Deep Inverse Reinforcement Learning

Matthew Alger
The Australian National University

In this report, we describe "inverse reinforcement learning". This is the problem of finding an unknown reward function for a Markov decision process (MDP), given an optimal policy for that MDP. We also describe existing methods to solve this problem, and investigate a pre-existing deep learning extension to these methods.

Introduction

There are many situations where we may want to model or reliably replicate some natural behaviour. Examples of this include road network navigation by taxis[1], the foraging behaviour of bees[2], or even the specifics of driving along highways[3]. These behaviours involve trade-offs that may be arbitrarily complex: Roads may have many differently-featured routes to get to the same location, foraging animals must make trade-offs between various risks and resource gains, and driving on a highway balances proximity to other cars and road features with ease of controlling the car and overall speed. The exact trade-offs made in these scenarios are generally guessed and subsequently modified by researchers until they give behaviour matching reality[3], but even the results of this approach are still very much an assumption which may be incorrect or break down in different conditions[4].

Inverse reinforcement learning (IRL) provides a systematic way of formally stating and solving this problem. The observed behaviour is treated as if it were the solution to a reinforcement learning problem, and the problem reduces to recovering the associated reinforcement learning reward function.

We will first introduce our notation, summarise the general topic of reinforcement learning, and introduce inverse reinforcement learning as first described by Russell[4].

Preliminaries

In this section, we will summarise the notation used later in this report and define the key concepts required to discuss the inverse reinforcement learning problem.

In reinforcement learning, an agent acts in an environment. In this report we assume that environments are **Markov decision processes** (MDPs). Agents occupy a state of the environment and perform actions to change the state they are in. After taking an action, they are given some representation of the state they are now in and some reward value associated with the action and new state.

An MDP, formally, is a tuple $(\mathcal{S}, \mathcal{A}, \{\mathcal{P}^a_{ss'}\}, R, \gamma)$, where:

- $\mathcal{S}$ is a (possibly infinite) set of **states**; we will only consider finite $\mathcal{S}$ in this report.
- $\mathcal{A}$ is a set of **actions**.

- $\{\mathcal{P}^a_{ss'}\}$ is a set of **transition probabilities**, where $\mathcal{P}^a_{ss'}$ is the probability of transitioning from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ after taking action $a \in \mathcal{A}$[5].
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the **reward function**, which describes how much **reward** an agent should get by taking an action when the agent is in a given state of the MDP. Reward is some real-valued performance measure; **long-term reward** is generally the sum of rewards accumulated over a long period of time. We will only consider reward functions that are independent of action in this report, and only rewards with absolute value bounded by R_{max}.
- $\gamma \in [0, 1)$ is called the **discount factor**, and describes how much a given reward is worth one step into the future compared to getting the same reward now[5]. For example, if $\gamma = 0.5$, then getting a reward of 1 now is worth as much as getting a reward of 2 one step into the future.

We define for later convenience a matrix $\boldsymbol{P}^a$, where $(\boldsymbol{P}^a)_{ij} = \mathcal{P}^a_{s_i s_j}$. Similarly, since the reward is only a function of state, we define a vector $\boldsymbol{R}$, where $\boldsymbol{R}_i = R(s_i)$.

The behaviour of the agent in the MDP is described by a **policy**. A policy is a function $\pi : \mathcal{S} \to \mathcal{A}$, where $\pi(s)$ gives the action the agent would take in state s. An **optimal policy** is a policy that maximises expected long-term reward for an agent following that policy. All optimal policies are denoted π^*[5]. For a stochastic policy, we may also write $\pi(s, a)$ as the probability of taking action a in state s under the policy.

A **path** taken through an MDP by an agent is denoted ζ, and is an ordered list of state and action tuples $\zeta = [(s_1, a_1), (s_2, a_2), \dots]$. Since the MDP may be stochastic depending on $\mathcal{P}^a_{ss'}$, there may be many different paths generated by the same policy. We generalise the reward function to act on paths by defining

$$R(\zeta) = \sum_{(s_i, a_i) \in \zeta} R(s_i) \tag{1}$$

The probability of taking a path according to $\{\mathcal{P}^a_{ss'}\}$ is denoted $P(\zeta)$; dependence on $\{\mathcal{P}^a_{ss'}\}$ is assumed.

We represent states as either integers or **feature vectors**. Let $\boldsymbol{\phi} : \mathcal{S} \to \mathbb{R}^D$, where D is some natural number. This is called the **feature map**; the feature vector representing state s is then $\boldsymbol{\phi}(s)$. It is not, in general, injective or surjective. D is the dimensionality of the feature space. We define the **feature counts** of a path ζ as the sum of feature vectors of the states visited along that path:

$$\boldsymbol{\phi}_\zeta = \sum_{(s_i, a_i) \in \zeta} \boldsymbol{\phi}(s_i) \tag{2}$$

We also define the **feature expectations** as the average feature counts over all paths:

$$\tilde{\boldsymbol{\phi}} = \sum_{\text{all paths } \zeta} P(\zeta) \boldsymbol{\phi}_\zeta \tag{3}$$

Reinforcement learning

The problem of reinforcement learning is now simple to state: For an agent in an environment with unknown transition probabilities, find, through trial and error, the optimal policy[5]. In this section, we describe some background and key results of reinforcement learning.

We first define the **value function** $V^\pi : \mathcal{S} \to \mathbb{R}$ as

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i R(s_{t+i+1}) \,\middle|\, s_t = s \right] \tag{4}$$

where s_t is the current state, s_{t+1} is the next state visited under π, and so on[5]. The value function $V^\pi(s)$ is the total discounted long-term reward expected to be obtained by an agent following the policy π and starting in state s. Informally, $V^\pi(s)$ describes how "good" it is for an agent to be in the state s if the agent follows the policy π. As with reward, we define for convenience a vector $\boldsymbol{V}^\pi$, where $(\boldsymbol{V}^\pi)_i = V^\pi(s_i)$.

For any policy π and state s, $V^\pi(s)$ satisfies a recursive relationship called the **Bellman equation for $\boldsymbol{V^\pi}$**[5]:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[R(s') + \gamma V^\pi(s') \right] \tag{5}$$

V^π uniquely solves the Bellman equation, so we can use the Bellman equation to find V^π.

