ARMA: Active Reward Modelling for Agent Alignment

Sam Clarke

September 2019

Abstract

Contents

Appendices

Part I

Background

Chapter 1

Introduction

1.1 Relation to Material Studied on the MSc Course

Chapter 2

Neural Networks

Or can I fit this in more naturally as part of the narrative? In any case, it need to come before I mention DQN.

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].

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.

3.1 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?

3.2 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...

3.2.1 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}.

3.2.2 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.

3.2.3 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]:*

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

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

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 (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.

3.2.4 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*

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

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

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:*

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

3.2.5 Optimal Policies and Optimal Value Functions

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]))**.**

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

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

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

There is a simple connection between optimal Q-function and optimal policy that will be used in Section 3.3:

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].*

3.2.6 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 3.2.4 and 3.2.5 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:*

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

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:*

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

3.3 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, the literature on RL solution methods focuses on finding approximate solutions.

In this section, we give a brief and incomplete taxonomy of RL solution methods, most of which give approximate solutions. Then, we focus on one such method, the deep Q-network, which is important for the rest of this thesis.

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

3.3.1 Taxonomy of RL Solution Methods

The following taxonomy draws on those given in [Sutton and Barto, 2018] and [Achiam, 2019].

```
RL Solution Methods
  Tabular Solution Methods
    Dynamic Programming
      Policy Iteration
      Value Iteration
    Monte Carlo Methods
      MC Prediction
      MC Control
    Temporal-Difference Learning
      Sarsa
      Q-Learning
  Approximate Solution Methods
    Deep RL
      Model-Free
        Policy Optimisation
          Policy Gradient
          A2C / A3C
          PPO
          TPRO
        Q-Learning
          DQN
          C51
        Policy Optimisation + Q-Learning
          DDPG
          TD3
          SAC
      Model-Based
        Given the Model
          AlphaZero
        Learn the Model
          I2A
```

3.3.2 Deep Q-network

The deep Q-network (DQN) agent approximates q_* using a deep convolutional neural network $Q(s, a; \theta)$ as the function approximator [Mnih et al., 2015]. θ are the parameters (weights) of the neural network, which is called a Q-network. It can be trained by performing gradient descent on the parameters θ_i at iteration i to reduce the mean-squared error between $Q(s, a; \theta_i)$ and the Bellman equation for $q_*(s, a)$, given in Proposition (4). Since we do not have access to the true value of this target, we instead use approximate target values $y = r + max_{a'}\ Q(s', a'; \theta_i^-)$, with θ_i^- some previous network parameters.

One could then perform standard gradient descent on this loss function using the experience collected by the agent $\langle(s_t, a_t, r_{t+1}, s_{t+1})\rangle_{t=0}^{T}$ as data, as one would do in the supervised learning setting. However, there are three important differences in the reinforcement learning setting: 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:

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

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 1. To ensure adequate exploration, the agent's policy is ϵ-greedy with respect to the current estimate of the optimal action-value function.

