Deep - Multiple Intentions Inverse Reinforcement Learning

Paras V. Prabhu

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Intelligent Systems)

Supervisor: Professor Ivana Dusparic

August 2019

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Paras V. Prabhu

August 13, 2019

Permission to Lend and/or Copy

Acknowledgments

Firstly and most importantly, I would like to express gratitude towards my supervisor, Prof. Ivana Dusparic, for her support, guidance and feedback throughout the dissertation. Her involvement and insights led to the completion of this project.
I would also like to thank my family and friends for their unconditional support and encouragement during this undertaking.

Paras V. Prabhu

University of Dublin, Trinity College

August 2019

Deep - Multiple Intentions Inverse Reinforcement Learning

Paras V. Prabhu, Master of Science in Computer Science

University of Dublin, Trinity College, 2019

Supervisor: Professor Ivana Dusparic

The entire field of Reinforcement Learning(RL) stands on the concept of reward, which is the way to order an agent to perform a task. A slight inaccuracy in specifying rewards might lead to an RL agent performing entirely different task than we prefer. Applicability of RL in real-world domain, hence, is limited considering resources spent in the process of adjusting rewards and re-training agent. Inverse Reinforcement Learning(IRL) comes to rescue in such circumstances with its distinctive approach of estimating rewards, using given environmental information called features and by observing experts performing the same tasks, This dissertation attempts to resolve two special cases of IRL in a single method 1. when expert observations are intermix of multiple rewards i.e multiple intentions are involved and 2. rewards are complex combination of features. We study the applicability of the combination of two techniques viz. Expectation-Maximization and Deep Neural Networks, in such situations. The proposed approach is evaluated against previously proposed Maximum Entropy based Linear-IRL, in a simulated environment viz. Objectworld, with two features. The proposed method matches the performance of the existing method in the experiment with one feature, while outperforms it with the other one; which gives the hint of the design's potential to handle both the above mentioned issues. Altogether, this dissertation puts research efforts in the idea of designing a single IRL approach to handle various issues involved.

Summary

Reinforcement Learning(RL) is a way of designing autonomous agents where agents interact with their environments to learn how to behave. An environment comprises of set of states and actions possible from those states which lead to another states. An agent's objective is to understand how its environment responds to the actions taken and use it in fulfilling the assigned task. The learning process is driven by what is called the reward maximization; a fresh agent randomly taking actions in its environment learns with time, what policy to adopt in order to reach some beneficial states which offer higher reward than others. Stating these reward states is the way of injecting human knowledge about the assigned task to guide AI agents. Recent breakthroughs in related domains whether it is Deep Neural Network architectures or advancement in GPU technology has lifted RL from books to mainstream autonomous agent designing approach, making it possible to target large-scale, complex problems. Almost all the domains in which AI is outperforming other Machine Learning agents or even humans, e.g. Game Playing, Resource Management, Traffic Light control, Chemical Reaction optimization, are such that a human system designer can clearly define a reward for them e.g. game points. If we have the right reward definition, the problem is reduced to an agent learning the right policies over the time, and can be solved with standard RL methods. On contrary, in other real world environment such as autonomous driving, virtual assistance, there may not exist a clear reward function. It is normal practice to hand-craft the reward function and manually tweak it until desired performance is achieved. A better approach of finding a fitting reward function has been put forward by Inverse Reinforcement Learning(IRL) which estimates reward by observing experts' behaviour. Ongoing research in this sub-field corroborates its potential in the environment such as simulated driving, robotic control etc.

This work targets the problem of extracting multiple reward functions from experts' demonstration when they are performed with more than one intentions in mind. Such technique is needed as we approach more real world domains such as autonomous driving where large amount of observations are available, but are unlabelled with regard to the intentions with which they are performed. The methods yet proposed to extract multiple rewards suits well in the situations when underlying reward is linear function of environment features, but do not address cases when it is non-linear one.

We propose the combination of Expectation-Maximization technique to sort demonstrations into clusters representing separate intentions and Deep Neural Network (DNN) based single intention IRL method to extract reward functions from each clusters. DNN lends itself naturally to the problem due to its capability to handle non-linearity, induced by the use of non-linear activation functions in its hidden layers. The combination of these two methods have never been implemented as per the literature review performed as part of this work. The design proposed puts efforts in the direction of creating a single method to target all IRL related issues.

The proposed design is evaluated in the classical objectworld environment against widely used Maximum Entropy based linear method. The author as an expert performed demonstrations with two reward intentions. These demonstrations along with two type of features are fed to the methods. The evaluations performed showed that, the proposed design gives comparable performance in situations when environment features are clearly stated. On the other hand, it outperforms the linear method by obtaining the exact positions of negative reward objects, even when features are vague. The evaluation gives hint of the potential of the proposed method to handle both multiple intentions and non-linearity in single process.

Contents

List of Figures

Chapter 1

Introduction

The aim of this chapter is to introduce the context of this work and state the need of research in the areas involved. The first section overviews what Reinforcement Learning (RL) is and its applications. It then underlines how RL's working is dependent correct reward specification and lists the reasons behind reward designing problem which resulted in the birth of a sub-field viz. Inverse Reinforcement Learning(IRL). After defining IRL and the way it aids RL, it notes what are some of the gray areas left in the ongoing research efforts. The contribution section states the areas we chose to work on during this dissertation and briefly spells out the contributions in this regard. We state some assumptions made during the process and structure of the remaining report in the final sections.

1.1 Motivation

The section outlines some of the basic terminologies in the context of the work and briefly states the problem and hence the need for research in the subject.

1.1.1 Reinforcement Learning

The Artificial Intelligence popular image and its various current applications depend on the availability of underlying data from which patterns can be recognized; these patterns are then used to predict or act on future data. This approach seems intuitive for the application areas where data is at the core and environment in which application will act is static e.g. image recognition. But this approach seems inadequate when it comes to applications such as game playing, car driving etc. which are more dynamic in nature.

On the contrary, Reinforcement Learning (RL) is a technique of learning from interaction with the surrounding environment. Here, an agent starts acting in an environment with limited or no knowledge, receives positive rewards for reaching desirable state and negative or no reward points otherwise. It gradually starts developing its policy to behave in that environment in such a way to maximize the reward received. To learn the optimal policy an agent should strike balance between intelligence to use the knowledge it has yet gathered viz. exploitation and courage to take calculated risk of going out-of-way to collect new knowledge viz. exploration. This approach seems very natural to train artificial agents, as this is the way we Human learn to live and progress in our lives. E.g. mastering car driving involves starting driving with basic knowledge, receiving negative criticism from tutor for mistakes such as harsh turns, urgent braking or positive comments for perfect lane changing. It is therefore no surprise that this approach started giving excellent performance in some domains, e.g. [30] illustrates how RL agent outperforms human in Atari game environment. This idea of reinforcement learning being natural and humanly way for learning is also supported by the work in behavioural studies [31] [12].

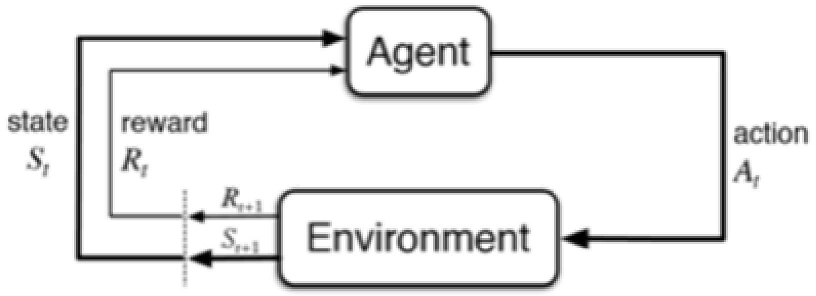
Summarizing all the above points, Reinforcement Learning can be depicted with the below Figure 1.1, where at any time step the agent senses the environment, takes an action according to its learned policy at that time and again senses the new state and the reward after taking the action.

1.1.2 Reward Engineering Problem with Reinforcement Learning

From the representation of RL it is clear that effectiveness of RL agent depends on how accurately it learns the policy to act. The closer it gets to the optimal policy the better. It can also be seen from the above illustration that the learned policy heavily depends on the reward function, which is often

[1]https://skymind.ai/wiki/deep-reinforcement-learning

Figure 1.1: Reinforcement Learning[1]



designed manually. A slight inaccuracy in the reward function has the potential to considerably impact the policy of an agent, and hence the performance. It also decides "How quickly" and "What things" an agent will learn while training.

The importance of reward can be explained with the term "the Cobra Effect", popularized by the late German economist Horst Siebert to illustrate the theory of unintended consequences. The anecdote goes as follows,

> "In a city, the government was concerned about the increased number of venomous cobras. The government therefore started a scheme offering reward for every dead cobra. The strategy proved successful initially, as large number of snakes were killed for the reward. Eventually, however, enterprising people started to breed cobras for income. When the government became aware of this, they decided to scrap the scheme, causing the cobra-breeders to set now-worthless cobras free. Consequently, the cobra population further increased"

It elegantly captures the importance of reward in reinforcement learning. The RL model designing experience shared here[2],describes on the similar lines, how a robotic arm learned to throw things away instead of placing them at a distance, after discovering its power to use torque.

Two of the prominent names in the field, Pieter Abbeel and Andrew Ng. emphasize the importance of reward in RL as,

[2]https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0

”the entire field of reinforcement learning is founded on the presupposition that the reward function, rather than the policy or the value function, is the most succinct, robust, and transferable definition of the task” [1]

A quick overview of the successful RL application till date e.g. [25], shows that most of them are the domains where environment can readily provide the reward. E.g. Atari game where winning is rewarded positively while losing rewarded negatively. But in the real world, it is often not clear at all what the reward should be and there are rarely any intrinsic reward signals such as game score. To continue with the car driving example given in the above section, consider how many possible parameters are there to be a good driver viz. lane changing, safe overtaking, understanding traffic signals and other vehicles' signal, safe turning, maintaining safe distance and many more. To add to the complexity there might also be possible relationships between them e.g. different safe distance for different speed. Specifying the perfect reward function means correctly assigning weights to all these parameters, so that an agent could judge the quality of the state it is in. Most of the time these weights are manually altered until performance is improved upto certain threshold. Clearly, considering the amount of computational power and time required to train RL agents, such trial-error approach is only suitable where reward function is simple enough to specify and possible states are few. As RL systems are becoming more general and autonomous, the reward engineering that elicit desired behaviours, is becoming both more important and challenging [10].

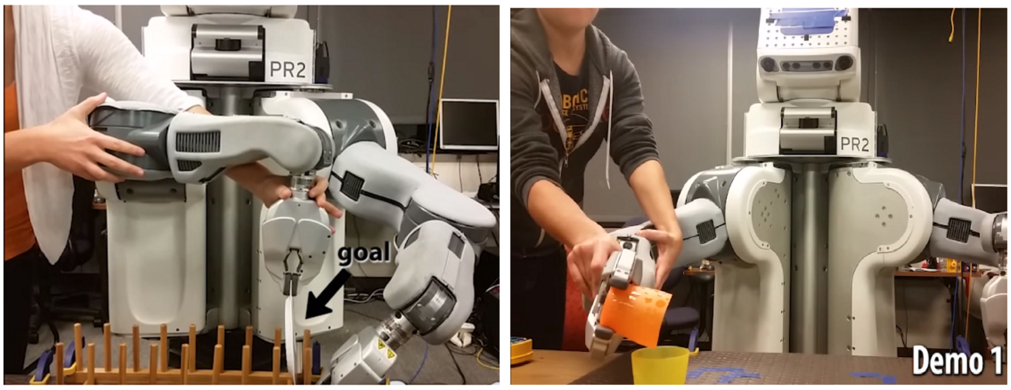
1.1.3 Inverse Reinforcement Learning

Inverse Reinforcement Learning (IRL) is a technique of extracting or approximating reward function by observing expert's behaviour[**3**]. Here, expert means anything from human to computer system, which are supposed to behave as optimally as possible. Similar to RL, IRL too finds co-relations with behavioural science and human learning process. Let us continue with the car driving case. Although, it is possible for a new learner to learn driving entirely by herself by trial-error, it could be time consuming if not disastrous. Instead she could learn comparatively quickly by observing how her instructor drives and so learning desirable traits while avoiding the common mistakes. Similarly, although ultimately an agent will be deriving its own policy to operate; a better reward function can help to arrive at the policy. IRL helps in this concern. Given experts' behaviour observation, IRL attempts to learn the underlying reward function that expert had in mind while demonstrating. To do so, it assumes entire state space as made up of some features and reward function is the function of these features. The job of IRL is then to approximate the parameters of this function i.e. feature weightage, from the available experts' demonstrations assuming that the expert behaved optimally in order to maximize certain feature counts while avoiding the remaining. E.g. frequent lane changing can be a feature which should be negatively weighted, while constant speed should be weighted positively. The learned reward function is then adopted in place of manually defined rewards in training process.

There has been rising research interest in the field recently which underlines its potential, e.g. [21] modelled a real mobile robot which learned from pedestrians demonstrations to navigate in an office environment in the presence of humans, [35] learns path planning tasks from partial observations. Figure 1.2 illustrates the experiment carried out at Berkeley AI-lab by Chelsea Finn [11]. Using IRL she taught two tasks to the robot arm shown; dish placement and pouring task. Descriptive video about the experiments can be found at this link [3].

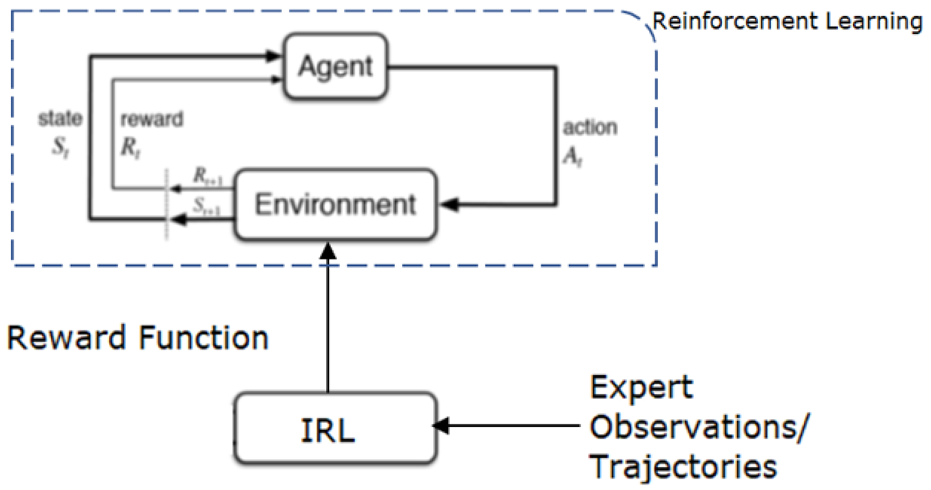
Summarizing all the above points, Inverse Reinforcement Learning can be depicted as in figure 1.3.

Figure 1.2: Inverse Reinforcement Learning-examples



[3]https://www.youtube.com/watch?v=hXxaepw0zAw

Figure 1.3: Inverse Reinforcement Learning



1.1.4 Issues in Inverse Reinforcement Learning

Being a developing technique, there have been lots of open issues to be addressed for IRL to be a full fledged reliable approach. An ideal IRL method would be the one which addresses all these simultaneously. Some of these issues are overviewed below.

1. Generalization

   It refers to extrapolating limited expert demonstrations available to deduce knowledge about the unobserved situations. E.g. a state never visited by the expert. The challenge is to generalize reward function which is not overfitted to sample observations and which deduce the overall goal of expert instead of just focusing on the states visited [32].

2. Feature selection

   From the high level functioning of IRL algorithm explained above, where reward is defined as the function of features; it is straightforward that the efficiency is directly associated with how correctly an environment is described as set of features [33]. It is a way to induce the prior knowledge. The problem becomes challenging when features defined can not model the expert preferences adequately.

3. Multiple Intentions

   The expert may demonstrate the related examples with different goals in mind. E.g. a driver may drive quiet and steadily when going on outing; on the other hand she may drive hastily when going to the office on the next day. An ideal IRL method should not confuse between such observations and be able to recover all the underlying reward functions.

