ARMA: Active Reward Modelling for Agent Alignment

Sam Clarke

September 2019

Abstract

Contents

I Background 4

1 Introduction 5

1.1 Relation to Material Studied on the MSc Course 5

2 Reinforcement Learning 6

2.1 Elements of Reinforcement Learning 6

2.2 Finite Markov Decision Processes 6

2.2.1 The Agent-Environment Interface 7

2.2.2 Goals and Rewards 8

2.2.3 Returns and Episodes 8

2.2.4 Policies and Value Functions 9

2.2.5 Optimal Policies and Optimal Value Functions 10

2.2.6 Bellman Equations 11

2.3 Reinforcement Learning Solution Methods 12

2.3.1 Deep Neural Networks 13

2.3.2 Deep Q-network 17

2.4 Reinforcement Learning from Unknown Reward Functions 19

2.4.1 Reward Learning from Trajectory Preferences in Deep RL . . 22

2.4.2 Reward Learning from Trajectory Preferences with Handcrafted Feature Transformations 26

3 Uncertainty in Deep Learning 28

3.1 Bayesian Neural Networks 29

3.1.1 Bayes by Backprop 30
3.2 Model Uncertainty in BNNs 30

4 Active Learning **31**
4.1 Acquisition Functions 32
4.1.1 Max Entropy 32
4.1.2 BALD 32
4.1.3 Variation Ratios 35
4.1.4 Mean STD 35
4.2 Applying Active Learning to RL without a reward function 36
4.2.1 APRIL 36
4.2.2 Active Preference-Based Learning of Reward Functions with handcrafted feature transformations 36
4.2.3 Deep RL from Human Preferences 36

II Innovation **37**

5 Method **38**
5.1 Possible failure modes of active reward modelling 38
5.1.1 Failure modes of active learning in general 39
5.1.2 Failure modes of active learning in the reward modelling setting 40
5.2 ARMA 42
5.3 Acquisition Functions and Uncertainty Estimates 43
5.4 Implementation Details 43

6 Experiments and Results **44**
6.1 Hypothesis 3: Reward model retraining 45
6.2 Hypothesis 4: Acquisition size 45
6.3 Hypothesis 5: Uncertainty estimates and Acquisition Functions . . . 46

7 Conclusions **48**

7.1 Summary 48

7.2 Evaluation 48

7.2.1 Personal Development 48

7.3 Future Work 48

References **49**

Appendices

A Appendix A **55**

A.1 CartPole Experimental Details 55

A.1.1 Ground truth reward function 55

A.1.2 Experiment 1 details 55

Part I

Background

Chapter 1

Introduction

1.1 Relation to Material Studied on the MSc Course

Chapter 2

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].

In this section we first introduce the elements of RL informally. We then formalise the RL Problem as the optimal control of incompletely-known Markov decision processes (finite? do I talk about PO?). We give a taxonomy of different RL solution methods and conclude with a description of one such method, Deep Q-Learning (DQN) that is of particular importance in this dissertation.

2.1 Elements of Reinforcement Learning

This subsection will introduce agent, environment, policy, reward signal, value function and [model] informally, similar to S&B 1.3. Is this necessary or should I skip straight to the formalism?

2.2 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. Then...

2.2.1 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.

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}. 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}.

2.2.2 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.

2.2.3 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]:

$$G_t := \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{2.1}$$

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 $\{R_{t+1}, R_{t+2}, R_{t+3}, \dots\}$.

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 (2.1) 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 (2.1) is then identical to summing over the episode, and that the sum is well-defined regardless of the discount factor γ.

