Teaching Agents with Deep Apprenticeship Learning

by

Amar Bhatt

A Thesis Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science
in Computer Engineering

Supervised by:

Assistant Professor Dr. Raymond Ptucha
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York

June 2017

Approved by:

Dr. Raymond Ptucha, Assistant Professor
Thesis Advisor, Department of Computer Engineering

Dr. Ferat Sahin, Professor
Committee Member, Department of Electrical Engineering

Dr. Iris Asllani, Assistant Professor
Committee Member, Department of Biomedical Engineering

Dr. Christopher Kanan, Assistant Professor
Committee Member, Department of Imaging Science

Louis Beato, Lecturer
Committee Member, Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Teaching Agents with Deep Apprenticeship Learning

I, Amar Bhatt, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Amar Bhatt

Date

Dedication

This thesis is dedicated to all those who have taught and guided me throughout my years...

My loving parents who continue to support me in my academic and personal endeavors. Their emphasis on respect, diligence, and never giving up crafted me as a student and the person I am today.

My teachers, professors, coaches, gurus, and senseis whose commitment to excellence, student development, and higher education brought light to what they taught me. Thank you for being my guide and imparting your knowledge on me.

Acknowledgments

I am grateful ...
For my advisor, Dr. Ptucha who brought me under his wing as a young undergraduate. His guidance, mentorship, and ability to unlock true potential shaped me as a student and an academic. I thank him for trusting me with Milpet, and being there for me in my journey through higher education.
For my thesis committee which is filled with a number of professors from different fields. Their diverse knowledge represents my passion for multidisciplinary research. They each have contributed to my academics in unique ways, and for that I am grateful.
For my friends for staying up with me late nights and collaborating on projects. To Luke Boudreau thank you for being a great friend and teammate. To Felipe Petroski Such, you are one of the most intelligent people I know, thank you for all of your insight. To Radha Mendapra for continually supporting me through my highs and my lows.
For my parents who funded my pursuit into higher education and whose unwavering support gave me the strength to continue forward.
For RIT, for giving me the opportunity to succeed both as an academic and as a student leader. This University's commitment to student excellence is incredible, and rare.

Abstract

Teaching Agents with
Deep Apprenticeship Learning

Amar Bhatt

Supervising Professor: Dr. Raymond Ptucha

As the field of robotic and humanoid systems expand, more research is being done on how to best control systems to perform complex, smart tasks. Many supervised learning and classification techniques require large datasets, and only result in the system mimicking what it was given. The sequential relationship within datasets used for task learning results in Markov decision problems that traditional classification algorithms cannot solve. Reinforcement learning helps to solve these types of problems using a reward/punishment and exploration/exploitation methodology without the need for datasets. While this works for simple systems, complex systems are more difficult to teach using traditional reinforcement learning. Often these systems have complex, non-linear, non-intuitive cost functions which make it near impossible to model. Inverse reinforcement learning, or apprenticeship learning algorithms, learn complex cost functions based on input from an expert system. Deep learning has also made a large impact in learning complex systems, and has achieved state of the art results in several applications. Using methods from apprenticeship learning and deep learning a system can be taught complex tasks from watching an expert. It is shown here how well these types of networks solve a specific task, and how well they generalize and understand the task through raw pixel data from an expert.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

When born, animals and humans are thrown into an unknown world forced to use their sensory inputs for survival. As they begin to understand and develop their senses they are able to navigate and interact with their environment. The process in which we learn to do this is called reinforcement learning. This is the idea that learning comes from a series of trial and error where there exists rewards and punishments for every action. The brain naturally logs these events as experiences, and decides new actions based on past experiences. An action resulting in a reward will then be higher favored than an action resulting in a punishment. Using this concept, autonomous systems, such as robots, can learn about their environment or how to do tasks in the same way. Reinforcement learning is used in many applications aimed to reflect the way a human/animal's brain learns/reacts [22]. Some of these applications include the inverse pendulum, mountain car problem [33], robotic navigation, and decision based systems. While traditional classification methods can assist in task mimicking, they do not perform well when introduced to new environments or new tasks. Reinforcement learning does not mimic the correct move, but instead develops a policy to choose actions to obtain the highest reward based on the current state.

Reinforcement learning techniques such as Q-Learning, are useful when the cost function is known, but this is not the case for many complex systems. Due to the vast number of variables in tasks such as driving, walking, or medical procedures; cost functions must be learned before they are applied. To do this, inverse reinforcement learning, or apprenticeship learning, is used. Apprenticeship learning extracts an action policy by observing an

expert's transitions over time and determining the cost function from the observations [24]. It is a Markov decision process, in that the result of the next action is based solely on the present state and not states that were observed previously. Therefore, inverse reinforcement learning sets to optimize the set of learned features in a particular state. Traditional inverse reinforcement learning uses this optimization of the linear combination of these features to determine the cost function of the given state. For static environments the traditional inverse reinforcement learning techniques are viable, however this is not the case for dynamic environments [36]. For dynamic networks, state features are constantly changing, which requires that the reward function and features be updated. Unfortunately, this no longer gives the most optimal solution but only an approximation [36].

These traditional methods do not scale well to systems with a large number of states. For example, if using images as an input for reinforcement learning or apprenticeship learning, the Q-table would be very large, and often times unsustainable. Therefore, combining these algorithms with deep learning where the rewards for state-action pairs are learned in a neural network, systems can be as complex or as large as needed [19]. Combining deep learning with reinforcement learning opens up a new dimension of learning, where complex correlations of environment and state features can be utilized to learn complex tasks.

The rest of this thesis will be structured as follows; Chapter 1 will end with a summary of the novel contributions in this research,Chapter 2 will explore past and current research in both reinforcement learning and apprenticeship learning, Chapter 3 will discuss the dataset, environment, and technologies used in experimentation, Chapter 4 will describe modifications to current apprenticeship architectures including new architectures and novel contributions, Chapter 5 will run-through the details on proposed architecture implementations and algorithms, Chapter 6 will show and explain the results each of the proposed architectures achieved with the datasets, and Chapter 7 will summarize the research and explore modifications to be made in future work.

The novel contributions and areas of exploration of this research are as follows:

- Reward Abstraction: Task level abstraction of rewards ($+$ if complete, $-$ if failed, 0 otherwise) for training on expert data and training the agent.
- Scheduled Shared Experience Replay: Combining expert experiences and agent experiences with the amount of expert experiences used in replay decreasing over multiple epochs.
- Target Q-Network Implementation: Implementing methods in DQN of stabilizing Q-networks with a target network updating frequency.
- Dueling DQN Utilization: Utilizing the concept of split value and advantage Q-values within expert data training and agent training.
- DRQN Comparison: Utilizing the concept of recurrent neural networks and LSTMs to learn sequential tasks by adding an LSTM layer to the end of the Dueling DQN architecture. This will then be compared to the Dueling DQN architecture without the LSTM layer.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement learning is used to teach systems based on trial and error. It uses a reward and punishment based cost function to help determine a policy for which to choose its next action. It is based around how living creatures learn using primitive reflexes at a young age. This methodology creates a balance between random decisions (exploration) and choosing the best learned policy over time (exploitation). These algorithms work with agent(s), state-action pairs, and policies. An agent is the system being operated on, such as, a car, robot, humanoid, or game solver. States represent different modes of the system, they can be any combination of environment or system specific features, such as, location of agent, location of obstacles, distance from goal, time, or speed. Actions can be anything that influences a transition from one state to another state, such as moving the agent, throwing at a target, or picking up an object. Actions will lead to some reward/punishment that will help train the system, such as an increase/decrease in score. Choosing an action at a specific state is called the system's policy. This is what will be learned using Reinforcement Learning. Therefore, reinforcement learning algorithms optimize over a known cost function and output a policy.