4. Nonlinear Rewards

   Reward function is most often assumed to be a linear combination of features. While this presumption suffices for many problem, it is still inadequate to model reward function for complex domains such as surveillance, where reward definition becomes complex. A reward function is said to be non-linear when there are more than one environmental features and a relationship between reward and one of the features becomes inconsistent in presence of at-least one other feature.

1.2 Contribution

This work targets the third and fourth of the issues listed in the section 1.1.4 i.e. extracting multiple non-linear reward functions from expert's unlabelled observations. The existing approach follows the two step process of first sorting the expert demonstrations into different clusters and then applying single intention IRL methods to extract rewards for different clusters. This work identifies the inefficiency of the second step while handling non-linear reward function and in order to alleviate the problem replaces it with Deep Neural Network(DNN) based IRL method. We show how DNN's primary capability to handle non-linearity can be employed to solve the problem. The proposed method is named as Deep- Multiple Intentions Inverse Reinforcement Learning (Deep-MIIRL).
The design is evaluated in the classical objectworld environment, where reward function is kept non-linear function of environmental features. The evaluation noticed the comparable performance of both the methods when features are clearly stated. On the other hand, even though Deep-MIIRL is not able to recover the exact reward function in case when features are vague, it outperforms

previous Linear-MIIRL.

Extracting underlying multiple reward functions from expert's demonstration is crucial while designing autonomous agents for real-world complex environment such as driving, where the reward function is not as readily available as in other cases. The availability of large data such as sensor data, archived logs, which encodes implied reward function in humans' actions, can be leveraged in order to address the problem. In general, the research contributes in efforts to devise such an approach using which the reward functions from unlabelled data can be extracted and used in training AI agents.

1.3 Assumptions

To match the time constraints and resources limitations, certain assumption are made in the process of implementation and evaluation. Evaluating the design in complex simulation environment or real-world domain would have cause investing considerable efforts in setting up experiments and training process. To keep the focus of the dissertation on the core part i.e. efficient reward extraction, a simulated toy environment is selected for evaluation process which provides comparable complexity to test the design and at the same time makes the environment setup hassle free. We believe that a system performing satisfactorily on such toy environment can be extended with little modifications to more complex domains in future.

The training data required for the evaluation is provided by the author himself carrying out the demonstrations, by assuming the role of an expert. The choice of simulation environment allows us to make this assumption. For real-world domains, this training data is usually gathered through complex data generation process such as collecting sensor data, processing archived big data etc.

1.4 Roadmap

The structure of the dissertation is as follows: Chapter 2 presents related research in the field. It explains formal definition, foundation and different flavours of RL, IRL and MIIRL. The Design chapter covers the approach we take to tackle the issues explained above, divides the system into two parts to handle each of them and formulates the algorithms. Chapter 4 talks about the implementation details involved. Chapter 5 describes the evaluation process and the results. Chapter 6 puts the design and evaluation in the frame to conclude the report. It also underlines some of the future work remained to be addressed.

Chapter 2

Background and Literature Review

This chapter formerly introduces all the concepts related to the thesis. It starts with describing terminologies and foundations of Reinforcement Learning. It then builds on it to describe IRL, different approaches taken to achieve it and a special case of IRL viz. multiple intentions IRL, which estimates more than one rewards. Lastly, we explain the concept of Deep Neural Network and its applications.

2.1 Reinforcement Learning

Most of the Artificial Intelligence techniques today are largely based on availability of data from which a model can derive some insights to act on similar data in future. These techniques, broadly known as Supervised Learning, are being successfully used to address a whole range of business challenges. However, these models are data-hungry and their performance relies heavily on the size of training data available. This role of datasets in AI projects has made Data *the oil of new world* [46] [41]. On the other hand, the reliance of AI models on available data exhibits its obvious limitation that, better modelling techniques would only enable us to extract whatever knowledge incorporated in the adopted datasets, but never to surpass it. It is evident that, this contradicts with the way human learn with the combination of all *data*, *experience* and *experiment*.

Reinforcement Learning presents an alternative approach to achieve machine intelligence which resembles with the human learning process. RL agents try out things on their own in the respective environments, perceive results of their actions in terms of rewards and in process learns how to behave in environment to maximize those rewards. Unlike supervised learning, RL agent does not depend on dataset and creates its own knowledge base by interacting with its environment. An ideal RL agent should make a good use of both the knowledge it has gathered to achieve maximal reward and also go out-of-box to try new things(actions) in order to explore greater reward. One can envisage the possibilities we can realize with this basic concept, provided with sufficient time and immense computational resources.

In most Artificial Intelligence designs, some mathematical framework is created to address problems. For RL, the solution is the Markov Decision Processes (MDP). It enables agent to determine the ideal behaviour in order to achieve certain state. This goal is regulated by agent's policy which determines the steps to be taken from the current states. This optimization is done with a reward feedback system,where different actions are weighted depending on the future state these actions will cause. The general terminology to explain MDPs is given below [39]:

1. Markov Property:
   It states,

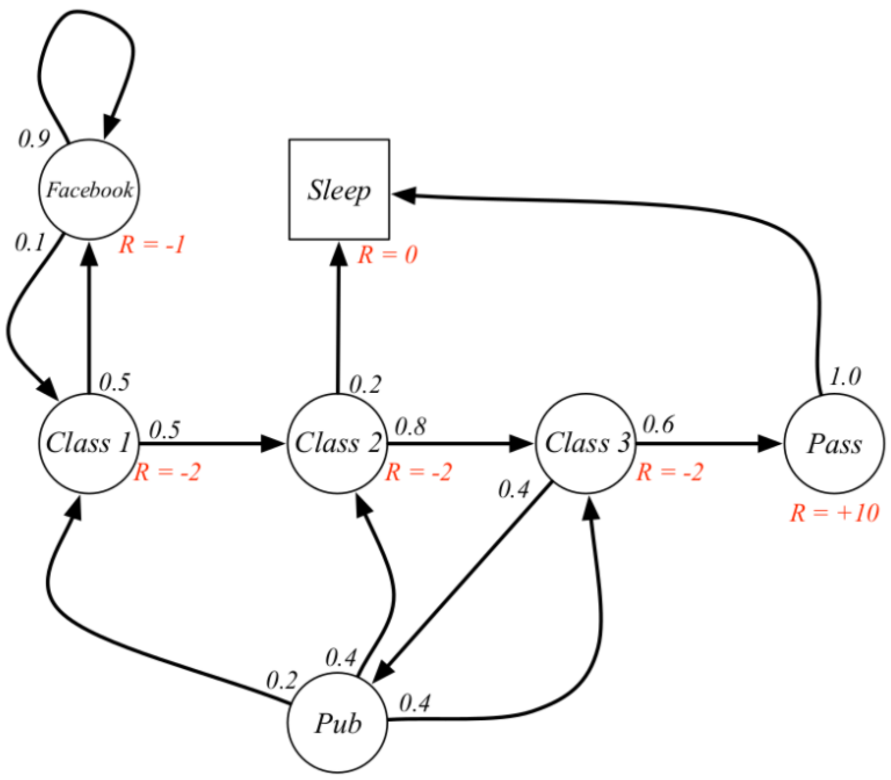
   "The future is independent of the past given the present."

   At any point in time, the future state only depends upon the current state and not on the history of states/path followed to reach the current state and hence history can be discarded.

2. Markov Chain/Markov Process:
   It is a memory-less random process i.e. the sequence of random states achieved by following the Markov Property. The figure 2.1 shows a Markov chain example of a student. E.g. from 'Class 1' he might go on to attend 'Class 2' or might log into Facebook with equal probabilities. A sample of states followed is called episode, which generally end with an absorbing state (Sleep).

[1]https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690

Figure 2.1: Markov Chain[1]



3. Stochastic Environment:

   An environment possesses Stochasticity, if taking action A from state S does not always leads to the same next state S' i.e. results of actions are non-deterministic. To describe such an environment, transition probability matrix is defined which lists probability of landing in every possible states in environment S' after taking action A from S for every state. The transition matrix can either be determined beforehand if the environment is simple enough or can be approximated on the fly by an agent when it starts acting.

4. Reward Function:

   It defines the utility of a state i.e. what reward agent gets being in that state.

5. Discount Factor:

   It is used to determine the desirability of states which do not have any intrinsic utility. E.g. if state A is one step away from state F which has reward of 10, then discount factor of 0.9

would make utility of state A equal to 9(10*0.9) and so on for other states. Discount factor aids in disseminating rewards from reward state back to initial states.

6. Markov Decision Process:
   Reinforcement Learning is formally defined using Markov Decision Process (MDP) as [39]:
   MDP consists of tuple (S, A, T, γ, R) where,
   S: finite set of environment states
   A: finite set of actions possible in the environment
   T: transition probability distribution i.e. a function T(s_t, a_t, s_{t+1}) giving probability of landing in state s_{t+1} from s_t by taking action a_t.
   R: reward function.
   γ: discount factor.

Some other concept which are derived from the basic RL definition and are frequently used in literature are stated below:

1. **Policy** is the function which determines agent's behaviour by taking current state and returning action to be taken; π: S -> A
   There exists the optimal policy π^*, by following which the agent has higher chances of achieving the maximum reward as compared to other sub-optimal policies. An RL agent's objective is to determine a policy as close as to the optimal policy.

2. **Value function** associates a value with every state representing expected reward starting from that state and following the policy.
   $V^{\pi}(s) = E[R(s_1) + \gamma\, R(s_2) + \gamma^2\, R(s_1) \ldots.. \mid \pi]$

3. **Optimal Value function** is the value function associated with the optimal policy i.e. it has highest expected reward. $V^*(s) = \max V^{\pi}(s), \forall\, s \in S$.

4. **Q-value:** Unlike value function which shows profitability of a state, Q-values represents profitability of an action from that state. We can easily imagine when considering Q-value is de-

sirable over Value functions e.g. the state in which a robot has reached a cliff does not have any sense until its next action decides whether it shall fall into valley or gets to diamonds on the other side.

5. **Bellman equation** helps to maximize the sum of cumulative rewards by considering both the immediate possible reward of taking an action and possible rewards in distant future. Optimal Q-value function is defined using it as,
$Q^*(s, a) = R(s, a) + \gamma\ [\sum_{s'} (T(s, a, s')\ V^*(s'))]$
Using this definition optimal value function can also be restated as,
$V^*(s) = \max_a Q^*(s, a)$
$V^*(s) = \max_a [\ R(s, a) + \gamma\ (\sum_{s'} (T\ (s, a, s')\ V^*(s')))\ ]$ .. (2.1)

6. **Value Iteration vs Policy Iteration** Value Iteration is an iterative method to solve for the equation 2.1 until state values converges. It involves repetitively solving for state value function (V) or state-action value function (Q) for the combination of each state and each action. The generated values can then be used by RL agent to decide an action to take from its current state so as to get to the next state with maximal value possible.

   We can imagine that as state-space or action-space increases, value iteration would take longer to converge. Also, an RL agent only needs policy to operate and not actual values i.e. values may take longer to converge but they started offering same policy much earlier than their convergence. Therefore, we can stop iterating as soon as policy converges. This approach is called policy iteration.

7. **Temporal Difference** It can be seen that both the above methods require environment model beforehand to iterate, that is to say these methods are suitable for model-based agents. For the situations when an environment model is not available, a modified version of the above methods viz. Temporal Difference (TD), is used to learn the policy. The basic idea behind TD Learning is to update Q-values on-the-fly as an agent starts interacting with environment, instead of computing them beforehand.

TD(s, a) = [R (s, a) + γ max$_{a'}$ Q (s', a')] - Q (s, a)

Q$_t$(s, a) = Q$_{t-1}$ (s, a) + α TD$_t$ (s, a)

where α is the learning rate to control how fast the values change.

TD updates the past Q-values estimation upon each new interaction with environment. This equation is bound to converge after finite iteration in non-changing environment and so we can use it to find out the policy. Even if environmental conditions change in the middle, it has the ability to adapt to them on-the-fly. Therefore, TD is more suitable to use in real world scenarios.

Thus, RL boils down to representing environments in terms of state and actions, forming reward function, specifying required discount factor and then solving it using any of the above method Value Iteration, Policy Iteration or TD to form policy to behave.

2.2 Inverse Reinforcement Learning (IRL)

In recent years, IRL has attracted researchers from Artificial Intelligence domain as well as from fields such as psychology, behavioral science, control theory etc. Its notion of making systems intelligent not only by leaving them learn by themselves but also combining it with the available human expertise looks promising considering the current state of the art [38] [34]. Some primary reasons of this developing interest are listed here.

2.2.1 Significance of IRL

Reward function inference

Generally when autonomous agent is designed, the problem is specified as forward learning/forward control task using RL methods(as explained above) or optimal control methods. The core part of this specification is representation of expected behaviour, preferences via a reward function. When it is specified manually, one needs to repeatedly tune it and test system. before we begin to get expected behaviour. Also the process gets cumbersome as environment becomes complex or parameters increases, and hence it is not practical to follow this process in real world applications. The issue constraints the RL approach to problems in which a lucid reward function is readily available. IRL aims to target this constraint by bypassing manual reward specification to directly infer it from examples of desired behaviour [4].

Enhanced Generalization

From all the other parameters of MDP listed above, reward function is the most succinct and representative of them for preferences of agents [1]. Policies i.e. state-to-action mapping, as put by Control Theory, is the more direct way to represent preferences; but this is a short-term solution as a slight change in environmental condition would leave policies worthless in new conditions. E.g. if few more states are added, then there will be no representation for them in terms of policy. We need to define state-action mapping each time a new state added, and while doing so the effect of new states on existing policy should also be considered. On the other hand, if preferences are modelled via reward function, then as long as new states are represented similarly as older ones, no change is needed. Using RL methods, a new policy for changed environment can be regenerated. If dynamic RL methods like TD learning explained above are used, then it is not even required to regenerate and agent can adjust with new environment on-the-fly. Thus, the characteristic of IRL to recover reward function than any other aspects of MDP make it more generic.

Possible Applications

From the conception of IRL, it possess the potential to contribute to modelling problems for human and animal behaviour [38]. E.g. Ullman[42] and Baker[6] formulated human behaviour as planning problem and recovered the human goals using IRL. The research in IRL has rapidly brought it across as a mainstream technique in variety of applications:

1. Intelligent Personalized Vehicle command by recovering user preferences from sample driving demonstrations. These include several types of control such as helicopter control [2], boat driving [33], socially aware robot navigation learning from humans to avoid colliding into

humans [22] [18].

The efficiency of hybrid vehicles i.e. mix of engine power & battery power, depends on parameters such as battery level, engine efficiency to operate at different speed. If the future power requirements are known, a more economical use of engine vs. battery power can be planned. [43] proposed model to predict the future routes so that the battery power scheduling can be efficient, without requiring any hardware improvements or change in driver behaviour.

2. Modelling some other agent's preferences to plan our policies. [48] applied it to predict route and destinations using model learned from preferences of 25 taxi drivers over 100,000 miles. [37] combined IRL with LEARCH(LEArning to searCH) approach to implement planning algorithm for legged locomotion robot. [49] designed robot-navigation model from goal-directed trajectories of pedestrians.

3. The capability to infer preferences from observations lends itself naturally in multi-agent setting where an agent would act either cooperatively with other agents to maximize the efficiency or adverserially to overpower the opposition. [7] employs IRL in adverserial multi-agent environment to penetrate a perimeter patrol by learning patrollers' policies and preferences.

2.2.2 Formal Definition of IRL

This definition builds on the terms and terminologies explained above in RL section.

Expert: A human or system which performs demonstrations and which is assumed to have been near-optimal while doing so.

Expert's Reward function(R_E): The reward function an expert had in mind while carrying out demonstrations.

MDP/R_E : MDP without reward function.

Trajectory/Observation/Sample (τ) : $\{(s_1, a_1), (s_2, a_2) \dots (s_t, a_t)\}$

It is an series of states and respective actions taken by an expert in a single episode, which likely to end in end/absorbing state.

Expert Demonstrations (D): $\{\tau_1,\tau_2 \, . \, . \, \tau_\mathrm{m}\}$

A collection of trajectories observed in a single session. It is assumed that all $\tau \in D$ are entirely observed.

IRL seeks to recover $\mathrm{R_E}$ that best describes the given observations.

2.3 Methods for IRL

This section introduces various methods proposed to solve the IRL problem above. To correlate different approaches, some common steps which generalizes the process are listed below as template.