2.2.4 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*

(2.1). Then the state-value function (or simply, value function) for policy π is:

$$v_\pi(s) := \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right]. \tag{2.2}$$

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:*

$$q_\pi(s, a) := \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right]. \tag{2.3}$$

2.2.5 Optimal Policies and Optimal Value Functions

**Some words here based on the TODO above. The problem of Reinforcement Learning is thus to find an optimal policy.

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 2.3:

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].*

2.2.6 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 2.2.4 and 2.2.5 satisfy slightly different equations. We prove the Bellman equation for the value function and state the remaining three 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:*

$$v_\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi(.|s)\\ s',r\sim p(.,.|s,a)}} \left[r + \gamma v_\pi(s')\right] \tag{2.4}$$

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:

$$q_\pi(s,a) = \mathop{\mathbb{E}}_{s',r\sim p(.,.|s,a)}\left[r + \gamma \mathop{\mathbb{E}}_{a'\sim\pi(.|s')}\left[q_\pi(s',a')\right]\right] \tag{2.5}$$

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:*

$$v_*(s) = \max_{a\in\mathcal{A}} \mathop{\mathbb{E}}_{s',r\sim p(.,.|s,a)}\left[r + \gamma v_*(s')\right] \tag{2.6}$$

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:*

$$q_*(s,a) = \mathop{\mathbb{E}}_{s',r\sim p(.,.|s,a)}\left[r + \gamma \max_{a'\in\mathcal{A}}\left[q_*(s',a')\right]\right] \tag{2.7}$$

2.3 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 approx-

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

imate 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 first provide background material on deep neural networks, necessary to understand how they approximate a Q-function or policy. We then explain in detail one particular deep reinforcement learning solution method, the deep Q-network, which is important for the rest of this thesis.

2.3.1 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 $\mathbf{Y}Y$ some imge 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 andsubsubsectionModel The model, of course, is a deep NN. This simplest form of deep NN is a *deep feedforward network*[3]. 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. 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*

[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.

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

layer. f is composed of three functions: $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$, where:

$$f^{(1)}(\mathbf{x}) = \mathbf{z} = \mathbf{W^1 x} \quad \textit{for some } 2 \times 2 \textit{ matrix } \mathbf{W^1}$$

$$f^{(2)}(\mathbf{z}) = \mathbf{a} = \text{ReLu}(\mathbf{a}) := \max\{0, \mathbf{a}\} \quad \textit{where } \max(.) \textit{ is applied pointwise}$$

$$f^{(3)}(\mathbf{a}) = \mathbf{W^2 a} \quad \textit{for some } 1 \times 2 \textit{ matrix } \mathbf{W^2}$$

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[4] [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 this modified function will give the same result and maximising the likelihood. So,

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

we define the negative log likelihood cost function $\text{NLL}(\theta)$ as

$$\begin{aligned}\text{NLL}(\theta) :=& -\frac{1}{m}\log L(\theta) \\ =& -\frac{1}{m}\sum_{i=1}^{m}\log p_{\text{model}}(\mathbf{x}_i;\theta) \\ =& -\mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}}\left[\log p_{\text{model}}(\mathbf{x};\theta)\right] \end{aligned} \tag{2.8}$$

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].

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 2.8 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 2.8 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]:

$$\nabla_\theta \text{NLL}(\theta) \approx \frac{1}{m'} \nabla_\theta \sum_{i=1}^{m'} -\log p_{\text{model}}(\mathbf{x}_i; \theta) \tag{2.9}$$

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.

2.3.2 Deep Q-network

We now have the necessary background to explain the deep Q-network (DQN). The idea 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 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 serrelations: both (i) in the data set, and (ii) between $Q(s, a; \theta_i)$ and the targets. AlsoFurti) 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 ex-

ploration, 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 2.2.3 whereby zero reward is given for all states after the terminal state. If the task were instead continuing, line 11 would simply be $y_j = r_{j+1} + \gamma \max_{a'} \hat{Q}(s_{j+1}, a_j; \theta^-)$.

2.4 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 approximate 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[5]. Given some assumptions about how the human's preferences relate to their latent reward function, we can again model $\hat{r}$ by

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

supervised learningI Then, standard RL algorithms can be applied to maximise the cted $\hat{r}$ over time.

Each of these methods have shown promising results, and some of the advantages and disadvantages are summarised in Table 2.1.

