \mid s, a) := \mathbb{P}\left(S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\right).$$

A *Markov* Decision Process is one where all states satisfy the Markov property. A state s_t of an MDP satisfies this property iff:

$$\mathbb{P}\left(s_{t+1}, r_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, \ldots, s_0, a_0\right) = \mathbb{P}\left(s_{t+1} \mid s_t, a_t\right).$$

This implies that the immediately preceding state s_t and action a_t are sufficient statistics for predicting the next state s_{t+1} and reward r_{t+1}.

3.1.2 Goals and Rewards

The reader may have noticed that we first introduced MDPs as a formalism for an agent interacting with its environment to achieve a goal, yet have since spoken instead of maximising a reward signal $R_t \in \mathbb{R}$ over time. Our implicit assumption is the following hypothesis:

Hypothesis 1 (Reward Hypothesis)**.** *All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward). [Sutton and Barto, 2018, p. 53]*

However, this hypothesis gives no information about how to construct such a scalar signal; only that it exists. Indeed, recent work has shown that it is far from trivial to do so; possible failure modes include negative side effects, reward hacking and unsafe exploration [Amodei et al., 2016]. This is central to the topic of this dissertation—our aim is to improve the sample efficiency of one particular method of reinforcement learning when the reward signal is unknown.

3.1.3 Returns and Episodes

Having asserted that we can express the objective of reinforcement learning in terms of scalar reward, we now formally define this objective. Consider the following objective:

Definition 1 ((Future discounted) return)**.** *Let a sequence of rewards between time step $t+1$ and T (inclusive) be $R_{t+1}, R_{t+1}, \ldots, R_T$. Let $\gamma \in [0, 1]$ be a discount factor of future rewards. Then we define the (future discounted) return of this sequence of rewards [Sutton and Barto, 2018, p. 57]:*

$$G_t := \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{3.1}$$

One reason for introducing a discount factor is because we would like this infinite sum to converge. Accordingly, we impose the condition that $\gamma < 1$ whenever the reinforcement learning task is *continuous*, that is to say, there may be an infinite number of non-zero terms in the sequence of rewards $\{R_{t+1}, R_{t+2}, R_{t+3}, \ldots\}$.

The other kind of task is called *episodic*. Here, interactions between the agent and environment occur in well-defined subsequences, each of which ends in a special *terminal state*. The environment then resets to a starting state, which may be fixed or sampled from a distribution. To adapt the definition in (3.1) to this case, we introduce the convention that zero reward is given after reaching the terminal state. This is because we typically analyse such tasks by considering a single episode—either because we care about that episode in particular, or something that holds

across all episodes [Sutton and Barto, 2018, p. 57]. Observe that summing to infinity in (3.1) is then identical to summing over the episode, and that the sum is well-defined regardless of the discount factor γ.

3.1.4 Policies and Value Functions

Policy determines the behaviour of the agent. Formally, a policy $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ defines a probability distribution over actions, given a state. That is to say, $\pi(a \mid s)$ is the probability of selecting action a if an agent is following policy π and in state s.

The *state-value function* $v_\pi : \mathcal{S} \mapsto \mathbb{R}$ for a policy π gives the expected return of starting in a state and following that policy. More formally,

Definition 2 (State-value function)**.** *Let π be a policy and $s \in \mathcal{S}$ be any state. We write $\mathbb{E}_\pi[.]$ to denote the expected value of the random variable G_t as defined in (3.1). Then the state-value function (or simply, value function) for policy π is:*

$$v_\pi(s) := \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]. \tag{3.2}$$

The *action-value function* $q_\pi : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ for a policy π is defined similarly. It gives the expected return of starting in a state, taking a given action, and following policy π thereafter.

Definition 3 (Action-value function)**.** *Let π be a policy, $s \in \mathcal{S}$ be any state and $a \in \mathcal{A}$ any action. Then the action-value function (or, Q-function) for policy π is:*

$$q_\pi(s, a) := \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]. \tag{3.3}$$

3.1.5 Optimal Policies and Optimal Value Functions

The problem of RL is to find an *optimal policy*: one which maximises expected return. All optimal policies share the same value functions. We call these the *optimal state-value function*, v_*, and the *optimal action-value function* q_*:

Definition 4 (Optimal state-value function (from [Sutton and Barto, 2018, p. 62]))**.**

$$v_*(s) := \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S}.$$

Definition 5 (Optimal action-value function (from [Sutton and Barto, 2018, p. 63]))**.**

$$q_*(s,a) := \max_\pi q_\pi(s,a) \quad \forall s \in \mathcal{S} \quad \forall a \in \mathcal{A}.$$

There is a simple connection between optimal Q-function and optimal policy that will be used in Section 3.2:

Claim 1. *If an agent has q_*, then acting according to the optimal policy when in some state s is as simple as finding the action a that maximises $q_*(s,a)$ [Sutton and Barto, 2018, p. 64].*

3.1.6 Bellman Equations

These value functions obey special recursive relationships called Bellman equations. The equations are proved by formalising the simple idea that the value of being in a state is the expected reward of that state, plus the value of the next state you move to. Each of the four value functions defined in Sections 3.1.4 and 3.1.5 satisfy slightly different equations. We prove the Bellman equation for the value function. The remaining three equations have similar proofs [Sutton and Barto, 2018] and are stated here for completeness.

Proposition 1 (Bellman equation for v_π [Sutton and Barto, 2018, p. 59])**.** *Let π be a policy, p the dynamics of an MDP, γ a discount factor and v_π a state-value function. Then:*

$$v_\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi(.|s) \\ s',r\sim p(.,.|s,a)}} \left[r + \gamma v_\pi(s')\right] \tag{3.4}$$

Proof.

$$
\begin{aligned}
v_\pi(s) &:= \mathbb{E}_\pi \left[G_t \mid S_t = s \right] \\
&= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \\
&= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \right] \\
&= \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s' \right] \right] \\
&= \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{\substack{s' \in \mathcal{S} \\ r \in \mathcal{R}}} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right] \\
&= \mathop{\mathbb{E}}_{\substack{a \sim \pi(.|s) \\ s', r \sim p(.,.|s,a)}} \left[r + \gamma v_\pi(s') \right]
\end{aligned}
$$

□

Proposition 2 (Bellman equation for q_π)**.** *Let π be a policy, p the dynamics of an MDP, γ a discount factor and q_π an action-value function. Then:*

$$
q_\pi(s, a) = \mathop{\mathbb{E}}_{s', r \sim p(.,.|s,a)} \left[r + \gamma \mathop{\mathbb{E}}_{a' \sim \pi(.|s')} \left[q_\pi(s', a') \right] \right] \tag{3.5}
$$

Proposition 3 (Bellman equation for v_* [Sutton and Barto, 2018, p. 63])**.** *Let p be the dynamics of an MDP, γ a discount factor and v_* an optimal value function. Then:*

$$
v_*(s) = \max_{a \in \mathcal{A}} \mathop{\mathbb{E}}_{s', r \sim p(.,.|s,a)} \left[r + \gamma v_*(s') \right] \tag{3.6}
$$

Proposition 4 (Bellman equation for q_* [Sutton and Barto, 2018, p. 63])**.** *Let p be the dynamics of an MDP, γ a discount factor and q_* an optimal Q-function. Then:*

$$
q_*(s, a) = \mathop{\mathbb{E}}_{s', r \sim p(.,.|s,a)} \left[r + \gamma \max_{a' \in \mathcal{A}} \left[q_*(s', a') \right] \right] \tag{3.7}
$$

3.2 Reinforcement Learning Solution Methods

One method of solving the reinforcement learning problem is to explicitly solve a set of Bellman optimality equations. For example, in a finite MDP with n states and m actions, the Bellman equations for q_* are a set of $n \cdot m$ equations in $n \cdot m$ unknowns[1]. Given the dynamics p of the MDP, standard techniques for solving systems of equations can be applied. Then, via Claim (1), the agent has an optimal policy [Sutton and Barto, 2018, p. 64].

However, in reality, we rarely have access to p, or sufficient computational resources to solve this system of equations exactly [Sutton and Barto, 2018, p. 66]. Thus, much of the recent literature on RL solution methods focuses on finding approximate solutions.

In particular, there has been rapid development in a class of solution methods called *deep reinforcement learning*. The idea is to use a deep neural network to approximate some function that will yield an optimal policy. Typically, we approximate the optimal value function—this is called *Q-learning*—or the optimal policy, directly—in which case it is termed *policy optimization*[2].

In this section, we explain in detail one particular deep reinforcement learning solution method, the deep Q-network, which is important for the rest of this thesis.

3.2.1 Deep Q-network

The idea of a deep Q-network (DQN) is simply to use a deep NN to approximate the optimal Q-function q_* using a deep neural network $Q(s, a; \theta)$ as the function approximator [Mnih et al., 2015]. The agent-environment interactions yield experience $\langle (s_t, a_t, r_{t+1}, s_{t+1}) \rangle_{t=0}^{T}$ which can be used as training data. We can then simply perform gradient descent on the parameters θ_i at iteration i to reduce the mean squared error between predictions $Q(s, a; \theta_i)$ and targets given by the Bellman equation for $q_*(s, a)$, from Proposition 4. Since we do not have access to the true value of these

[1] This assumes that the agent can take any action in any state.

[2] Some methods, such as DDPG [Lillicrap et al., 2015], TD3 [Fujimoto et al., 2018] and SAC [Haarnoja et al., 2018] approximate both the optimal value function and the optimal policy.

targets, we instead use approximate target values $y = r + max_{a'}\ Q(s', a'; \theta_i^-)$, with θ_i^- some previous network parameters.

So far, this looks very similar to the supervised learning setting. However, there are three important differences that come with reinforcement learning. There are two sources of correlations: both (i) in the data set, and (ii) between $Q(s, a; \theta_i)$ and the targets. Furthermore, (iii) updates to Q may change the policy and thus change the data distribution. This leads to instability in training. To address this, the authors propose two algorithmic innovations. Firstly, instead of training on experience in the order that it is collected, the agent maintains a buffer of experience $D_t = \{e_1, e_2, \ldots, e_t\}$ where $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. When making learning updates, drawing minibatches uniformly at random from this buffer breaks correlations in the experience sequence and smooths over changes in the data distribution, alleviating problems (i) and (iii). This is termed *experience replay*. Secondly, to reduce correlations between Q and the targets and alleviate problem (ii), the approximate target values are updated to match the parameters Q only every C steps for some hyperparameter $C > 1$.

With these changes, we arrive at a loss function $\ell_i(\theta_i)$ for each learning update i:

$$\begin{aligned}
\ell_i(\theta_i) &= \mathbb{E}_{s,a,r}\left[\left(\mathbb{E}_{s'}\left[y \mid s, a\right] - Q(s, a; \theta_i)\right)^2\right] \\
&= \mathbb{E}_{s,a,r}\left[\mathbb{E}_{s'}\left[y - Q(s, a; \theta_i) \mid s, a\right]^2\right] \\
&= \mathbb{E}_{s,a,r}\left[\mathbb{E}_{s'}\left[\left(y - Q(s, a; \theta_i)\right)^2 \mid s, a\right] - \mathbb{V}ar_{s'}\left[y - Q(s, a; \theta_i)\right) \mid s, a\right]\right] \\
&= \mathbb{E}_{s,a,r,s'}\left[\left(y - Q(s, a; \theta_i)\right)^2\right] - \mathbb{E}_{s,a,r}\left[\mathbb{V}ar_{s'}\left[y\right]\right]
\end{aligned}$$

where the expectations and variances are with respect to samples from the experience replay. This loss function is then optimized by stochastic gradient descent with respect to the network parameters θ_i. Note that the final term is independent of these parameters, so we can ignore it. Finally, the authors found that stability is improved by clipping the error term $y - Q(s, a; \theta_i)$ to be between -1 and 1.

We summarise this training procedure in Algorithm 2. To ensure adequate exploration, the agent's policy is ϵ-greedy with respect to the current estimate of the optimal action-value function.

Algorithm 2 Deep Q-learning with experience replay.