Input: Expert Demonstrations (D) $\{\tau_1,\tau_2 \, . \, . \, \tau_\mathrm{m}\}$
Output: Recovered Reward function $\mathrm{R_E}$
Steps:

1. Create MDP without Reward function from expert's demonstrations.
2. Describe reward as a function of some parameters such as states, state features etc.
3. Figure out MDPs behaviour (e.g. in terms of policy, state-visitation frequency etc.) under current reward function
4. Update reward function parameter to lessen the difference between MDP's current behaviour and expert's observed behaviour.
5. Iterate the steps until the difference is reduced.

Different approaches assumes Expert's Reward function in different format e.g. linear function of state features, probability distribution over states, probability distribution over possible actions etc. The solution is then to update this assumption to new values which would explain the possibility of expert's demonstration more certainly.

2.3.1 Maximum Margin

This approach seeks such a reward function which explains observed demonstrations better than other possible policies by a margin.

Linear Programming

This is the initial approach proposed by Ng and Russell [34] which generate reward function which would produce the inputted expert's policy as optimal solution when solved with MDP. Note, it requires the exact policy of an expert and not the demonstrations. In order to maximize the margin, it seeks to maximize the difference between action values for each state of solved MDP and the second best MDP possible. It assumes reward function as linear weighted sum of basis functions as,

$R_{\mathrm{E}} = w_1\phi_1(s) + w_2\phi_2(s) + ..w_{\mathrm{i}}\phi_{\mathrm{i}}(s)$

,where ϕ_{i} defines the basis function and $\mathrm{w_i}$ is the weight assigned.

The method proposed originally for a discrete state-space. If state-space is continuous then, Ng and Russel suggest, we can fragmentize it and take samples to generalize it into discrete state space.

Apprenticeship Learning

The Linear Programming method has limitations as it requires the policy of an expert which is rarely available directly. Instead, expert demonstrations are readily available in most cases. Apprenticeship Learning [1] modifies linear programming approach to learn from demonstrations. For this purpose, it defines the term *feature count*. Environment is assumed to have been made up of set of features possessing values and each state is characterized with some of those features. *Expected feature count* is sum of feature values under a policy.

$\mu(\pi) = E\Bigg[\sum_{t=0}^{\infty}\gamma^{\mathrm{t}}\phi(s_{\mathrm{t}})|\pi\Bigg]$

For set of demonstrations, feature count is given as,

$\mu(\pi_{\mathrm{E}}) = E\Bigg[\frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{\infty}\gamma^{\mathrm{t}}\phi({s^{\mathrm{i}}}_{\mathrm{t}})|\pi\Bigg]$

Then, to find a reward function so as to minimize the divergence between expert's feature count and assumed policy would be the task to do. The algorithm repetitively adjust '*w*'s to do so. Abbeel and Ng states that, as long as feature count and reward function are simple enough to be correlated, a reward function would be close enough with the original one. [33] solves the formed problem using gradient descent method in reward space. Syed[40] approaches the problem in an unique way to formulate it as zero-sum game between the max player choosing a policy and adversary replying with reward function.

2.3.2 Entropy Optimization

The maximum margin algorithms solely depend upon matching feature expectation of expert's demonstrations. The problem with this approach is that many policies and hence many reward functions may satisfy a feature expectation constraint. For example, reward function giving zero reward for every state explains every observed demonstration quite perfectly. This is known as *degeneracy problem* which makes IRL an ill-posed problem. As a remedy to it, *Entropy Optimization* methods resort to maximum entropy principle [16].

Maximum Entropy

This approach initially introduced *Maximum Entropy* in context of IRL. As explained above Maximum Margin methods results in degeneracy problem. Apart from that, it also assumes that demonstrations given are all optimal ways of doing thing. In reality, there could be some noises and suboptimal behaviour demonstrated by expert in some cases. To continue with the car driving example, car instructor could always have some glitches in his driving, but the leaner is not expected to learn from it and should neglect such noisy behaviour. Maximum Entropy deals with these issue by leveraging probabilistic approach of maximum entropy, which allows to extract the policy distribution from expert trajectories which only depends on feature expectation and no other aspects [47]. This approach allows the algorithm to deal with noise and probable suboptimal behaviour in expert demonstration.

In order to do so, it defines reward function as linear combination of state features,

$R_\theta = \theta^T . f(s) = \sum_{s \in \tau} (\theta^T f_s)$ . . (2.2),

where f_τ is feature representation of trajectory state space and θ is feature weight vector to be determined.

At the core of the method, there is a probabilistic technique viz. **maximum likelihood estimation (MLE)**, which is elaborated in detail in Appendix A for reference. To briefly elucidate it here, it takes statistical model and data as input, and outputs parameter values of that model that maximizes the likelihood of generating such a data under that model.

In case of Maximum Entropy IRL, the statistical model is assumed as follows. The probability of a trajectory being performed by an expert is exponentially proportional to its expected reward.

i.e. $\Pr(\tau) \propto e^{R\theta(\tau)}$

i.e. $\Pr(\tau) = e^{R\theta(\tau)}/Z$ .. (2.3)

where Z is the normalization term defined as $Z= \sum_{\tau} e^{R\theta(\tau)}$

So now that we have probabilistic model of the trajectory distribution, we can find the parameters (θ) of reward function associated with it by formulating it as optimization problem using MLE i.e. search for θ such that it will maximize the log-likelihood of the probability distribution function (2.3).

$L = \text{argmax}_{\theta}$ [$\log \prod_{\tau_{\mathrm{d}} \epsilon D} (\Pr(\tau_{\mathrm{d}}))$]

L= - argmin_{θ} $[(\frac{1}{M}) (\sum_{\tau_{\mathrm{d}} \epsilon \mathrm{D}} (\log (e^{R_{\theta}(\tau)}/Z)))]$

($(\frac{1}{M})$ is taken just for the mathematical convenience and it doesn't affect the optimization problem in hand)

L= argmin_{θ} $[((\frac{1}{M}) \sum_{\tau_{\mathrm{d}} \epsilon \mathrm{D}} R_{\theta}(\tau_{\mathrm{d}})) + \log \sum_{\tau} e^{R_{\theta}(\tau)}]$

Differentiating with respect to reward parameter θ,

$\nabla_{\theta} L = (\frac{1}{M}) (\sum_{\tau_{\mathrm{d}} \epsilon \mathrm{D}} \nabla_{\theta}(R_{\theta}(\tau_{\mathrm{d}}))) - (\sum_{\tau} \Pr(\tau) \nabla_{\theta}(R_{\theta}(\tau)))$

The second term here can be converted to the function of all states S in the trajectories τ.

$\nabla_{\theta} L = (\frac{1}{M}) (\sum_{\tau_{\mathrm{d}} \epsilon \mathrm{D}} \nabla_{\theta}(R_{\theta}(\tau_{\mathrm{d}}))) - (\sum_{\mathrm{S}} \Pr(\mathrm{S}) \nabla_{\theta}(R_{\theta}(\tau)))$

Here the first term is called **expert feature expectation** which is the summation over each feature of each state visited by expert in given demonstrations.

The second term is called **learned feature expectation** which is feature expectation calculated similarly as above; only the demonstrations are generated by following the policy generated under learned reward function. The required demonstrations are generated by starting from the same start states as of expert's, and then following the learned policy. The required calculation os performed by dynamic algorithm.

The equation is written in short hand as,

$$\nabla_\theta L = \mu_\text{D} - \mathbb{E}(\mu)..(2.4)$$

Finally, equation (2.4) is the optimization target which can be solved by simply calculating minima using gradient descent following below steps.

Input: Expert Demonstrations (D) $\{\tau_1,\tau_2 \,..\, \tau_\text{m}\}$
Output: Reward function parameter θ
Steps:

1. Iterate over given expert demonstrations to calculate μ_D
2. Randomly initialize θ.
3. Using R_θ, find current policy π(a|s) (using value/policy iteration).
   Calculate learned feature expectation $\mathbb{E}(\mu)$ using Dynamic Programming.
4. Compute gradient in equation (2.4)
5. Using gradient update θ.
6. Go to step 3.

Relative Entropy

Maximum Entropy IRL requires to calculate policy at each iteration, as evident from the steps above. Satisfying this requirement would start going out of hand, as state space start getting larger and continuous. Relative Entropy IRL [8] seeks to eliminate the dependence on learning environment dynamics.
Initially random trajectories having feature counts close to that of demonstrated ones are created. Secondly another set of trajectories are sampled under some uniform baseline policy assumed. Then reward function can be learned by iteratively minimizing the relative entropy between these sets of distribution. Finn et al. [11] implemented REIRL by employing neural networks to represent reward where difference in SVF between two sets of policies is backpropogated as error signals.

2.3.3 Machine Learning techniques

A few attempts to address IRL problem using classical ML techniques such as regression or classification have also been made.

Feature creation

All the IRL methods implemented relies on availability of environment described in term of features. Levine et al. [23] devised method to address both reward function recovery and environment feature creation. It begins with empty set of features. The first step viz. Optimization step constructs reward function for current set of features and the next step viz. Fitting step creates new set of features that describes constructed reward function more precisely than previous set of features. The process involves expressing states in form of minimal regression tress for rewards which results in new fine tuned features which makes R_E appropriate with demonstrations.

Classification

Klein et al. [20] formulated IRL as a classification problem where the loss function of a classifier is parameterized with feature expectation of trajectories. It also formulated Q-values in terms of feature expectation as, which made it possible to connect Q-values and IRL through feature weights. The algorithm then minimizes the loss function by adjusting feature weights through gradient descent method. Klein [19] further extended the method to cope up with unknown environment dynamics, estimating it using given demonstration through regression.

2.3.4 Bayesian approach

Bayesian inference is a method in which Bayes' theorem is used to update the probability for a hypothesis as more evidence or information becomes available. For example, if the problem is to find the probability if a person having cancer, initially it is straightforward to answer with whatever percent of the population has cancer. As we get some additional information such as the person is a chain-smoker, the probability can be safely increased, This is known as Bayesian update.

Bayes rule is give as,

$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$

,where A is the event we want to find probability of and B is the new evidence/information. $P(A|B)$ is the probability of event A given evidence B about it, called *posterior*. $P(B|A)$, called *likelihood*, is the probability of observing evidence B, under initial hypothesized probability of A. P(A), called *prior*, is the initial probability of happening even A. P(B), called *marginal likelihood*, is the general probability of observing evidence B. Ramachandran and Amir [36] defined posterior distribution over reward function as,

$P(R_{\mathrm{E}}|\tau) = \frac{P(\tau|R_{\mathrm{E}})P(R_{\mathrm{E}})}{P(\tau)}$

where $P(\tau|R_{\mathrm{E}}) = \prod_{(s,a)\in\tau} Pr((s,a)|R_{\mathrm{E}})$

Under the given reward hypothesis, some state-action pairs are more probable to occur. The reward hypothesis is continuously updated as more number of trajectories are considered to fit correctly to observations. Michini et al. [28][29] extended this approach by dividing a trajectory into sub-parts and designate each of it with separate reward function to represent their own sub-goals.

2.4 Multiple Intentions Inverse Reinforcement Learning (MIIRL)

Many a times a large amount of data for a particular domain is available to us as result of archived data or collected through modern-day smart sensors technology. E.g. nuScenes [9] recently published data collected through 13 high-end sensors fitted to cars including camera, RADAR and LIDAR. Such datasets can be leveraged in designing reward functions for AI agents performing IRL on them. The problem arises in straightforward use of the datasets due to the contexts in which these demonstrations were observed. For example, a person might be rushing to office one day or driving safely while on family outing the next day. Simply feeding such observations to any IRL method would produce chaotic reward function which would not be able to catch any of the intentions correctly. MIIRL solves this problem of extracting multiple reward functions from given demonstrations.

Vroman et al [5] aggregates Linear IRL with clustering technique which sorts demonstrations into different goals clusters. The process is iterative where reward functions for each cluster are found as per initial assumptions and then each demonstration is weighted for each cluster as per how much it adheres to the policy generated under the reward function of that cluster. In the next iteration, a demonstrations input to calculate the reward function of a cluster is weighted against the likelihood of adherence to that cluster found in previous step. The process requires the number of clusters beforehand.

Gleave and Habryka [13] extended this method in the context of Entropy Optimization explained in 2.3.2 to make it more sample efficient i.e. it could learn from small numbers of demonstrations. It does so by simply adding regularization term to the loss function. The underlying assumption is that, all the clusters' intentions are closely related to each other and differs very slightly; therefore the reward weights (θ) for each cluster are equally distributed around some mean. The regularization term added to the loss bars the reward weights (θ) from moving too far away from the mean.

Bogert and Doshi [7] formulated this problem as multiple experts interacting with each other through acting in the environment. The interaction is framed as general-sum strategic game between two players. In the process, one agent learns the preferences and therefore, distribution over reward function of the other one. [26] extended the approach to suit multiple agent environment.

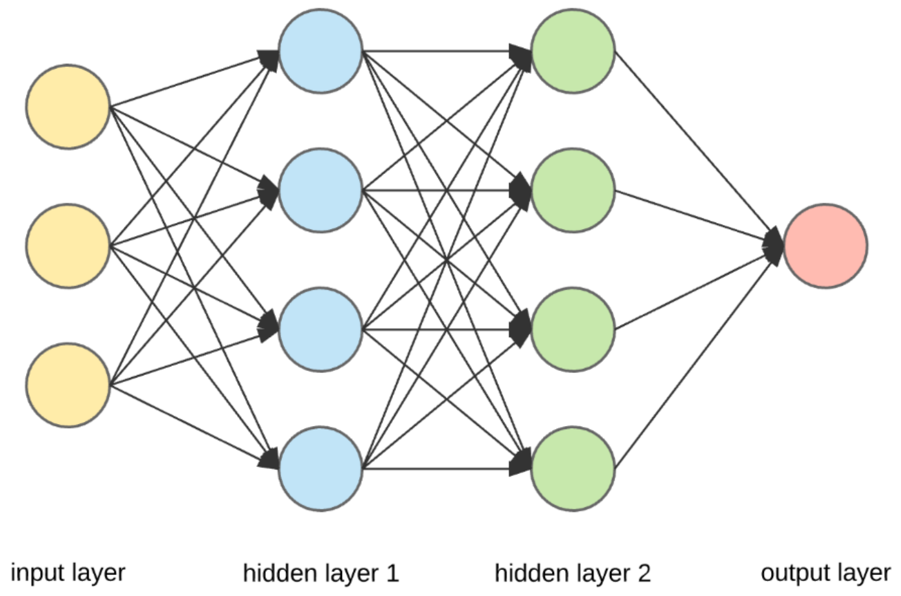
2.5 Deep Neural Networks

2.5.1 What is it?

Deep Neural Network is the computation model which simulates human brain's working in order to perform and which in theory has capability to represent any complex relationships. A high level picture of a DNN is depicted below.

DNN is made up of number of layers where each layer contains neurons in it. Each neuron receives information from every neuron from the layer behind it, does some computation on the received information and forwards it to the next layer. With enough training data feed in to a DNN, it learns what transformations to perform on it to convert it to the desired output in the last layer.

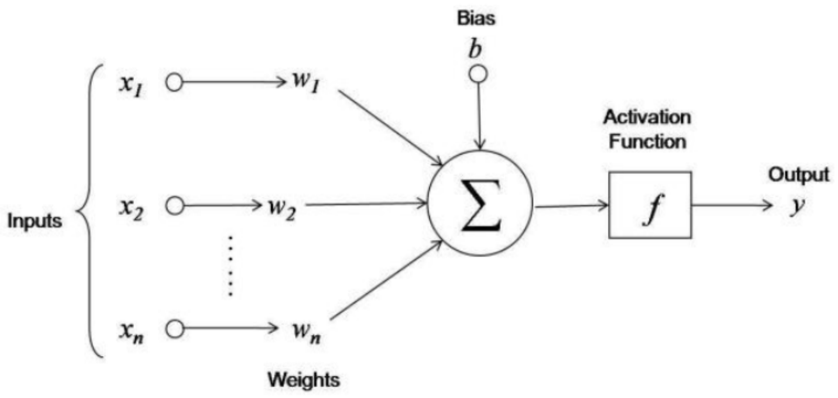
Figure 2.2: Deep Neural Network[2]



2.5.2 Artificial Neuron

Primary component of DNN is artificial neuron which non-linearly transforms the input received into output.