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[6]	Well suited, since demonstrations can guide exploration	Limited[7]
Communication efficiency	On the order of hundreds of bits per human hour	Much richer in information than trajectory preferences [8]	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 [9]

Table 2.1: 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 intuitive to think that a combination of feedback methods will give better performance than either one individually. Recent work has confirmed this intuition. Specifically,

[6]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. Since exploration is determined by the inferred reward func

[7]See footnote 6

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

[9]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 do not allow for a fair comparison.

[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[10]. [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.

2.4.1 Reward Learning from Trajectory Preferences in Deep RL

Setting

The agent-environment interface is as described in Section 2.2.1, with the modification that on step t, instead of receiving reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, there is an *annotator* 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

[10]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]

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[11], 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 ** 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 $\sigma^1 \succ \sigma^2$; $\mu(1) = 0.5, \mu(2) = 0.5$ if $\sigma^1 \sim \sigma^2$; or $\mu(1) = 0, \mu(2) = 1$ if $\sigma^2 \succ \sigma^1$. The triples

[11]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.

$(\sigma^1, \sigma^2, \mu)$ 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,* $\hat{P}(\sigma^1 \succ \sigma^2)$*, depends exponentially on the value of* $\hat{r}$ *summed over the clips.*

This allows us to write:

$$\hat{P}(\sigma^1 \succ \sigma^2; \hat{r}) = \frac{\exp \sum_t \hat{r}(s_t^1, a_t^1)}{\exp \sum_t \hat{r}(s_t^1, a_t^1) + \exp \sum_t \hat{r}(s_t^2, a_t^2)} \tag{2.10}$$

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 2.10 and the annotator's labels.

$$\text{loss}(\hat{r}) = - \sum_{(\sigma^1, \sigma^2, \mu) \in \text{A}} \mu(1) \log \hat{P}(\sigma^1 \succ \sigma^2; \hat{r}) + \mu(2) \log \hat{P}(\sigma^2 \succ \sigma^1; \hat{r}) \tag{2.11}$$

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).

2.4.2 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 features $\mathbf{w} = [w_1, w_2, w_3, w_4, w_5]$ 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 2.10), 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.

Chapter 3

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 encounter 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, rather than blindly taking its best guess. 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 2.3.1, we explained that NNs output probability distributions. For example, with categorical data, the final layer of the network is typically a 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]. The field of *Bayesian Deep Learning* aims to equip neural networks with the ability to accurate uncertainty alongside point estimates. To do so, we place probability distributions over each weight in the network, rather than single values as in standard NNs. Such models are called Bayesian neural networks (BNNs). We then perform Bayesian inference on the weights on the network, i.e. we 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 proper 'dogmatic' Bayes approach is intractable: taking an expectation under the posterior distribution on weights effectively means computing a forward pass with an infinite ensemble of neural networks, each with one possible configuration of the weights, then weighting their outputs according to the posterior distribution.

When we cannot do exact Bayesian inference, there are two common approximation methods: Markov Chain Monte Carlo and Variational Inference (VI). The paper proposes to use the latter.

Historically, many of the attempts to do so were not very practical. For example, one algorithm, 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. However, recent techniques allow almost any network trained with a stochastic regularisation technique, such as dropout, to, given an input, obtain a predictive mean and variance (uncertainty), without any complicated augmentation to the network [Gal, 2017, p. 15].

3.1 Bayesian Neural Networks

From Andreas: 2.3 Bayesian Neural Networks (BNN) In this paper we focus on BNNs as our Bayesian model because they scale well to high dimensional inputs, such as images. Compared to regular neural networks, BNNs maintain a distribution over their weights instead of point estimates. Performing exact inference in BNNs is intractable for any reasonably sized model, so we resort to using a variational approximation. Similar to Gal et al. [10], we use MC dropout [9], which is easy to implement, scales well to large models and datasets, and is straightforward to optimise.

Explain what VI is by lifting words from your BBB report.