1: Initialise replay memory D to capacity N
2: Initialise neural network Q with random weights θ as approximate optimal action-value function
3: Initialise neural network $\hat{Q}$ with identical weights $\theta^- = \theta$ as approximate target action-value function
4: Reset environment to starting state s_0
5: **for** $t = 0, \ldots, T$ **do**
6: With probability ϵ execute random action a_t
7: otherwise execute action $a_t = \arg\max_a Q(s_t, a; \theta)$
8: Observe next state and corresponding reward $s_{t+1}, r_{t+1} \sim p(., . \mid s_t, a_t)$
9: Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in D_t
10: Randomly sample minibatch of transitions $(s_j, a_j, r_{j+1}, s_{j+1}) \sim D_t$
11: Set $y_j = \begin{cases} r_{j+1} \text{ if episode terminates at step } j+1 \\ r_{j+1} + \gamma \max_{a'} \hat{Q}(s_{j+1}, a_j; \theta^-) \text{ otherwise} \end{cases}$
12: Do gradient descent on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t network parameters θ
13: Every C steps set $\theta^- = \theta$
14: **end for**

Note that line 11 assumes we are training DQN to perform an episodic task, hence the first case which follows the convention given in Section 3.1.3 whereby zero reward is given for all states after the terminal state. If the task were instead continuing, line 11 would simply be $y_j = r_{j+1} + \gamma \max_{a'} \hat{Q}(s_{j+1}, a_j; \theta^-)$.

3.3 Reinforcement Learning from Unknown Reward Functions

So far, we have assumed that as the agent interacts with the environment, it receives both information about the next state and the associated scalar reward. This presents an obvious challenge if we want to apply RL to solve real-world problems: since the world does not give scalar rewards, it seems we would have to manually specify a reward function mapping states of the world to rewards. For complex or poorly defined goals, this is difficult to do. If we try instead to design an approxi-

mate reward function, an agent optimizing hard for this objective will do so at the expense of satisfying our preferences [Christiano et al., 2017, p. 1].

To circumvent this issue, a growing body of work studies how to do RL from various forms of human *feedback*, instead of an explicit reward function. Three main feedback methods have been studied, all of which involve a human-in-the-loop providing information to the agent about the desired behaviour. Firstly, an agent may learn from expert demonstrations. This may involve using demonstrations to infer a reward function, an approach known as *inverse reinforcement learning* [Ng and Russell, 2000, Ziebart et al., 2008]. One can then use a standard RL algorithm on this recovered reward function. Other methods involve training a policy directly from demonstrations, referred to as *imitation learning* [Ho and Ermon, 2016, Hester et al., 2017].

Secondly, an agent may learn from feedback on its current policy in the form of scalar rewards [Knox and Stone, 2009, Warnell et al., 2017]. Instead of providing a set of demonstrations, the human-in-the-loop observes the agent's behaviour and gives an appropriate reward. If it is assumed that the human provides this reinforcement according to some latent reward function $\hat{r} : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$, then standard supervised learning techniques can be applied to model this function. The agent can then select actions so as to maximise expected modelled reward. This approach differs from manually specifying a reward function because the human does not provide a precise function mapping all possible state-action pairs to rewards in advance. Rather, the human remains in the loop, providing reinforcement to the agent in an online fashion.

Finally, an agent may learn from binary preferences over trajectories [Wilson et al., 2012, Christiano et al., 2017]. As with the *policy feedback* method, the human-in-the-loop observes the agent's behaviour. However, instead of giving reinforcement in the form of scalar rewards, they are periodically presented with a pair of trajectories and must indicate which they prefer[3]. Given some assumptions about how the hu-

[3]The human also has the option of expressing indifference, or that the trajectories are incomparable.

man's preferences relate to their latent reward function, we can again model $\hat{r}$ by supervised learning. Then, standard RL algorithms can be applied to maximise the expected $\hat{r}$ over time.

Each of these methods have shown promising results, and some of the advantages and disadvantages are summarised in Table 3.1.

Property	**Trajectory preferences**	**Expert demonstrations**	**Policy feedback**
Demandingness for human	Human only needs to judge outcomes	Human needs to perform task (expertly)	Human needs to provide suitable scalar rewards
Upper bound on performance?	Superhuman performance is possible	Impossible to significantly surpass performance of expert	Superhuman performance is possible
Suitability to exploration-heavy tasks	Limited[4]	Well suited, since demonstrations can guide exploration	Limited[5]
Communication efficiency	On the order of hundreds of bits per human hour	Much richer in information than trajectory preferences [6]	Scalar rewards provide richer information than binary preferences over trajectories, but not as rich as demonstrations
Computational efficiency (in simple Atari environments)	On the order of 10 million RL time steps	On the order of 10 million RL time steps	On the order of thousands of learning time steps [7]

Table 3.1: Summary of the properties of using different forms of human feedback in RL without a reward function.

Noticing that the properties on which learning from preferences performs poorly are precisely those on which learning from demonstrations performs well, it is intu-

[4]The human can only give feedback on states visited by the agent. If the agent does not explore well, this limits the amount of information the human can convey. Naively, exploration is determined by the inferred reward function, which may lead to a problematic circularity: if the initialised reward model maps some subset of the state space to low reward, the agent will never explore here, and so the human will never have the chance to give feedback about these states.

[5]See footnote 4

[6][Ibarz et al., 2018] show that demonstrations half the amount of human time required to achieve the same level of performance

[7]Note that the work which prototypes this method trains, with an unspecified amount of compute, a deep autoencoder to extract 100 features from the Atari game screen *before* commencing the RL stage. The other methods do not do such pretraining, thus the reported results may not give a fair comparison.

itive to think that a combination of feedback methods will give better performance than either one individually. Recent work has confirmed this intuition. Specifically, [Ibarz et al., 2018] test this method on nine Atari games [Bellemare et al., 2013], and show that combined preferences and demonstrations outperform only demonstrations on eight games, and only preferences on the four exploration-heavy games[8]. [Palan et al., 2019] show that in a 2D driving simulator [Bıyık and Sadigh, 2017] and two OpenAI Gym environments [Brockman et al., 2016b], using just one expert demonstration reduces by a factor of 3 the number of human preferences required to achieve the same performance [Palan et al., 2019, p. 6].

This dissertation concerns using active learning to improve the performance of reward learning from trajectory preferences. So far, work on feedback from preferences has taken two different approaches: training a reward model on handcrafted features of the environment, and taking the deep learning approach of training a reward model end-to-end without handcrafted features. As active learning has already been shown to improve performance in the former setting [Bıyık and Sadigh, 2017], we focus on applying active learning in the latter setting. Algorithmic innovation in this setting is also more exciting, because if we want to scale RL to real-world tasks, it is unlikely that we will be able to handcraft all the correct features.

In the remainder of this section, we explain in detail the algorithm used for the latter approach, which was developed in [Christiano et al., 2017]. We then summarise briefly the difference in learning from preferences with handcrafted features, in which active learning has already been applied successfully.

3.3.1 Reward Learning from Trajectory Preferences in Deep RL

Setting

The agent-environment interface is as described in Section 3.1.1, with the modification that on step t, instead of receiving reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, there is an *annotator*

[8]The contribution of demonstrations in games without difficult exploration is not significant, except for in two games where demonstrations are harmful compared to using only preferences. The authors hypothesise that this is due to the relatively poor performance of the expert [Ibarz et al., 2018, p. 6]

who expresses preferences between *trajectory segments*, or *clips*. A clip is a finite sequence of states and actions $((s_0, a_0), (s_1, a_1), \ldots, (s_{k-1}, a_{k-1})) \in (\mathcal{S} \times \mathcal{A})^k$. If the annotator prefers some clip σ^1 to another clip σ^2, write $\sigma^1 \succ \sigma^2$. The annotator may also be indifferent between the clips, in which case we write $\sigma^1 \sim \sigma^2$. The agent does not see the annotator's implicit reward function $r : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{R}$, and must instead use the preferences expressed by the annotator to maximise r over time. This is the goal of reward learning from trajectory preferences.

If the annotator could write down their true r, then clearly we could perform traditional RL instead of taking the reward learning approach. However, as we noted above, we are interested in applying RL to tasks for which we do not have the true r, which is when reward learning is useful. Nonetheless, for the purposes of quantitatively evaluating the method, we consider tasks for which we do have access to the true r.

Method

The method has two components: a policy $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ and an estimate of the annotator's reward function, $\hat{r} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{R}$, called a *reward model* or *reward predictor*. Both components are parametrised by a deep neural network. The agent learns to maximise the annotator's implicit reward function over time by iterating through the following three processes:

1. Reinforcement learning by a traditional deep RL algorithm whereby policy π interacts with the environment for T steps and updates its parameters to optimise $\hat{r}$ over time. Agent experience $\mathrm{E} = ((s_0, a_0), (s_1, a_1), \ldots, (s_{T-1}, a_{T-1}))$ is stored.

2. Select pairs of clips (σ^1, σ^2) from E, request a preference μ from the annotator on each sampled pair, and add the labelled pair to the annotation buffer A.

3. Supervised learning to train the reward model on A, the preferences expressed by the annotator so far.

More detail on each process is provided below.

Process 1: Training the policy

As mentioned, process 1 is akin to traditional RL except $\hat{r}$ is used in place of the environment reward r. There are two subtleties to mention. Firstly, since $\hat{r}$ is learned while RL is taking place, [Christiano et al., 2017] prefer policy optimization methods over Q-learning, as these have been successfully applied to RL tasks with a non-stationary reward function [Ho and Ermon, 2016]. Specifically, they use A2C [Mnih et al., 2016] and TPRO [Schulman et al., 2015]. However, the follow up paper [Ibarz et al., 2018] uses a Q-learning method[9], DQfD [Hester et al., 2017], and it is not clear that performance is impaired. Hence, the literature is ambiguous on whether a non-stationary reward function is necessarily problematic for Q-learning.

Secondly, since the reward model $\hat{r}$ is trained only on pairwise comparisons, its scale is underdetermined. Previous work therefore proposes periodically normalising $\hat{r}$ to have zero mean and constant standard deviation over the examples in A. This is crucial for training stability since deep RL is sensitive to the scale of rewards.

Process 2: Selecting and annotating clip pairs

Previous work uses two methods for selecting clip pairs. [Ibarz et al., 2018] sample uniformly at random from E. [Christiano et al., 2017] train an ensemble of three reward predictors and select the clip pairs with the maximum standard deviation across the ensemble. Roughly this favours clip pairs on which the model is most uncertain, with the hope of improving sample efficiency (that is to say, being able to learn $\hat{r}$ with fewer labels from the annotator). However, they found that relative to random selection, this sometimes impaired performance and slowed down training. In Section 5.3 we consider what caused this.

The selected clip pairs σ^1, σ^2 are then annotated with a label μ, indicating which clip is preferred. μ is a distribution over $\{1, 2\}$ where $\mu(1) = 1, \mu(2) = 0$ if $\sigma^1 \succ \sigma^2$;

[9]The reason for deviating from the recommendation in [Christiano et al., 2017] is that [Ibarz et al., 2018] combine reward learning from trajectory preferences and expert demonstrations, and DQfD is state-of-the-art for the latter problem.

$\mu(1) = 0.5, \mu(2) = 0.5$ if $\sigma^1 \sim \sigma^2$; or $\mu(1) = 0, \mu(2) = 1$ if $\sigma^2 \succ \sigma^1$. The triples $(\sigma^1, \sigma^2, \mu)$ are then added to A. The majority of previous work uses a *synthetic annotator* rather than an actual human. Labels are simply generated according to the (hidden) ground truth reward function r, where $\sigma^1 \succ \sigma^2$ if $\sum_t r(s_t^1, a_t^1) > \sum_t r(s_t^1, a_t^1)$; $\sigma^1 \sim \sigma^2$ if $\sum_t r(s_t^1, a_t^1) = \sum_t r(s_t^1, a_t^1)$; and $\sigma^2 \succ \sigma^1$ otherwise. This facilitates quicker experimentation and more clear performance metrics (by using hidden ground truth reward function to evaluate agent performance and reward model alignment).