Figure 2.3: Artificial Neuron[3]



For example. the neuron in the figure receives three inputs(*'X's*). Each of them are multiplied by some weights(*'W's*) before taking summation over them.

[2]https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6
[3]https://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network

$f(x)= b + w_1.x_1+ w_2.x_2. . . + w_n.x_n = b+W^TX$

Then, instead of directly passing this output to the next layer it applies a some non-linear activation function e.g. sigmoid on it.

$y=\sigma(f(x))$

2.5.3 Train a network

Training DNN means learning proper values for all W and b. It is achieved through the process of **feed forward** and **backpropagation**. In feed forward stage, each training example is passed through a network and network transforms it to some output using neuron computations explained above. The next stage is to calculate **Loss (*L*)** i.e. the estimate of how much the predicted output diverges from the actual one. Ideally, the loss function should be zero. The training process aims to make it as close as possible to zero.

Once loss is determined, the next stage is to adjust each W and b so as to minimize that loss. **Gradient Descent** is the mathematical technique based on calculus which is used for this purpose. Gradient value of loss with respect a weight i.e $\frac{\partial(L)}{\partial(w_1)}$ is calculated, which determines responsibility of w_1 for the current loss. Using that gradient value, w_1 is nudged as below.

$w_1 = w_1 + \frac{\partial(L)}{\partial(w_1)} \cdot \alpha$

,where α is called **learning rate** which determines how fast network weights are to be adjusted (usually 0.001).

It starts from calculating gradient for the last layers' neuron and then going backward for each layers' neuron, hence the name backpropagation. As we can see, calculating gradient and adjusting values for each and every weight in network is very computationally heavy task which requires considerable amount of time and resources. After enough of training, each of the W and b are so adjusted that when the new data comes in from the input the desired output can be expected from a network.

The real power of DNN lies in the activation functions which controls firing up of neuron and gives non-linearity to networks. Without them, a neural network is just a combination of multiple linear functions. Sigmoid, ReLu, Tanh are some flavours of activation functions. All these factors together give DNN the capability to represent any complex continuous function viz. *Universal Approximation* [15], which sets it apart from other Machine Learning techniques.

2.5.4 Applications

The *Universal Approximation* characteristic make DNN popular implementation choice to solve RL problems. Mao et al. [27] employed RL to autonomously decide to allocate and schedule machine resources to waiting jobs to minimize the average job waiting time. Current resource allocation and jobs' profile forms the state space which is fed to DNN. Loss function is defined as sum of negative inverse of waiting time for each job. The network is created with single hidden layer of 20 neurons fully connected to input and output layers. Hausknecht and Stone [14] used a variant of DNN viz. RNN, which gives a network the capacity to memorize. They trained the system to play Atari 2600 games, which takes only screen pixels as input and acts in the available action space of a game. The loss function is the game score given by the game engine. The memory element enables the agent to take decision not only depending on the current screen situation but also considering some previous screen memorized. The architecture surpassed the human level performance baseline as well as the previous milestone setup by the DNN architecture without memory [30].

Chapter 3

Design

The previous chapters introduced the overall need for IRL, the special case viz. MIIRL for extracting multiple reward functions from demonstrations, the attempts in the direction and some open areas. The scope of this work is to develop a single method which handles both multiple intentions and non-linear rewards. The literature review showed that although these have been addressed separately, there is no common approach to target both. We propose one approach called Deep-MIIRL to handle both these issues under one roof, using mathematical technique of Expectation-Maximization(EM) and Deep Neural Network(DNN).
Before moving forward, it is advised to get acquainted with EM algorithm explained in Appendix B, entropy optimization and MLE from section 2.3.2 and basics of MIIRL from section 2.4, on which the design is based on and refers to in order to avoid repetition.

3.1 Overview

The overall system can be viewed as a process which is fed with environment details such as states & features of each state, expert demonstration samples and number of intentions with which those demonstrations are carried out and it outputs the reward function parameters for each intention.
Let us lay down the basic terminologies,
$\mathcal{J}$: number of intentions i.e. number of clusters.

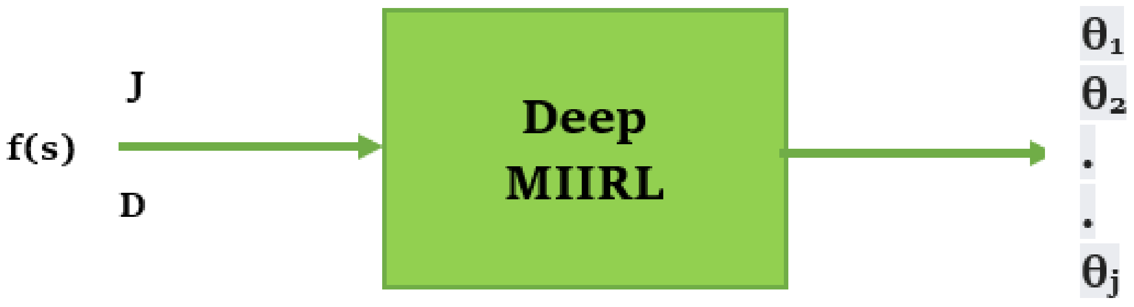
$f(\mathrm{s})$: F-dimensional vector per state representing values for F number of features.

$\mathrm{R}_{\theta} = \theta^{\mathrm{T}} \cdot f(\mathrm{s})$: This definition of reward function is extended in later sections to handle non-linear reward functions; but the basic idea remains same as in section 2.3 i.e. reward is the function of state features. θ is called reward function parameter.

τ: a sequence of state-action pairs $\{(s_1,a_1), (s_2,a_2) \ldots (s_t,a_t)\}$ called a trajectory.

D: expert demonstrations consisting m-number of trajectories $(\tau_1, \tau_2 \ . \ . \ \tau_m)$, where each trajectory is performed by an expert with any one of the J-intentions in mind.

Figure 3.1: Deep-MIIRL Overview



3.2 Handling Multiple Intentions

The high-level frame of the architecture is of a sorting algorithm which separates trajectories into clusters representing different intentions. Once we have trajectories separated into clusters, single intention IRL is applied to each cluster to extract θ. But as we can see, in this case sorting in one-go is impractical, considering limited information in hand. Hence, we resort to EM algorithm which instead of doing hard-sorting, does soft-sorting by assigning weights(within 0-1) to each trajectory for each cluster as per how likely it belongs to that cluster. This likelihood is the possibility of a trajectory being generated by a policy under the current reward function of that cluster. Once soft-sorting weights are determined in this step, single intention IRL is applied for each cluster. Here, input of each trajectory is weighted against its soft-sorting weight for that cluster.

Let us state few more notations,

$\mathrm{Z_{ij}}$: soft-sorting weight of trajectory τ_i belonging to cluster j.

ρ_j : prior-probability of cluster j (between 0-1) i.e. what percentage of trajectories belong to cluster j.

$\pi_{\theta_\mathrm{j}}(s,a)$: probability of taking action '*a*' from state '*s*' when following policy π generated under reward function with parameter θ_j. Policy can be found by any standard RL method (value iteration or policy iteration, explained in previous chapters).

In E-step, we approximate $\mathrm{Z_{ij}}$ for every τ_i and cluster j, as follows,

$$z_\mathrm{ij} = \prod\nolimits_{(s,a)\in\tau_\mathrm{i}} \frac{\pi_{\theta_\mathrm{j}}(s,a)\rho_\mathrm{j}}{N} \tag{3.1}$$

Product of probabilities for all (s,a) pairs in τ_i gives joint-probability of τ_i as a whole being generated by following policy π generated under reward function with parameter θ_j. This term is weighted against cluster's prior-probability(ρ_j). N is just a normalization term to keep $\mathrm{Z_{ij}}$ from growing out of proportion. The equation as a whole gives probability of τ_i belonging to cluster j.

In M-step, we update our hypothesis about ρ and θ to make them maximally likely under new Z values.

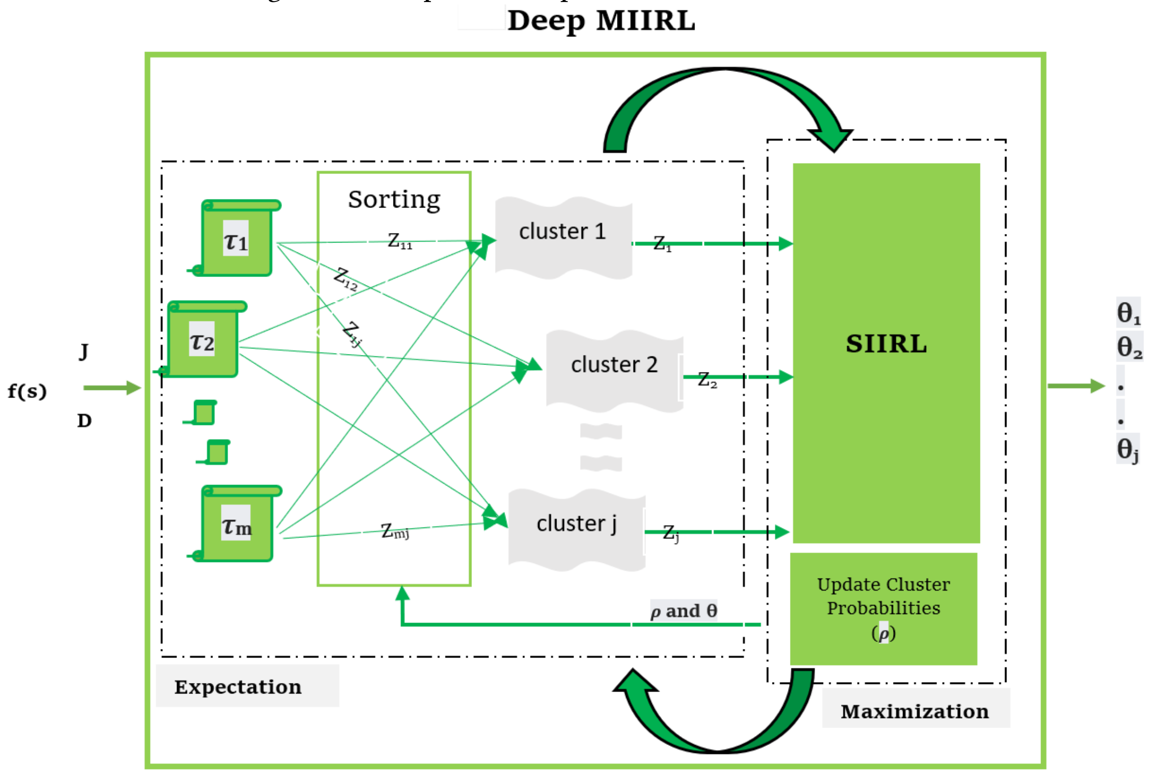
Updating ρ is a trivial task, as it involves simply averaging over the probabilities with which trajectories belong to a cluster, as follows,

$$\rho_\mathrm{j} = \sum\nolimits_i \frac{z_\mathrm{ij}}{M} \tag{3.2}$$

Updating θ is the critical step here. Any single intention IRL(SIIRL) can be plugged here with one modification: input of a trajectory τ_i in calculating θ_j is considered only as much as it belongs to the cluster-J i.e. $\mathrm{Z_{ij}}$. At this point, we consider it as a black-box which takes in demonstrations(D), features $f(\mathrm{s})$, soft-sorting weights(Z) and returns updated reward parameters θ. In the next section, the specifics on how to use DNN based SIIRL here to handle non-linearity are detailed. Both the steps are iterated until values begin to converge or until specific number of iterations.

All the procedure is summarized below in the form of pseudo-code in Algorithm 1 and also depicted in the Figure 3.2.

Figure 3.2: Deep-MIIRL Expectation-Maximization view



3.3 Dealing with Non-linearity

As we have the larger picture in place now, let us concentrate on its sub-part viz. SIIRL which finds reward parameters(θ) for every cluster separately. As pointed out in earlier section, we can plug-in any SIIRL method in here. We based our choice for this method on the Maximum Entropy based IRL's (see section 2.3.2) enhancement provided by Wulfmeier et al. [44], which proved DNN's suitability in SIIRL problem. The choice of this architecture is natural considering DNN's intrinsic capability to handle non-linearity (see section in 2.5).

Algorithm 1 Handling Multiple Intentions

```
    Input $J, f(s), D(\tau_1, \tau_2 .. \tau_n)$
    Output $\theta$
 1: procedure
 2:     $\theta \leftarrow (J - dimensional)$
 3:     $\rho \leftarrow (J - dimensional)$
 4:     $z \leftarrow (J * M - dimensional)$
 5:     for every j in (1 to J) do
 6:         Using $\theta_j$, find current policy $\pi_j$ (using value iteration/policy iteration).
 7:         $N \leftarrow 0$
 8:         for every $\tau_i$ in D do
 9:             $joint - probability_i \leftarrow 1$
10:             for every (s,a) pair in $\tau_i$ do
11:                 $joint - probability_i \leftarrow joint - probability_i * [probability\ of\ (s, a)\ under\ \pi_j]$
12:         $N \leftarrow N + joint - probability_i$
13:         for every $\tau_i$ in D do
14:             $z_{ij} \leftarrow \frac{joint-probability_i * \rho_j}{N}$
15:     for every j in (1 to J) do
16:         $total - probability \leftarrow 0$
17:         for every i in (1 to M) do
18:             $total - probability \leftarrow total - probability + z_{ij}$
19:             $\rho_j \leftarrow \frac{total-probability}{M}$
20:     for every j in (1 to J) do
21:         $\theta_j \leftarrow SIIRL(f(s), D)$
22:     Go to step 5.
```

At the core of Maximum Entropy IRL and methods developed from it, the assumption is that reward function is linear combination of environmental features. As stated by equation (2.2),

$$\mathrm{R}_\theta = \theta^{\mathrm{T}} f(\mathrm{s})$$

The assumption poses the obvious limitation in situations when reward function can not be explained in linear terms. To remedy this, our approach twists this assumption to make reward function non-linearly transformed linear combinations.

$$\mathrm{R}_\theta = \theta^{\mathrm{T}} \psi(\mathrm{s})$$

$$\psi(\mathrm{s}) = \sigma(\mathrm{W} \cdot f(\mathrm{s}))$$

Here, Sigmoid (σ(e)) is a basic non-linear activation function applied to every element e. It can be replaced with any other non-linear function to suit a problem in hand. The same logic can be stretched to arbitrary length as,

$$\mathrm{R}_\theta = \theta^{\mathrm{T}} \psi_1(\mathrm{s})$$

$$\psi_2(\mathrm{s}) = \sigma(\mathrm{W}_1 \cdot \psi_1(\mathrm{s}))$$

$$\psi_3(\mathrm{s}) = \sigma(\mathrm{W}_2 \cdot \psi_2(\mathrm{s}))$$

$$\cdot$$

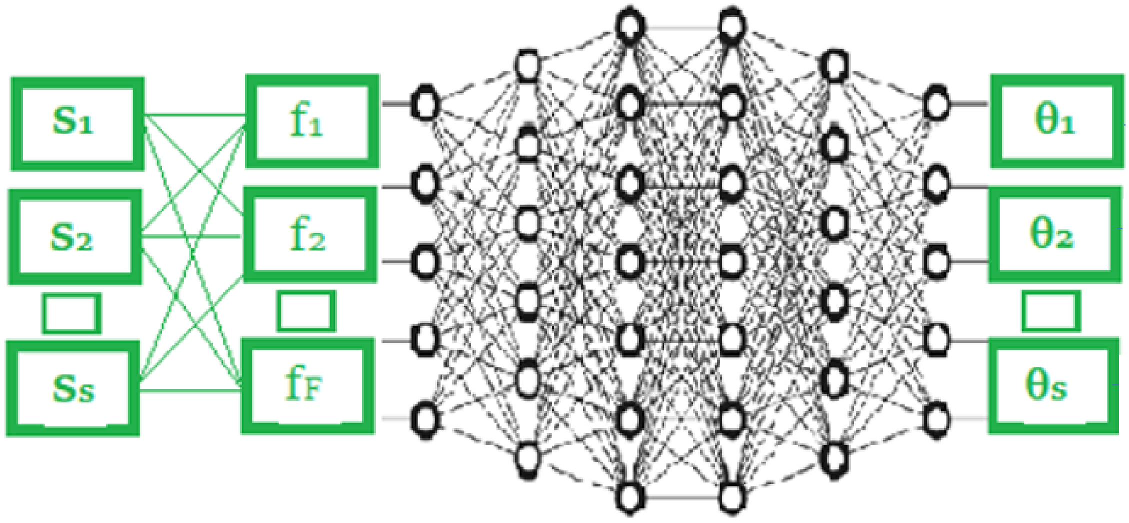
$$\cdot$$

$$\psi_{\mathrm{n}}(\mathrm{s}) = \sigma(\mathrm{W}_{\mathrm{n-1}} \cdot f(\mathrm{s}))$$

Therefore, we can approximate any reward function with state features and suitable values for 'W's.