2.1.1 Temporal Difference Learning

Temporal difference learning is a branch of reinforcement learning that attempts to model the way an animal learns by predicting a reward in terms of a given stimulus. The core of this algorithm comes from its prediction

error signal, which constantly adjusts expected rewards of a state [25]. This algorithm estimates the value function of a state, allowing it to estimate the value at each state for each action taken. If this value were not estimated, and instead calculated to be exact, the agent would need to wait until a terminal state is reached before updating state and action reward values. This calculation is inefficient, and impossible to compute unless the value at every state was computed at the same time. The estimation of the value over time allows for incremental updates to the states value as more information is given to the system. The non-estimated value function is shown in (2.1) [9].

$$V^*(s_t) = V(s_t) + \alpha[R_t - V(s_t)] \tag{2.1}$$

This function returns a new value (*V**) when given a state (*s*) at time step (*t*) which is represented by s_t. The value $V(s_t)$ is the current value or reward at a specific state. If $V(s_t)$ is very high, then that state will carry a higher importance upon evaluation. Therefore, when evaluating this type of algorithm, the next states with the largest values will be chosen to be transitioned to. The learning rate, α, is used to determine the proportion of update. If α is very large, then the update at each time step, *t*, will also be large. R_t is the total reward of the system when proceeding from state s_t to a terminal state. This non-estimating function assumes that the final reward function R_t exists as shown in (2.2) [19].

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{n-t} r_n \tag{2.2}$$

The temporal difference estimation of the value function is shown in (2.3) [9]. The immediate reward received at s_t is defined by r_t, therefore the reward for a state one time step out (s_{t+1}) is r_{t+1}. The discount rate, γ, represents the weight (between 0 and 1) that the next state value will be multiplied with. With a discount rate of 1, the entire value of the next state will be considered in the state value update. A discount rate of less than 1 will decrease the importance of the value of the next state in the state value update. This function only accounts for the next immediate state. After one iteration through all of the states, the values for all states will be incorrect due to this one step update, therefore it takes multiple iterations to obtain

the optimal state values.

$$V^*(s_t) = V(s_t) + \alpha[r(s_t) + \gamma V(s_{t+1}) - V(s_t)] \tag{2.3}$$

Estimating in this way allows for the continuous update of state values as more information is provided. As shown, the major difference between the original value function and its estimate is the consideration of the reward in the next step as well as the difference in expected reward between the current state and next state. The $r(s_t)$ value in (2.3) is the estimated reward for the action selected at that state in current time, whereas R_t in (2.1) is the total reward from a state to a terminal state.

Temporal difference learning requires several iterations, called episodes, to converge. An episode starts at any given state, but must end at a terminal state. A system can have more than one terminal state (e.g. end of a game, reaching the goal state, etc.). Examples of the usage of the value estimation equation (2.3) are shown in subsequent sections.

There are two types of Temporal Difference Learning methods; Off-Policy and On-Policy. Off-Policy methods, such as Q-learning, can update state value functions using actions that have not been tried. On-Policy methods, such as Sarsa, update state value functions using only actions it has tried. Both methods choose a different balance between exploration and exploitation. Exploration is the idea of choosing random actions to learn the entire state-space. Exploitation is choosing only those actions that proved to be high in reward.

2.1.1.1 Off-Policy: Q-Learning

Q-learning is an off-policy method of temporal difference learning. It is the one of the most widely used algorithms in this domain. The Bellman Equation shown in (2.4) [19] is used to update its value function denoted as *Q*.

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma * max_{a_{t+1}}(Q(s_{t+1}, a_{t+1})) \tag{2.4}$$

As shown, the *Q-value* at state s_t and action a_t at time step *t* is based on the current value and the next best state-transitions value multiplied by

the discount factor (γ). In Q-learning, the *Q(s,a)* values are represented as a table with states as its rows and actions at its columns. Therefore, each Q-value at a particular state-action pair is the total reward at that state assuming that optimal actions are taken from that state onward. This requires knowledge of future states and rewards. Therefore, this algorithm requires several episodes starting at different states to reach convergence and update the Q-table [38]. Unlike other neural-network based applications, Q-Learning is unsupervised and must learn an environment dynamically, which is equivalent to creating its own dataset rather than having one provided. Typically, this causes massive overhead when scaling this algorithm to larger environments, causing the traditional Q-Learning methods to be unpopular [26]. However, advances in this algorithm have generated new ways to define its policy making it capable to scale to larger environments, such as applying an ϵ-greedy method for environment exploration [1].

To iteratively update Q-values over time, the value function in (2.5) is used.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r(s_t, a_t) + \gamma * max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.5}$$

In this equation, α is the learning rate usually held between 0 and 1. If this value is set to 1 then the Q-value is updated fully without considering the previous value, $Q(s_t, a_t)$, which will cause the algorithm to learn very quickly. However, this may result in divergence of the algorithm. If α is set to 0 then the Q-value is not updated. Therefore, to ensure the system is learning and can converge to a solution α needs to be set between 0 and 1. The exact value of the learning rate needs to be determined through experimentation. Over many episodes, the update to the $Q(s_t, a_t)$ value will result in no change, because the difference between $r(s_t, a_t) + \gamma * max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$ and $Q(s_t, a_t)$ will be close to 0. At this point, the system knows it can stop learning because the $Q(s_t, a_t)$ values have saturated.