The **action–value function** $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is similarly defined as a measure of discounted long-term reward expected to be obtained by taking an action in a state and then following π thereafter[5]:

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[\sum_{i=0}^{\infty} \gamma^i R(s_{t+i+1}) \,\middle|\, s_t = s, a_t = a \right] \tag{6}$$

Q^π and V^π are related by

$$Q^\pi(s, a) = \mathbb{E} \left[R(s_{t+1}) + \gamma V^\pi(s_{t+1}) \,\middle|\, s_t = s, a_t = a \right] \tag{7}$$

We can now formally restate what an optimal policy is. A policy π is optimal iff

$$\forall s \in \mathcal{S}. \quad \forall a \in \mathcal{A} \setminus \pi(s). \quad Q^\pi(s, \pi(s)) \geq Q^\pi(s, a) \tag{8}$$

There are many algorithms for finding such policies. There are two main classes of policy-finding algorithms: **Model-based**, and **model-free**. A model-based algorithm either knows the transition probabilities of the MDP, or forms some explicit approximation of them from observed trajectories. A model-free algorithm does not.

In the experiments covered in this report, we use the **value iteration** algorithm. This is a model-based algorithm that relies on knowledge of the transition probabilities. It estimates the value function for each state; the policy in any given state is then whichever action is expected to lead to the highest-valued state. The value iteration algorithm is

reproduced here as algorithm 1.

Algorithm 1: Value iteration[5].

Input: $\mathcal{P}^a_{ss'}, R(s), \mathcal{S}, \mathcal{A}, \gamma$
foreach $s \in \mathcal{S}$ **do**
  $V(s) \leftarrow 0$
end
$\Delta \leftarrow \infty$;
while $\Delta > \epsilon$ **do**
  $\Delta \leftarrow 0$;
  foreach $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$;
    $V(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}(R(s') + \gamma V(s'))$;
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
  end
end
foreach $s \in \mathcal{S}$ **do**
  $\pi(s) \leftarrow \operatorname{argmax}_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'}(R(s') + \gamma V(s'))$;
end
return π

An example of a model-free algorithm is **SARSA**, which learns an optimal policy by repeatedly estimating Q^π and then modifying π to be more optimal[5]. We do not discuss SARSA or other policy-finding algorithms here.

Inverse reinforcement learning

Inverse reinforcement learning (IRL) is the problem of finding the environment's reward function given observations of the behaviour of an optimally-behaving agent, i.e., given either an optimal policy π^* or sample paths of an agent following π^*[4]. This is motivated by two main ideas.

Firstly, we may want to replicate behaviour of some entity by using well-tested reinforcement learning methods, but a reward function suited to this task may be arbitrarily complex, abstract, or difficult to describe. An example of this is the problem of driving a car; to quote Abbeel & Ng's 2004 paper[3] on IRL:

> Despite being able to drive competently, the authors do not believe they can confidently specify a specific reward function for the task of "driving well".

By finding a reward function, we can then train an agent to find a policy based on that reward function and hence replicate the original behaviour. This kind of problem is called **apprenticeship learning**. By recovering a reward function and learning a policy from it, rather than directly learning a policy from behaviour (e.g. by function approximation of π^*), we may be able to find more robust policies that can adapt to perturbations of the environment[3, 6]. We may also be able to solve the related problem of **transfer learning**, where the abstract goal the agent is trying to achieve is similar, but the specifics of the environment differ; a policy learned directly from another environment's optimal policy will probably not be successful in the new environment[7].

Secondly, we may want to know the reward function itself, to explain the behaviour of an existing agent. In the example of modelling the behaviour of bees[2], there is no real interest in recovering the behaviour of the bees, but instead in modelling their motivations.

IRL as stated is not a problem without issues. The biggest issue is that of reward function multiplicity: For any given policy π, there are many reward functions for

which π is optimal. As an example, for the trivial reward function $R(s) = const$, every policy is optimal[8]. In practice, degeneracy in reward functions is resolved with use of heuristics to favour specific reward functions over trivial ones, but even this only partially resolves the problem as there can still be many non-trivial reward functions matching the observed policy.

Review of IRL Algorithms

In this section, we review existing IRL algorithms by Ng & Russell[8], and Ziebart et al.[1], and summarise their derivations and methods.

Linear programming formulation

Ng & Russell[8] introduced the first algorithms to solve the IRL problem. These algorithms work differently for small and large state spaces; we begin by looking at the small state space approach. This approach explicitly requires both the policy π^* and the transition probabilities $\{\mathcal{P}^a_{ss'}\}$.

By writing the Bellman equation (equation 5) in matrix form, we can explicitly solve for $\boldsymbol{V}^\pi$. Writing the action chosen by π^* in any given state as a^*, we find

$$\boldsymbol{V}^{\pi^*} = (\boldsymbol{I} - \gamma \boldsymbol{P}^{a^*})^{-1} \boldsymbol{R} \tag{9}$$

We can also rewrite the condition for a policy to be optimal (equation 8) in terms of the value function.

$$\forall a \in \mathcal{A} \setminus a^*. \quad \boldsymbol{P}^{a^*} \boldsymbol{V}^{\pi^*} \geq \boldsymbol{P}^a \boldsymbol{V}^{\pi^*} \tag{10}$$

Substituting equation 9, we find

$$\forall a \in (\mathcal{A} \setminus a^*). \quad (\boldsymbol{P}^{a^*} - \boldsymbol{P}^a)(\boldsymbol{I} - \gamma \boldsymbol{P}^{a^*})^{-1} \boldsymbol{R} \geq \boldsymbol{0} \tag{11}$$

This is a key result of the paper: The reward function R is constrained by the actions chosen by π^*. This constraint, then, can in principle be solved for $\boldsymbol{R}$. The degeneracy problem is immediately apparent, since $\boldsymbol{R} = const$ is a solution no matter what MDP we have. A further constraint is thus introduced to reduce the number of solutions. We choose the solution to equation 11 that maximises

$$\sum_{s \in \mathcal{S}} \left(Q^{\pi^*}(s, a^*) - \max_{a \in (\mathcal{A} \setminus a^*)} Q^{\pi^*}(s, a) \right) \tag{12}$$

The intuitive meaning of equation 12 is that deviating from the optimal policy should reduce the total reward as much as possible. We also include an L_1 regularisation term to favour "simpler" solutions over "complicated" ones; with L_1 regularisation we expect sparse solutions for $\boldsymbol{R}$. We can write this as a linear programming problem, which can then be solved with well-known linear programming methods:

$$\text{maximise} \sum_{i=1}^{|\mathcal{S}|} \min_{a \in (\mathcal{A} \setminus a^*)} ((\boldsymbol{P}^{a^*})_i - (\boldsymbol{P}^a)_i)(\boldsymbol{I} - \gamma \boldsymbol{P}^{a^*})^{-1} \cdot \boldsymbol{R} - \lambda ||R||_1 \tag{13}$$

$$\text{s.t. } \forall i \in 1, \ldots, |\mathcal{S}|. \forall a \in (\mathcal{A} \setminus a^*). -((\boldsymbol{P}^{a^*})_i - (\boldsymbol{P}^a)_i)(\boldsymbol{I} - \gamma \boldsymbol{P}^{a^*})^{-1} \cdot \boldsymbol{R} \leq 0 \tag{14}$$

$$\text{and } \forall i \in 1, \ldots, |\mathcal{S}|. |R_i| \leq R_{\max} \tag{15}$$