This structure is similar to deep neural network(DNN), which maps input to output governed by network parameters. Figure 3.3 explains the concept intuitively where any state can be mapped to its reward parameter by the network. The states' features are fed from input layer and a trained network is expected to map them to the respective reward parameters, governed by network weights representing 'W's. The non-linear transformation is handled by all the activation-functions of the neurons.

Figure 3.3: State to Features to Reward Mapping using DNN



3.3.1 Network Training

The training process for the network architecture proposed here would differ from traditional DNN applications such as computer vision, in terms of input data. These applications requires different instances of input, so a network could learn to map them to output by adjusting the weights. In our design, the input data would always be the same i.e. the state features, as the number of states in an environment is always going to be a fix number once defined. Therefore, it is the loss measure of a network and how we backpropagate it to adjust weights, which would dominate the training process mainly.

To define the loss measure, we rely on the loss-gradient defined by Maximum-Entropy method(section

2.3.2) as in equation (2.4)

$$\nabla_\theta L = \frac{\partial L}{\partial \theta} = \mu_\mathrm{D} - \mathbb{E}(\mu)$$

,where μ_D is **expert feature expectation** and $\mathbb{E}(\mu)$ is **learned feature expectation**. Minimizing this loss using gradient-descent would lead to proper θ values.

$\mu_\mathbf{D}$ can be calculated using the same process given by section 2.3.2 i.e. to iterate over expert trajectories to sum up feature count for each state each feature. This is the one time process as expert demonstrations are going to be same always once given[1]. The one thing needs to be modified here to make it suitable for *multiple intentions*, as specified in section 3.2 i.e. weight each trajectory against how much it belong to the cluster we are currently calculating reward parameters for. In other words, τ_i's feature inputs are multiplied with $\mathbf{Z}_\mathrm{ij}$ while training for cluster j.

On the other hand, the process for calculating $\mathbb{E}(\mu)$ differs from Maximum-Entropy. We use the current reward generated by the network to obtain the current policy. Then sample trajectories are generated under that policy. We start from the same *start states* as that of in expert demonstrations and follow the generated policy to create sample trajectories in the same number as that of expert's. Once we have the sample trajectories, calculating $\mathbb{E}(\mu)$ is similar to finding μ_D explained above, except that this calculation is performed repeatedly after every forward pass.

The loss-gradient defined above is in relation with θ. As we need it in relation with network weight W's ($\frac{\partial L}{\partial W_i}$) to perform backpropagation, the chain-rule is applied to find this relationship as follows,

$$\nabla_{\mathrm{W}_i} L = \frac{\partial L}{\partial \theta} \cdot \frac{\partial \theta}{\partial W_i} \qquad , \forall\, i$$

$$\nabla_{\mathrm{W}_i} L = (\mu_\mathrm{D} - \mathbb{E}(\mu)) \cdot \frac{\partial \theta}{\partial W_i} \qquad , \forall\, i$$

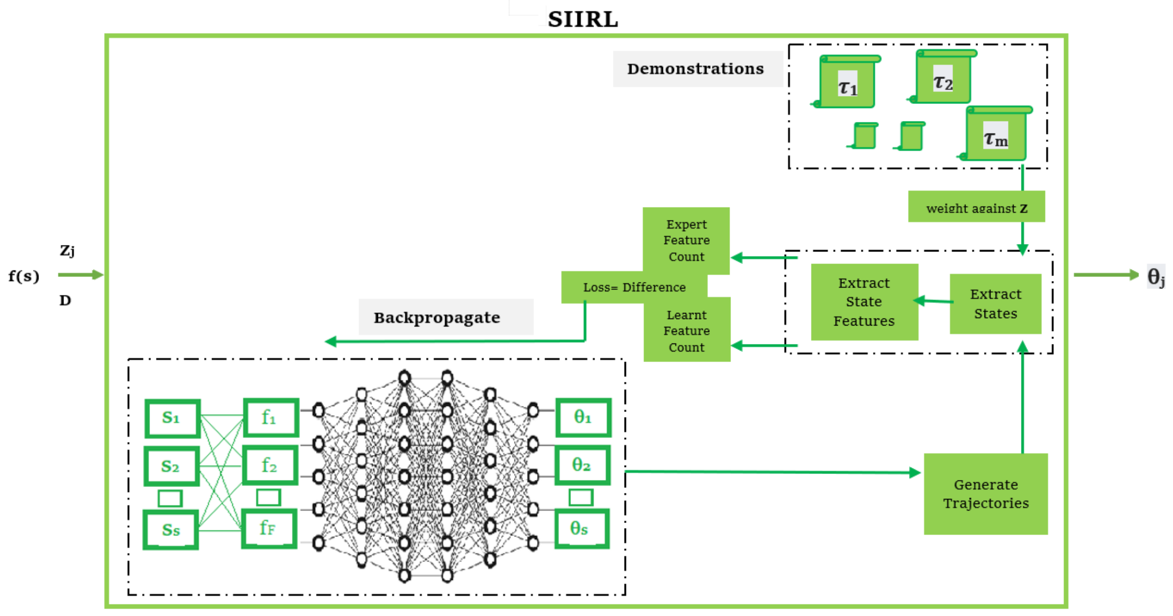
Here the second term is the normal network gradient showing the relationship between the output reward θ and a weight W_i. This gives us our final gradient which is backpropagated to tweak the

[1]Demonstrations would be changing in special cases such as life-long learning, active learning where an agent keeps on updating throughout its life. In such cases, μ_D needs to be recalculated time-to-time.

network weights.

The procedure is summarized in Algorithm 2 and Algorithm 3 through the pseudo-code and also depicted in Figure 3.4. Note that, this component fits in the place of **SIIRL** of section 3.2. In the pseudo-code, FORWARD_PASS and BACKPROPAGATE are standard DNN methods to transform input to output and backpropagate loss measure to adjust the weights respectively. GENERATE_POLICY is a standard RL method which creates a policy using value iteration/policy iteration from the reward given.

Figure 3.4: SIIRL using DNN



Algorithm 2 SIIRL

```
    Input $Z_j, f(s), D(\tau_1, \tau_2 .. \tau_n)$, epochs
    Output $\theta_j$
1:  procedure
2:      $F \leftarrow (no.\ of\ features)$
3:      $S \leftarrow (no.\ of\ states)$
4:      $expert\ feature\ expectation \leftarrow (F - dimensional)$
5:      $learned\ feature\ expectation \leftarrow (F - dimensional)$
6:      $expert\ feature\ expectation \leftarrow 0$
7:      $learned\ feature\ expectation \leftarrow 0$
8:      $\theta\_learned \leftarrow (S - dimensional)$
9:      $Reward\_learned \leftarrow (S - dimensional)$
10:     for every $\tau_i$ in D do
11:         for every (s) in $\tau_i$ do
12:             for feature in (1 to F) do
13:                 $expert\ feature\ expectation_{feature} \leftarrow [expert\ feature\ expectation_{feature}] + z_j * [f_{feature}(s)]$
14:     for e in (1 to epochs) do
15:         for every (s) in States do
16:             $\theta\_learned_s \leftarrow FORWARD\_PASS(s)$
17:             $Reward\_learned_s \leftarrow (\theta\_learned_s * f(s))$
18:         $D_{sampled} \leftarrow GENERATE\_TRAJECTORIES(Reward\_learned)$
19:         for every $\tau_i$ in $D_{sampled}$ do
20:             for every (s) in $\tau_i$ do
21:                 for feature in (1 to F) do
22:                     $learned\ feature\ expectation_{feature} \leftarrow [learned\ feature\ expectation_{feature}] + f_{feature}(s)$
23:         $LOSS \leftarrow (expert\ feature\ expectation - learned\ feature\ expectation)$
24:         $BACKPROPAGATE(LOSS)$
```

Algorithm 3 GENERATE_TRAJECTORIES

```
    Input Reward_learned, $D_{\text{expert}}$
    Output $D_{\text{sampled}}$
 1: procedure
 2:     $M \leftarrow (length(D))$
 3:     $D_{\text{sampled}} \leftarrow (M - dimensional)$
 4:     $\pi_{\text{current}} \leftarrow (GENERATE\_POLICY(Reward\_learned))$
 5:     for every $\tau_{\text{i}}$ in D do
 6:         $state \leftarrow (\tau_{\text{i}}[0])$
 7:         $L \leftarrow (length(\tau_{\text{i}}))$
 8:         $\tau_{\text{sample}} \leftarrow (L - dimensional)$
 9:         for l in (1 to L) do
10:             $next\_state \leftarrow (\pi_{\text{current}}(state))$
11:             $\tau_{\text{sample}}.add(next\_state)$
12:             $state \leftarrow (next\_state)$
13:         $D_{\text{sampled}}.add(\tau_{\text{sample}})$
```

3.4 Summary

Our design's high-level architecture is primarily based on Expectation-Maximization technique which aids in simultaneously sorting demonstrated trajectories into different clusters and calculating reward functions for each clusters using single intention IRL. We proposed the use of DNN based single intention IRL method in the second part, to handle non-linearities in reward functions. The training process for the network design here differs from other traditional applications in that, we input the same data i.e. state features in each forward pass and get reward function parameters from output. Thus, the entire training process relies on how we calculate loss measure; for which we used loss measure provided by earlier stable technique viz. Maximum-Entropy IRL. In effect, our design is a consolidation of two of the existing approaches which enables us to extract multiple non-linear reward functions.

Chapter 4

Implementation

This chapter describes the implementation of MIIRL agent. We extend the existing framework which implements the necessary basic components. We detail how these components are improved upon to suite the needs and also about the necessary layer added on top of them. The system working is explained through the high-level UML class diagrams which presents overview of the components involved. It is followed by the sequence diagram which explains flow of actions. Interface of a few crucial methods and their implementation is described wherever necessary.

4.1 IRL framework

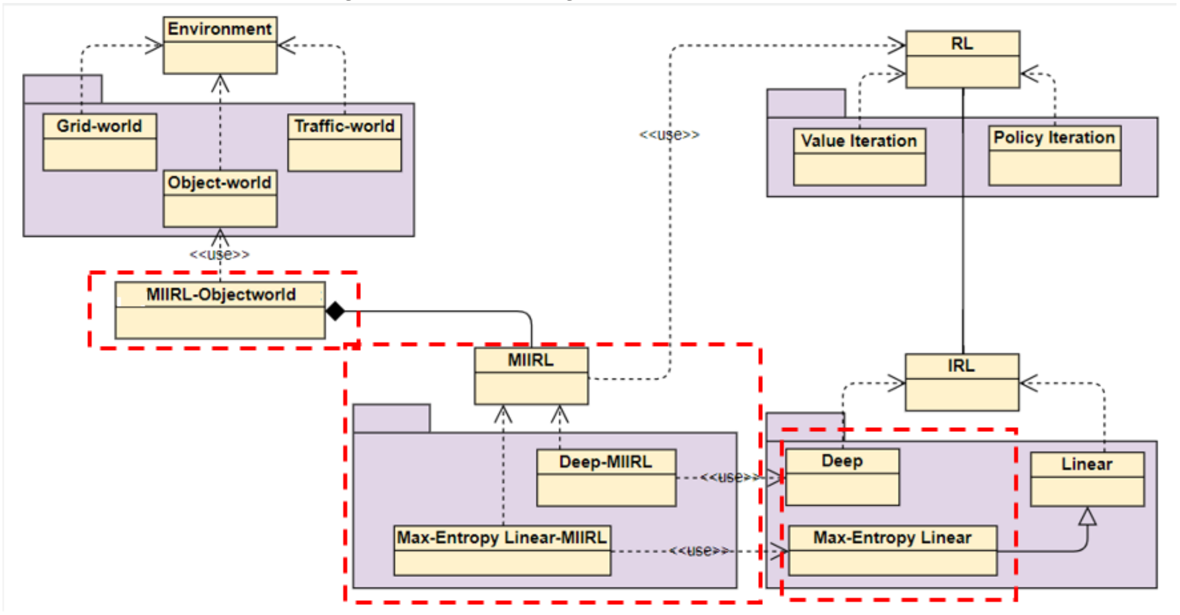
For implementation of the proposed Deep-MIIRL technique, we use the framework provided by Alger[3] which provides the primary components such as simulation environment, policy iteration, Linear IRL, Maximum-Entropy based IRL, Deep IRL etc., which are building blocks for our design. It is implemented in Python-3.7 using following libraries,

- NumPy
- Theano (for DNN)
- scikit-learn

- MatPlotLib (for graph plotting)

The UML class diagram for the framework is shown in Fig. 4.1 which depicts only crucial parts for our project, as the entire framework is vast enough to portray. The parts highlighted in red are the components we added or changed as part of this project.

Figure 4.1: Class Diagram for IRL-framework



4.1.1 Environment module

This is the parent class for all the environments provided. It provides the interface for creating a simulation environment with all the essentials. It was primarily designed for RL problems and therefore has provisions for specifying terms related to it also. We deal with the Object-world environment in our case. Its constructor method's signature is provided below.

```
class Objectworld(Gridworld):
def init(self, gridsize, nObjects, nColours, wind, discount):
```

- gridsize: no. of blocks in row and column of a grid. (a grid is always a square)
- nObjects: no. of objects placed

- nColours: no. of colors of placed objects
- wind: uncertainty in the environment. It defines stochasticity i.e. the possibility of an agent taking random actions.
- discount: RL related term (γ) (refer section 2.1)

The original code was to randomly distribute objects over grid; we changed that part to define fix positions so as to repeat the same experiments with different settings.

```
def featureMatrix(self, featureMap="manhattan"):
```

We added this method to specify environmental features. The default one is *manhattan distance*. The other one used in the experiments is *Effect range*. These are crafted manually considering the context of the project, but can also be auto-specified using other improvements in IRL methods.

4.1.2 Reinforcement Learning module

The entire module is primarily designed for RL related problems. We use *Value Iteration* part of it to generate policy of learned rewards to sample trajectories. These trajectories are then used to calculate *learned feature expectation*, as explained in the design chapter.

```
def findPolicy(environment, reward, stochastic=True):
```

The method uses Q-learning approach to determine policy. Reward here is the *learned reward* obtained from the *learned reward parameters* at the end of forward pass through network.

4.1.3 IRL module

It provides components for single intention-IRL with three different flavours: Linear-IRL, Maximum Entropy based Linear IRL, and Deep IRL. Last two are used in the project. Below given is the interface method for the IRL class which each of these three types implements with different approaches.

```
def irl(environment, featureMatrix, trajectories, epochs,
        learningRate):
```

- environment: Objectworld in our case.
- featureMatrix: features
- trajectories: expert trajectories provided.
- epochs: no. of time all the trajectories will pass through the process.
- learningRate: rate for gradient descent/ network training

As explained in the design chapter, we needed to improve on this method to consider soft-sorting weight (i.e Z_{ij}), which gives probabilities of a trajectory belonging to a cluster. The above method's signature and internal working are so modified to accommodate this change, as below.

```
def irl(environment, featureMatrix, expertTrajectories, epochs,
        learningRate, softSortingWeights):
```

Here, *softSortingWeights* is of same dimension as *expertTrajectories*, expressing probabilities of each of trajectories belonging to the cluster currently under consideration.