This algorithm learns soft policies, meaning it has a high factor of exploration. Off-Policy algorithms update value functions using assumed actions which have not been tried, as shown by taking the maximum valued action

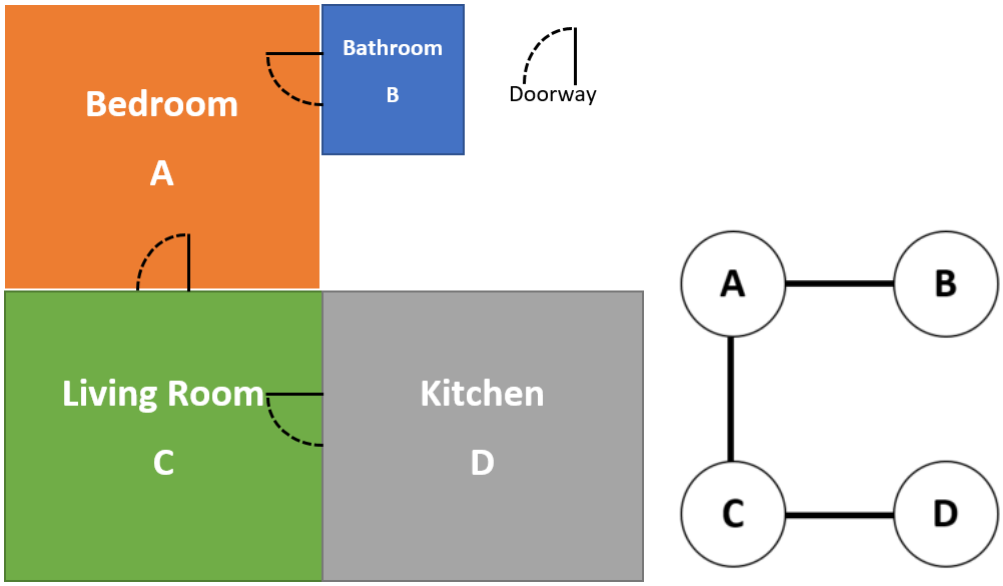


Figure 2.1: Floor plan for a one-story house for temporal difference learning example as a picture (left) and graph (right).

at any given state even if the action has never been tried. This allows the agent to learn actions it did not actually perform [9]. This shows that this algorithm favors exploration more so than exploitation as will be shown in the example below.

2.1.1.1.1 Q-Learning Example

Suppose there exists an environment that is a one-story house with four rooms (which can be considered states); bedroom (A), bathroom (B), living room (C), kitchen (D), as shown in Fig. 2.1. With this environment comes a "hungry" agent who is trying to navigate to the food in the kitchen. Therefore, the kitchen is the goal, so if the agent is put into any room it will need to learn to go to the kitchen using an optimal path. The agent must transition to another room, unless it is at the terminal state. Therefore, a reward is introduced to each room transition, as shown in Fig. 2.2. As shown, all doorways are two-way and any doorway leading to the kitchen is given a reward of 100, all other transitions are given a reward of zero. The kitchen is a terminal, absorbing state, which is denoted by the self-rewarding loop at that state. This reward graph can also be shown as a transition table with

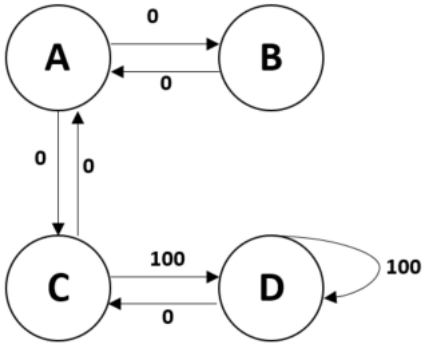


Figure 2.2: Rewards graph for room transitions for Fig. 2.1.

R	Possible next state of agent				
Current state of agent		A	B	C	D
	A	-	0	0	-
	B	0	-	-	-
	C	0	-	-	100
	D	-	-	0	100

Figure 2.3: State transition table with reward values. A "$-$" sign denotes that there does not exist a state transition. For example, state A cannot go to state D.

rewards as shown in Fig. 2.3 and is denoted as R. A Q-matrix is then created in the same format as the state transition table with rewards. This matrix is initialized with zeros, which shows that at iteration zero (Q^0) the agent has no information, this matrix is shown in Fig. 2.4. Say that the agent starts in the bedroom (state A) and transitions to the living room (state B), using the Bellman equation (2.4) [19] with γ equal to 0.9, the Q-value for state A will be updated as:

Q^0	Possible next state of agent				
Current state of agent		A	B	C	D
	A	0	0	0	0
	B	0	0	0	0
	C	0	0	0	0
	D	0	0	0	0

Figure 2.4: Q-table initialized to zero denoting the lack of information the agent has at time step zero.

Q^1	Possible next state of agent				
Current state of agent		A	B	C	D
	A	0	0	**0**	0
	B	0	0	0	0
	C	0	0	0	**100**
	D	0	0	0	**100**

Figure 2.5: Q-table update after first episode iteration.

$Q(A, C) = R(A, C) + \gamma * max_a Q(C, a) = 0 + 0.9 * 0 = 0.$
The Q-value for this state does not change. Now the agent is in state C or the living room. Say from random selection the agent chooses to transition to state D. The Q-value for state C will be updated as:
$Q(C, D) = R(C, D) + \gamma * max_a Q(D, a) = 100 + 0.9 * 0 = 100.$
The agent is now in state D and it chooses to transition back to state D, causing the update to look like:
$Q(D, D) = R(D, D) + \gamma * max_a Q(D, a) = 100 + 0.9 * 0 = 100.$
These updates can be shown in Fig. 2.5. The agent is now in the terminal state so its location resets to explore additional states.

The agent is now in state B or the bathroom, and it transitions to state A. The update for the Q-value at state B is:
$Q(B, A) = R(B, A) + \gamma * max_a Q(A, a) = 0 + 0.9 * 0 = 0.$
The agent is now in state A or the bedroom, and it transitions to state C. The update for the Q-value at state A is:
$Q(A, C) = R(A, C) + \gamma * max_a Q(C, a) = 0 + 0.9 * 100 = 90.$
The agent is now in state C or the living room, and it randomly transitions to state D. The update for the Q-value at state C is:
$Q(C, D) = R(C, D) + \gamma * max_a Q(D, a) = 100 + 0.9 * 100 = 190.$
The agent is now in state D and it chooses to transition back to state D, causing the update to look like:
$Q(D, D) = R(D, D) + \gamma * max_a Q(D, a) = 100 + 0.9 * 100 = 190.$
These updates can be shown in Fig. 2.6. The agent is now in the terminal state so it resets to a random state again.

If the agent is in state B again, and it transitions to A the update for the Q-value at state B is:

Q^2	Possible next state of agent				
		A	B	C	D
Current state of agent	A	0	0	**90**	0
	B	**0**	0	0	0
	C	0	0	0	**190**
	D	0	0	0	**190**

Figure 2.6: Q-table update after second episode iteration.

Q^3	Possible next state of agent				
		A	B	C	D
Current state of agent	A	0	0	90	0
	B	**81**	0	0	0
	C	0	0	0	190
	D	0	0	0	190

Figure 2.7: Q-table update after third episode iteration.

$Q(B, A) = R(B, A) + \gamma * max_a Q(A, a) = 0 + 0.9 * 90 = 81.$

These updates can continue until the optimal path is found, after just three update iterations the agent has the optimal policy to get to the kitchen from any state by choosing the maximum Q-value for the next state, as shown in Fig. 2.7. As shown in this example, this algorithm is more inclined to update with actions it has never used, thus showing it prefers exploration over exploitation.