This can be written as a set of matrix equations. Taking $\boldsymbol{M}$ and $\boldsymbol{u}$ as dummy vectors, the problem in block matrix form becomes:

$$\text{maximise } \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \\ -\lambda\mathbf{1} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{M} \\ \boldsymbol{u} \end{bmatrix} \tag{16}$$

$$\text{s.t. } \begin{bmatrix} -(\boldsymbol{P}^{a^*} - \boldsymbol{P}^a)(\boldsymbol{I} - \gamma\boldsymbol{P}^{a^*})^{-1} & \boldsymbol{I} & \mathbf{0} \\ -(\boldsymbol{P}^{a^*} - \boldsymbol{P}^a)(\boldsymbol{I} - \gamma\boldsymbol{P}^{a^*})^{-1} & \mathbf{0} & \mathbf{0} \\ -\boldsymbol{I} & \mathbf{0} & -\boldsymbol{I} \\ \boldsymbol{I} & \mathbf{0} & -\boldsymbol{I} \end{bmatrix} \cdot \begin{bmatrix} \boldsymbol{R} \\ \boldsymbol{M} \\ \boldsymbol{u} \end{bmatrix} \leq 0 \tag{17}$$

In larger state spaces, this formulation becomes intractable. This can be resolved this by approximating the reward function as a linear combination of feature vectors of the states:

$$R(s) = \boldsymbol{\alpha} \cdot \boldsymbol{\phi}(s) \tag{18}$$

The value function for the optimal policy with this reward function is linear in $\boldsymbol{\alpha}$:

$$V^{\pi^*}(s) = \sum_i \alpha_i V_i^{\pi^*}(s) \tag{19}$$

where V_i^π is the value function for the policy π when $R(s) = \alpha_i \phi_i(s)$. We thus have a generalisation of equation 11 for this approximation:

$$\forall s \in \mathcal{S}. \forall a \in (\mathcal{A} \setminus a^*). \mathop{\mathbb{E}}_{s' \sim \mathcal{P}^{a^*}_{ss'}} \left[V^{\pi^*}(s') \right] \geq \mathop{\mathbb{E}}_{s' \sim \mathcal{P}^{a}_{ss'}} \left[V^{\pi^*}(s') \right] \tag{20}$$

where $\mathbb{E}_{s' \sim \mathcal{P}^a_{ss'}}$ represents an expectation value over states s' weighted by the probability of transitioning to s'. Since V^{π^*} is linear in $\boldsymbol{\alpha}$, we can interpret this as a set of constraints on $\boldsymbol{\alpha}$. In practice, since we are dealing with large state spaces, we have a very large number of these constraints. To rectify this, we only consider constraints on states from a sample $S_0 \subset \mathcal{S}$.

This results in another linear programming formulation for large state spaces:

$$\text{maximise } \sum_{s \in \mathcal{S}} \min_{a \in (\mathcal{A} \setminus a^*)} p\left(\mathop{\mathbb{E}}_{s' \sim \mathcal{P}^{a^*}_{ss'}} \left[V^{\pi^*}(s') \right] - \mathop{\mathbb{E}}_{s' \sim \mathcal{P}^{a}_{ss'}} \left[V^{\pi^*}(s') \right] \right) \tag{21}$$

$$\text{s.t. } \forall i \in 1, \ldots, D. |\alpha_i| \leq 1 \tag{22}$$

where

$$p(x) = \begin{cases} x, & x \geq 0 \\ mx, & x < 0 \end{cases} \tag{23}$$

Since the reward is now linearly approximated, it may no longer be possible to find a non-trivial reward function that makes the observed policy optimal. To get around this, we relax the optimality condition by introducing the function $p(x)$. $p(x)$ represents a penalty for suboptimal actions leading to higher value states than optimal actions would under our recovered reward function. This penalty lets us change our strict optimality constraint into a relaxed constraint, where violating optimality increases the value we are trying to minimise by some dynamic amount mx. m is how strongly this penalty is enforced. Ng & Russell found empirically that $m = 2$ was an effective penalty coefficient[8].

The nonlinear functions p and min can be eliminated by adding additional variables. We can once again write this as a set of matrix equations. Taking $\boldsymbol{z}$, $\boldsymbol{y}$, and $\boldsymbol{x}$ as dummy vectors, the block matrix form is

$$\text{maximise} \quad \begin{bmatrix} \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}^T \begin{bmatrix} \boldsymbol{z} \\ \boldsymbol{y} \\ \boldsymbol{x} \\ \boldsymbol{\alpha} \end{bmatrix} \tag{24}$$

$$\text{s.t.} \begin{bmatrix} \mathbf{0} & \boldsymbol{I}_j & -\boldsymbol{I}_j & -\boldsymbol{v}_{ij}^T \end{bmatrix} \begin{bmatrix} \boldsymbol{z} \\ \boldsymbol{y} \\ \boldsymbol{x} \\ \boldsymbol{\alpha} \end{bmatrix} = \mathbf{0} \tag{25}$$

$$\text{and} \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{I} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & -\boldsymbol{I} \\ \mathbf{0} & -\boldsymbol{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & -\boldsymbol{I} & \mathbf{0} \\ \mathbf{1}\boldsymbol{I}_l^T & -\delta_{il}\boldsymbol{I} & 2\delta_{il}\boldsymbol{I} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{z} \\ \boldsymbol{y} \\ \boldsymbol{x} \\ \boldsymbol{\alpha} \end{bmatrix} \leq \begin{bmatrix} \mathbf{1} \\ \mathbf{1} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \tag{26}$$

where

$$(\boldsymbol{v}_{ij})_k = \mathop{\mathbb{E}}_{s' \sim \mathcal{P}^{a^*}_{ss'}} \left[V_k^{\pi^*}(s') \right] - \mathop{\mathbb{E}}_{s' \sim \mathcal{P}^{a}_{ss'}} \left[V_k^{\pi^*}(s') \right] \tag{27}$$

The final linear programming formulation in Ng & Russell's paper describes IRL from sampled paths, with no explicit π^* or $\{\mathcal{P}^a_{ss'}\}$, though it does require the ability to run arbitrary policies through the MDP. This is effectively an approximation to the previous formulation of the problem, with slight changes to the eventual linear programming formulation, so we do not go into detail on this here.