We also detail other helper methods in the module which are required in the process.
In both Maximum-Entropy based Linear IRL and Deep IRL, training is based on the difference between features of expert trajectories and sampled trajectories. The signature of the method performing the task is given below. It returns result of the same dimension as that of no. of features. As with the above method, this one is also modified to accommodate soft-sorting weights.

```
def findFeatureExpectations(featureMatrix, trajectories,
        softSortingWeights):
```

Below given method is used to generate sampled trajectories from the current reward obtained from the current guess of the reward parameters. It starts from the same start-states as that of expert and then follows the current policy obtained using *findPolicy()* method given above, to generate the same length of trajectory as that of expert. It repeats the procedure for all the expert trajectories and then returns the generated sampled trajectories as output.

```
def generateSampledTrajectories(environment, reward,
        expertTrajectories):
```

4.1.4 MIIRL module

This method updates soft-sorting weights(Z_{ij}) as per current cluster-priors (ρ) and reward parameters (θ).

```
def computeSoftSortingWeights(theta,clusterPriors):
```

The below method is the interface for all the above methods. We specify the required data to this module and it takes care of calling the necessary methods in order, returns extracted reward parameters and in the end also presents the color-map of grid for sake of lucidity (the same color-maps are used for evaluation purpose).
The snapshot of a few instructions is also given. The arguments of the called methods are avoided for simplicity. At the start of each iteration, current soft-sorting weights are computed for all the clusters and then for each cluster reward parameters are updated.

```
def miirl(noOfCusters, noOfIterations, expertTrajectories,
environment, featureMatrix, method, networkStructure):
    for i in range (noOfIterations):
        softSortingWeights= computeSoftSortingWeights()
        for j in range (noOfCusters):
            theta[j] = irl()
```

- noOfCusters: number of intentions of expert trajectories have to be provided beforehand.
- noOfIterations: iterations of Expectation-Maximization algorithm.
- method: linear/max-ent-linear/deep. In our experiments, we used the last two.
- networkStructure: number and shape of hidden layers.

4.1.5 MIIRL-Objectworld module

It's the entry-point/main method for the framework. It initializes the required components and then passes them to the interface *miirl()* method above. The objecworld environment is initialized as:

```
import objectworld as objectworld
environment = objectworld.Objectworld(10, 10, 3, 0.2, 0.9)
```

Then a feature matrix is fetched. The method does not take any arguments and hence fetches *manhattan distance* by default.

```
featureMatrix = ow.featureMatrix()
```

The expert demonstrations are performed in a separate session and the trajectories are stored in a log file. *requestExpertTrajectories()* fetches those trajectories from the log file.

```
expertTrajectories = ow.requestExpertTrajectories()
```

The related configuration variables are also defined here.

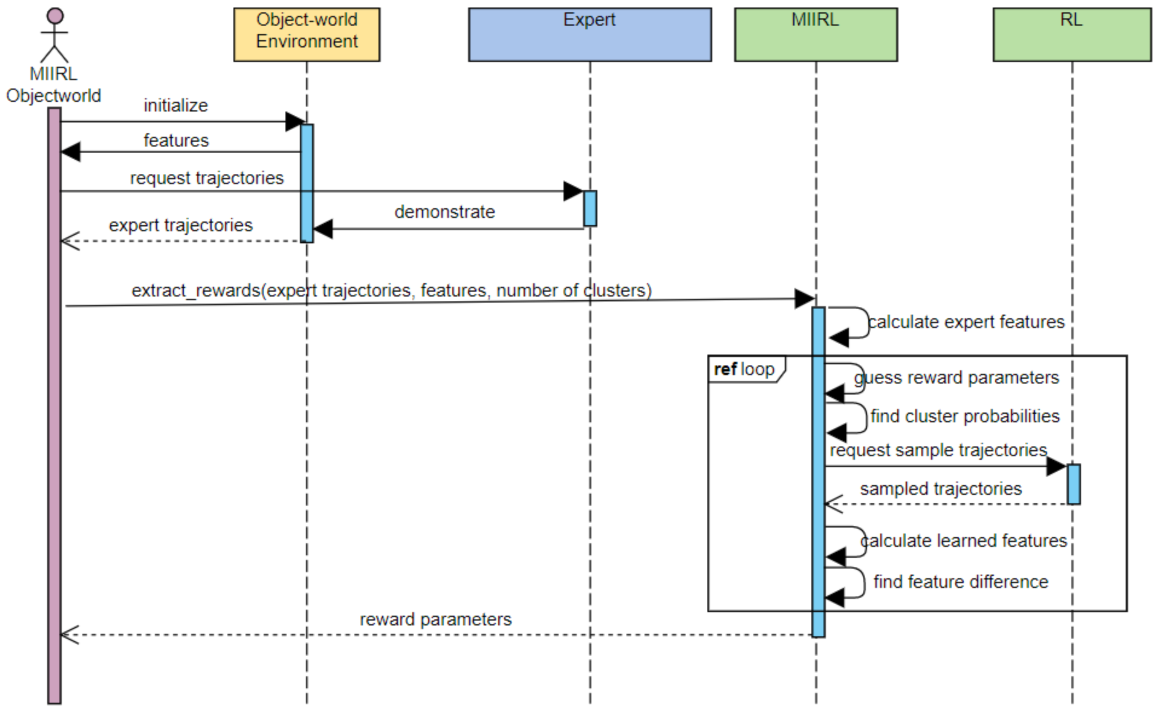
```
noOfClusters=2
noOfIterations=50
networkStructure=(15,15)
method="deep"
```

No. of hidden layers and their shape can be changed by changing *networkStructure* variable. For instance, there are two hidden layers with 15 neurons in each.

4.2 Actions sequence

The previous section explained the system from component point of view. Let us focus on how these components interact. The sequence diagram of actions occurring in the system in shown in Fig. 4.2. As mentioned in earlier section, MIIRL-Objectworld is the entry-point of the system where all the

Figure 4.2: MIIRL sequence diagram



required configuration is initialized. It boots an environment, in our case Objectworld, with number of objects, colors and type of features. It then requests expert to perform demonstrations. An expert carries out demonstrations in the environment. The actual expert demonstrations are performed in separate session in this project. Therefore, *request demonstrations* action is fulfilled by fetching them from log file. With all the configured information, MIIRL interface method is instantiated. This call is handed over to either Max-Ent based Linear MIIRL or Deep-MIIRL depending on the *method* variable. MIIRL module shown in the sequence diagram is the combination of both IRL and MIIRL components of the above section. The steps illustrated in design chapter are then followed between

MIIRL and RL, which ultimately returns reward parameters.

4.3 Summary

This chapter presented implementation details in accordance with the design proposed. The existing RL+IRL framework available with various basic components is enhanced upon in order to do so. We presented the snapshot of the relevant module from the entire framework in form of the class diagram and indicated the parts we improved and newly added. The interface of a few crucial methods, their brief functioning is given. The parts of actual code are attached wherever necessary. After components are laid, their interaction in chronological order is depicted through the sequence diagram.

Chapter 5

Evaluation

This chapter presents the details on the evaluation process of the project. It begins by explaining why the Objectworld environment is selected for testing the proposed approach. We then give details about how its employed in our experiments, types of environmental features used and two different intentions with which demonstrations are carried out. The comparison of our approach with the Linear-MIIRL method shows increasing accuracy as environment is defined with clearer features.

5.1 Objective

As mentioned before in Design chapter, the aim is to target two core issues simultaneously,
a. extracting multiple reward functions from expert demonstrations,
when **b.** they are non-linear functions of environmental features.
Maximum Entropy based Linear-MIIRL is selected to compare the results due to its widespread use as evident from literature review. It has shown remarkable performance to extract multiple linear reward. Our method is expected to keep its performance and also to extend it when rewards are non-linear.
We fed both expert demonstrations and features to these method and rewards extracted from them are compared against the known ground-truth. Th experiments are run until values began to converge. Our method is expected to outperform Linear-MIIRL by recovering more part of ground-truth

reward. This advantage must be consistent for different features, to confirm that it does not favour only one kind of feature.
We also compare both the methods based on training time required. As Deep-MIIRL involved training neural network which is computationally intensive, its expected to require longer training duration than linear method.

5.2 Simulation Environment

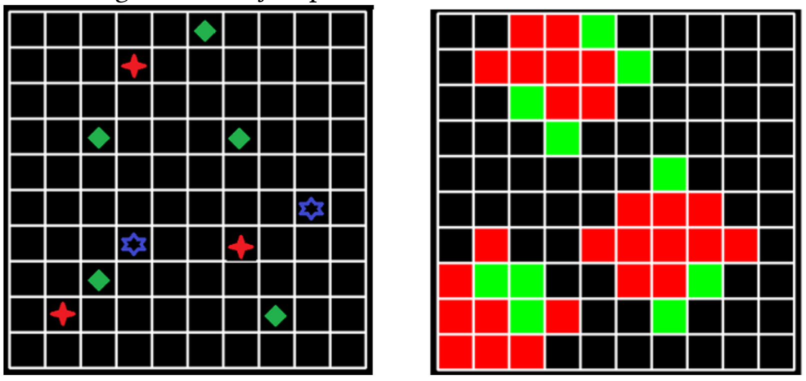
5.2.1 Objectworld - the choice of environment

Instead of directly testing on real-world problem simulations, we have chosen Objectworld environment, which is an elementary environment for primarily testing the suitability of the proposed approach. In Objectworld, colored objects having some range of effect are placed randomly on a grid which assigns some characteristics to the states surrounding it. Rewards generated are such that placement of an object makes reward generated by some other object inconsistent, which makes it non-linear. This provides the essential complexity required for evaluation, while keeping the efforts required for recreating the experiments minimal. Once a method has passed the primary evaluation phase, the same environment has the facility to increase the complexity, or one can opt for more real world environment. The similar environment was used by previous researches like Levine et al. [24], Wulfmeier et al. [44], Jin and Spanos [17].

5.2.2 Object Placement

We considered 3 colors viz. *red* having effect range up to 2 steps, *green* up to 1 step and *blue* also up to 1 step. Placement of these objects on the grid generates two types of rewards. The states within range of both red and green have one type of reward marked by Green color. On other hand, states within range of red only posses other type of reward marked by Red color. Blue objects served as distractors having no effect. Here, placement of green object is making rewards by red objects inconsistent. Figure 5.1 shows how objects are placed for the experiments and the reward map generated by it.

Figure 5.1: Object placement and Generated Rewards

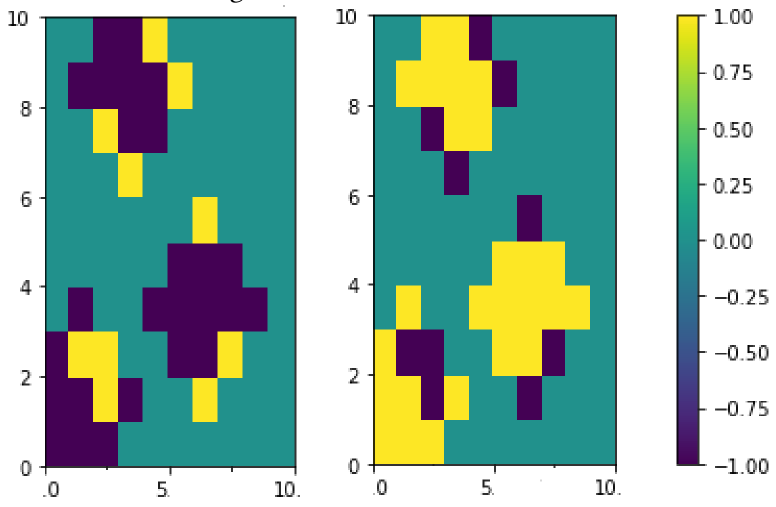


5.2.3 Intentions and Rewards

Two types of intentions are assumed for the experiments:

- getting to green regions.
- getting to red regions.

Figure 5.2: Ground-truth Rewards



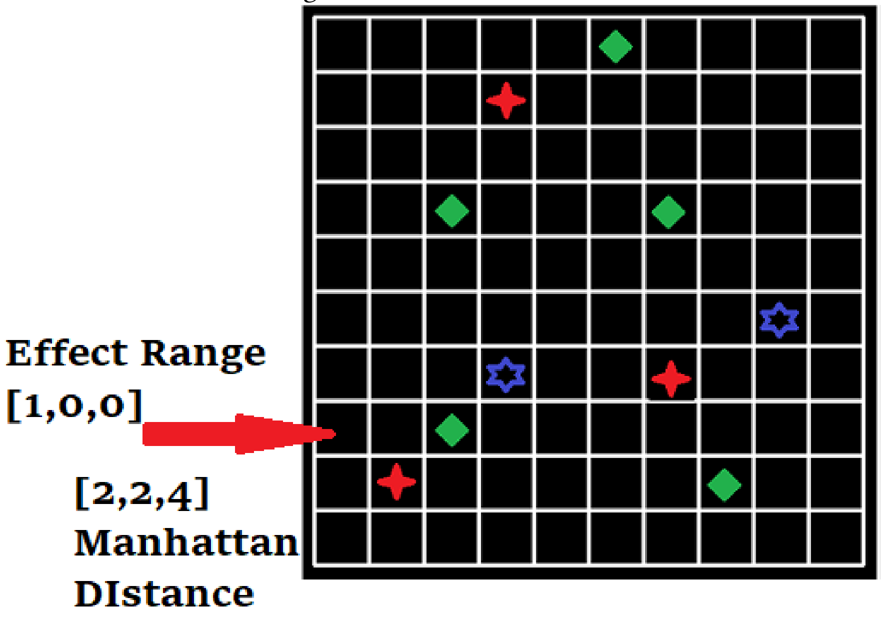
The ground-truth rewards for both the intentions is shown in figure 5.2. The figure is a replica of the grid above and numbers on left and bottom are the row and column numbers, provided for ease of understanding. Here, *yellower* a state, higher the reward and *bluer* it is lesser the reward. The legend on the right side conveys the same clue. This divides the entire state space into positive, negative and neutral reward regions. *These same assumptions are continued further while comparing the results.*

5.2.4 Environmental Features

Two widely used features from the previous experiments in literature are selected for the experiments. These features are manually encoded and provided to the algorithms. The Experiments are repeated for both of them, keeping the other conditions maintained.

1. *Manhattan Distance* The distance to the nearest object of particular color.
2. *Effect range* It is a binary feature showing 1 if a state comes under the effect of an object. As we can notice, this one is more precise than the above, giving exact information about a state.

Figure 5.3: Both the Features



The figure 5.3 shows both the feature values for the state at 3rd row-1st column of the grid. 1st element gives value for Red object, 2nd element for Green and 3rd for Blue. Effect range feature for the state is [1,0,0] which shows it is in effect of a Red object and not the other two. Similarly, Manhattan

Distance feature for the state is [2,2,4] which shows that it is 2 step away from the nearest Red & Green and 4 steps away from the nearest Blue object.

5.2.5 Expert Demonstrations

The author assuming the role of an expert carried out demonstrations for both the intentions, by controlling the agent. Total of 1200 demonstrations are performed; 480 for the first intentions(green region) are remaining for the other(red region).
The expert starting from a random state had 9 steps to complete a single episode i.e. a trajectory is of length 9. The expert had to take agent to its desirable region for a particular intention. Five legal actions are allowed from each state viz. up, down, left, right and stay. If the agent is on edge of the grid, and an action is taken to take it off the grid, it stays in the same state. The environment is kept to be stochastic; 80% of time agent takes the same action that the expert performed and 20% of time a random one. Therefore, while performing demonstrations the expert took care of choosing states as far as possible from the undesirable states, to avoid accidentally falling into those states due to random actions. All the episodes are recorded in the log file, which is then fed to the algorithm.
The link to a short video of the expert carrying out the demonstrations is given here[1].