2.1.1.2 On-Policy: Sarsa

Sarsa is an on-policy method of temporal difference learning. In this case the value function in (2.6) [32] is updated according to a policy designed more around experience. As shown, this equation is updated by using the current policy (f^{π}) found to determine a_{t+1} instead of using the maximum value. Therefore, this algorithm does not favor exploration because it defaults to choosing next actions based on what it knows. This ties this algorithm into considering control and exploration as one entity, whereas its off-policy counterpart can separate the two.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r(s_t, a_t) + \gamma f^{\pi}(Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t)] \quad (2.6)$$

Unlike the Q-Learning value function equation, Sarsa does not consider the maximum value of an action at a given state, but instead takes actions that have been taken in the past [9]. This shows that this algorithm favors exploitation more so than exploration. This also allows this algorithm to populate Q-values with a closer true approximate than the Q-Learning algorithm [32]. Therefore, it can be said that while Q-Learning updates based on states, Sarsa updates in accordance to state-action pairs. This allows for a better convergence policy, and leads to less unnecessary exploration by the system at hand [33]. However, because this algorithm lacks in exploration, it may converge to a local minima or non-optimal solution.

2.1.1.3 Epsilon Selection Policies

Both off-policy and on-policy methods of temporal difference learning rely on the selection of states and actions which is called a policy. Policies are meant to find a balance between exploration and exploitation. Popular policies used in both Q-Learning and Sarsa are ϵ-greedy, ϵ-soft, and softmax. In both ϵ-greedy and ϵ-soft selection policies an action is chosen with the highest estimated reward for most of the iterations. They choose a random action with a probability of ϵ for ϵ-greedy and probability 1-ϵ for ϵ-soft where ϵ is a small value between 0 to 1. ϵ-greedy is chosen to limit exploration using ϵ between 0 to 0.5. This results in 0% to 50% of the trials having actions chosen at random. ϵ-soft is chosen to favor exploration using ϵ between 0 to 0.5. This results in 50% to 100% of the trials having actions chosen at random [9]. These ϵ values can be dynamically adjusted based on number of episodes completed, number of steps within an episode, or based on how well the algorithm is converging. Therefore, common practice is to have high exploration (random action) at the start of learning, and high exploitation towards the end of learning [1]. Unlike the ϵ selections, softmax (in terms of selection policies not neural networks) chooses random actions with respect to weights assigned to each action. These weights are adjusted

over time as the system learns. This makes it so undesirable actions are less likely to be chosen over time, whereas in the ϵ selections the worst action has the same probability as the best action of being randomly chosen. Each selection method has its own benefits, but none have proven to be better than the other [9].

2.1.2 Q-Learning and Sarsa Implementations

Using simulated sensory data from ultrasonic sensors, moisture sensors, encoders, shock sensors, pressure sensors, and steepness sensors, a robotic system will be able to make decisions on how to navigate through its environment to reach a goal. The robotic system will not know the source of the data or the terrain it is navigating. Given a map of an open environment simulating an area after a natural disaster, the robot will use model-free temporal difference learning with exploration to find the best path to a goal in terms of distance, safety, and terrain navigation. Two forms of temporal difference learning will be tested; off-policy (Q-Learning) and on-policy (Sarsa). Both of these algorithms will be tested and compared against using the simulated environment and sensory data. Through experimentation with several world map sizes, it is found that the off-policy algorithm, Q-Learning, is the most reliable and efficient in terms of navigating a known map with unequal states, as opposed to comparing it to the results of the Sarsa algorithm that chose more costly paths and would often not converge to a solution.

2.1.2.1 Application

In many cases robotic path planning is used in controlled environments, whether in a hospital, nursing home, hotel, etc. Reinforcement learning is also used to navigate known city maps and other open environments [26]. However, many of these cases fall into the category of finding the best path of a known map which can also be achieved with less invasive path planning algorithms. Consider the case of a city that underwent a natural disaster. While the city map has stayed the same, the terrain of the city may have been vastly altered. Clear roads may be flooded with water, and buildings

may have been knocked down forcing the creation of alternative paths. In this case, it will be pertinent for a land-based robotic system to efficiently navigate the city to reach potential resources and survivors in the safest way. If the system took the known path as shown on a city map before the disaster, the robot may get stuck, hurt, or take too much time. Understanding the landscape and making decisions based on the type of terrain can ensure an efficient and safe path is chosen.

2.1.2.2 Methods

To test the temporal difference learning algorithms, Q-Learning and Sarsa, MATLAB was used. An $M \times N$ matrix was created consisting of $M * N$ states. Each state had between 2-4 neighbors, depending on where on the grid it fell. Therefore, the agent could move right, down, left, and up to reach a new state. The agent would not be able to move off the grid. Each state also was associated to one of five terrain types; sand, forest, pavement, water, or mountainous rock/debris. These states represented an area on a map after a natural disaster. This meaning the defined paths from a start to a goal have been severely altered. An example map is shown in Fig. 2.8. Each terrain was associated to certain punishments, as decided by the simulated sensor readings on the robot. The sensors used were, encoders (speed detection), ultrasonic sensors (visibility), shock sensor (smoothness), moisture sensor, pressure sensor (stability), and steepness sensor. Each sensor reading was statically defined and normalized between 0 and 1, where 0 is the safest/most efficient and 1 is the most dangerous/least efficient for that particular sensor. Further testing of the system would call for the sensor readings to be introduced to natural noise.

The world map matrices were first predefined to initialize the two algorithms (defined path from start to goal) and to find appropriate parameter values for the algorithms (punishment weight, discount factor, goal reward, etc). The maps were then randomized for further testing. These parameter tuning results for optimal path selection are shown in Fig. 2.9 for Q-Learning and Fig. 2.10 for Sarsa. As shown, both were tested in an environment with punishment and without. The parameters for the two algorithms were then tuned until the optimal path was chosen. They both produce the

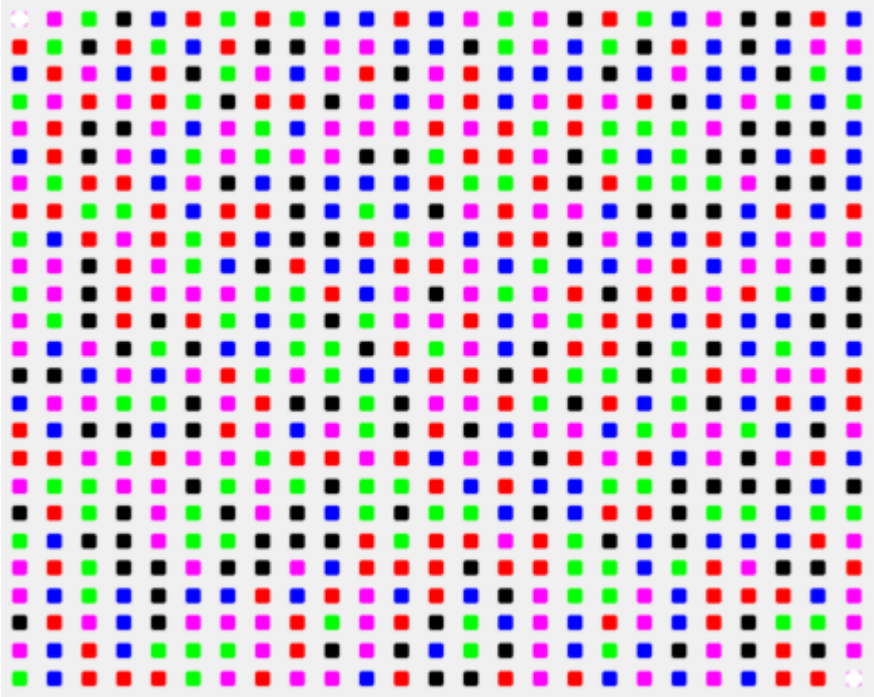


Figure 2.8: Map Example with Terrain (Sand = Magenta, Forest = Green, Pavement = Black, Water = Blue, Misc. Debris = Red, Start/Goal = White dots).