Process 3: Training the reward model

The training of $\hat{r}$ is based on the following assumption:

Assumption 1. *The annotator's probability of preferring clip 1 to clip 2, $\hat{P}(\sigma^1 \succ \sigma^2)$ is equal to the softmax function evaluated on the value of $\hat{r}$ summed over the two clips.*

More precisely:

$$\hat{P}(\sigma^1 \succ \sigma^2; \hat{r}) = \frac{\exp \sum_t \hat{r}(s_t^1, a_t^1)}{\exp \sum_t \hat{r}(s_t^1, a_t^1) + \exp \sum_t \hat{r}(s_t^2, a_t^2)} \tag{3.8}$$

We can then fit the parameters of $\hat{r}$ by treating the problem as binary classification. In other words, we can use standard supervised learning techniques to optimize the parameters of $\hat{r}$ so as to minimise the cross-entropy loss between the predictions in 3.8 and the annotator's labels.

$$\text{loss}(\hat{r}) = - \sum_{(\sigma^1, \sigma^2, \mu) \in \text{A}} \mu(1) \log \hat{P}(\sigma^1 \succ \sigma^2; \hat{r}) + \mu(2) \log \hat{P}(\sigma^2 \succ \sigma^1; \hat{r}) \tag{3.9}$$

Assumption 1 follows the Elo rating system developed for chess [Elo, 1978]. Given a zero-sum game and two players with a scalar rating, Elo specifies a mapping from player ratings (in our case: reward) to the probability of each player winning (in our case: the probability of each clip being preferred by the annotator).

3.3.2 Reward Learning from Trajectory Preferences with Handcrafted Feature Transformations

Setting

[Bıyık and Sadigh, 2017] consider a more narrow setting. There are two vehicles, H which is "human driven", and R which is a "robot". The vehicles are in a 2D environment with each obeying a simple point-mass dynamics model, with state space $\mathcal{S}$:

$$[\dot{x} \quad \dot{y} \quad \dot{\theta} \quad \dot{v}] = [v \cdot \cos(\theta) \quad v \cdot \sin(\theta) \quad v \cdot u_1 \quad u_2 - \alpha \cdot v]$$

where v and θ are vehicle velocity and direction respectively. The action space $\mathcal{A}$ is $[u_1, u_2]$ which represent steering and acceleration respectively. α is a friction coefficient.

They assume that $\hat{r} : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ is a linear combination of a set of five features:

$$\hat{r}(s, a) = \mathbf{w}^T \phi(s, a).$$

These five weights $\mathbf{w} = [w_1, w_2, w_3, w_4, w_5]$ are handcrafted, and correspond to (i) distance to road boundary, (ii) distance to centre of road lane with an extra penalty for shifting lane, (iii) difference between vehicle speed and the speed limit, (iv) dot product between θ and a vector pointing along the road, and (v) a non-spherical Gaussian over the distance from R to H (to penalise collisions).

Like [Christiano et al., 2017], there is annotator who is queried for preferences on trajectory segments. Some $\mathbf{w}_{\text{true}}$ is specified in advance, and the annotator uses this to answer the queries as in Process 2 of [Christiano et al., 2017].

Method

[Bıyık and Sadigh, 2017] do not attempt to train a policy using the learned reward model. Their performance metric is simply the expected similarity of the true and

predicted feature weights:

$$m = \mathbb{E}\left[\frac{\mathbf{w} \cdot \mathbf{w}_{\text{true}}}{|\mathbf{w}||\mathbf{w}_{\text{true}}|}\right]$$

They generate queries by solving a constrained optimization problem to find the two trajectory segments that will maximise the volume removed from the hypothesis space. Using the same mapping from reward space to preference space as [Christiano et al., 2017] (Equation 3.8), they start with a uniform prior over the space of all $\mathbf{w}$ (uniform over the unit ball), generate a query, get a preference from the annotator, and perform a Bayesian update on this labelled clip pair. Thanks to the simple function form of their reward model, this Bayesian update has a closed form solution. They repeat this procedure until the reward model is close to the true reward function, according to m.

This concludes the background material on Reinforcement Learning. We move to the final chapter of part I, which provides context on uncertainty in deep learning.

Chapter 4

Uncertainty in Deep Learning

If we are to use deep NNs for practical applications, it is crucial that they output not only point estimates, but also their uncertainty in those estimates. For example, suppose a deep NN is being used to drive an autonomous vehicle. If it encounters a situation in which it is uncertain about whether to brake or not, we probably want it to hand over control to a human driver. Equally, if a deep NN is being used for medical diagnosis and is confronted with an unfamiliar stimulus, it should request more data or alert a human doctor. In Section 2, we explained that NNs can output probability distributions. For example, with categorical data, the final layer of the network is typically the softmax function, which gives the probability of the input being in each of the possible classes. However, such probability distributions do not, in fact, accurately reflect uncertainty [Gal, 2017, p. 13].

In this chapter, we first summarise some of the techniques used to equip NNs with the ability to output uncertainty estimates alongside point estimates. We then explain how such uncertainty estimates can be applied to the problem of active learning (AL). Having understood AL, the reader will be equipped with all the prerequisites to understand the application of AL to Reward Modelling.

4.1 Bayesian Neural Networks

Bayesian neural networks (BNNs) different from standard NNs in that they maintain a distribution over their weights rather than point estimates. We require an algorithm for learning from data. The mathematically optimal way to do this for BNNs is to perform Bayesian inference on the weights on the network, i.e. compute a posterior distribution over the weights, given the training data. Then, given the posterior distribution over weights, to answer predictive queries we take expectations under this distribution:

$$P(\hat{y} \mid \hat{x}) = \mathbb{E}_{P(w|D)} \left[P(\hat{y} \mid \hat{x}, w) \right]$$

However, this method of inference is intractable for deep learning models that have millions, or even billions, of parameters. Instead, we need to find an approximation to the true posterior.

Variational Inference (VI) is one such approximation method. Historically, many of the attempts to perform VI on BNNs were impractical. For example, *Bayes by Backprop* [Blundell et al., 2015], requires doubling the number of model parameters, making training more computationally expensive, and is very sensitive to hyperparameter tuning. An alternative is *MC-Dropout*, where we use *dropout*[1] at test time. [Gal, 2017] showed that this approximates VI on BNNs.

The second method is by using a *Deep Ensemble* [Lakshminarayanan et al., 2017]. The idea is simply to train a collection of NNs, with different random weight initialisations, using different randomly sampled minibatches of training data. Computing a forward pass through component of the ensemble separately then functions as if we were drawing samples from a posterior distribution. This does not actually perform a Bayesian approximation but nonetheless has been shown to give good quality uncertainty estimates in practice [Beluch et al., 2018].

[1] Dropout [Srivastava et al., 2014] was first formulated as a regularisation technique. The idea is to remove a randomly sampled subset of NN weights when computing each forward pass through the network.

Equipped with a method to draw samples from an approximate posterior, we can use this method to get uncertainty estimates. In this dissertation, we are interested in the classification setting. The next section explains how to get uncertainty estimates in this setting, and use them in active learning.

4.2 Active Learning

Supervised learning makes use of labelled training data. However, for many real-world problems, obtaining labelled data is expensive. For example, constructing a dataset for image classification requires a human to specify the class of every image in training data. Active learning [Cohn et al., 1996] is a framework for training a model to a particular accuracy while minimising the need for labelled data. The idea is to acquire only the data that is most informative about the model parameters. The key ingredient in active learning is called an *acquisition function*. Given a model $\mathcal{M}$ and pool data $\mathcal{D}_{pool} \subset X$, an acquisition function $a : X \times \mathcal{M} \mapsto \mathbb{R}$ quantifies how informative the label of an element $\mathbf{x} \in \mathcal{D}_{pool}$ would be to the model. This determines the next $\mathbf{x}$ to query the *oracle* for a label. An oracle is operationalised as a function which can be queried for the true label of an example, incurring some cost [Gal et al., 2017]. More precisely, we acquire each new training datum $\mathbf{x}^*$ according to:

$$\mathbf{x}^* = \arg\max_{\mathbf{x} \in \mathcal{D}_{pool}} a(\mathbf{x}, \mathcal{M})$$

Acquisition functions are often based on uncertainty estimates, corresponding with the intuition that it is good to acquire data about which the model is currently uncertain. In this section, we review some common acquisition functions for classification tasks[2] and then discuss how BNNs can provide the uncertainty estimates that these functions require.

[2]Different acquisition functions are used for regression tasks, typically based on predictive variance [Gal, 2017, p. 47] which we do not cover here.

4.2.1 Max Entropy

One natural idea is to acquire $\mathbf{x} \in \mathcal{D}_{pool}$ with the highest entropy [Shannon, 1948] given the current training data $\mathcal{D}_{train}$. Entropy is a key concept in information theory. Given a predictive distribution $p : X \times Y \mapsto [0, 1]$ (i.e. a model which predicts the probability of input $\mathbf{x}$ being in class y) trained on dataset $\mathcal{D}_{train}$, the *predictive entropy* of a new input $\mathbf{x}$ formalises how uncertain p is about the label of input $\mathbf{x}$ [Gal, 2017, p. 52]:

$$\mathbb{H}[y \mid \mathbf{x}, \mathcal{D}_{train}] = -\sum_{c} p(y = c \mid \mathbf{x}, \mathcal{D}_{train}) \log p(y = c \mid \mathbf{x}, \mathcal{D}_{train}) \tag{4.1}$$

where the sum is over the possible classes c of label y.

Example 2. *Consider the binary classification case where we have two classes i.e.* $c \in \{0, 1\}$. *Given some input* $\mathbf{x}$*, if the model* p *predicts* 0.5 *for both classes, i.e.* $p(y = 0 \mid \mathbf{x}, \mathcal{D}_{train}) = p(y = 1 \mid \mathbf{x}, \mathcal{D}_{train}) = 0.5$ *then* $\mathbb{H}[y \mid \mathbf{x}, \mathcal{D}_{train}]$ *will take its maximum value of* $\log(2)$*, corresponding with the intuition that the model is maximally uncertain as to the label of* $\mathbf{x}$ *because it predicts label* 0 *and label* 1 *with equal probability. The other extreme is when the model predicts exactly* 0 *or* 1 *for the label of* $\mathbf{x}$*. In this case* $\mathbb{H}[y \mid \mathbf{x}, \mathcal{D}_{train}] = 0$*. The model is already certain about the label of* $\mathbf{x}$ *(and it would be pointless to acquire its label).*

4.2.2 Bayesian Active Learning by Disagreement

Whilst acquiring points by maximising entropy seems like a reasonable idea, one might wonder whether the data on which the model is the most uncertain are actually the most informative data to acquire. Consider that the pool might contain data which are inherently ambiguous, like a handwritten digit that is a borderline case between a 1 and a 7. It might not be very helpful to acquire such points, because their labels do not actually resolve any uncertainty about the model parameters. Such data are said to have high *aleatoric uncertainty*, i.e. "irreducible" uncertainty, due to noise inherent in the data, which cannot be resolved given more data [Gal, 2017,

p. 7].

On the contrary, it seems better to acquire data about which the model is uncertain, but which also help to resolve uncertainty. Such data is said to have high *epistemic* or *model uncertainty*. We claimed that predictive entropy as in Equation 4.1 represents a model's total uncertainty in its prediction. Total uncertainty comprises both that arising from noisy data, and uncertainty about model parameters and class. In other words, predictive uncertainty is sum of aleatoric and epistemic uncertainty.

Now, how can we specify an acquisition function that selects data which has high epistemic but low aleatoric uncertainty? For reasons which will soon become clear, we denote epistemic uncertainty as $\mathbb{I}[y, \mathbf{w} \mid \mathbf{x}, \mathcal{D}_{train}]$ where $\mathbf{w}$ are the parameters of our model. This is called *mutual information* between the prediction y and the model parameters $\mathbf{w}$. Since we now consider uncertainty in both data and the model parameters, we need some extra notation. Consider our model parameters $\mathbf{w} \sim p(. \mid \mathcal{D}_{train})$ to be sampled from a distribution over model parameters, which depends on the dataset on which the model has been trained. Now, we can write aleatoric uncertainty as expected predictive entropy, where the expectation is over draws of our model parameters: $\mathbb{E}_{p(\mathbf{w}|\mathcal{D}_{train})}[\mathbb{H}[y \mid \mathbf{x}, \mathbf{w}]]$. This formalises the intuition that aleatoric uncertainty is high if, even when model uncertainty is removed (because we consider only a *single* draw of the model parameters), predictive entropy is still high. In this case, the uncertainty can only be coming from noise in the data. Conversely, if we take average over the model parameters, leaving only aleatoric uncertainty, and predictive entropy is low, then aleatoric uncertainty must be low.