5.3 Results & Discussion

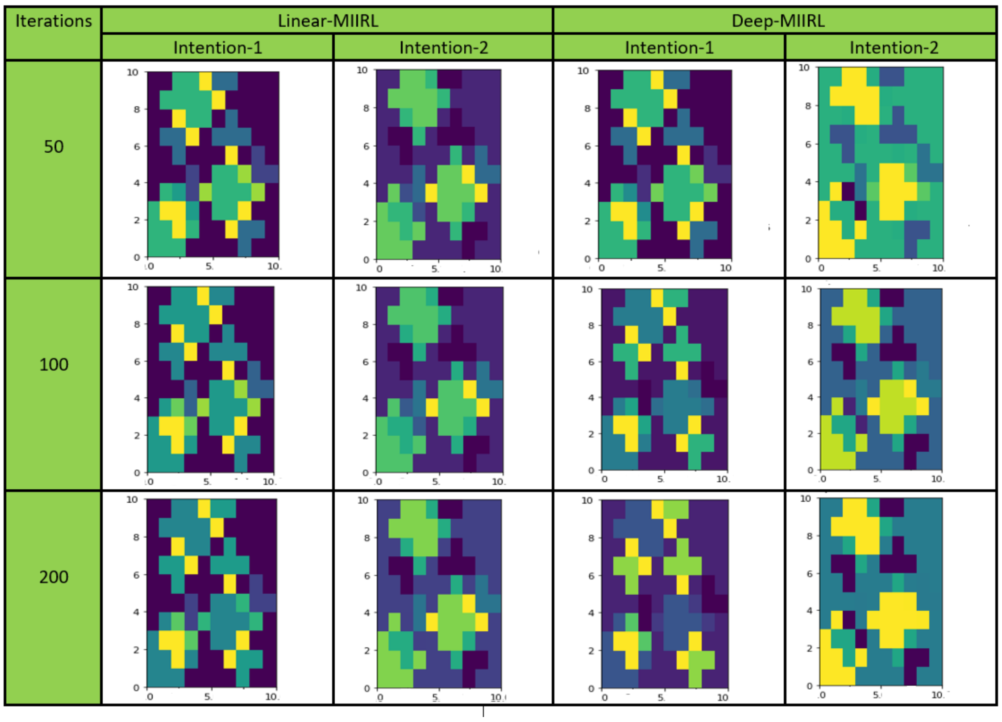
5.3.1 Feature: Effect range

Figure 5.4 compares results for experiments when Effect range is the feature fed to the algorithms. Comparing it to the ground-truth reward given above shows, both the method have obtained all the positive reward states(*yellower*) for 1st intention as early as at 50 iterations; while not able to recover negative reward states and neutral states quite perfectly until last iteration.
On the other hand for the 2nd intention, Deep-MIIRL have obtained positive reward states perfectly when Linear-MIIRL has not achieved the comparable results. In case of negative reward states, both

[1]https://youtu.be/4DQNBIdrE7M

methods shows identical performance, while for neutral states Deep-MIIRL can be seen to have obtain almost all the states correctly which is not the case for the other method.

Figure 5.4: Results(Feature- Effect Range)[2]

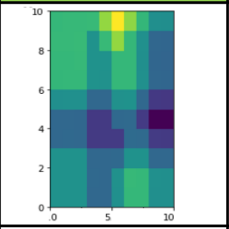
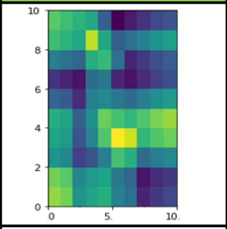
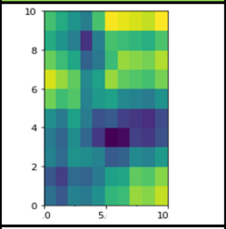
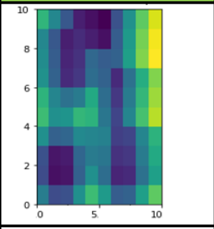
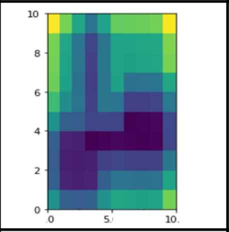
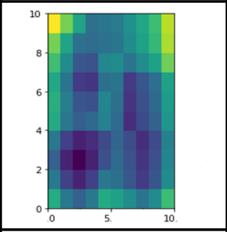
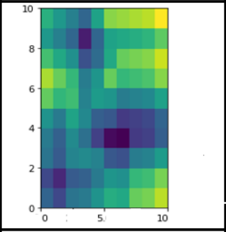
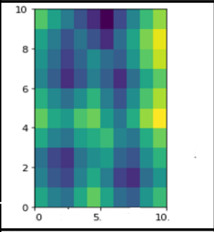
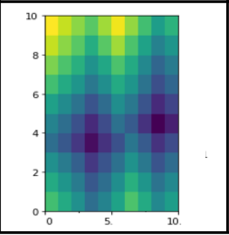
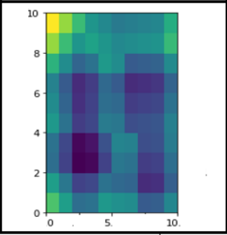
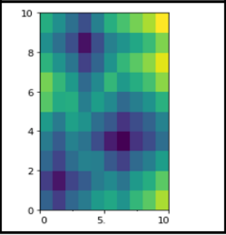
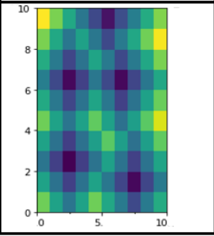


5.3.2 Feature: Manhattan Distance

Figure 5.5 compares results for experiments when Manhattan Distance is the feature fed to the algorithms. Comparing it to the ground-truth reward given above shows that, for both the intentions Linear-MIIRL is unable to achieve satisfactory performance to identify either of positive, negative or neutral state rewards. On the other hand, although Deep-MIIRL has also been not able to identify the correct complete regions for all the rewards, it has found the exact position of objects which placement is causing negative reward regions (These positions are highlighted in dark blue) i.e. in case of 1st intention, the positions of red objects and in case of 2nd intention, the positions of green objects.

[2]The color-maps continue with the same assumptions about colors and numbers, as stated in section 5.2.3

Figure 5.5: Results(Feature- Manhattan Distance)[3]

Iterations	Linear-MIIRL		Deep-MIIRL	
	Intention-1	Intention-2	Intention-1	Intention-2
50				
100				
200				

5.3.3 Training Time

Both the algorithms are also compared against the time required to complete 50, 100 and 200 iterations. We noted that, Linear-MIIRL results began to converge around 120-130 iterations, while for Deep-MIIRL this number was around 150-180. Hence, 200 iterations is chosen as threshold to allow some margin and keep the experiments comparable.

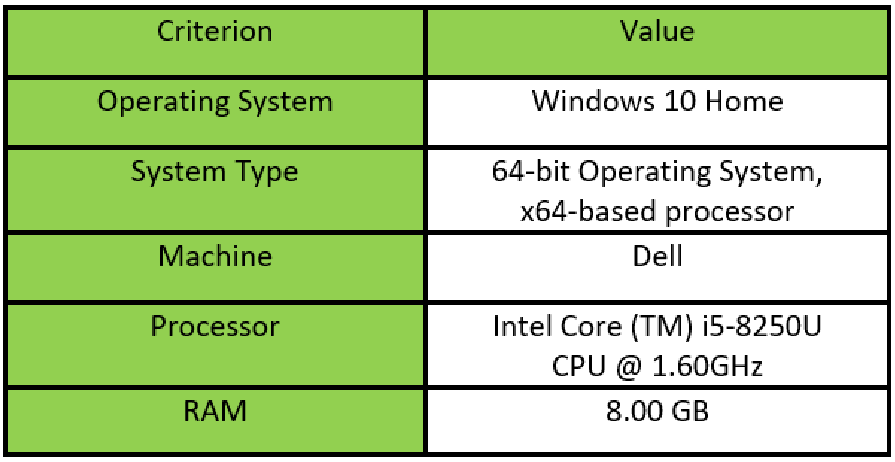
Figure 5.6: Training Time in minutes

Iterations	Linear-MIIRL	Deep-MIIRL
50	1.52	4.57
100	3.20	7.35
200	5.13	13.03

[3]The color-maps continue with the same assumptions about colors and numbers, as stated in section 5.2.3

Figure 5.6 compares the time for both the experiments to complete various iterations. Deep-MIIRL is evidently taking much longer than the other method, as expected due to computationally heavy task of training the network. The system configuration of the machine on which the experiments are performed is given in Figure 5.7.

Figure 5.7: System Configuration



Criterion	Value
Operating System	Windows 10 Home
System Type	64-bit Operating System, x64-based processor
Machine	Dell
Processor	Intel Core (TM) i5-8250U CPU @ 1.60GHz
RAM	8.00 GB

5.4 Discussion

This chapter detailed the experiments performed to evaluate the proposed Deep-MIIRL against previously proposed Maximum Entropy based Linear MIIRL. Objectworld simulation environment is used in the process with 10 objects of 3 colors, which generates two types of rewards. The demonstrations carried out by the expert targeting these two rewards, are fed to both these methods along with two features.

We observe that, although Deep-MIIRL could not extract the exact ground-truth, its performance is superior than the other method in case of both the features. This highlights the consistency of the method across the features. Both methods are almost equivalent in the case when the feature is very precise (Effect range). On the other hand, Deep-MIIRL outperforms in case of vague features (Manhattan Distance) by identifying the exact positions of the objects which caused negative reward regions. The linear method failed to identify either of positive, negative or neutral regions in this case.

Inability to obtain certain reward areas (e.g. neutral and negative reward state with Effect range feature) by Deep-MIIRL might be related to number of demonstrations provided and features considered. More efforts can be put to reason appropriately about this phenomenon. For example, experiments with different features and various numbers of demonstrations can be performed to understand if it is the number of demonstrations or the feature specification, that is affecting the outcome.

Even though it needs further corroboration from different experiments involving more complex environments, range of different features and more number of intentions; the evaluation, however, gives a hint of the potential of the proposed method to handle multiple intentions and non-linear rewards across the features. Previously, these two issues were handled separately by two different methods. Our approach combined these two into a single method, which is a small step towards designing a common method to handle all the issues mentioned in section 1.1.4. Further research can be carried out to incorporate the remaining two issues with the proposed method. .

Chapter 6

Conclusion

This dissertation broadly concentrated on the problem of reward engineering in reinforcement learning and an approach proposed to handle it called Inverse Reinforcement Learning which estimates rewards by observing an expert. We addressed the specific case of IRL when expert demonstrations are intermix of different rewards i.e. generated having different intentions in mind and when reward is non-linear function of environmental features. As per the literature review performed as part of this project, these two issues were never handled in a single method before. We evaluated the proposed approach against a previously used method, which resulted in indicating positive signs regarding the potential of the approach. In this chapter, we overview the entire dissertation, briefly covering the contents of each chapter, the overall contribution and concluding with limitations of the approach and possible future undertakings.

6.1 Overview

The first chapter starts with introducing the basics of reinforcement learning field and how reward specification poses challenges to applicability of RL in real-world complex domains. It expounded on a distinctive approach of *learning-from-mentor* taken by IRL to tackle the challenge, the open issues for IRL including two of the issues we focused on during this work. It concluded with stating that an ideal IRL method would have the capability to tackle all these issues in single approach and hence, underlined the necessity of the research effort in this direction.

The next chapter established common related literature terms and concept, around which the dissertation is organized and the existing applications of the concepts. Various primary components of the later proposed approach are illustrated here which we refer in many places. The literature review highlighted the need for efforts to combine various approaches proposed to handle the issues in IRL into a single approach. At the same place we introduced Deep Neural Network, its characteristic which is used in the design and its applications.

The design targeted two issues of IRL mentioned above and proposed the combination of Expectation-Maximization technique and DNN to tackle multiple intentions and non-linear rewards respectively. The larger framework of EM algorithm performs iterative sorting of demonstrations into separate clusters, where each cluster represents an intention. The single intention IRL is applied on each cluster to estimate rewards of intentions they represent. We chose DNN based Single intention IRL at this place to tackle non-linear rewards, by extending the reward hypothesis proposed under popularly used Max-Ent based Linear IRL.

The proposed design is implemented under the existing framework for RL and IRL which provides various ready primary component. Some of these are improved upon to suit our purpose and some were added as required by the design. The implementation chapter laid the component-wise and action-sequence-wise explanation of the system, along with interfaces signature and the snapshots from the code wherever necessary.

The classical Objectworld environment is chosen to evaluate the proposed approach against previous Max-Ent based Linear IRL. The object placement in the environment generated two types of rewards, which the author as an expert chased. These interactions of the expert with environment are stored in log file and fed to the two methods along with two types of environment features. The results showed that both the methods are comparable when clear features are available and recovered most of the ground-truth. On the other hand, even though the proposed method was not able to recover most part of the ground-truth, in case when the feature was vague, it certainly outperformed the previous approached by recovering the positions of the objects causing negative reward regions.

The evaluation gave the hint of the potential of the proposed approach to handle both of the mentioned issues viz. handling multiple intentions and non-linear rewards, in a single process. There remains a few gray areas in the approach and in relation with it. The below section lists some of them. Nonetheless, the proposed method combining two of the IRL issues in single process, is a small step towards an ideal IRL method which is able to handle all of the issues mentioned in section 1.1.4 in single process.

6.2 Limitations & Future Work

The scope and time constraints of the project left many open areas regarding the experiments performed as part of it as well as in the design. The project can be extended in future on these fronts. Some of the related points are noted here in brief.

Number of Intentions

The experiments involved extracting only two intentions at this point. Although the design, in theory, is capable to handle more than two intentions, actual experiments with more intentions would further corroborate this factor. Also, the proposed method requires the number of intentions beforehand. It might not always be possible to provide this information while working in real-world domains. Further exploration is required in the Expectation-Maximization part which carries responsibility of handling intentions.

Different Environments

Current evaluation environment i.e Objectworld, as mentioned earlier, is a primary tool for assessing new approaches in Reinforcement Learning. It provided certain elementary factors for the purpose. There are more complex simulation environment in literature e.g. highway car driving, which involves high dimensional state spaces, immensely non-linear rewards which would facilitate thorough evaluation. The only things which need to be changed in current implementation is state-space definition and the way to record expert demonstrations. Such environments would also enable testing using range of features.

Limited Demonstrations

As observed. our design required considerable number of demonstrations in order to achieve a fair performance. Every domain might not permit this facility of executing as many demonstrations as required. Design must cope up with available demonstrations, generalize them and keep updating current conjecture as new data comes in. The literature study revealed a few experiments which deals with solving IRL with few demonstrations e.g. Xu et al. [45]. The proposed design has scope for improvement on this front.

Improved Networks

DNN used is implemented with elementary settings to validate its usability in handling non-linear rewards. If fine-tuned with all its assets such as parallel computations using GPU, faster optimizers etc., it could fill the time lag against Linear-MIIRL. Faster computation would make the design more suitable for high-dimensional state spaces as well as for life-long learning applications.

Number of iterations

Lastly, for the scope of project, we stop iterations either at certain number or as we get close enough to ground-truth. This is possible as ground-truth is available to us. While applying IRL, we will need a better mechanism to decide when to stop iterations. One way would be to check average change in reward values for each state after each iteration (or after some number of iterations) and stop iterating if it remains below a threshold for certain period. Another approach could be to sample trajectories using the policy under current reward function and compare those against the expert demonstrations provided. Iterations can be stopped when difference between them becomes minimal.

Appendix A

Maximum Likelihood Estimation

This section explains a probabilistic technique viz. maximum likelihood estimation which is the basis for the IRL approaches such as Maximum-Likelihood IRL, Multiple Intentions IRL etc. The technique finds the best suited parameter values for a statistical model that maximizes the probability of observing given data.

A statistical model is the description of a process that resulted in generation of some observed data. For example, one may design a model to predict the students' grade from the time they took to complete the exam. Such a model can be represented linearly as, $y= mx + c$, where y is predicted grades, x is exam completion time, and m & c are called the model parameters which are the actual things to be determined in the creation of a model. Different values of these parameters would give the different linear models as depicted in the figure A.1.

So parameters define the blueprint for a model. It is only when specific values are chosen for parameters, that we get an instantiation for a model that describes a given phenomenon. The MLE seeks the parameter values, such that they maximize the likelihood that the process described by a model produced the observed data. To illustrate further, suppose we observed 10 data points representing the exam completion time in minutes for students. These 10 data points are shown in the figure A.2.