same results, but the parameters for them to do so were different between the two algorithms. The simulated robot navigating the map from the start (top left corner) to the goal (bottom right corner) had no prior knowledge of the terrain of the state it was in. It did know the map structure, however. As each algorithm was run, the value at each state was updated using a reward for the action chosen as well as a punishment which was determined by the sensor readings for each state. On a normalized average the punishments for each terrain type was as follows: $sand \rightarrow 1$, $forest \rightarrow 2$, $pavement \rightarrow 0$, $water \rightarrow 3$, $mountainousrock/debris \rightarrow 4$. This is much like the penalties incurred in James Sutton's puddle world problem [33], however, in this case the robot does not know about the terrain in its world, only what it reads through its sensors. Varying punishments made one state more favorable than another in terms of proximity to the goal, safeness, and efficiency. A rewards matrix of size $(M * N) \times (M * N)$ was created, where each row symbolized a state and each column symbolized an action (next state).

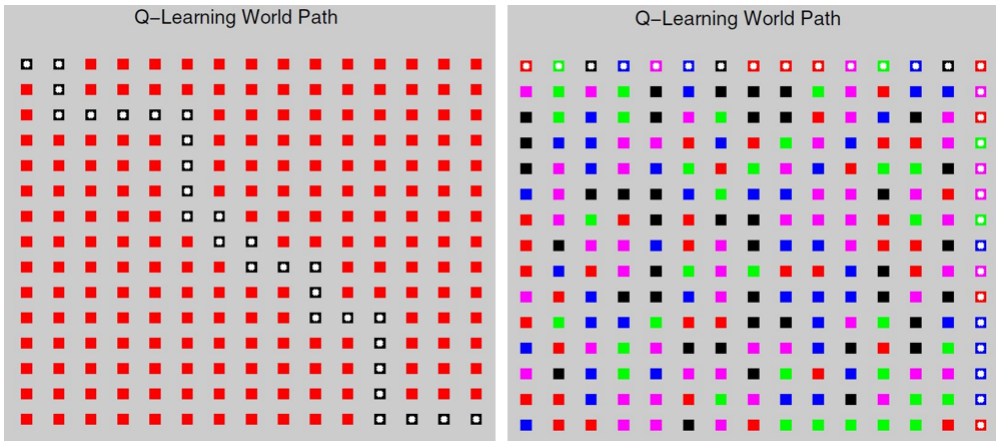


Figure 2.9: Parameter tuning for optimal path selection on controlled environments for Q-Learning. (left) Environment with punishment, (right) Environment without punishment.

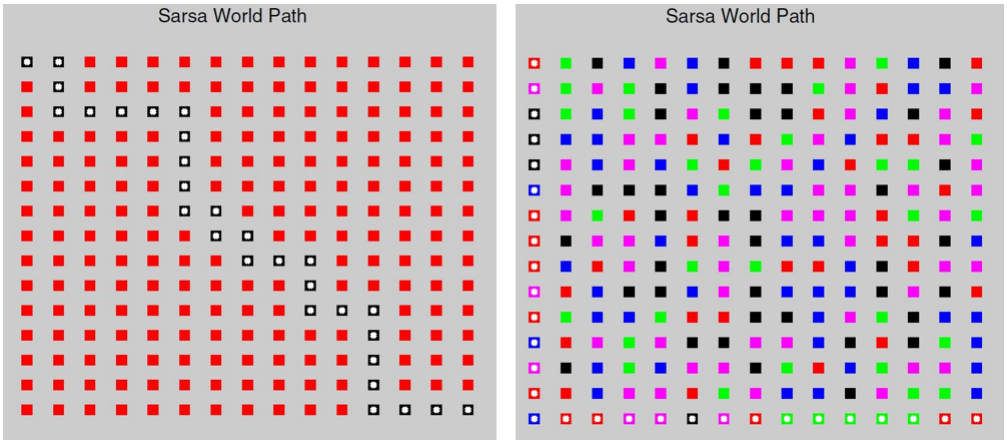


Figure 2.10: Parameter tuning for optimal path selection on controlled environments for Sarsa. (left) Environment with punishment, (right) Environment without punishment.

Each neighbor state in the actions column were given a value of 0, non-neighbors were given a value of -∞. The goal state and any state that was a neighbor to the goal received the highest reward value. A Q-matrix of size $(M * N) \times (M * N)$ was also created to store the values of each state to be updated by the temporal difference learning algorithms. Both the reward and Q-matrix represented the true map of the world without terrain information [2]. Other factors needing tuning were the goal reward, the discount factor, and the punishment weight. These were found for both algorithms through the control testing on a predefined map as shown in Fig. 2.9 for Q-Learning and Fig. 2.10 for Sarsa. Also, each algorithm was run for 10,000 episodes with an epsilon of 0.2 meaning that 20% of the episodes would have next state actions chosen at random under the ϵ-greedy selection policy. The updated Q-matrix is used to determine the best path based on the highest rewards for each action at a given state.

2.1.2.2.1 Q-Learning Algorithm

To implement the Q-learning algorithm, the high-level steps outlined in Algorithm 1 were used. As shown by the algorithm, a random state is chosen from the generated Q-matrix used to hold the values to determine the best path. Then an action is taken in the current state based on the ϵ-greedy selection policy. Using this action, the next state transition is determined as well as the reward for taking the action. These values then update the Q-matrix at the current state as described by (2.5). This is then repeated with the next state, until the current state is the goal. As shown in Algorithm 1, the Q-matrix at each state is updated based on a learning parameter, α which is set to 1 in this application to assure quick learning. This matrix also depends on a discount factor denoted by γ which is set to 0.8 to give importance to future rewards [9]. This was repeated for 10,000 episodes where at the start of each episode a random starting state was chosen.

2.1.2.2.2 Sarsa Algorithm

To implement the Sarsa algorithm, the high-level steps outlined in Algorithm 2 were used. As shown by the algorithm a random state is chosen

Algorithm 1 Q-Learning [9]

for e $\in$ number of episodes **do**
  Pick random state s
  while s is not the goal **do**
    Choose action a from s using epsilon policy from Q
    Take action a, observe reward r and next state s'
    Update Q-values as follows:
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r(s,a) + \gamma * max_{a'}(Q(s',a')) - Q(s,a)]$
    $s \leftarrow s'$

from the generated Q-matrix used to hold the values to determine the best path. Then an action is taken in the current state based on the ϵ-greedy selection policy. Then until the state present is not the goal, the action will be taken and the next state transition is determined as well as the reward for taking the action. Another action is chosen using the ϵ-greedy selection policy for the next state. The Q-matrix at the current state is then updated as described in (2.6). As shown in the algorithm, the Q-matrix at each state is updated based on a learning parameter, α which is set to 1 to assure quick learning. This matrix also depends on a discount factor denoted by γ which is set to 0.3 to give importance to future rewards [9]. This was repeated for 10,000 episodes where at the start of each episode a random starting state was chosen.