Putting this all together, we arrive at a formalisation of epistemic uncertainty, in terms of the difference between predictive and aleatoric uncertainty [Gal, 2017,

p. 53]:

$$\begin{aligned}\mathbb{I}\left[y; \mathbf{w} \mid \mathbf{x}, \mathcal{D}_{train}\right] &= \mathbb{H}\left[y \mid \mathbf{x}, \mathcal{D}_{train}\right] - \mathbb{E}_{p(\mathbf{w}|\mathcal{D}_{train})}\left[\mathbb{H}\left[y \mid \mathbf{x}, \mathbf{w}\right]\right] \\ &= -\sum_c p(y = c \mid \mathbf{x}, \mathcal{D}_{train}) \log p(y = c \mid \mathbf{x}, \mathcal{D}_{train}) \\ &\quad + \mathbb{E}_{p(\mathbf{w}|\mathcal{D}_{train})}\left[\sum_c p(y = c \mid \mathbf{x}, \mathbf{w}) \log p(y = c \mid \mathbf{x}, \mathbf{w})\right] \quad (4.2)\end{aligned}$$

There is an alternative way to arrive at this formalisation of epistemic uncertainty, which is the reason for denoting it as $\mathbb{I}\left[y, \mathbf{w} \mid \mathbf{x}, \mathcal{D}_{train}\right]$. The mutual information between two random variables $\mathbb{I}\left[X; Y\right]$ quantifies the information gained about X by observing Y. Thus, $\mathbb{I}\left[y, \mathbf{w} \mid \mathbf{x}, \mathcal{D}_{train}\right]$ quantifies the information gained about the model parameters by observing the label y of input $\mathbf{x}$, given the current training data $\mathcal{D}_{train}$, which sounds like a good metric for an acquisition function. By the definition of mutual information we can arrive at the same result as in Equation 4.2:

$$\mathbb{I}\left[y; \mathbf{w} \mid \mathbf{x}, \mathcal{D}_{train}\right] := \mathbb{H}\left[y \mid \mathbf{x}, \mathcal{D}_{train}\right] - \mathbb{E}_{p(\mathbf{w}|\mathcal{D}_{train})}\left[\mathbb{H}\left[y \mid \mathbf{x}, \mathbf{w}\right]\right]$$

Using this objective as an acquisition function was first proposed in [Houlsby et al., 2011]. Their intuition that it seeks to acquire examples on which particular settings of the model parameters are highly confident, but in disagreement with each other, or in the language used above, on which aleatoric uncertainty is low but epistemic uncertainty is high. They called this objective Bayesian Active Learning by Disagreement (BALD).

Example 3. *Again, consider the binary classification setting. In the second case presented in Example 2 when individual draws of the model parameters always predict either* 0 *or* 1*, then (as with Max Entropy),* $\mathbb{I}\left[y; \mathbf{w} \mid \mathbf{x}, \mathcal{D}_{train}\right] = 0$*. There is no disagreement on the label of* $\mathbf{x}$ *between different draws of the model parameters, and so the epistemic uncertainty is zero. However, in the first case, where individual draws of the model parameters always predict* 0.5*, then we get* $\mathbb{I}\left[y; \mathbf{w} \mid \mathbf{x}, \mathcal{D}_{train}\right] = \mathbb{H}\left[y \mid \mathbf{x}, \mathcal{D}_{train}\right] - \mathbb{E}_{p(\mathbf{w}|\mathcal{D}_{train})}\left[\mathbb{H}\left[y \mid \mathbf{x}, \mathbf{w}\right]\right] = \log(2) - \log(2) = 0$*. While this input has*

high predictive entropy, its label would not be very informative to the model, because the predictive entropy is due entirely to noise (which cannot be explained away given more data) rather than epistemic uncertainty. For an example which will score highly with respect to BALD, suppose that sequential draws of the model parameters predict the label of some input $\mathbf{x}$ *to be* $0, 1, 0, 1, 0, 1 \ldots$. *Then* $\mathbb{I}[y; \mathbf{w} \mid \mathbf{x}, \mathcal{D}_{train}] = \log(2) - 0 = \log(2)$. *For each draw of the model parameters, the resulting prediction has high confidence; but there is high disagreement between these predictions given different model parameters.*

4.2.3 Variation Ratios

Maximising the Variation Ratios [Freeman, 1965] is similar to Max Entropy in that it seeks the $\mathbf{x}$ on which the model has high predictive uncertainty. The difference is that it does not have an information theoretic formalisation. Instead,

$$\text{variation-ratio}[\mathbf{x}] := 1 - \max_c p(y = c \mid \mathbf{x}, \mathcal{D}_{train})$$

Observe that this metric will be low in the second case presented in Example 2, because for the class y that is always predicted by the model, $p(y \mid \mathbf{x}, \mathcal{D}_{train}) = 1$ and so variation-ratio$[\mathbf{x}] = 0$. Conversely, it will achieve its maximum value of 0.5 (in the binary setting) in the first case in Example 2, because $p(y = 0 \mid \mathbf{x}, \mathcal{D}_{train}) = p(y = 1 \mid \mathbf{x}, \mathcal{D}_{train}) = 0.5$. Thus, it is open to the same failure mode as Max Entropy: acquiring data with high aleatoric but low epistemic uncertainty.

4.2.4 Mean STD

Finally, an approach which derives from the regression literature, but can also be applied to classification, is to acquire points that maximise the mean standard deviation $\sigma(\mathbf{x})$, where the mean is taken over the different classes c that input $\mathbf{x}$ can

take [Kampffmeyer et al., 2016, Kendall et al., 2015].

$$\sigma_c = \sqrt{\mathbb{E}_{p(\mathbf{w}|\mathcal{D}_{train})}\left[p(y=c \mid \mathbf{x}, \mathbf{w})^2\right] - \mathbb{E}_{p(\mathbf{w}|\mathcal{D}_{train})}\left[p(y=c \mid \mathbf{x}, \mathbf{w})\right]^2}$$
$$\sigma(\mathbf{x}) = \frac{1}{C}\sum_c \sigma_c \tag{4.3}$$

This acquisition function has similar properties to BALD in that it standard deviation, like disagreement, will avoid data with high aleatoric and low epistemic uncertainty. This can be seen as follows: if different draws of the model parameters predict the label of input $\mathbf{x}$ to be always 0.5, as in the first case in example 2, the standard deviation of these samples is zero.

4.2.5 Approximating acquisition functions with Bayesian Neural Networks

In essence, evaluating each of the above acquisition functions requires computing one or both of the following quantities:

$$p(y=c \mid \mathbf{x}, \mathcal{D}_{train}) \qquad \text{for some class } c \tag{4.4}$$
$$\mathbb{E}_{p(\mathbf{w}|\mathcal{D}_{train})}\left[\sum_c p(y=c \mid \mathbf{x}, \mathbf{w}) \log p(y=c \mid \mathbf{x}, \mathbf{w})\right]. \tag{4.5}$$

To approximate quantity 4.4, we simply take an average over the predicted values of $p(y=c \mid \mathbf{x}, \mathcal{D}_{train})$ resulting from the different samples from the approximate posterior (or components of the deep ensemble) [Gal et al., 2017]. More precisely,

$$p(y=c \mid \mathbf{x}, \mathcal{D}_{train}) \approx \frac{1}{T}\sum_{i=1}^{T} p_i(y=c \mid \mathbf{x}, \mathcal{D}_{train}) \tag{4.6}$$

where $p_i(y=c \mid \mathbf{x}, \mathcal{D}_{train})$ is the softmax output for class c from the i^{th} sample from the approximate posterior. For example, to approximate max entropy (quantity

4.1), we do:

$$\mathbb{H}\left[y \mid \mathbf{x}, \mathcal{D}_{train}\right] \approx -\sum_{c}\left(\frac{1}{T}\sum_{i=1}^{T} p_i(y=c \mid \mathbf{x}, \mathcal{D}_{train})\right)\log\left(\frac{1}{T}\sum_{i=1}^{T} p_i(y=c \mid \mathbf{x}, \mathcal{D}_{train})\right)$$

which effectively *first* averages predictions over different draws from the approximate posterior, *then* computes the entropy using these averaged predictions.

To approximate quantity 4.5 we *first* compute entropies, *then* average across the resulting entropies from different samples from the approximate posterior [Gal et al., 2017]. More precisely,

$$\begin{aligned}&\mathbb{E}_{p(\mathbf{w}\mid\mathcal{D}_{train})}\left[\sum_{c} p(y=c \mid \mathbf{x}, \mathbf{w})\log p(y=c \mid \mathbf{x}, \mathbf{w})\right]\\ &\approx \frac{1}{T}\sum_{i=1}^{T}\sum_{c} p_i(y=c \mid \mathbf{x}, \mathbf{w})\log p_i(y=c \mid \mathbf{x}, \mathbf{w})\end{aligned} \tag{4.7}$$

where again $p_i(y=c \mid \mathbf{x}, \mathbf{w})$ is the softmax output for class c from the i^{th} sample from the approximate posterior.

Part II

Innovation

Chapter 5

Method

Our aim is to find out whether active learning can be used to decrease the number of queries required to align an RL agent with the intentions of a user. Building on previous work in [Christiano et al., 2017], in which the results were mixed, we form hypotheses for the possible failure modes of active reward modelling, and experiments to test each hypothesis. Our results suggest that the success of active reward modelling depends on properties of the environment and task (as specified by the intentions of the user), which is consistent with the mixed results in previous work.

The first section of this chapter outlines the acquisitions functions and uncertainty estimates that used for active learning, and how they are applied in the reward modelling setting. Section 2 details the core active reward modelling training protocol, which we modify in various ways to test our hypotheses. Section 3 explains these hypotheses, each of which corresponds to a different possible failure mode of active reward modelling. Section 4 outlines and motivates our choices of how to implement the method.

5.1 Applying acquisition functions to reward modelling

[Christiano et al., 2017] use only the mean STD acquisition function. We tried each of the four functions explained in Section 4.2. Each require the ability to get uncertainty estimates by sampling from some appropriate posterior to the reward model

$\hat{r}$ that we are optimizing. For this, we follow the deep ensemble approach explained in Section 4.1 and parametrise $\hat{r}$ with an ensemble of 5 neural networks. Each network has a different random initialisation and is trained independently, that is to say, using independent random minibatches for gradient descent. For a given input $(s, a) \in \mathcal{S} \times \mathcal{A}$, we compute 5 estimates of $\hat{r}(s, a)$: one for each forward pass through a network in the ensemble. In some experiments, we also compare this approach to MC-dropout. We primarily use deep ensembles because it has been shown to give high quality uncertainty estimates in active learning, and also introduces no additional hyperparameters. This minimises the number of possible failure modes of our implementation, facilitating easier diagnosis of unsuccessful experiments.

Applying these acquisition functions to reward modelling requires computing quantities 4.4 and 4.5 in this setting. Quantity 4.4 becomes:

$$
\begin{aligned}
p(y = 0 \mid \mathbf{x}, \mathcal{D}_{train}) &= \hat{P}(\sigma^1 \succ \sigma^2; \hat{r}) \\
&\approx \frac{1}{T} \sum_{i=1}^{T} \hat{P}_i(\sigma^1 \succ \sigma^2; \hat{r}_i) \\
&= \frac{1}{T} \sum_{i=1}^{T} \frac{\exp \sum_t \hat{r}_i(s_t^1, a_t^1)}{\exp \sum_t \hat{r}_i(s_t^1, a_t^1) + \exp \sum_t \hat{r}_i(s_t^2, a_t^2)} \\
p(y = 1 \mid \mathbf{x}, \mathcal{D}_{train}) &= 1 - p(y = 0 \mid \mathbf{x}, \mathcal{D}_{train})
\end{aligned}
$$

where $\hat{r}_i(s, a)$ is the output of the i^{th} component in the ensemble of reward predictors, evaluated on input (s, a). We use $\hat{P}_i(\sigma^2 \succ \sigma^1; \hat{r}_i)$ to denote the annotator's probability of preferring σ^1 to σ^2, using this component of the reward model ensemble, according to equation 3.8. Quantity 4.5 becomes:

$$
\begin{aligned}
& \mathbb{E}_{p(\mathbf{w} \mid \mathcal{D}_{train})} \left[\sum_c p(y = c \mid \mathbf{x}, \mathbf{w}) \log p(y = c \mid \mathbf{x}, \mathbf{w}) \right] \\
\approx \quad & \frac{1}{T} \sum_{i=1}^{T} \hat{P}_i(\sigma^1 \succ \sigma^2; \hat{r}_i) \log \hat{P}_i(\sigma^1 \succ \sigma^2; \hat{r}_i) + \hat{P}_i(\sigma^2 \succ \sigma^1; \hat{r}_i) \log \hat{P}_i(\sigma^2 \succ \sigma^1; \hat{r}_i)
\end{aligned}
$$

5.2 Active Reward Modelling

In this section we present the training protocol we used in our experiments. This is essentially a modification of Algorithm 1 in [Ibarz et al., 2018], excluding the imitation learning and adding active reward learning.