3.1.1 Bayes by Backprop

Typically, when we train a neural network, it learns single point estimates for each of its weights, to minimise some loss function, given some data. To do bayesian inference on a neural network, we calculate the posterior distribution of the weights given the training data, $P(w|D)$. Figure **??** illustrates the difference.

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 proper 'dogmatic' Bayes approach is intractable: taking an expectation under the posterior distribution on weights effectively means computing a forward pass with an infinite ensemble of neural networks, each with one possible configuration of the weights, then weighting their outputs according to the posterior distribution.

When we cannot do exact Bayesian inference, there are two common approximation methods: Markov Chain Monte Carlo and Variational Inference (VI). The paper proposes to use the latter.

3.2 Model Uncertainty in BNNs

This section would say basically the same stuff as active learning, because the acquisition functions we consider are precisely the ways to get model uncertainty with a BNN in classification setting. Not sure if I need to include it, or maybe I should collapse the sections.

Chapter 4

Active Learning

Supervised learning requires labelled training data. However, for many real-world problems, obtained labelled data is expensive. For example, constructing a dataset for image classification requires a human to specify the category of every image in training data. Active learning [Cohn et al., 1996] is a framework for training a model using less data to achieve the same performance, by acquiring only the data that is most informative to the model. 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. More precisely, we acquire each new training datum x^* according to [Gal et al., 2017]:

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

Acquisition functions are often based on uncertainty estimates, corresponding with the intuition that it is good to acquire data on which the model is currently uncertain. In this section, we review some common acquisition functions for classification tasks[1] and then discuss how BNNs can provide the uncertainty estimates that these functions require.

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

4.1 Acquisition Functions

4.1.1 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, which is a mathematical formalisation of uncertainty. More specifically, 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).*

4.1.2 BALD

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. Such data are said to have high

aleatoric uncertainty, which is to say uncertainty due to noise inherent in the data, which cannot be resolved given more data [Gal, 2017, p. 7]. Even if we had a many labels of 7's that look like 1's, or 1's that look like 7's, given a new ambiguous 1 or 7, we will still (correctly) be uncertain about its label.

On the contrary, it seems better to acquire data which the model is uncertain about, but which also help to resolve uncertainty. Such data 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 have 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 noisy 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 a single draw of 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 sound 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, consider if sequential draws of the model parameters predict the label of some input $\mathbf{x}$ *to be* $0, 1, 0, 1, \ldots$. *Then* $\mathbb{I}[y; \mathbf{w} \mid \mathbf{x}, \mathcal{D}_{train}] = \log(2) - 0 = \log(2)$, *corresponding with the high confidence, but high disagreement between different model parameters.*

4.1.3 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,

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

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.

4.1.4 Mean STD

Finally, an approach with less theoretical grounding that is nonetheless used in the literature 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].

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

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 is easy to see: 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 Applying Active Learning to RL without a reward function

Discussion of previous work.

4.2.1 APRIL

4.2.2 Active Preference-Based Learning of Reward Functions with handcrafted feature transformations

4.2.3 Deep RL from Human Preferences

[Christiano et al., 2017] compute the standard deviation of each clip pair (σ^1, σ^2) across the ensemble. In the language of acquisition functions, they use $a_{mean\_std}((\sigma^1, \sigma^2), \hat{r})$, which in this setting is written as:

$$a_{mean\_std}((\sigma^1, \sigma^2), \hat{r}) = \sqrt{\frac{1}{3}\sum_{i=1}^{3}\left(\hat{P}_i(\sigma^1 \succ \sigma^2; \hat{r}) - \overline{\hat{P}(\sigma^1 \succ \sigma^2; \hat{r})}\right)^2},$$

where $\hat{P}_i(\sigma^1 \succ \sigma^2; \hat{r})$ is the prediction $\hat{P}(\sigma^1 \succ \sigma^2; \hat{r})$ according to component i of the ensemble, and $\overline{\hat{P}(\sigma^1 \succ \sigma^2; \hat{r})}$ is the average of these predictions.