Algorithm 2 Sarsa [9]

for e $\in$ number of episodes **do**
  Pick random state s
  Choose action a from s using epsilon policy from Q
  while s is not the goal **do**
    Take action a, observe reward r and next state s'
    Choose action a' from s' using epsilon policy from Q
    Update Q-values as follows:
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r(s,a) + \gamma * (Q(s',a')) - Q(s,a)]$
    $s \leftarrow s'$
    $a \leftarrow a'$

2.1.2.3 Results

To validate the effectiveness of the two algorithms, each was run with a 5×5, 10×10, 15×15, and 25×25 size world with randomly generated terrains. Each was run in ten different terrain maps at each world size. The results from this can be found in Table 2.1. This table shows the average path length the agent took over the ten terrain maps. It also shows the total average cost for all ten different terrain maps, the average cost per step, the average convergence time, and percent convergence for all ten terrain maps. These parameters were measured for both the Q-learning and Sarsa implementations. As shown, it is apparent that the Q-Learning algorithm performs best in terms of cost. It chooses the shortest and safest path. Also, as the world size gets bigger, the Q-Learning algorithm takes advantage of the increase in space and terrain to lower its cost per move. Sarsa also does this, but only slightly. The Sarsa algorithm is the fastest, while it may not be the most efficient. It is shown to finish well before the Q-Learning algorithm on larger maps. However, the Q-Learning algorithm has a higher rate of convergence, where the Sarsa algorithm is prone to diverge at larger map sizes. In the 25×25 world size, the Sarsa algorithm failed to converge, increasing the discount factor for the algorithm at this world size to 0.8 allowed for convergence. Examples of paths found by the Q-Learning algorithm and Sarsa algorithm at a world size of 25×25 are shown in Fig. 2.11 and Fig. 2.12, respectively.

In consideration of natural disaster relief and rebuilding, the Q-Learning algorithm is the best. It takes the longest to converge, but will find the most efficient path in terms of time and safety for itself. Sarsa, tends to favor the goal over safety in these situations because of its exploitation over exploration strategy.

2.1.2.4 Conclusion

Reinforcement learning is a series of methods and algorithms used to mimic the way a living being makes decisions. Just like a living being, a decision cannot be proven right or wrong until it has been made [2]. This methodology gives way for a system to learn its environment and discover patterns not

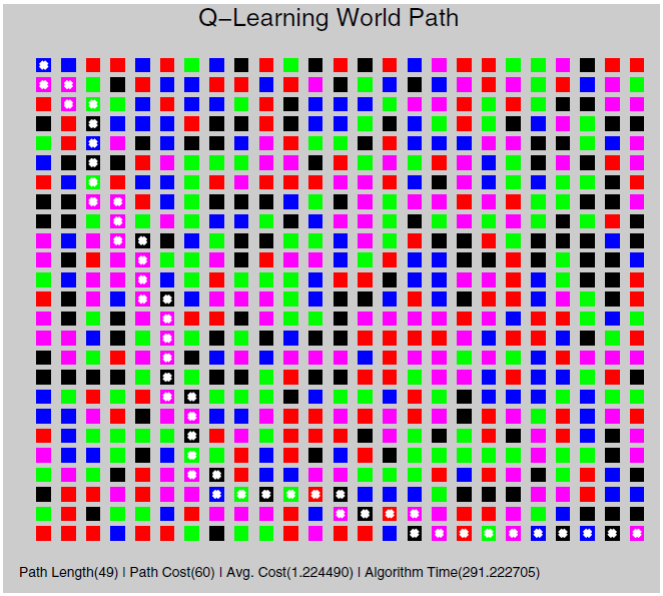


Figure 2.11: Q-Learning Algorithm Path Example with Punishment.

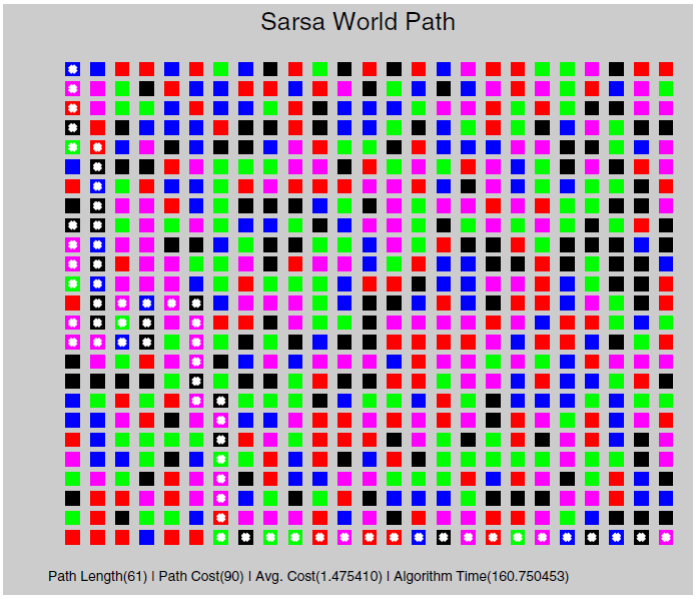


Figure 2.12: Sarsa Algorithm Path Example with Punishment.

Table 2.1: Q-Learning and Sarsa results across several world map sizes.

		Q-Learning	**Sarsa**
5x5	Avg. Path	9	9
	Avg. Cost	12.3	18
	Avg. Unit Cost	1.3667	2.000
	Avg. Time (s)	2.396	2.2731
	% Converge	100	100
10x10	Avg. Path	19	23.667
	Avg. Cost	23.4	38.8
	Avg. Unit Cost	1.2316	1.6260
	Avg. Time (s)	12.7255	12.2337
	% Converge	100	60
15x15	Avg. Path	29	60.75
	Avg. Cost	32.2	80.0
	Avg. Unit Cost	1.1103	1.3110
	Avg. Time (s)	37.8544	33.4482
	% Converge	100	40
25x25	Avg. Path	49	63.2*
	Avg. Cost	51.3	101*
	Avg. Unit Cost	1.0469	1.6005*
	Avg. Time (s)	337.8118	186.467*
	% Converge	100	50*
	*with a discount factor of 0.8		

easily recognizable. Used in the case of natural disaster response or ruins exploration, reinforcement learning, specifically temporal difference learning, can be used to explore the area, as well as build an efficient navigation map. In terms of the experiments implemented, Q-Learning performs better than Sarsa in creating an efficient and safe path from a start to a goal. Using multiple synchronized systems across a map, where each system would represent an episode, could reduce the time limitation found in Q-learning. In this type of problem, Q-Learning does the best because of the importance it places on exploration. On known, non-variable maps, Sarsa will do better because it can exploit rewards in future states early.