Maximum entropy formulation

Ziebart et al.[1] build on Abbeel & Ng's[3] approach to removing multiplicity in the possible reward functions. They introduced a method of matching feature expectations between observed paths and optimal paths for recovered reward functions. Intuitively, if we generate a policy which is optimal for our recovered reward function, we would expect that on average it generates the same paths as the optimal policy for the true reward function. This approach thus recovers reward functions that can be learned from using standard reinforcement learning methods to recover policies similar to the optimal policy. The observed feature expectations from our N observed trajectories is a simple average over feature counts.

$$\tilde{\phi} \approx \tilde{\phi}_{\text{obs}} = \frac{1}{N} \sum_{i=1}^{N} \phi_{\zeta_i} \tag{28}$$

Ziebart et al. extend this by using the principle of maximum entropy. First, they restate the problem probabilistically. Consider a distribution over all possible paths through the MDP. Due to the stochastic nature of the environment, there may be many paths which match feature expectations, and these paths may be constrained in ways that are not implied by the feature expectations[1]. To fix this problem, we use the maximum entropy distribution over paths, which has the least additional constraints other than what we know from our feature expectation matching. The maximum entropy distribution is described in more detail by Jaynes[9]. The result of this is that paths yielding higher total reward should be exponentially more likely to be chosen, i.e.:

$$P(\zeta) = \frac{1}{Z} \exp(R(\zeta)) \tag{29}$$

where Z is called the **partition function** in analogy with statistical physics and is a normalisation constant obtained by summing $\exp(R(\zeta))$ over all paths.

One key benefit of this probabilistic approach is that we implicitly handle the uncertainty and noise in our observed paths through the MDP, potentially leading to obtaining clearer or more robust reward functions.

For a non-deterministic MDP, the maximum entropy distribution is intractable, but under the assumptions that the partition function is constant and that the stochasticity in the MDP has little effect on long-term behaviour of the policy, there is a tractable approximation:

$$P(\zeta \mid \boldsymbol{\alpha}) \approx \frac{\exp(\boldsymbol{\alpha} \cdot \boldsymbol{\phi}_\zeta)}{Z(\boldsymbol{\alpha})} \prod_{(s_t, a_t), (s_{t+1}, a_{t+1}) \in \zeta} \mathcal{P}^{a_t}_{s_t s_{t+1}} \tag{30}$$

This gives a stochastic policy, where we choose an action with a probability proportional to the sum of all probabilities of taking paths that begin with that action, i.e.,

$$P(a \mid \boldsymbol{\alpha}) \propto \sum_{\zeta \text{ s.t. } a \in \zeta_1} P(\zeta \mid \boldsymbol{\alpha}) \tag{31}$$

To find $\boldsymbol{\alpha}$, we want to maximise the log-likelihood of observing the observed paths under this distribution. This is an optimisation problem, with the optimal $\boldsymbol{\alpha}$ given by

$$\boldsymbol{\alpha}^* = \operatorname*{argmax}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = \operatorname*{argmax}_{\boldsymbol{\alpha}} \sum_{i=1}^{N} \log P(\zeta_i \mid \boldsymbol{\alpha}) \tag{32}$$

Ziebart et al. assume that the reward function is linear in the states (as in the linear programming approach, equation 18) and hence find the gradient of $L(\boldsymbol{\alpha})$ to be

$$\frac{\partial L}{\partial \boldsymbol{\alpha}} = \tilde{\boldsymbol{\phi}}_{\text{obs}} - \sum_{i=1}^{N} P(\zeta_i \mid \boldsymbol{\alpha}) \boldsymbol{\phi}_{\zeta_i} = \tilde{\boldsymbol{\phi}}_{\text{obs}} - \sum_{s \in \mathcal{S}} D(s) \boldsymbol{\phi}(s) \tag{33}$$

where $D(s)$ is called the **expected state visitation frequency** of state s and represents the probability of being in a given state. $D(s)$ can be efficiently computed using a dynamic programming algorithm which takes transition probabilities and a reward function, and returns $D(s)$. The transition probabilities are ideally given, but can be approximated by observed trajectories. This algorithm has been reproduced here as algorithm 2. $\boldsymbol{\alpha}$ can then be found by standard gradient descent methods, where $D(s)$ is recalculated using algorithm 2 after each gradient descent step based on the new value of $\boldsymbol{\alpha}$.

Algorithm 2: Dynamic programming algorithm for finding D[1].

Input: $\{\mathcal{P}^a_{ss'}\}, R(s), \mathcal{S}, \mathcal{A}$
Output: $D(s)$
$Z_{s_{\text{terminal}}} \leftarrow 1$;
loop N **times**
  foreach $s \in \mathcal{S}$ **do**
    foreach $a \in \mathcal{A}$ **do**
      $Z_{sa} \leftarrow \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} Z_{s'} \exp R(s)$;
    end
  end
  foreach $s \in \mathcal{S}$ **do**
    $\sum_{a \in \mathcal{A}} Z_{sa} + \mathbf{1}_{s=s_{\text{terminal}}}$;
  end
endloop
foreach $s \in \mathcal{S}$ **do**
  foreach $a \in \mathcal{A}$ **do**
    $P(a \mid s) \leftarrow Z_{sa}/Z_s$;
  end
  $D_0(s) \leftarrow P(s = s_{\text{initial}})$;
end
foreach $t \in 1, \ldots, N$ **do**
  foreach $s \in \mathcal{S}$ **do**
    $D_t(s) \leftarrow \sum_{s' \in \mathcal{S}} \sum_{a \in \mathcal{A}} D_{t-1}(s') \mathcal{P}^a_{s's} P(a \mid s')$;
  end
end
foreach $s \in \mathcal{S}$ **do**
  $D(s) \leftarrow \sum_{t=0}^{N} D_t(s)$;
end
return $D(s)$

Deep maximum entropy formulation

The maximum entropy formulation can be simply extended to use deep learning. This extension has also been derived by Wulfmeier et al.[10]

In the maximum entropy formulation, we assumed that the reward function is a linear combination of feature vectors, as in equation 18.

$$R(s) = \boldsymbol{\alpha} \cdot \boldsymbol{\phi}(s)$$

This has a very obvious limitation: It might not be possible to accurately approximate the reward function as a linear combination of these feature vectors in the given basis $\boldsymbol{\phi}$. To remedy this, we can approximate the *basis functions* as non-linearly transformed linear combinations:

$$R(s) = \boldsymbol{\alpha} \cdot \boldsymbol{\varphi}(s) \tag{34}$$
$$\boldsymbol{\varphi}(s) = \sigma(\boldsymbol{W} \cdot \boldsymbol{\phi}(s)) \tag{35}$$

Here, $\sigma(\boldsymbol{x})$ is the sigmoid (logistic) function[1] (equation 36), applied elementwise to the

[1]We can use any non-linear function instead of sigmoid. In this context the function we use is called the

vector $\boldsymbol{x}$.