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 the previous work in [Christiano et al., 2017], in which active learning did not help, we form hypotheses to explain this failure. We form an experiment to test each hypothesis. Our results suggest that the success of active reward modelling depends on properties of the environment and intentions of the user, making it difficult to answer the question in general.

The first section of this chapter explains our hypotheses, each of which corresponds to a different possible failure mode. Section 2 details the core active reward modelling training protocol, which we modify in various ways to test our hypotheses. Section 3 ... Section 4 outlines and motivates our choices of how to implement the method.

5.1 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. Hypotheses 1-4 apply to active learning in general, whilst hypotheses 5-7 apply specifically to learning from preference in deep RL.

1. Uncertainty estimate method

2. Quality of acquisition function
3. Not reinitialising reward model
4. Acquisition size
5. Pool dataset too uniform
6. Pool dataset is sufficiently non-uniform but we cannot search $O(n^2)$ MI matrix, and the solution of randomly sampling $10x$ clip pairs does not suffice to find the rare clips
7. Some problem of applying acquisition functions to preference space

5.1.1 Failure modes of active learning in general

The quality of uncertainty estimates and acquisition function are two possible failure modes. Indeed, [Christiano et al., 2017, p. 6] explain the failure of their implementation as due to $a_{mean\_std}((\sigma^1, \sigma^2), \hat{r})$ across an ensemble of three networks as a "crude approximation" to reward predictor uncertainty. Possible solutions to this failure mode are to use a different acquisition function (such as BALD [Houlsby et al., 2011]), or to derive uncertainty estimates using a different method (such as MC-Dropout [Gal and Ghahramani, 2015] or Bayes by Backprop [Blundell et al., 2015]).

Thirdly, the implementation 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. As training continues, the model will have been trained on preferences gathered early during training much more than those gathered towards the end of training. This may decrease the quality of the uncertainty estimates; preferences gathered later may still have relatively high uncertainty. [Christiano et al., 2017] propose to alleviate this problem by maintaining only the most recent 3000 preferences in the annotation buffer, but it is not clear that this unprincipled method will lead to well-calibrated uncertainty estimates.

Fourthly, when performing *batch acquisition*, that is, acquiring the top b points that maximise an acquisition function [Gal et al., 2017], may lead to the acquisition

of points that are informative individually, but jointly are much less informative than the sum of their parts. In particular, the acquisitions may not be very diverse. Indeed, [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 BALD several classes are under-represented in the acquisitions made by BALD. [Christiano et al., 2017] use acquisition sizes of up to 500[1].

5.1.2 Failure modes of active learning in the reward modelling setting

Unlike active learning in the standard supervised case, the pool dataset in active reward modelling is not known in advance, but is gathered as the RL agent encounters new states. If the trajectories of the RL agent at a certain stage of training are all similar, then the informativeness of the clip pairs in the pool dataset may be close to uniform. In this case, we would not expect active learning to improve on random acquisition. This is a fifth possible failure mode of active reward modelling.

A further difference between active supervised learning and active reward modelling setting is that 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 a_{mean_std}. 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

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

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.

A seventh possible failure mode, 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[2]; and uncertain whether p_2 has label 0 or 1. Now, $a_{mean\_std}(c_1, \hat{r}) =$, whereas $a_{mean\_std}(c_2, \hat{r}) =$, 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.

Finally, it is worth noting that sometimes random acquisition just does perform strongly. At the beginning of training, the uncertainty estimates used by the acquisition function may have biased noise, while random acquisition has no such bias. At the end of training, if most of the pool data have been acquired, active learning will show little improvement over random acquisition. Thus, depending on the setting, there is a potentially small window in which active learning may help. The usefulness of active learning depends on how large this window is. In active reward

[2]TODO I may need to spell this out more, in terms of draws from the posterior giving something like 0, 0.5, 0, 0.5 ... Can I borrow Yarin's presentation in thesis of a similar case?

modelling, is it an open question how large this window is for different tasks.