Algorithm 1 Deep Q-learning with experience replay.

```
 1: Initialise replay memory $D$ to capacity $N$
 2: Initialise neural network $Q$ with random weights $\theta$ 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 $\epsilon$ 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 $\theta$
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 3.2.3 whereby zero reward is given for all states after the terminal state. If the task were instead continuing, line 11 would simply be $y_j = r_{j+1} + \gamma \max_{a'} \hat{Q}(s_{j+1}, a_j; \theta^-)$.

3.4 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 modelled reward.

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 scalar rewards, she is periodically presented with a pair of trajectories and must indicate which they prefer[2]. In a similar fashion to the above, we can model the human's preferences $\hat{r} : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$. Then, standard RL algorithms can be applied to maximise the expected (modelled) reward over time.

Each of these methods have shown promising results, and some of the advantages

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

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[3]	Well suited, since demonstrations can guide exploration	Limited[4]
Communication efficiency	On the order of hundreds of bits per human hour	Much richer in information than trajectory preferences [5]	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 [6]

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

Since this dissertation concerns reward learning from trajectory preferences specifically, the remainder of this section explains in more detail the algorithms used to do so. There are two branches of work on this question: that which learns a reward function based on handcrafted features of the environment state, and that which takes the deep learning approach of training a reward model end-to-end without handcrafted features.

[3] 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

[4] See footnote 3

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

[6] 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.

3.4.1 Reward Learning from Trajectory Preferences with Handcrafted Feature Transformations

Active Preference-Based Learning of Reward Functions

Batch Active Preference-Based Learning of Reward Functions

DemPref

3.4.2 Reward Learning from Trajectory Preferences in Deep RL

Setting

The agent-environment interface is as described in Section 3.2.1, 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 $((s_0, a_0), (s_1, a_1), \dots, (s_{k-1}, a_{k-1})) \in (\mathcal{S} \times \mathcal{A})^k$. 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[7], 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.

[7] 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.

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 $(\sigma^1, \sigma^2, \mu)$ 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:

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

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.

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

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).

Chapter 4

Uncertainty in Deep Learning

Standard deep learning models output point estimates. For example, a model trained to classify pictures of dogs according to their breed takes a picture of a dog and outputs its predicted breed. However, what will the model do if it is given a picture of a cat? [Gal, 2017].

We probably want the model to be able to recognise that this is an out of distribution example, and request more training data, or simply say that it doesn't know the answer. However, since standard deep learning models output only point estimates, the model will just go ahead and classify the cat as some breed of dog, just as confidently as any other input.

Thus, the field of Bayesian Deep Learning aims to equip neural networks with the ability to output a point estimate along with its uncertainty in that estimate. Historically, many of the attempts to do so were not very practical. For example, one algorithm, Bayes by Backprop, 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].

4.1 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.

4.2 Model Uncertainty in BNNs

Chapter 5

Active Learning

5.1 Acquisition Functions

5.1.1 Max Entropy

5.1.2 Variation Ratios

5.1.3 Mean STD

5.1.4 BALD

5.2 Applying Active Learning to RL without a reward function

Discussion of previous work.

5.2.1 APRIL

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

Active Preference-Based Learning of Reward Functions

Batch Active Preference-Based Learning of Reward Functions

DemPref

5.2.3 Deep RL from Human Preferences

Part II

Innovation

Chapter 6

Method

6.1 Possible failure modes of active reward modelling

Before developing our method, it was important to consider the possible failure modes of the previous attempt to apply active learning to reward modelling. [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 $a_{mean\_std}(\sigma^1, \sigma^2)$, which in this setting is written as:

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

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

The quality of uncertainty estimates and active learning method is one possible failure mode. Indeed, [Christiano et al., 2017, p. 6] explain the failure of their implementation as due to $a_{mean\_std}(\sigma^1, \sigma^2)$ “crude approximation” to reward predictor uncertainty.

Additionally, there are two features of the application of active learning to reward modelling that are not present in the standard supervised learning case. Firstly, the pool dataset is not known in advance, but is gathered as the RL agent encounters new states. Secondly, since the objects that we acquire are *clip pairs* rather than, for example, single images, yet the (growing) dataset is comprised of single clips, evaluating the acquisition function over the dataset is a complexity $O(n^2)$ operation, for n the number of clips acquired. This makes the standard active learning procedure of

There are other possible failure modes that apply to active learning in general. 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.

Anything else?

6.2 Applying acquisition functions to reward modelling

6.3 ARMA

In this section we present the training protocol we developed.

6.4 Acquisition Functions

6.5 Uncertainty Estimates

6.6 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,

Algorithm 2 ARMA: Active Reward Modelling for Agent Alignment.

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

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., 01], which is also built on top of Python. This gives finer control over data representation and sampling.

Chapter 7

Experimental Details

We used the environments provided by OpenAI Gym to run our experiments.

7.1 CartPole-v0

Chapter 8

Results

8.1 CartPole-v0

Chapter 9

Conclusions

9.1 Summary

9.2 Evaluation

9.3 Future Work

References

[Achiam, 2019] Achiam, J. (2019). Spinning Up in Deep RL.

[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.

[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*.

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

[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.

[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.

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

[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.

[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.

[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.

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

[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

Some Appendix Material