Algorithm 3 Active Reward Modelling.

1: Initialise RL agent
2: Initialise neural network $\hat{r}$ as reward model
3: Initialise experience buffer E for sampling clip pairs
4: Initialise annotation buffer A for storing labelled clip pairs
5: Define acquisition function $a((\sigma^1, \sigma^2), \hat{r})$
6: For each round $i = 1, \ldots, N$ fix some number m_i of labels to request from the annotator in that round
7: Without updating its parameters, run the agent in the environment and add experience to E
8: **for** $i = 1, \ldots, N$ **do**
9: Sample $10m_i$ clip pairs from E
10: Acquire the m_i clip pairs that maximise $a(.,.)$
11: Request labels on these clip pairs (from the annotator) and add them to A
12: (Optionally) reinitialise reward model $\hat{r}$
13: Train $\hat{r}$ to convergence with the preferences in A, by doing gradient descent on loss function 3.9
14: Reinitialise RL agent
15: Clear experience buffer E
16: Train RL agent to convergence with rewards from $\hat{r}$, adding experience to E
17: **end for**

In our implementation, we use DQN for our RL agent. Thus line 16 represents calling Algorithm 2 as a subroutine, except with rewards from $\hat{r}$ instead of from the environment. Following the majority of previous work, all our experiments will use a synthetic annotator to label clip pairs i.e. for each clip pair (σ^1, σ^2) sent for evaluation, we query the ground truth reward function of the environment for the sum of the predicted rewards of each state-action pair in each of the two clips, and return a preference according to which clip has higher total reward. Note that unlike in previous work, on 13 we reinitialise the RL agent and train it to convergence on each iteration. This way, we can be sure the DQN agent is not failing due to its value function having been trained using outdated rewards.

5.3 Possible failure modes of active reward modelling

In this section, we list our hypotheses about different possible failure modes of active reward modelling and then explain each in more detail.

1. Not retraining reward model from scratch
2. Acquisition size is too large
3. Choice of acquisition function
4. Choice of uncertainty estimate method
5. Uncertainty estimate quality is poor in general
6. Learning the reward model is too easy
7. Too few trajectories generated by the agent are disproportionately informative

5.3.1 Not retraining reward model from scratch

The implementation of active reward modelling in [Christiano et al., 2017] initialises the reward model once at the beginning of training, then finetunes that model i.e. trains on more preferences as they are acquired. However, it is standard practice in the Active Learning setting to reinitialise and retrain models from scratch after every acquisition [Kirsch et al., 2019, p. 3]. This is because as training continues, a model that is only initialised once at the beginning of training will have been trained on data gathered early during training much more than those acquired late in training. This may bias the uncertainty estimates; data gathered later may still have incorrectly high uncertainty. [Christiano et al., 2017] propose to alleviate this problem by maintaining only the most recent 3000 preferences in the annotation buffer, but this method does not have a clear theoretical justification, and it is not obvious that it will yield well-calibrated uncertainty estimates.

5.3.2 Acquisition size is too large

Batch acquisition means acquiring the top b points that maximise an acquisition function [Gal et al., 2017]. This may lead to the acquisition of points that are informative individually, but jointly are much less informative than the sum of their parts. For instance, [Kirsch et al., 2019, p. 8] show that with acquisition size 5, BALD underperforms random acquisition on the EMNIST image dataset [Cohen et al., 2017] when acquiring new images with acquisition size 5. Specifically, they observe that several classes are under-represented in the acquisitions made by BALD. [Christiano et al., 2017] use acquisition sizes of up to 500[1].

5.3.3 Choice of acquisition function

No singular acquisition function has yet been shown to give consistently superior performance. Some functions seem to work better than others is different settings, and it is standard practice to compare all the common functions. [Christiano et al., 2017] use only Mean STD.

5.3.4 Choice of uncertainty estimate method

Likewise, no uncertainty estimate method has been shown to consistently outperform the others. Uncertainty estimates for NNs is not yet a well-understood field, and so best practice is to compare the estimates from different methods. [Christiano et al., 2017] use only the deep ensemble approach.

5.3.5 Uncertainty estimate quality

Furthermore, it is still an open question under what conditions any of the uncertainty estimate methods give well-calibrated estimates. There are three possible failure modes which would result in poor quality uncertainty estimates. Firstly, each method requires specifying several hyperparameters; if these are inappropriate,

[1]The first batch of acquisitions is of size 500. For subsequent acquisitions, it is unclear what acquisition size they use.

then the uncertainty estimates will be of poor quality. Two examples of hyper-parameters are the number of components of a deep ensemble and the number of samples with different randomly dropped out weights in MC-dropout. For these, using higher values tends to give more accurate uncertainty estimates, whilst incurring a larger computational cost.

Secondly, it is not clear that the standard acquisition functions can be applied out of the box to learning *in the preference space*. Consider the following example: we are deciding whether to acquire clip pair $c_1 = (\sigma^1, \sigma^2)$ or $c_2 = (\sigma^3, \sigma^4)$ to acquire. Suppose further that the reward model is uncertain whether c_1 has label 0 or 0.5; and uncertain whether p_2 has label 0 or 1. Now, $a_{mean\_STD}(c_2, \hat{r})$ is higher than $a_{mean\_STD}(c_1, \hat{r})$, and thus we will acquire c_2. Yet, when learning in the preference space, using disagreement between models in an ensemble as the basis for an acquisition function may not capture all that we care about. It may be important, for example, to acquire clip pairs that allow the model to make deductions based on transitivity of the preference relation. For suppose that we have already acquired some clip pair (σ^0, σ^1). Then acquiring (σ^1, σ^2) would in effect give for free the label of (σ^0, σ^2), whereas the acquisition of (σ^3, σ^4) would not. Thus, we may need a better proxy than simply disagreement for active learning in the preference space.

Thirdly, my implementation of the acquisition functions may contain bugs.

5.3.6 Learning the reward model is too easy

It is worth noting that sometimes random acquisition just does perform strongly. Specifically, at the beginning of training, the uncertainty estimates used by the acquisition function may have biased noise, while random acquisition has no such bias. In other words, the uncertainty estimates are of poor quality *towards the beginning of training*. Accordingly, we see that in Figure 1 of [Gal et al., 2017], there is no significant difference between the performance of BALD and random acquisition prior to acquiring the 50^{th} example. If, for a given task and environment, a high quality reward model can be learned with a small number of examples, then

we should not expect active learning to show improvement over a random baseline.

5.3.7 Too few trajectories generated by the agent are disproportionately informative

An important difference between active supervised learning and active reward modelling is that in the latter, the objects we acquire are *clip pairs* rather than, for example, single images. This means that evaluating an acquisition function over the dataset is a complexity $O(n^2)$ operation, for n the number of clips acquired. Therefore, the standard active learning procedure of evaluating the acquisition function on each point in the pool dataset and picking that which maximises it, quickly becomes unfeasible as the dataset grows in size. To circumvent this issue, in order to acquire k clip pairs, [Christiano et al., 2017] propose randomly sampling $10k$ clip pairs and selecting the k with the highest score according to the mean STD acquisition function. However, it is not clear that this method suffices to find clip pairs that are more informative than random acquisition: a factor of 10 may simply be too small. If it is important to query the annotator about behaviour that occurs rarely, this behaviour may never be included in the pool of clip pairs over which the acquisition function is evaluated. This is related to exploration problems. In general, RL methods struggle to solve tasks which require difficult exploration, for example the Atari game Montezuma's Revenge. However, exploration is a doubly hard problem for RL from preferences: not only does the agent have to explore states of the environment which are difficult to reach, but also the trajectories generated in that exploration have to be sampled at random into the pool of clip pairs over which the acquisition function is evaluated.

5.4 Implementation Details

Our implementation is available here. The training protocol is implemented mostly in Python [Van Rossum and Drake Jr, 1995]. We use gradient descent to optimize the parameters of the deep Q-network and reward model. Pytorch [Paszke et al., 2017]

is an open source machine learning library, built on top of Python, which provides tools to perform automatic differentiation. This allows us to compute gradient without differentiating by hand our loss function with respect to our model parameters. We implement the buffers for collecting agent experience, storing annotated clips in SciPy [Jones et al., 2001], which is also built on top of Python. This gives finer control over data representation and sampling.

Chapter 6

Experiments and Results

6.1 CartPole experiments

Our first set of experiments use the *CartPole* environment, a classic RL task formulated in [Barto et al., 1983]. As shown in Figure 6.1, the environment features a pole attached to a cart, which moves on a one-dimensional line. The pole starts upright; the objective is to keep the pole from falling by applying a force of $+1$ or -1 to the cart. The episode ends when the cart is more than 2.4 units from the centre or the pole falls to more than 15 degrees from vertical. We used the implementation of this environment provided by *OpenAI Gym* [Brockman et al., 2016b] to run our experiments.

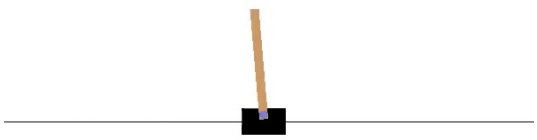


Figure 6.1: The CartPole environment [Brockman et al., 2016a]

As mentioned in Section 5.2, clips are annotated according to the ground truth

reward function of the environment[1]. This reward function is hidden from the agent; it receives only rewards from the reward model, which is trained on preferences according to the ground truth reward. To evaluate the performance of agents, we also use the ground truth reward function as in [Christiano et al., 2017]. As with many RL environments provided by OpenAI Gym, there is a defined threshold above which the agent is considered to have "solved" the environment. Our performance metric is the number of preferences required for an agent to reach this threshold.

Figure 6.2 shows our first attempt to train agents using reward modelling in CartPole. As the reward model is trained on an increasing number of labels, the mean episode return of the agents tends to improve. For clarity, the plot shows only random acquisition and BALD, though we tried all four acquisition functions and found no significant differences between them. We replicate the finding in [Christiano et al., 2017] that active learning does not significantly improve on random acquisition. Our hypotheses about the cause of this negative result will be tested in the following sections. For reference, the mean episode return achieved by a standard RL agent (which can solve CartPole perfectly); taking random actions (*random policy*); and training on a randomly initialised reward model (*random reward model*) are also shown.

We found that even in a task as simple as CartPole, the performance of agents trained via reward modelling has high variance and low stability. For instance, using rewards from a reward model trained on *more* labels sometimes leads to lower mean episode return. The results in Figure 6.2 were averaged over 40 repeats, which was required to decrease standard error from its initial high value. This suggests that whilst reward modelling has been shown, in some sense, to "work", it

[1]The OpenAI Gym implementation of CartPole gives +1 reward on every time step. Since we follow the approach taken in [Christiano et al., 2017] where all clip pairs are of equal length, using this reward for the synthetic evaluation of clips would give the same total reward to every clip. Therefore, we actually give preferences according to the reward function given in [Sutton and Barto, 2018, p. 59]: 0 on every time step, except for failure steps—when the cart moves more than 2.4 units from the centre or the pole falls to more than 15 degrees from vertical—for which −1 is given. Note, however, that we use the OpenAI Gym reward for *testing* agent performance, because there is a defined threshold for "solving" the environment according to this reward function, which is a useful performance metric.

is not a mature technology and requires much more development before it could be applied to more real-world tasks. High variance and low stability is also reported by [Christiano et al., 2017, p. 7], but they do not address the issue in much depth. It is worth noting that deep RL in general suffers from these same problems [Irpan, 2018].