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{36}$$

This concept can be extended to arbitrary depth.

$$R(s) = \boldsymbol{\alpha} \cdot \boldsymbol{\varphi}_n(s) \tag{37}$$

$$\boldsymbol{\varphi}_n(s) = \sigma(\boldsymbol{W}_n \cdot \boldsymbol{\varphi}_{n-1}(s)) \tag{38}$$

$$\vdots$$

$$\boldsymbol{\varphi}_1(s) = \sigma(\boldsymbol{W}_1 \cdot \boldsymbol{\phi}(s)) \tag{39}$$

This structure is called a **neural network**, and when $n > 1$ we call it a **deep neural network**[12]. By choosing suitable values for the parameters $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_n$, we can approximate any continuous feature map[13, 11] and hence, in principle, any reward function.

We can still find the $\boldsymbol{\alpha}$ that minimises loss L with gradient descent, by using a variation of equation 33. This variation is shown in equation 40; $\tilde{\boldsymbol{\varphi}}_{\text{obs},n}$ is the observed feature expectations evaluated with $\boldsymbol{\varphi}_n$ as the feature map.

$$\partial_{\boldsymbol{\alpha}} L = \tilde{\boldsymbol{\varphi}}_{\text{obs},n} - \sum_{s \in \mathcal{S}} D(s) \boldsymbol{\varphi}_n(s) \tag{40}$$

Define $D_{\text{obs}}(s)$ as the **observed state visitation frequency** of state s; $D_{\text{obs}}(s)$ is the number of times the state s was visited in the observed trajectories, divided by the total number of states visited. We can then rewrite the observed feature expectations as $\sum_{s \in \mathcal{S}} D_{\text{obs}}(s) \boldsymbol{\varphi}_n(s)$ and hence we can rewrite $\partial_{\boldsymbol{\alpha}} L$ as

$$\partial_{\boldsymbol{\alpha}} L = \sum_{s \in \mathcal{S}} (D_{\text{obs}}(s) - D(s)) \boldsymbol{\varphi}_n(s) \tag{41}$$

In this form, the connection between the derivative with respect to $\boldsymbol{\alpha}$ and the original equation becomes clear. We can now easily find gradients of L with respect to $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_n$. Generalising the Ziebart et al. gradient for $\boldsymbol{\alpha}$ (equation 41), we find the gradient with respect to $\boldsymbol{W}_i$ to be

$$\partial_{\boldsymbol{W}_i} L = \sum_{s \in \mathcal{S}} \frac{\partial L}{\partial R(s)} \frac{\partial R(s)}{\partial \boldsymbol{W}_i} \tag{42}$$

$$= \sum_{s \in \mathcal{S}} (D_{\text{obs}}(s) - D(s)) \partial_{\boldsymbol{W}_i} R(s) \tag{43}$$

The gradients $\partial_{\boldsymbol{W}_i} R(s)$ are standard neural network derivatives, so all gradients can be calculated using the backpropagation algorithm[14]. From the gradients, we can find good values for $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_n$ and $\boldsymbol{\alpha}$ using standard gradient descent methods, where $D(s)$ is recalculated using algorithm 2 after each gradient descent step. The reward function input to algorithm 2 is given by equations 37 – 39 using the new values of $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_n$ and $\boldsymbol{\alpha}$.

This approach has recently been tested by Wulfmeier et al.[10] and they obtain good results. We attempt here to replicate these results on an MDP with reward non-linear in the state vectors, using the above equations.

activation function in analogy with biological neurons. We choose sigmoid as it is differentiable and its derivative can be written in terms of itself. These properties make the process of finding optimal parameters $\boldsymbol{W}$ and $\boldsymbol{\alpha}$ more efficient. An additional constraint on the activation function is that it should be non-polynomial; polynomial activation functions lead to equations that *cannot* approximate arbitrary continuous functions[11].

Experiments and Results

We loosely followed the experimental method of Wulfmeier et al.[10] and Levine et al.[7] and compared the maximum entropy IRL algorithm (MaxEnt) to the deep maximum entropy IRL algorithm (DeepMaxEnt) on the objectworld MDP, and compared our results to the results presented in the Wulfmeier et al. paper. We set $\gamma = 0.9$ and used a learning rate of $\xi = 0.01$. The deep maximum entropy network consisted of two hidden layers of dimension 3; Wulfmeier et al. found this to be a good configuration of the network[10]. We also included bias vectors on the inner layers. All layers of the network were initialised with random normal values; bias vectors were initialised as zero. The equations describing the maximum entropy reward and deep maximum entropy reward are equation 44 and equation 45 respectively, where $\boldsymbol{\phi}_{\text{cts}}$ is the matrix with column i equal to $\boldsymbol{\phi}_{\text{cts}}(s_i)$, $\boldsymbol{b}_i$ are bias row vectors, and $\boldsymbol{W}_i$ are weight matrices. Gradient descent was performed using AdaGrad[15]. The expected state visitation frequency $D(s)$ was recalculated based on the current reward function at each iteration step using algorithm 2, and this was then used to calculate the gradient as described in equation 43. The reward was z-scored before each calculation of $D(s)$ to prevent the exponentiation of the reward function in algorithm 2 from overflowing.

$$\boldsymbol{R}_{\text{ME}} = \boldsymbol{\alpha}^T \boldsymbol{\phi}_{\text{cts}} \tag{44}$$

$$\boldsymbol{R}_{\text{DME}} = \boldsymbol{\alpha}^T \sigma(\boldsymbol{b}_2 + \boldsymbol{W}_2 \sigma(\boldsymbol{b}_1 + \boldsymbol{W}_1 \boldsymbol{\phi}_{\text{cts}})) \tag{45}$$

Both the maximum entropy tests and the deep maximum entropy tests were performed with the same code framework. This is possible since the maximum entropy algorithm is a special case of the deep maximum entropy framework with no hidden layers. By using the same code for both algorithms, we hope to eliminate systematic errors in the implementation which would bias results toward or against either algorithm. The implementation is available on GitHub [2].

We evaluated the performance of each algorithm by using the expected value difference (EVD) score[7]. This is computed by finding the optimal policy under the reward function recovered by each algorithm and then finding the expected total discounted reward for this policy under the *original* reward function. The expected value difference is then given by the difference between this value and the expected total discounted reward for the optimal policy, averaged over initial states.

Evaluation MDP

Experiments were run on the **objectworld** MDP[7], which has states that are cells in an $N \times N$ grid. An agent can take five actions in each state: up, down, left, right, or stay. Each action corresponds to movement in the given direction (or no movement at all for stay), but has a 30% chance of moving in a random direction instead. If the agent tries to move off the grid, it remains in the same state. Agents start in random states.

The grid contains some number of "objects", distributed at random through the grid. Each object has a position on the grid, an inner colour, and an outer colour; there are C colours. The reward of a state is 1 if the state is both within 3 cells of an outer red object and within 2 cells of an outer green object, -1 if the state is within 3 cells of an outer red object, and 0 otherwise. The inner colours and any colours other than red or green are unrelated to the reward and serve as distractors.