2.2 Deep Reinforcement Learning

An advantage to using neural networks in reinforcement learning, specificially Q-learning, is that it can replace the Q-table. For complex systems with numerous state-action pairs, maintaining a Q-table with that much data becomes infeasible. For example, Q-tables work well for "closed-world" problems, but not for typical "open-world" scenarios. Also, reinforcement learning stems from that fact that not everything can be observed. The demonstrator, or "expert", may not reach every state or may move in a stochastic fashion. Therefore, neural networks allow the system to generalize for unknown states and obstacles. Neural Networks are great for finding complex features in complex functions, therefore, the traditional *Q-table* is replaced by a network which approximates the value for each state-action pair [19]. The Neural Inverse Reinforcement Learning algorithm in [40] is a model-free implementation of deep reinforcement learning. It uses several sensors on the robot as inputs into the neural network. It also uses the current state, and any obstacle detected as inputs. Therefore, this method separates a robot's coordinates with its environment so that it can better generalize its navigation [40]. Beyond just using neural networks for advanced reinforcement learning techniques, deep networks have become common practice. Deep networks allow agents to learn environments without the need to handcraft features or to have full observability of the environment. Often these networks use image data and several network layers to learn complex

tasks [22].

Atari has been a common platform for Deep Reinforcement Learning. Specifically, Deep Q-networks (DQN) developed by DeepMind (owned by Google) use raw pixels and scores from classic Atari games to learn the cost function for the games to maximize the score [21]. Most neural network implementations of Q-learning are unstable or fail to converge due to non-linear functions. These non-linear functions result in complex games (such as in Atari) or tasks whereby the underlying reward function (or score) is determined by complex combinations of the state features. By using deep convolutional neural networks, complex state features can be learned to better model a non-linear reward function. Storing an agent's experience in replay memory also gives more stability to the network. Therefore, the system is generating its own dataset to learn from. This method applies Q-learning updates on mini-batches of experience that are drawn at random from a pool of samples generated by the system's own exploration. While this has been a proven method to increase the efficiency of the system, especially when using a GPU, this also allows the system to not get stuck in a Markov loop which causes divergence because of high correlations between state spaces. This idea was further built upon by the authors in a paper published one year later [20]. In this paper, the algorithm was modified to use multiple agents to learn rather than batch-processing (replay). This method further uncorrelated state spaces because different agents would be at different states. This method proved to converge much faster, and also learn unknown environments better [20]. However, having multiple agents in a non-simulated environment can prove to be a challenge, therefore batch-processing as introduced in [21] is a more feasible option.

Another implementation of this algorithm is Maximum Entropy Deep Inverse Reinforcement Learning from [39]. This algorithm applies wide convolutional layers to learn more relevant spatial features in the data it is trying to learn from. These wider convolutional layers essentially represent fully convolutional neural networks with width one. This essentially means that there is no pooling using this methodology to ensure every bit of raw input is used.

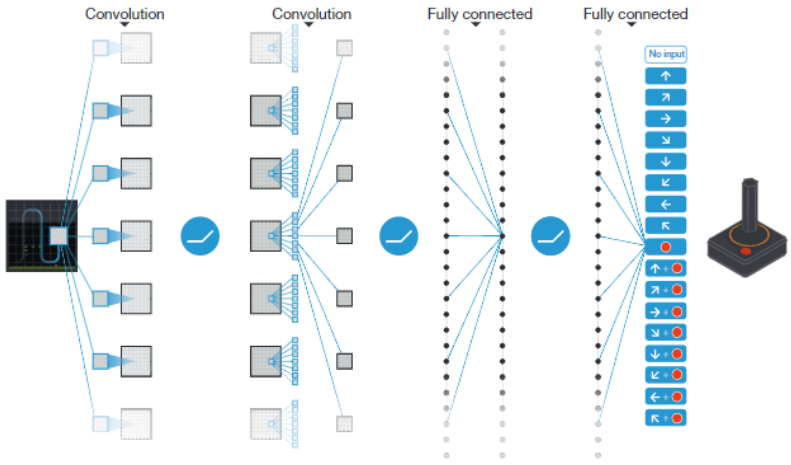


Figure 2.13: Deep Q-Network Architecture. The input consists of an 84x84x4 image. Each hidden layer is followed by a rectifier non-linearity (max(0,x)) [22].

2.2.1 Deep Q-Networks

Mnih et al. [22] pioneered the use of deep neural networks in reinforcement learning to learn a variety of Atari 2600 games. This work has been highly cited, and has led as a base for further advancements in deep reinforcement learning [22]. They use the idea of Deep Q-Networks (DQN) which uses a deep neural network to approximate the Q-table using (2.5). This network takes as an input raw image data from a game (state) and outputs an action. The score of the game at each given state is fed in as the reward function, and the weights are iteratively optimized to achieve the highest score. The game state image passes through several convolutional layers before going into a fully connected layer to determine the next action out of 18 possible actions, this is shown in Fig. 2.13. As shown, the 18 possible actions include 8 cardinal directions, a button press, and 8 cardinal directions with a button press. However, do to the unstable nature of reinforcement learning when using nonlinear function approximators (neural networks), this method has many faults. Correlations in sequences of observations may cause Q-values to drastically change during training causing the algorithm to diverge. Two methods used to overcome this instability are experience replay and a separate target network. Using these methods, DQN was able to provide a general framework to learn many Atari 2600 games at human

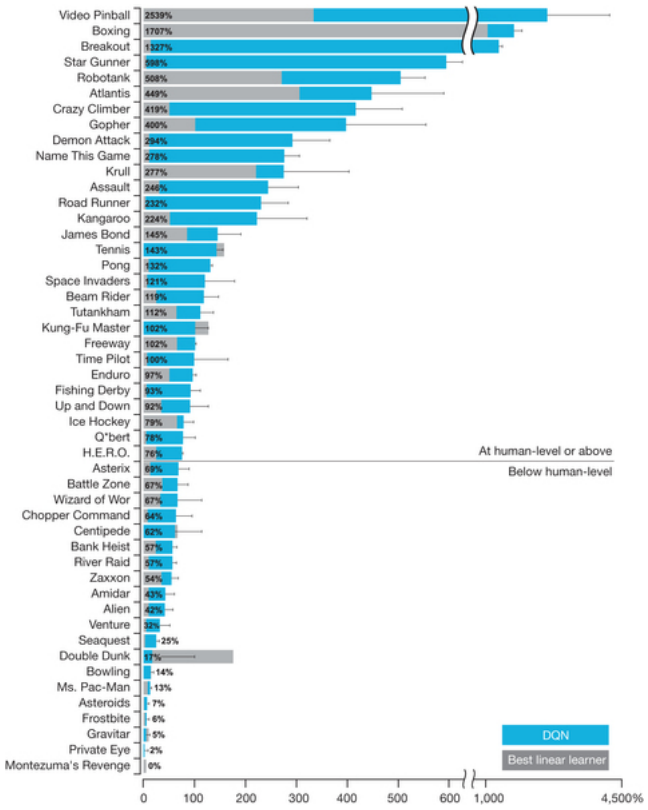


Figure 2.14: Deep Q-Network results on Atari 2600 games when compared to a linear learner [22].

or super-human levels as shown in Fig. 2.14. Basic games such as Pinball, Pong, Space Invaders, and Breakout out performed humans (super human level). More complicated games with multiple goals or sequence restricted navigation performed well below human level such as Ms. Pac-Man, Alien, and River Raid [22].