5.2 ARMA

6 In this section we present the training protocol we developed. Specifically, we explain how to apply acquisition functions to reward modelling. In our implemen-

Algorithm 3 ARMA: Active Reward Modelling for Agent Alignment.

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: Reinitialise reward model $\hat{r}$
13: Train $\hat{r}$ to convergence with the preferences in A, by doing gradient descent on loss function 2.11
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**

tation, 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 total reward of each state-action pair in each of the two clips, and return a preference according to which clip has higher total reward.

5.3 Acquisition Functions and Uncertainty Estimates

We tried each of the four acquisition functions explained in Section 4.1. 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 parameterise $\hat{r}$ 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[3]. For a given input $(s, a) \in \mathcal{S} \times \mathcal{A}$, we can draw 5 samples from the approximate posterior of $\hat{r}$: one for each forward pass through a network in the ensemble. There are other methods for sampling from an approximate posterior (see Section 3.1), but we choose this method because while it is computationally more expensive than, for example, MC-Dropout, it introduces no additional hyper-parameters to be tuned. This minimises the number of possible failure modes of our implementation, facilitating easier diagnosis of unsuccessful experiments.

5.4 Implementation Details

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.

[3] TODO mention that there is no principled way to implement ensemble, and that using different minibatches is a design choice?

Chapter 6

Experiments and Results

Our first set of experiments test hypotheses 1-4: failure modes of active learning in general. We use the *CartPole* environment, a classic RL task formulated in [**?**]. As shown in Figure **??**, 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.

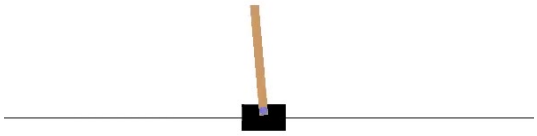

Figure 6.1: The CartPole environment [Brockman et al., 2016a]

As mentioned in Section , clips are annotated according to the ground truth reward function of the environment. There are several ways of implementing this function which all encode the same goal of keeping the pole upright; see Appendix

A for the function we use. 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 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.

6.1 Hypothesis 3: 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, 10 preferences are acquired.

As shown in Figure ??, 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 it gives equal weight to all the acquired data (the reason for which we see improvement in random acquisition, too).

6.2 Hypothesis 4: 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, and find that performance improves

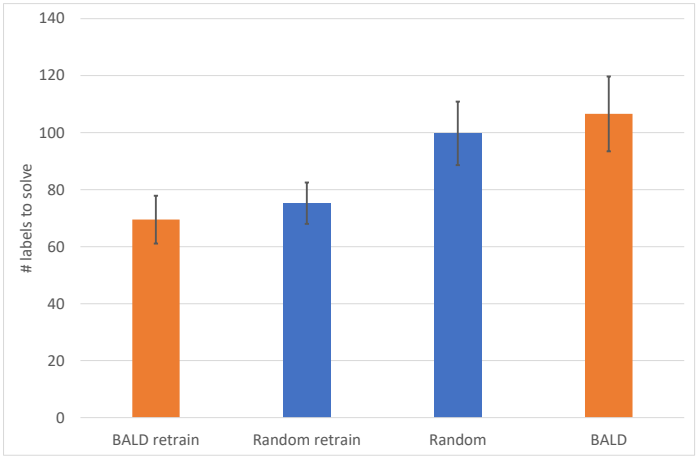

Figure 6.2: Number of labels required to solve CartPole by reward modelling with random acquisition and BALD, with and without retraining reward model after each acquisition. Results are averaged over 20 runs; error bars show standard error.

when making acquisitions of size 1, as per Figure ??[1]. This indicates that 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.

6.3 Hypothesis 5: Uncertainty estimates and Acquisition Functions

Having established... we try all the acquisition functions. Plus MC-Dropout.

[1]Note that the performance of BALD with acquisition size 10 is worse than that shown in Figure ?? because we performed the acquisition size experiment before realising that not retraining the *agent* from scratch on each round also impairs performance, as mentioned in Section 6. In neither condition of this experiment do we retrain the agent.