[2] `https://github.com/MatthewJA/Inverse-Reinforcement-Learning`

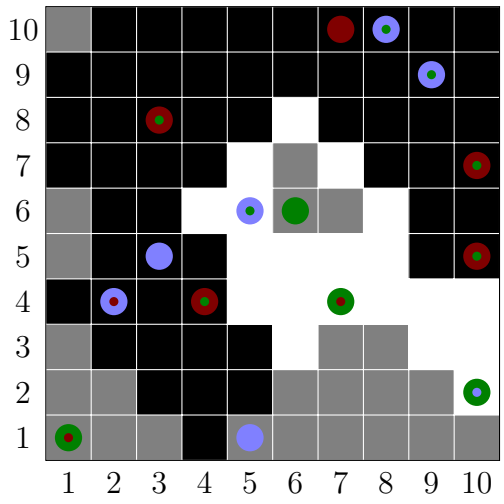


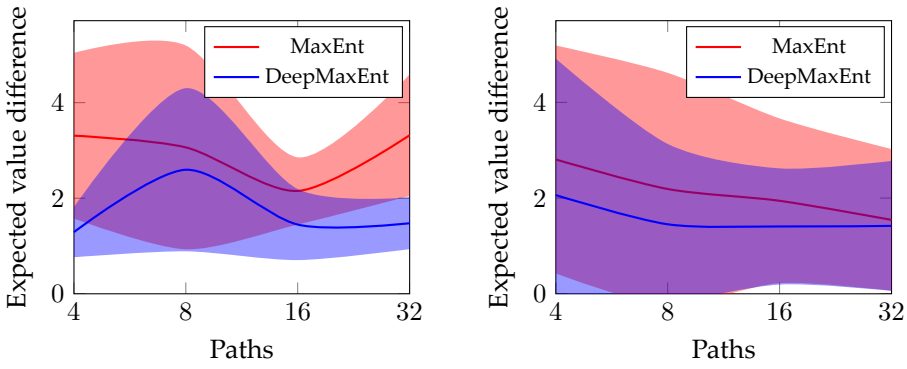
Figure 1: Example 10×10 objectworld. Cells coloured grey have reward 0, cells coloured white have reward 1, and cells coloured black have reward -1. Objects are represented by circles with their inner and outer colours represented as the colour of the inner circle and the colour of the outer circle respectively.

Two different feature maps can be used represent the states: a continuous map, and a discrete map. The continous map, ϕ_{cts}, gives $2C$-dimensional real-valued feature vectors. The elements of the feature vectors are distances to objects: The first element is the distance to the nearest inner red object, the second element is the distance to the nearest outer red object, the third element is the distance to the nearest inner green object, and so on through all colours. The discrete map, ϕ_{dsc}, gives $2CN$-dimensional binary-valued feature vectors. For each continuous feature element, there are N discrete elements, with the dth element being 1 if the corresponding continuous feature element is less than d. We only used the continuous feature map in our experiments.

Results

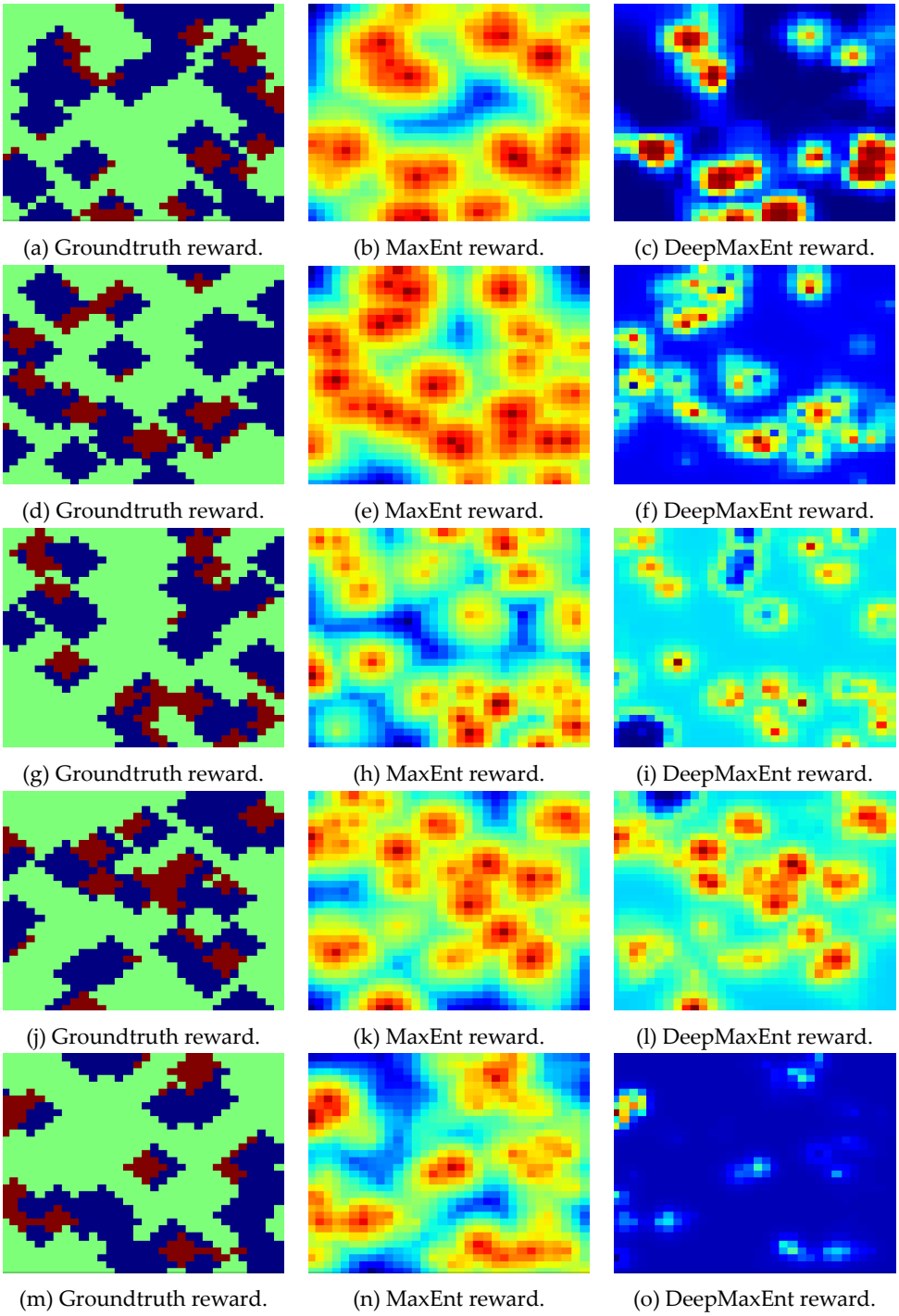
We tested the maximum entropy and deep maximum entropy algorithms on 32×32 objectworlds with different numbers of sampled paths. 50 objects were placed randomly on the grid. Two colours were used. Each path sampled as input to the IRL algorithms was of length 8. Each trial was repeated five times. A similar test was run on 16×16 objectworlds. Since these MDPs are smaller, the algorithms tested were much faster, so more trials could be run. All parameters were the same as for the 32×32 tests, but each trial was repeated 40 times. The results for both tests are plotted in figure 2.