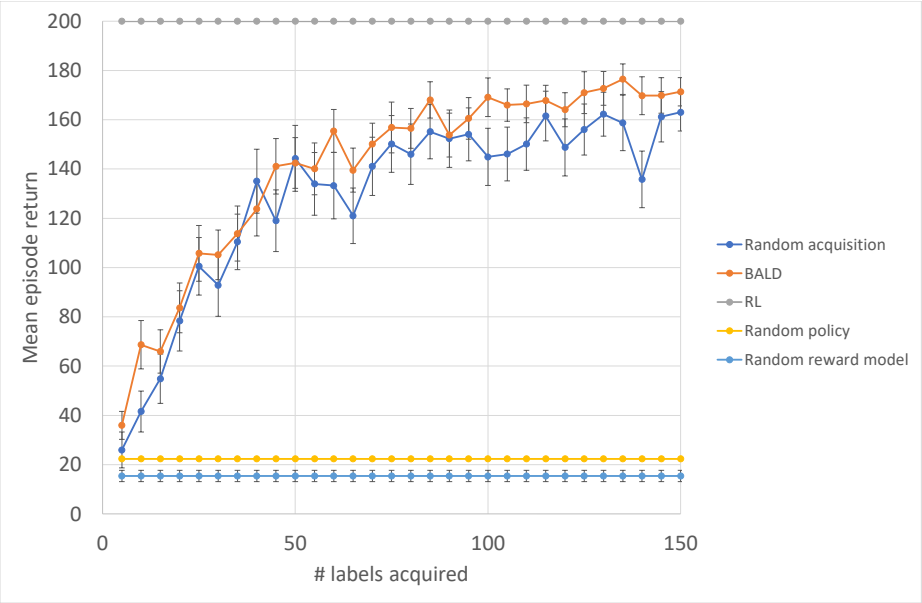


Figure 6.2: Mean episode return of agent trained via reward modelling using random acquisition and BALD. The performance of standard RL, a random policy, and an agent trained on a randomly initialised reward model are also shown. Results are averaged over 40 runs except for the random reward model condition which is averaged over 20. Error bars show standard error. Note that the performance of the agents trained via reward modelling is worse than those in the following sections due to less extensive hyperparameter tuning.

6.2 Hypothesis 1: Reward model retraining

Our first experiment tests the hypothesis that not retraining the reward model from scratch after every acquisition gives low quality uncertainty estimates. We run the training protocol given in Algorithm 3 twice in the CartPole environment, changing whether the reward model is reinitialised in this manner. On each round, 5 preferences are acquired.

As shown in Figure 6.3, we find that under both conditions, random acquisition

and BALD show similar performance, but reward model retraining improves the performance of both. Therefore, whilst it is possible that the uncertainty estimates are harmed by not retraining, it is unclear how much of BALD's performance improvement is due to reward model retraining being better practice in general, because the learned model parameters give equal weight to all the acquired data (the reason for which we see improvement in random acquisition, too).

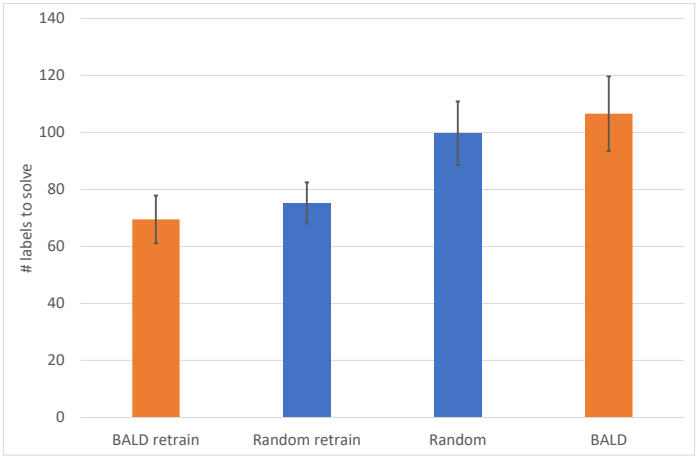


Figure 6.3: Number of labels required to solve CartPole by reward modelling with random acquisition and BALD, with and without retraining reward model after each acquisition. Results are averaged over 20 runs; error bars show standard error.

6.3 Hypothesis 2: Acquisition size

Next, we test the hypothesis that making acquisitions of size greater than 1 harms the performance of BALD-driven reward modelling. We run the training protocol in CartPole with acquisition size 10 and 1, retraining the reward model from scratch after each acquisition, and find that performance improves when acquisition size is reduced from 10 to 1, as per Figure 6.4. This is good evidence that in the reward modelling setting, when the acquisition size is too large, BALD acquires clip pairs that are individually informative, but are jointly less informative than the sum of their parts. So we have good evidence that using too large an acquisition size is one

failure mode of active reward modelling.

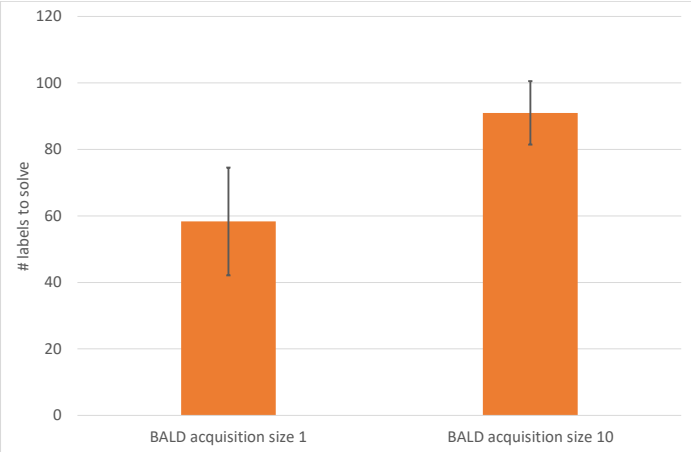


Figure 6.4: Number of labels required to solve CartPole by reward modelling with BALD, using acquisition sizes 1 and 10. Results are averaged over 20 repeats for acquisition size 10 and 6 repeats for acquisition size 1; error bars show standard error.

6.4 Hypotheses 3 and 4: Uncertainty estimate method and acquisition functions

Having established that—in the CartPole environment—not retraining the reward model from scratch after every acquisition, and using large acquisition sizes harms the performance of active reward modelling, we retest all four acquisition functions with reward model retraining and acquisition size 1 compared to the random acquisition baseline. We also evaluate random acquisition using an ensemble of reward models, to ensure that any performance gains from active learning are not simply due to ensembling. Finally, we compare using MC-Dropout for estimating uncertainty with the ensemble approach taken so far.

Our results, summarised in Figure 6.5, show improved performance in general, but still no significant difference between random acquisition and active learning[2].

[2]Note that BALD and random outperform their best results in the first two experiments because in between these experiments we performed extensive DQN hyperparameter tuning. Additionally, we found that not retraining the *agent* from scratch on each round also impairs performance, as

This provides evidence against the hypothesis that the choice of acquisition function or uncertainty estimate method has a significant effect on the performance of active reward modelling. Contrary to the suggestion in [Christiano et al., 2017, p. 6], merely changing these methods does not make active reward modelling improve on random acquisition, at least in the CartPole environment.

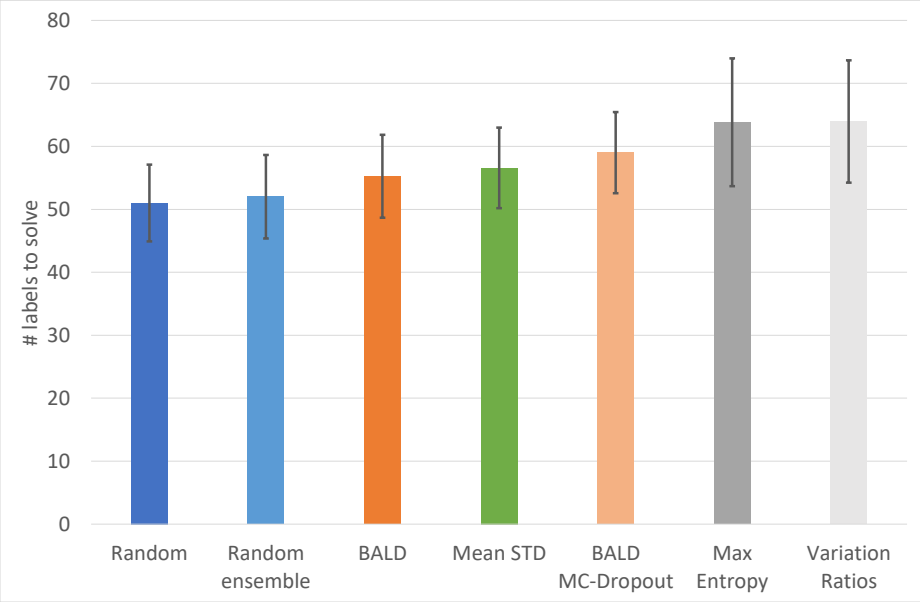


Figure 6.5: Number of labels required to solve CartPole by reward modelling using various acquisition functions. The performance of using random acquisition with an ensemble of reward models, and BALD with uncertainty estimates from MC-Dropout with is also shown. Results are averaged over 20 repeats; error bars show standard error.

In the experiments that follow, unless otherwise stated we continue to do reward model retraining and use acquisition size 1. Incorporating the latter modification by acquiring only 1 label per round, i.e. training the agent on the new reward model after every new acquisition, would make each experiment take a long time to run because agent training is a costly procedure. Therefore, we continue to acquire multiple labels per round, whilst using acquisition size 1. The procedure for doing so

mentioned in Section 5.2. Being independent of the reward modelling aspect of the training protocol, such modifications affect all conditions equally and thus do not invalidate the results; they merely help to speed up experiments because agent performance performance is not harmed by having been trained (without subsequent reinitialisation) on a lower quality reward model.

is shown in Algorithm 4; this essentially replaces line 9 to 12 of the original training protocol given in Algorithm 3. The idea is simple: we repeatedly acquire 1 clip pair, query its label, reinitialise the reward model and train it to convergence, until m_i clip pairs are acquired and we proceed to agent training, and then onto the next round[3].

Algorithm 4 Acquiring a batch of clip pairs with acquisition size 1.

1: Sample $10m_i$ clip pairs from E
2: **repeat**
3: Acquire the single clip pair that maximises $a(.,.)$
4: Request label on this clip pair (from the annotator) and add it to A
5: Reinitialise reward model $\hat{r}$
6: Train $\hat{r}$ to convergence with the preferences in A, by doing gradient descent on loss function 3.9
7: **until** m_i clip pairs have been acquired

6.5 Hypothesis 5: Uncertainty estimate quality

Active reward modelling may be unable to outperform random acquisition due to poor quality uncertainty estimates, due to any of the reasons discussed in Section 5.3.5, or an implementation bug.

To test these hypotheses, we set up an experiment in which many duplicate clip pairs are added to the pool dataset. Essentially, this adds redundancy to the pool: once one of the duplicated clip pairs is acquired, no further information is gained about the annotator's latent reward function by acquiring exact copies of it.

More precisely, our experimental setup is to run the training protocol with both random acquisition and BALD, acquiring 5 labels per round (using acquisition size 1), with the following modification: after sampling $10 \times 5 = 50$ clip pairs from the experience buffer E (line 1 in Algorithm 4), we make 50 identical copies of one of the clip pairs. Now, this set of 100 clip pairs is used as the pool dataset over which the

[3]Because reward model training also takes time, this modification will still slow down the training protocol. Future work could investigate whether a recent improvement on the BALD algorithm, called BatchBALD [Kirsch et al., 2019], could successfully optimize this subroutine by removing the need to iteratively acquire single examples and retrain the model.

acquisition function is evaluated on line 3. We call this setup *Repeated CartPole*[4].

Modifying the procedure in this way will impair the performance of random acquisition. In expectation, half of the 5 randomly sampled clip pairs will be exact duplicates, meaning that the reward model is trained on a poorer quality dataset due to the redundant acquisitions. However, the performance of BALD should be unimpaired, because after acquiring one of the duplicate clip pairs, the information gained by acquiring the same clip again is zero, so BALD should not reacquire the example, but instead pick 4 other clip pairs from the original set. If, on the other hand, BALD is giving poor quality uncertainty estimates, then it will erroneously acquire duplicate examples and so the performance of BALD will also be significantly impaired in Repeated CartPole.