One first has to decide which model best describes the process of generating the data. Lets assume

Figure A.1: Similar Models with different parameters

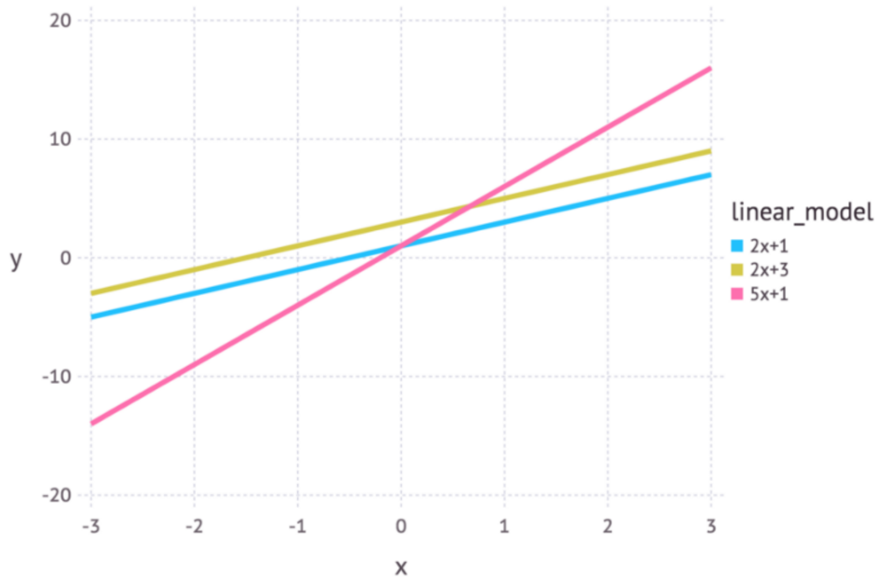
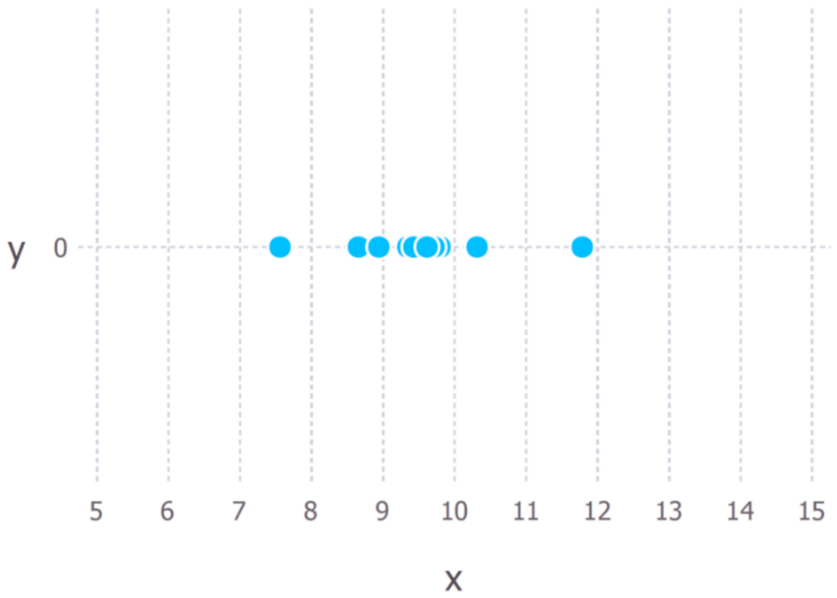
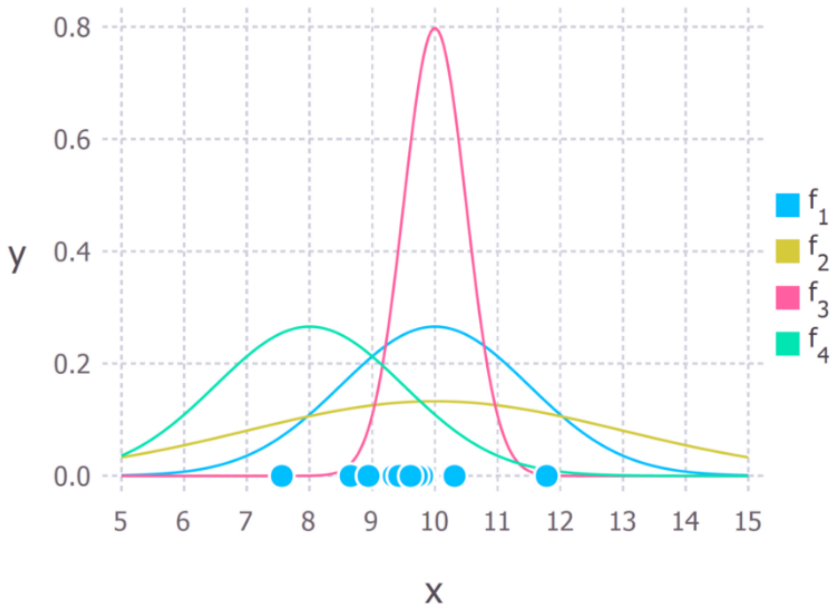


Figure A.2: The 10 Observed points



that the data generation process can be adequately described by a Gaussian (normal) distribution. Visual inspection of the figure suggests that a Gaussian distribution is plausible because most of the 10 points are clustered in the middle with few points scattered to the left and the right. The Gaussian distribution has 2 model parameters; the mean, μ, and the standard deviation, σ. Different values of these parameters result in different curves (like the straight lines above). Maximum likelihood estimation is a method that will find the values of μ and σ that select the curve that best fits the data from all the other possible curves (Figure A.3).

Figure A.3: The possible distributions' curves



Now that it is intuitively clear what MLE is, let's move on to actually find out those parameter values. The probability density of observing a data point can be given by Gaussian distribution formula as:

$P(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}exp(\frac{-(x-\mu)^2}{2\sigma^2})$

The joint probability of observing all the 10 points is given by:

$P(x_1, x_2..x_{10},;\mu,\sigma) = \prod_{i=1}^{10} \frac{1}{\sigma\sqrt{2\pi}}exp(\frac{-(x_\text{i}-\mu)^2}{2\sigma^2})$

Now we are left with figuring out the values for μ and σ which would result in finding the maximum value for the above equation. This can be achieved by calculating maxima using calculus. Just calculating the partial derivatives of the above equation with respect to μ and σ, setting it to zero and rearranging it to find a parameter of interest would result in MLE values of the parameters.

As can be seen, differentiating the above equation becomes complicated as we repeatedly need to apply Chain-Rule in order to solve the each and every term. So instead of directly differentiating the above terms, it can be simplified by applying logarithm on both sides, which converts the multiplication terms into summation and therefore each term can be solved individually without applying the Chain-Rule.

$log(P(x_1, x_2..x_{10}, ; \mu, \sigma)) = \sum_{i=1}^{10} log(\frac{1}{\sigma\sqrt{2\pi}} exp(\frac{-(x_i-\mu)^2}{2\sigma^2}))$

$\frac{\partial log(P(x_1,x_2..x_{10},;\mu,\sigma))}{\partial \mu} = 0$

$\frac{\partial log(P(x_1,x_2..x_{10},;\mu,\sigma))}{\partial \sigma} = 0$

Solving above equation results in MLE values of μ, σ.

The Gaussian model used on the illustrative example above can be replaced with any other model and MLE values for the respective model's parameters can be approximated. In case of Maximum Entropy IRL, the maximum entropy takes place of Gaussian model and reward function parameters take place of μ and σ.

Appendix B

Expectation-Maximization Algorithm

Maximum Likelihood Estimation (MLE), the mathematical method explained in Appendix A, is the way to determine model parameters when all the data is available. For the situations when not all the data is either available or observable, the need of deriving model parameters is fulfilled by Expectation-Maximization (EM) algorithm- that is to say EM is advance version of MLE used in more complex situations. While MLE considers all the data and finds the most-likely model parameters for that data; EM first guesses the missing data from the available data and then using whole of the data to update model parameters.

Steps:

1. Model parameters are initialized randomly. Incomplete data is provided as input, with the assumption that the data is generated under above model.

2. *Expectation:*

   Guess missing related data from the available data i.e. updating the variables.

3. *Maximization:*

   Use complete data from *Expectation* step to twist the model parameters to make that data most-likely possible under those parameters i.e updating the hypothesis

4. Repeat steps 2 and 3 until values converge or upto certain iterations.

The algorithm is guaranteed to be *stable* i.e. estimates would keep on improving with iterations. Due to this characteristic, it has formed the basis for unsupervised machine learning methods. It has also been employed for estimating latent variables, hidden markov decision models' parameters. Along with these benefits, it also comes with some downside such as slow convergence, getting stuck in local optima etc. which requires careful design measures to cope with.

Appendix C

Links

The codebase for the project can be found at the below GitHub link:

https://github.com/pparas007/dissertation-irl

A short video of the author (assuming the role of an expert) carrying out the demonstrations. These demonstrations are used as data for experiments:

https://youtu.be/4DQNBIdrE7M

Bibliography

[1] Pieter Abbeel and Andrew Y. Ng. "Apprenticeship learning via inverse reinforcement learning". In: *ICML*. 2004.

[2] Pieter Abbeel et al. "An Application of Reinforcement Learning to Aerobatic Helicopter Flight". In: *NIPS*. 2006.

[3] Matthew Alger. *Inverse Reinforcement Learning*. 2016. DOI: `10.5281/zenodo.555999`. URL: `https://doi.org/10.5281/zenodo.555999`.

[4] Brenna Argall et al. "A survey of robot learning from demonstration". In: *Robotics and Autonomous Systems* 57 (2009), pp. 469–483.

[5] Monica Babes-Vroman et al. "Apprenticeship Learning About Multiple Intentions". In: *ICML*. 2011.

[6] Chris L. Baker, Rebecca Saxe, and Joshua B. Tenenbaum. "Action understanding as inverse planning". In: *Cognition* 113 (2009), pp. 329–349.

[7] Kenneth D. Bogert and Prashant Doshi. "Multi-Robot Inverse Reinforcement Learning Under Occlusion with State Transition Estimation". In: *AAMAS*. 2015.

[8] Abdeslam Boularias, Jens Kober, and James Peters. "Relative Entropy Inverse Reinforcement Learning". In: *AISTATS*. 2011.

[9] Holger Caesar et al. "nuScenes: A multimodal dataset for autonomous driving". In: *arXiv preprint arXiv:1903.11027* (2019).

[10] Daniel Dewey. “Reinforcement Learning and the Reward Engineering Principle”. In: *AAAI Spring Symposia*. 2014.

[11] Chelsea Finn, Sergey Levine, and Pieter Abbeel. “Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization”. In: *ArXiv* abs/1603.00448 (2016).

[12] Willem Eduard Frankenhuis, Karthik Panchanathan, and Andrew G. Barto. “Enriching behavioral ecology with reinforcement learning methods”. In: *Behavioural processes* (2018).

[13] Adam Gleave and Oliver Habryka. “Multi-task Maximum Causal Entropy Inverse Reinforcement Learning”. In: 2018.

[14] Matthew J. Hausknecht and Peter Stone. “Deep Recurrent Q-Learning for Partially Observable MDPs”. In: *AAAI Fall Symposia*. 2015.

[15] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2 (1989), pp. 359–366.

[16] Edwin T. Jaynes. “Information Theory and Statistical Mechanics”. In: 1957.

[17] Ming Jin and Costas J. Spanos. “Inverse Reinforcement Learning via Deep Gaussian Process”. In: *CoRR* abs/1512.08065 (2015). arXiv: `1512.08065`. URL: `http://arxiv.org/abs/1512.08065`.

[18] Beomjoon Kim and Joelle Pineau. “Socially Adaptive Path Planning in Human Environments Using Inverse Reinforcement Learning”. In: *International Journal of Social Robotics* 8 (2016), pp. 51–66.

[19] Edouard Klein et al. “A Cascaded Supervised Learning Approach to Inverse Reinforcement Learning”. In: *ECML/PKDD*. 2013.

[20] Edouard Klein et al. “Inverse Reinforcement Learning through Structured Classification”. In: *NIPS*. 2012.

[21] Henrik Kretzschmar et al. “Socially compliant mobile robot navigation via inverse reinforcement learning”. In: *I. J. Robotics Res.* 35 (2016), pp. 1289–1307.

[22] Henrik Kretzschmar et al. “Socially compliant mobile robot navigation via inverse reinforcement learning”. In: *The International Journal of Robotics Research* 35.11 (2016), pp. 1289–1307. doi: 10.1177/0278364915619772. eprint: https://doi.org/10.1177/0278364915619772. url: https://doi.org/10.1177/0278364915619772.

[23] Sergey Levine, Zoran Popovic, and Vladlen Koltun. “Feature Construction for Inverse Reinforcement Learning”. In: *NIPS*. 2010.

[24] Sergey Levine, Zoran Popovic, and Vladlen Koltun. “Nonlinear Inverse Reinforcement Learning with Gaussian Processes”. In: *NIPS*. 2011.

[25] Yuxi Li. “Deep Reinforcement Learning: An Overview”. In: *CoRR* abs/1701.07274 (2017).

[26] Xiaomin Lin, Peter A. Beling, and Randy Cogill. “Multi-agent Inverse Reinforcement Learning for Zero-sum Games”. In: *ArXiv* abs/1403.6508 (2014).

[27] Hongzi Mao et al. “Resource Management with Deep Reinforcement Learning”. In: *HotNets*. 2016.

[28] Bernard Michini, Mark Cutler, and Jonathan P. How. “Scalable reward learning from demonstration”. In: *2013 IEEE International Conference on Robotics and Automation* (2013), pp. 303–308.

[29] Bernard Michini and Jonathan P. How. “Bayesian Nonparametric Inverse Reinforcement Learning”. In: *ECML/PKDD*. 2012.

[30] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518 (2015), pp. 529–533.

[31] P. Read Montague et al. “Bee foraging in uncertain environments using predictive hebbian learning”. In: *Nature* 377 (1995), pp. 725–728.

[32] Thibaut Munzer et al. “Inverse Reinforcement Learning in Relational Domains”. In: *IJCAI*. 2015.

[33] Gergely Neu and Csaba Szepesvári. "Apprenticeship Learning using Inverse Reinforcement Learning and Gradient Methods". In: *UAI*. 2007.

[34] Andrew Y. Ng and Stuart J. Russell. "Algorithms for Inverse Reinforcement Learning". In: *ICML*. 2000.

[35] Xinlei Pan et al. "Human-Interactive Subgoal Supervision for Efficient Inverse Reinforcement Learning". In: *AAMAS*. 2018.

[36] Deepak Ramachandran and Eyal Amir. "Bayesian Inverse Reinforcement Learning". In: *IJCAI*. 2007.

[37] Nathan D. Ratliff, David Silver, and J. Andrew Bagnell. "Learning to search: Functional gradient techniques for imitation learning". In: *Autonomous Robots* 27 (2009), pp. 25–53.

[38] Stuart J. Russell. "Learning Agents for Uncertain Environments (Extended Abstract)". In: *COLT*. 1998.

[39] Richard S. Sutton and Andrew G. Barto. "Reinforcement Learning, Second Edition An Introduction. MIT Press, second edition". In: (2018).

[40] Umar Syed and Robert E. Schapire. "A Game-Theoretic Approach to Apprenticeship Learning". In: *NIPS*. 2007.

[41] "The World's Most Valuable Resource Is No Longer Oil, But Data". In: *The Economist* (2017).

[42] Tomer Ullman et al. "Help or Hinder: Bayesian Models of Social Goal Inference". In: *NIPS*. 2009.

[43] Adam Vogel et al. "Improving Hybrid Vehicle Fuel Efficiency Using Inverse Reinforcement Learning". In: *AAAI*. 2012.

[44] Markus Wulfmeier, Peter Ondruska, and Ingmar Posner. "Maximum Entropy Deep Inverse Reinforcement Learning". In: 2015.

[45] Kelvin Xu et al. *Few-Shot Intent Inference via Meta-Inverse Reinforcement Learning*. 2019. URL: https://openreview.net/forum?id=SyeLno09Fm.

[46] Xiangxin Zhu et al. "Do We Need More Training Data?" In: *International Journal of Computer Vision* 119 (2015), pp. 76–92.

[47] Brian D. Ziebart et al. "Maximum Entropy Inverse Reinforcement Learning". In: *AAAI*. 2008.

[48] Brian D. Ziebart et al. "Navigate like a cabbie: probabilistic reasoning from observed context-aware behavior". In: *UbiComp*. 2008.

[49] Brian D. Ziebart et al. "Planning-based prediction for pedestrians". In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2009), pp. 3931–3936.