Plots of the 32×32, 16 path trials are shown in figure 3 for illustrative purposes. Additionally, the recovered reward and value functions for two specific examples of the 16 path trials are shown in figure 4; these were the trials where deep maximum entropy achieved the best and worst results. Figures 4a – 4f are from the trial where deep maximum entropy maximally outperformed maximum entropy (EVD 0.74 and EVD 1.95 respectively), and 4g – 4l are from the only trial where maximum entropy outperformed deep maximum entropy (EVD 1.96 and EVD 2.62 respectively).



(a) 32×32 objectworld (b) 16×16 objectworld

Figure 2: Mean expected value difference for different numbers of sampled paths on different sized objectworlds. Error bars represent standard deviations.



(a) Groundtruth reward. (b) MaxEnt reward. (c) DeepMaxEnt reward.

(d) Groundtruth reward. (e) MaxEnt reward. (f) DeepMaxEnt reward.

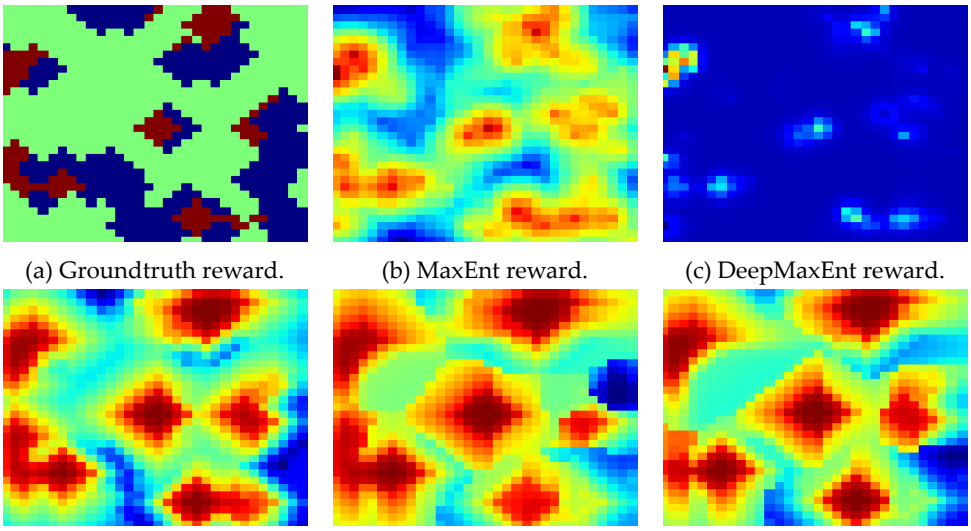
(g) Groundtruth reward. (h) MaxEnt reward. (i) DeepMaxEnt reward.

(j) Groundtruth reward. (k) MaxEnt reward. (l) DeepMaxEnt reward.

(m) Groundtruth reward. (n) MaxEnt reward. (o) DeepMaxEnt reward.

Figure 3: Groundtruth and recovered reward functions from the 32×32, 16 path trials. Each row represents one trial. Red is the most positive reward, and blue is the most negative reward, with other hues representing intermediate rewards.

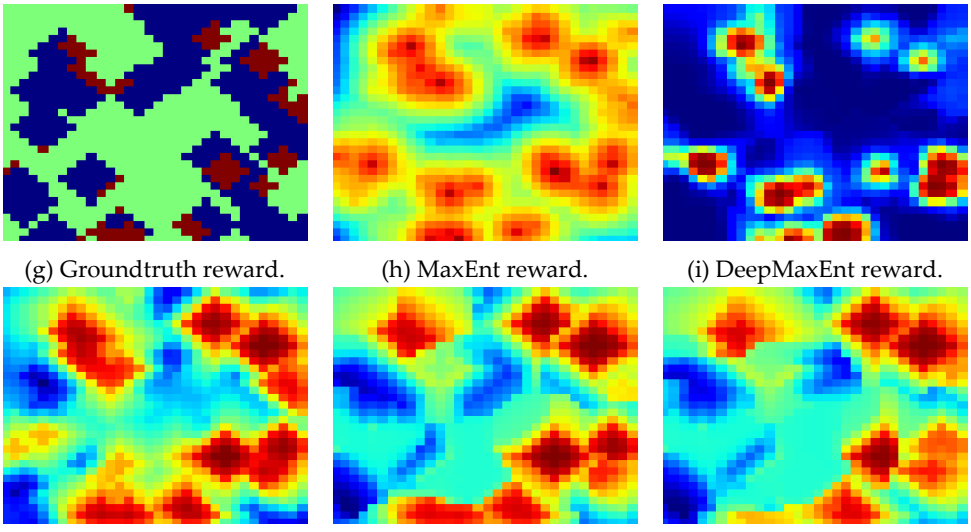
Best:



(a) Groundtruth reward. (b) MaxEnt reward. (c) DeepMaxEnt reward.

(d) Groundtruth value. (e) MaxEnt value. (f) DeepMaxEnt value.

Worst:



(g) Groundtruth reward. (h) MaxEnt reward. (i) DeepMaxEnt reward.

(j) Groundtruth value. (k) MaxEnt value. (l) DeepMaxEnt value.

Figure 4: Groundtruth and recovered reward and value functions from the best- and worst-performing 32×32, 16 path trials. Each row represents one trial. Red is the most positive reward, and blue is the most negative reward, with other hues representing intermediate rewards; a similar scale applies for the value function.

Discussion

The most striking property of the reward functions recovered by the deep maximum entropy algorithm (figure 3) is that they are much sharper than those recovered by the maximum entropy algorithm. Having sharp peaks in recovered reward functions makes sense within the context of maximum entropy: The simplest reward function that gives policies matching feature expectations are point sources of reward located in states that the optimal policy aims to occupy. In the objectworld case, the most occupied states are those that are maximally far away from low reward areas, i.e., the centre of positive reward areas. This is because of the 30% random action chance — being moved at random from the centre of these areas will usually put the agent in an area with the same positive reward, whereas being moved away from the edge of these areas has a higher chance of moving the agent to a region of lower reward. Having peaks in the reward function at the centre of these regions will reproduce policies that aim to get to and stay in the centres, and so the recovered reward functions should reproduce policies similar to the true optimal policies.