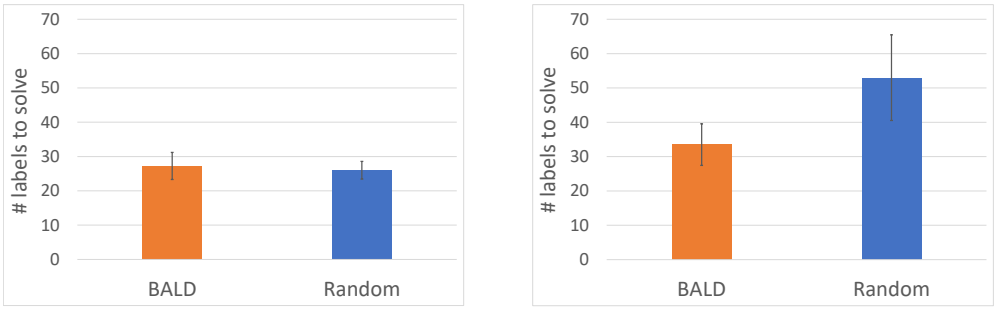
As shown in Figure 6.6, we find that BALD's performance is not significantly harmed, whilst random acquisition performs poorly[5]. This provides some evidence that poor quality uncertainty estimates are not a failure mode of BALD in this environment: if they were, then we would expect BALD to perform as poorly as the random baseline in Repeated CartPole. In particular, hyperparameter tuning or bugs in the code are unlikely to be crippling the method, and it seems that standard acquisition functions can be applied out-of-the-box in the preference space.

6.6 Hypothesis 6: Learning the reward model is too easy

It may be that the CartPole task is simply too easy for active reward modelling to show improvement over random acquisition. As mentioned in section 5.3.6, we

[4]This experiment was inspired by the *Repeated MNIST* setup in [Kirsch et al., 2019], which makes a somewhat similar modification to test a different hypothesis.

[5]The performance of BALD and random acquisition in the standard CartPole are both substantially better than in previous experiments. Having noticed the high variance of repeated experiments, we began testing agent performance several times on each iteration of the training protocol, rather than just at the end of agent training (testing agent performance means measuring mean episode return of the current policy in greedy forms, i.e. without taking random actions some ϵ fraction of the time, over a fixed number of episodes, typically 100). This follows the testing regime used in [Mnih et al., 2015]. Since this change is independent of the reward modelling aspect of the training protocol, it does not invalidate our previous results.



(a) Standard CartPole **(b)** Repeated CartPole

Figure 6.6: Number of labels required to solve two variants of CartPole, using BALD and random acquisition functions for sampling clip pairs. Results are averaged over 20 repeats; error bars show standard error.

	CartPole	Repeated CartPole
Average MI of BALD over pool dataset	0.06 ± 0.01	0.16 ± 0.02

Table 6.1: Mean mutual information over the pool dataset in CartPole and Repeated CartPole experiments, averaged over all acquisitions, with standard error from four repeat trials.

expect uncertainty estimates to be of poor quality until approximately 50 examples have been acquired. So, given that reward modelling can solve CartPole with around 30 labels, we conjecture that BALD's inability to outperform random acquisition in this setting can be explained by the fact that learning the reward model for this environment and task is simply too easy.

We gathered some evidence for this conjecture by comparing the mutual information scores in Repeated CartPole with those in standard CartPole. Since there are many redundant examples in the pool dataset of Repeated CartPole, we expect the average mutual information over this dataset (if uncertainty estimates are well-calibrated) to be lower than that in standard CartPole. However, as shown in Table 6.1, we find the opposite result. This supports the conjecture that the uncertainty estimates are of low quality given this small number of examples (though they are clearly of high enough quality of outperform random acquisition for Repeated CartPole).

To summarise the results so far, we have shown that in the CartPole environment,

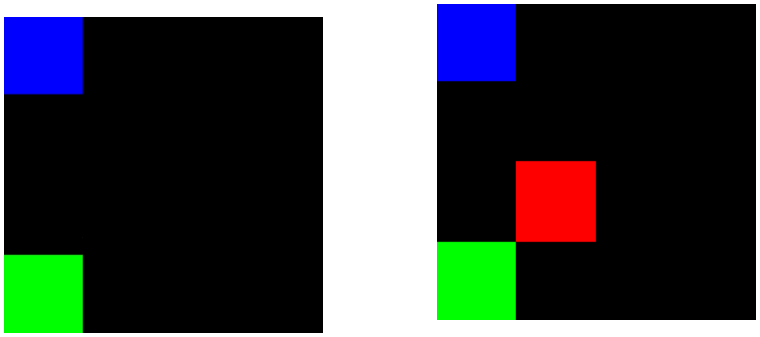
not reinitialising the reward model and using large acquisition sizes harms active reward modelling, and the former problem also harms non-active reward modelling. This provides some evidence for hypothesis 1 being a failure mode of active learning in this environment, and good evidence for hypothesis 2 being a failure mode. With these changes, however, active reward modelling still cannot outperform random acquisition, so these are not the only failure modes. We then provided evidence against hypotheses 3 and 4 by comparing four common acquisition functions and two uncertainty estimates methods and finding that no combination improves on random acquisition. Finally, the Repeated CartPole experiment suggests that the uncertainty estimates are of questionable quality in CartPole, where only a small number of examples are required to learn a good reward model. Thus, hypotheses 6 and 7 remain plausible failure modes.

Now, we seek to gather more evidence for these results by trying to replicate them in a different environment.

6.7 Gridworld experiments

A gridworld is a simple two-dimensional environment of size 4x4, containing an agent (blue), one goal cell (green) and an optional lava cell (red). On each time step, the agent receives an observation of the environment, which is an RGB pixel value for each cell, and takes one of four actions (up, down, left or right). The environment gives rewards of $+1$ for reaching the goal, -1 for moving onto the lava, and 0 otherwise. Episodes terminate when the goal or lava is reached, or when 50 time steps have passed. An example state of the environment is shown in Figure 6.7. This is similar to the environments used in [Kenton et al., 2019] and here. We define solving this environment as getting a mean episode return of greater than 0.95 across 100 episodes, equivalent to reaching the goal in 95 of 100 episodes.

The simplest configuration of this gridworld has no lava cell and a goal cell with a fixed location. We use this simplest configuration to try to replicate the finding that in simple tasks and environments, active reward modelling is unable



(a) Gridworld with one goal cell.

(b) Gridworld with one goal and one lava cell.

Figure 6.7: Two examples of possible gridworld environments. The blue, green and red cells are the agent, goal and lava, respectively.

to outperform random acquisition. Specifically, we fix the goal location to be in the bottom left corner, and the agent starting location to be in the top left corner, corresponding to Figure 6.7a. By a quick simulation, we found that 17% of the clips generated by an agent taking random actions in this environment have total reward +1. This should mean that a decent proportion of the clip pairs acquired at random will contain information about synthetic annotator's preferences and so non-active reward modelling should be able to solve the environment with very few acquisitions, whilst BALD should suffer from poor quality uncertainty estimates, having only been trained on a small number of examples. Indeed, as seen in Figure 6.8 the results affirmed this. This is consistent with the finding that hypothesis 6 is one remaining failure mode. We also observe that BALD's mutual information estimates are high on acquisitions 10-30, but low on 0-10 and 30-40. Their being (erroneously) low on 0-10 provides additional evidence for this finding.

6.8 Hypothesis 7: Too few trajectories generated by the agent are disproportionately informative

The next reasonable step to prune the hypothesis space is to make the task harder such that the reward model harder to learn. This will provide evidence about whether hypothesis 6 is the only remaining failure mode. So, we introduce a lava cell into the gridworld, also with a fixed position, placing it close to the goal cell, as

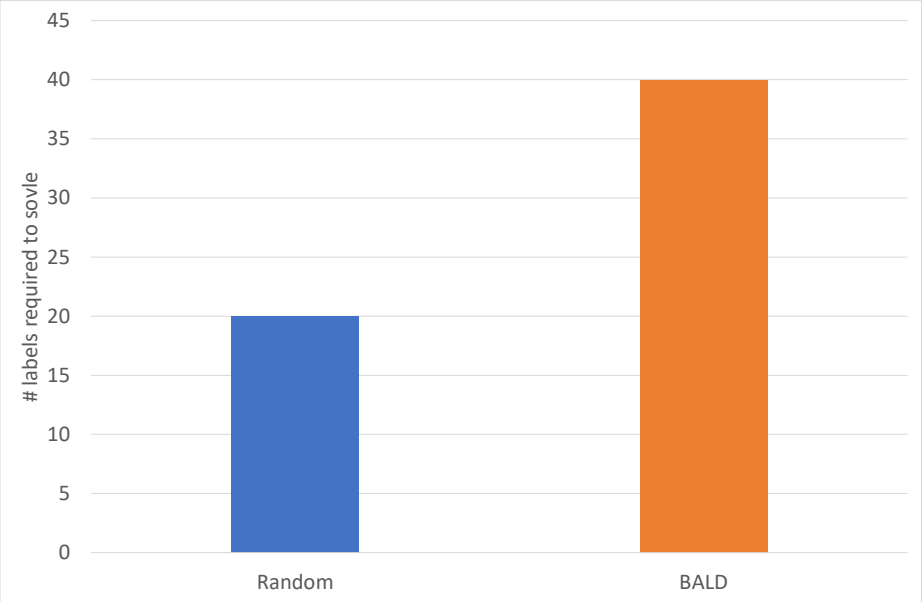


Figure 6.8: Number of labels required to solve the gridworld shown in Figure 6.7a via reward modelling using BALD and random acquisition. Results shown are only one run.

shown in Figure 6.7b. By a simulation, we found that around 95% of clips sampled from the trajectories of a random policy have zero return, around 1% have return greater than or equal to $+1$, and around 4% have return less than or equal to -1[6]. Because of the infrequency of clips with positive return, random sampling will collect few such clips, and therefore non-active reward modelling will struggle to learn the annotator's reward function. However, since such clips contain vital information, active reward modelling should acquire these clips. If it does not, this is evidence that hypotheses 5 or 7 are failure modes.

As shown in Figure 6.9, the results provides good evidence that the uncertainty estimates in active reward modelling are of good enough quality to outperform random acquisition in this more complicated environment (against hypothesis 5) and that enough trajectories generated in this environment are disproportionately infor-

[6]Note that we modified the clip length to be 1 for this experiment, because we realised that single state-action pairs (rather than sequences, like in CartPole) is the level of granularity on which most information about the task is contained.

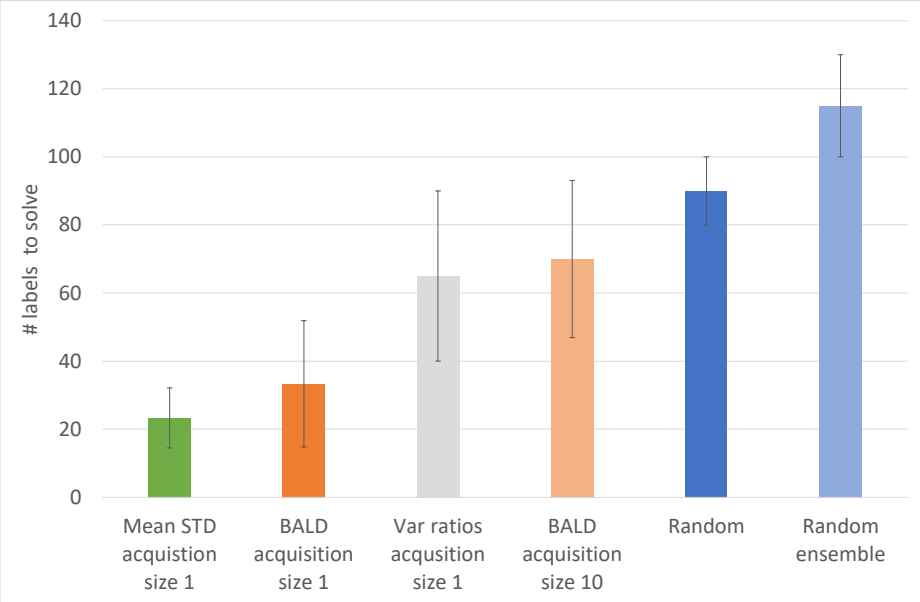


Figure 6.9: Number of labels required to solve the gridworld shown in Figure 6.7b via reward modelling using random acquisition, and mean STD, BALD and variation ratios with acquisition size 1. BALD with acquisition size 10 is also shown. Results are averaged over 3 random seeds.

mative (against hypothesis 7). We also evaluate BALD using acquisition size 10, and replicate the finding of section 6.3 that this hurts the performance of BALD. Surprisingly, variation ratios performs significantly worse than mean STD, and using an ensemble in random acquisition seems to hinder performance relative to not ensembling.