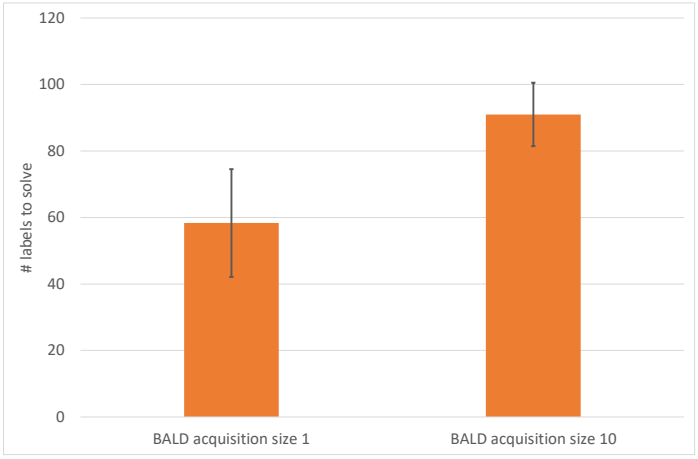

Figure 6.3: Number of labels required to solve CartPole by reward modelling with BALD, using acquisition sizes 1 and 10. Results are averaged over 20 repeats for acquisition size 10 and 6 repeats for acquisition size 1; error bars show standard error.

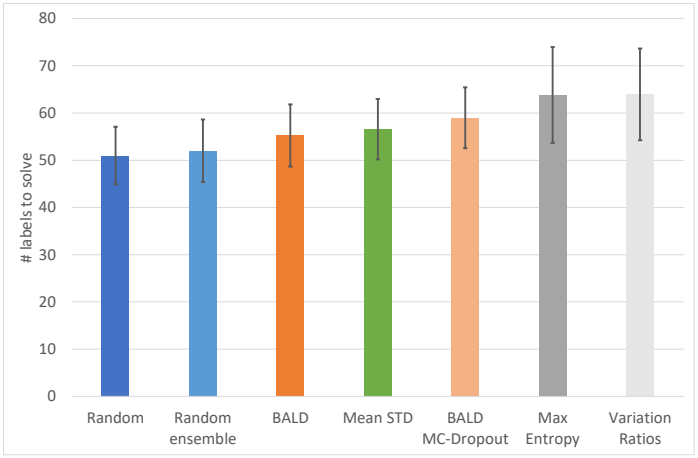

Figure 6.4: Number of labels required to solve CartPole by reward modelling with.... Results are averaged over 20 repeats; error bars show standard error.

Chapter 7

Conclusions

7.1 Summary

7.2 Evaluation

7.2.1 Personal Development

Not to seek to make the thing work, but to wonder, why? To notice confusion rather than running from it. 'Acrobot shouldn't solve in one label'. Led to us finding it was a bad environment for testing reward modelling. Random acquisition with ensemble underperforming random without ensemble early in training -¿ realised that not implementing per-component normalisation of reward models; not normalising at training time correctly. (Well, in fact this was a case of running from this confusion for ages, before finally—and begrudgingly—facing up to it. In some sense this was a key learning point; feeling that noticing confusion pays off in the long run.) Ignoring occasional NaNs was another example. When I finally debugged it rationally, I found that normalising observations greatly improves reward model performance.

7.3 Future Work

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.

[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 and Ghahramani, 2015] Gal, Y. and Ghahramani, Z. (2015). Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. 48.

[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*.

[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.

[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*.

[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.

[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.

[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.

[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.

[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.

Appendices

Appendix A

Appendix A

A.1 CartPole Experimental Details

A.1.1 Ground truth reward function

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 instead use 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.

A.1.2 Experiment 1 details

Train agent for 3000 steps (sufficient for convergence) Train reward model for 2000 epochs (sufficient for convergence) Hyperparameters (get from cartpole-defaults)