Indeed, both linear and deep approaches have spikes, but the maximum entropy algorithm "spreads out" the reward over larger regions. This is likely due to the fact it is a linear approximation, so the reward is constrained to change smoothly between regions, whereas in the deep maximum entropy approach, non-linearity allows the reward function to drop off steeply.

Another interesting difference between the reward functions recovered by maximum entropy and those recovered by deep maximum entropy is that while the maximum entropy reward functions vary fairly evenly over the range of possible rewards, the deep maximum entropy reward functions are mostly flat with sharp negative and positive peaks. This, again, may be a result of the heuristic employed of matching feature expectations, which aims only to recover policies similar to the optimal policy, but that deep maximum entropy returns reward functions that are the same in areas of zero and negative true reward is unexpected.

From the expected value differences (figure 2), we can see that the deep maximum entropy approach performs somewhat better than the maximum entropy approach. We would expect that the deep maximum entropy approach would at minimum be equally good, since it reduces to the maximum entropy approach, and this is mostly confirmed by the consistent better performance of deep maximum entropy. In some trials, deep maximum entropy did perform worse, though not by a very large amount. We can examine figure 4 to see a specific case where deep maximum entropy performs well, and a specific case where it performs poorly. When deep maximum entropy did well, it had very sharp peaks in its recovered reward function. As a result, the value function of the recovered optimal policy on the true reward looks very similar to the true optimal policy's value function. Conversely, maximum entropy performs poorly here, as it links areas of high reward with areas of medium reward, even though there are negative reward regions between the high reward areas. When deep maximum entropy does poorly, it seems to have overestimated how large the high reward regions are — the reward here is much more spread out than in other trials. This objectworld seems fairly similar to other trialled objectworlds, so the poor performance of deep maximum entropy here may be due to the specific trajectories that were observed for this trial rather than properties of the objectworld. We note that maximum entropy did not do any poorer than average for this trial, so it is possible that deep maximum entropy is more sensitive to observed trajectories and this may have caused the poor performance here.

Comparing our expected value difference results to those obtained by Wulfmeier et al.[10], we see far less improvement with deep maximum entropy compared to maxi-

mum entropy. While our deep maximum entropy did perform better than maximum entropy, Wulfmeier et al. report that the expected value difference for deep maximum entropy is on the order of five or more times lower than that of maximum entropy. However, we also note that our maximum entropy performed far better than the results presented in Wulfmeier et al. — we considered that this may be due to our use of AdaGrad in our maximum entropy implementation, but removing AdaGrad did not noticably worsen the performance of the maximum entropy algorithm.

For better results in future experiments, we could run more trials. This was not possible here, since trials were quite slow and we had limited computational power. However, from comparing figure 2a (which had 5 trials per point) to figure 2b (which had 40 trials per point), it seems that increasing the number of trials does not affect the standard deviation much. This effect seems to come from occasional failure of both maximum entropy and deep maximum entropy to recover a good reward function. While we are not sure why this occurs, we note that our standard deviations are on the same order as those presented in other results[10, 7]. One hypothesis is that the random initialisation of the reward function occurs in a local minimum that gradient descent cannot escape from; random restarts (such as those used by Wulfmeier et al.) may solve this problem.

Conclusion

In this report, we have described the linear programming inverse reinforcement learning algorithm[8] and the maximum entropy inverse reinforcement learning algorithm[1]. We have also derived the deep inverse reinforcement learning approach taken by Wulfmeier et al.[10], and tested this approach on the objectworld MDP.

We found that the deep maximum entropy approach achieved lower expected value differences than the maximum entropy approach, and is thus better at recovering reward functions.

Future work could involve trialling random restarts in the gradient descent, though this raises the question of how to combine the multiple recovered reward functions. Experimentation on the size and structure of the neural network used would also be useful, since this is not covered by either this report or Wulfmeier et al.[10]; it is possible that deeper or higher dimension networks may increase the effectiveness of the deep maximum entropy approach.

Finally, it would be beneficial to find out why maximum entropy and deep maximum entropy sometimes recover very poor reward functions — this is the reason the standard deviation of our results was so high, and is the most obvious limitation of the two maximum entropy approaches.

References

[1] B. D. Ziebart, A. L. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In *AAAI*, pages 1433–1438, 2008.

[2] P. Montague, P. Dayan, C. Person, T. J. Sejnowski, et al. Bee foraging in uncertain environments using predictive hebbian learning. *Nature*, 377(6551):725–728, 1995.

[3] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on machine learning*, page 1. ACM, 2004.

[4] S. Russell. Learning agents for uncertain environments. In *Proceedings of the eleventh annual conference on computational learning theory*, pages 101–103. ACM, 1998.

[5] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction, 1998.

[6] Bilal Piot, Matthieu Geist, and Olivier Pietquin. Learning from demonstrations: Is it worth estimating a reward function? In H. Blockeel, K. Kersting, S. Nijssen, and F. Železný, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 8188 of *Lecture Notes in Computer Science*, pages 17–32. Springer Berlin Heidelberg, 2013.

[7] S. Levine, Z. Popovic, and V. Koltun. Nonlinear inverse reinforcement learning with gaussian processes. In *Advances in Neural Information Processing Systems*, pages 19–27, 2011.

[8] A. Y. Ng and S. Russell. Algorithms for inverse reinforcement learning. In *ICML*, pages 663–670, 2000.

[9] E. T. Jaynes. Information theory and statistical mechanics. *Phys. Rev.*, 106:620–630, May 1957.

[10] M. Wulfmeier, P. Ondruska, and I. Posner. Deep inverse reinforcement learning. *arXiv preprint arXiv:1507.04888*, 2015.

[11] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

[12] Y. Bengio. Learning deep architectures for AI. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[13] B. C. Csáji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24, 2001.

[14] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[15] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

Appendix — Description of code produced

This appendix describes code implemented as part of this project. All algorithms described in this report were implemented in Python 3.4.0 with the NumPy, CVXOPT, and Theano libraries. The algorithms implemented were:

- Linear programming inverse reinforcement learning (Ng & Russell, 2000)
- Large state space linear programming inverse reinforcement learning (Ng & Russell, 2000)
- Maximum entropy inverse reinforcement learning (Ziebart et al., 2008)
- Deep maximum entropy inverse reinforcement learning (Wulfmeier et al., 2015)
- Value iteration (Sutton & Barto, 1998)

Two MDPs were also implemented:

- Gridworld (Sutton & Barto, 1998)
- Objectworld (Levine et al., 2011)

All of the above code is available on GitHub at `https://github.com/MatthewJA/Inverse-Reinforcement-Learning`.