However, we remain somewhat unsure about hypothesis 7. Whether this is a failure mode depends on the frequency with which informative clip pairs are generated by the agent. In turn, this frequency depends on the task and environment, as well as the agent's exploration method and the way clip pairs are annotated. There certainly are environments in which too few informative trajectories are informative, as seen in Figure 3 of [Christiano et al., 2017]: the difficulty of passing other cars in Enduro by random exploration means that few or no clip pairs used to train reward model actually feature good behaviour[7]. Active learning would not

[7]This is only true when the ground truth reward is used to annotate the clips. Human annotators

improve over random acquisition in this case. The open question is: as reward modelling is scaled to more complicated environments and tasks, will the method used in [Christiano et al., 2017] to avoid evaluating the acquisition function over the entire set of clip pairs (which has size $O(n^2)$ for n clips from which the pairs are sampled) suffice to identify informative clip pairs? Or, do we tend to see informative pairs missed out? If so, then the success of active reward modelling will require some new method to efficiently find informative clip pairs in a set of size $O(n^2)$.

prefer clips in which even small progress is made towards passing other cars, which circumvents this issue. Nonetheless, the basic problem remains that on some tasks, random exploration may not induce behaviours that have sufficient differences in quality for even human annotators to distinguish between.

Chapter 7

Conclusions

7.1 Summary

The aim of this dissertation was to explore the environments and tasks in which active reward modelling can improve on a the baseline of random acquisition. We summarise our findings as follows. (1) If the task and environment can be learned by reward modelling with a small number of labels, then active reward modelling is unlikely to show improvement over random acquisition because uncertainty estimates tend to be poorly calibrated with a small number of labels. (2) Else, if the agent rarely generates trajectories on which the preferences give good information about the annotator's latent reward function, then the pool dataset will contain even fewer informative clip pairs. In the absence of informative clip pairs in the pool dataset, active reward modelling will not beat random acquisition. Indeed, standard reward modelling is similarly prone to failure in such environments. (3) Else, if the agent generates informative trajectories with a frequency such that *some* informative clip pairs end up in the pool dataset, then we expect active reward modelling to improve on random acquisition.

7.2 Future Work

Firstly, as argued in section 6.1, reward modelling is not a mature technology and requires much development before it could be applied to more real-world tasks. In particular, there is high variance and low stability between repeats of the same experiment with different random seeds. It would be useful to determine whether this is purely due to the brittleness of current deep RL methods. If so, we can expect these instabilities to be resolved with general progress in deep RL. Otherwise, further work on the brittleness of reward learning will be required. One might gather data on this by finding how well the current reward learning approach generalises to scenarios not in the training data, and compare its generalisation ability to that of deep RL.

Secondly, as explained in section 6.8, we would like to know if some new method of efficiently finding informative clip pairs in a set of size $O(n^2)$ is required. If so, some possibilities include: modifying the method of fitting the reward model to be based on TrueSkill [Herbrich et al., 2007] instead of Elo, which explicitly maintains uncertainty estimates for each "player" (i.e. clip pair); pairing each clip in the set with some reference clip and estimating the uncertainty of these pairs; or specifically generating behaviour on which the preferences will be informative, rather than always trying to maximise expected return.

Finally, and most importantly, I am uncertain about how the results about the failure modes of active reward learning in my testing environments will transfer to more complicated environments. Specifically, hypotheses 6 and 7 have strong dependencies on the environment. Given this, it seems important to think clearly about the desiderata for the next set of environments in which reward modelling technology is developed. Innovation should proceed in environments which increasingly resemble the real world as opposed to games, otherwise any progress may simply be overfitting to particular properties of the environments in which the "innovation" was tested. For instance, we would like the environment to have a ground truth reward function such that the preferences it induces are qualitatively similar to those

of a human annotator (in hindsight, my gridworld experiments, which used a sparse ground truth reward function, were far from ideal for this reason).

7.3 Relation to Material Studied on the MSc Course

The *Advanced Machine Learning* course was the most closely related to the material covered in this dissertation. I learned the basics of implementing deep learning algorithms, as well as an overview of reinforcement learning and uncertainty in deep learning. In *Computational Learning Theory*, I learned to be more rigorous in formalising and reasoning about algorithms that learn from experience. The *Probability and Computing* course was also invaluable in equipping me with a better understanding of probability theory.

7.4 Personal Development

My most significant personal development in undertaking this project was to cultivate a shift in attitude when an idea or implementation does not work as expected. Towards the beginning of the project, I made desperate attempts to make active reward modelling give the results I wanted. Later on, I began to pay closer attention to exactly what was going wrong, and use these findings to inform my next actions, rather than fighting against them. I learned to notice confusion, rather than avoiding it. For instance, early in the code development, random acquisition with an ensemble was consistently underperforming random acquisition without an ensemble. I avoided this problem for some time, but eventually dug deeper and realised that not implementing per-component normalisation of reward models was impairing performance, which turns out to be crucial for the proper performance of the algorithm. Similarly, I spent a week running experiments using the OpenAI Gym environment *Acrobot*. Sometimes, this environment was solved with almost no reward model training, which I initially passed over as a fluke. Only when the results began to look increasingly dubious did I look deeper into these runs and find

that Acrobot can actually be solved (for some random seeds) using a randomly initialised reward model, rendering invalid all findings in that environment. Following confusions like this as soon as they arose would have resulted in faster and deeper progress on my research question.

References

[Amodei et al., 2016] Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mané, D. (2016). Concrete Problems in AI Safety. pages 1–29.

[Barto et al., 1983] Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846.

[Bellemare et al., 2013] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

[Beluch et al., 2018] Beluch, W. H., Genewein, T., Nürnberger, A., and Köhler, J. M. (2018). The power of ensembles for active learning in image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9368–9377.

[Blundell et al., 2015] Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight Uncertainty in Neural Networks. 37.

[Brockman et al., 2016a] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016a). The cartpole-v0 environment.

[Brockman et al., 2016b] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016b). Openai gym. *arXiv preprint arXiv:1606.01540*.

[Bıyık and Sadigh, 2017] Bıyık, E. and Sadigh, D. (2017). Active Preference-Based Learning of Reward Functions.

[Christiano et al., 2017] Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S., and Amodei, D. (2017). Deep reinforcement learning from human preferences.

[Cohen et al., 2017] Cohen, G., Afshar, S., Tapson, J., and van Schaik, A. (2017). Emnist: an extension of mnist to handwritten letters. *arXiv preprint arXiv:1702.05373*.

[Cohn et al., 1996] Cohn, D. A., Ghahramani, Z., and Jordan, M. I. (1996). Active learning with statistical models. *Journal of artificial intelligence research*, 4:129–145.

[Elo, 1978] Elo, A. E. (1978). *The rating of chessplayers, past and present*. Arco Pub.

[Freeman, 1965] Freeman, L. C. (1965). *Elementary applied statistics: for students in behavioral science*. John Wiley & Sons.

[Fujimoto et al., 2018] Fujimoto, S., van Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*.

[Gal, 2017] Gal, Y. (2017). Uncertainty in Deep Learning. *Phd Thesis*, 1(1):1–11.

[Gal et al., 2017] Gal, Y., Islam, R., and Ghahramani, Z. (2017). Deep Bayesian Active Learning with Image Data.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[Haarnoja et al., 2018] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*.

[Henderson et al., 2018] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018). Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[Herbrich et al., 2007] Herbrich, R., Minka, T., and Graepel, T. (2007). TrueSkill(TM): A Bayesian Skill Rating System - Microsoft Research. *Adv. Neural Inf. Process. Syst. 20*, pages 569–576.

[Hester et al., 2017] Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Dulac-Arnold, G., Osband, I., Agapiou, J., Leibo, J. Z., and Gruslys, A. (2017). Deep Q-learning from Demonstrations.

[Ho and Ermon, 2016] Ho, J. and Ermon, S. (2016). Generative Adversarial Imitation Learning.

[Houlsby et al., 2011] Houlsby, N., Huszár, F., Ghahramani, Z., and Lengyel, M. (2011). Bayesian Active Learning for Classification and Preference Learning. pages 1–17.

[Ibarz et al., 2018] Ibarz, B., Leike, J., Pohlen, T., Irving, G., Legg, S., and Amodei, D. (2018). Reward learning from human preferences and demonstrations in Atari. (2017):1–20.

[Irpan, 2018] Irpan, A. (2018). Deep reinforcement learning doesn't work yet. https://www.alexirpan.com/2018/02/14/rl-hard.html.

[Jones et al., 2001] Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open source scientific tools for Python. [Online; accessed 24/08/2019].

[Kampffmeyer et al., 2016] Kampffmeyer, M., Salberg, A.-B., and Jenssen, R. (2016). Semantic segmentation of small objects and modeling of uncertainty in urban remote sensing images using deep convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 1–9.

[Kendall et al., 2015] Kendall, A., Badrinarayanan, V., and Cipolla, R. (2015). Bayesian segnet: Model uncertainty in deep convolutional encoder-decoder architectures for scene understanding. *arXiv preprint arXiv:1511.02680*.

[Kenton et al., 2019] Kenton, Z., Filos, A., Evans, O., and Gal, Y. (2019). Generalizing from a few environments in safety-critical reinforcement learning. *arXiv preprint arXiv:1907.01475*.

[Kirsch et al., 2019] Kirsch, A., van Amersfoort, J., and Gal, Y. (2019). BatchBALD: Efficient and Diverse Batch Acquisition for Deep Bayesian Active Learning.

[Knox and Stone, 2009] Knox, W. B. and Stone, P. (2009). Interactively shaping agents via human reinforcement: The Tamer Framework. *Proc. fifth Int. Conf. Knowl. capture - K-CAP '09*, page 9.

[Lakshminarayanan et al., 2017] Lakshminarayanan, B., Pritzel, A., and Blundell, C. (2017). Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems*, pages 6402–6413.

[Leike et al., 2018] Leike, J., Krueger, D., Everitt, T., Martic, M., Maini, V., and Legg, S. (2018). Scalable agent alignment via reward modeling: a research direction.

[Lillicrap et al., 2015] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. 48.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski,

G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[Ng and Russell, 2000] Ng, A. and Russell, S. (2000). Algorithms for inverse reinforcement learning. *Proc. Seventeenth Int. Conf. Mach. Learn.*, 0:663–670.

[Palan et al., 2019] Palan, M., Landolfi, N. C., Shevchuk, G., and Sadigh, D. (2019). [DemPref] Learning Reward Functions by Integrating Human Demonstrations and Preferences.

[Paszke et al., 2017] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*.

[Schulman et al., 2015] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2015). Trust Region Policy Optimization.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction (2nd Edition, in preparation)*.

[Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

[Todorov et al., 2012] Todorov, E., Erez, T., and Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE.

[Van Rossum and Drake Jr, 1995] Van Rossum, G. and Drake Jr, F. L. (1995). *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Chung, J., Mathieu, M., Jaderberg, M., Czarnecki, W., Dudzik, A., Huang, A., Georgiev, P., Powell, R., Ewalds, T., Horgan, D., Kroiss, M., Danihelka, I., Agapiou, J., Oh, J., Dalibard, V., Choi, D., Sifre, L., Sulsky, Y., Vezhnevets, S., Molloy, J., Cai, T., Budden, D., Paine, T., Gulcehre, C., Wang, Z., Pfaff, T., Pohlen, T., Yogatama, D., Cohen, J., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Apps, C., Kavukcuoglu, K., Hassabis, D., and Silver, D. (2019). AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/.

[Warnell et al., 2017] Warnell, G., Waytowich, N., Lawhern, V., and Stone, P. (2017). Deep TAMER: Interactive Agent Shaping in High-Dimensional State Spaces. pages 1545–1553.

[Wilson et al., 2012] Wilson, A., Fern, A., and Tadepalli, P. (2012). A Bayesian approach for policy learning from trajectory preference queries. *Adv. Neural Inf. Process. Syst.*, 2:1–9.

[Ziebart et al., 2008] Ziebart, B., Maas, A., Bagnell, J., and Dey, A. (2008). Maximum entropy inverse reinforcement learning. *Proc. AAAI*, (January):1433–1438.