2.2.1.1 Experience Replay

Experience replay is inspired by nature in the idea that learning a task can be learned better if an agent (human or animal) uses their past learnings. Therefore, as the network is being trained experience data from the agent is stored in state-action-reward-nextState pairs ($< s, a, r, s^* >$). Batches of these experiences are drawn at random during training from an experience replay buffer which contains a set number of experiences. As a new experience is learned it is added to the buffer and the oldest replay is removed.

These batch experiences are used to update the weights of the network as the network navigates through a sequence of observations. This allows the network to learn from a variety of past experience instead of finding a local minima in the immediate episode it is learning in [22].

2.2.1.2 Separate Target Network

One main issue with Q-learning based reinforcement learning is that Q-values are constantly shifted by small amounts during every iteration. These constant shifts in values can cause a network to easily diverge, especially early on when the updates have a larger magnitude. Using a separate target network with Q values that is only updated periodically is one solution to this problem. The network continues to update Q-values iteratively, however, it will use the target network Q-values in its calculations. The target Q values are then updated periodically (adjustable hyper-parameter) with the current calculated Q-values in the network. This gives the system stability, and removes tightly coupled correlations from influencing the network weights [22].

2.2.2 Double Deep Q-Networks

One set back found from using the traditional DQN [22] is that it may overestimate Q-values for certain actions in a state [35]. This poses a problem if Q-values for actions were not overestimated equally which is usually the case. Therefore, during training there is a high chance that some suboptimal actions may be given a high Q-value early on which would cause the system to fall into a local minima. Therefore Hasselt et al. [35] developed a technique called Double DQN (DDQN) which utilized the network Q-values and target-Q values. In DQNs the max over Q-values is used to compute the target-Q value. In double DQN the Q-values from the primary network are used to choose an action while the target-Q network is used to generate the target Q-value for the chosen action. Therefore, the action choice and target Q-value generation are decoupled which reduces overestimation and provides greater stability. In [35] the DDQN outperformed DQN on every Atari 2600 game except for two as shown in Fig. 2.15. The

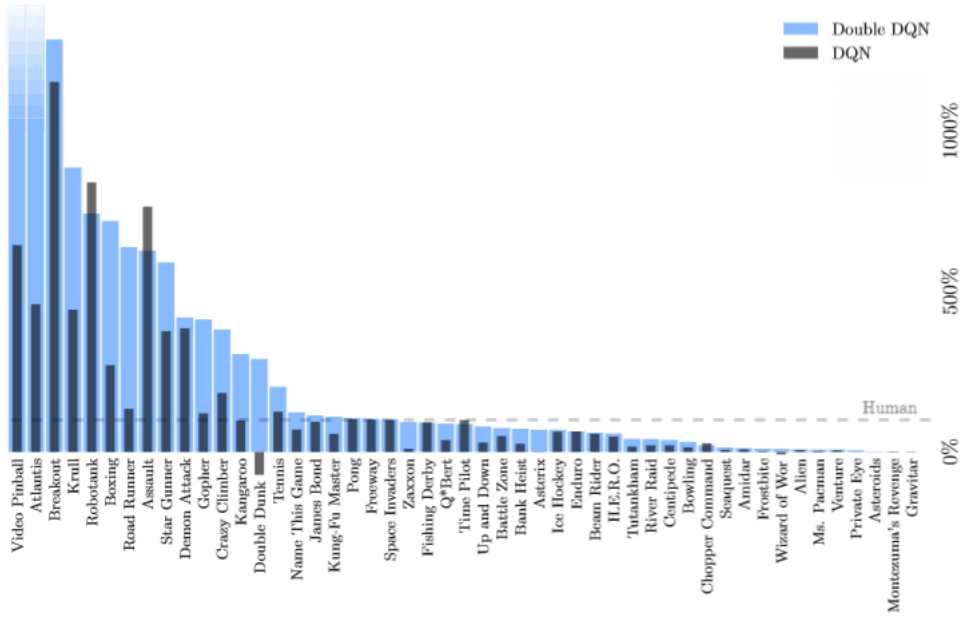


Figure 2.15: Double Deep Q-Network results on Atari 2600 games when compared to DQN [35].

DDQN was also able to achieve super human level in games where DQN could not.

2.2.3 Dueling Deep Q-Networks

Q-values directly correspond to how beneficial it is to take action (a) in a given state (s). Wang et al. [37] splits these Q-values into two separate values. One of the values is $V(s)$ which indicates how good it is to be in a given state. The second value is $A(a)$ which is the advantage function indicating the advantage of taking action a compared to the other possible actions. Therefore, Q is decomposed as show in (2.7) [37].

$$Q(s,a) = V(s) + A(a) \tag{2.7}$$

Therefore, dueling DQNs split the last layer in the DQN into two sub-networks, one to compute $V(s)$ and one to compute $A(a)$. The outputs of these networks are then combined to find the final Q-value ($Q(s,a)$) as shown in Fig. 2.16 [37].

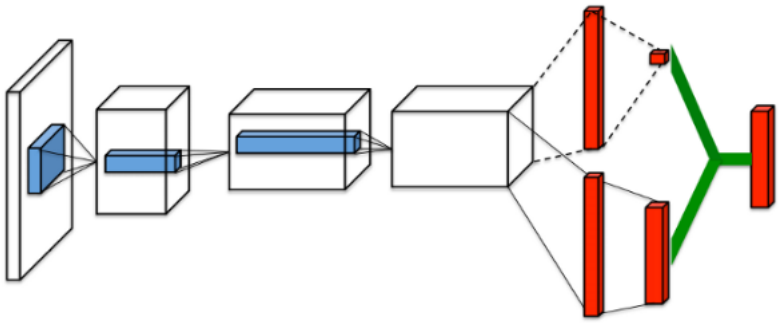


Figure 2.16: Dueling Deep Q-Network Architecture. Two networks are used and combined at the end [37].

The idea of separating Q-values into these two value functions is to be able to create robust state value estimates without having it be attached to a specific action. For example, consider an agent in a room where it receives a very high reward of being in the green zone and a very low reward of being in the red zone. If the agent is in the green zone it is highly rewarding to be in that zone, and no action needs to be taken to receive a reward. Therefore, it does not make sense in this case to consider the value of being in the green zone state being coupled with an action. By decoupling the value of being in a state and the advantage of taking an action, more complex and robust estimates can be made that allow for greater stability and faster learning. The Atari 2600 games were used as the benchmark for testing and it outperformed both DDQN and DQN architectures producing state-of-the-art results [37]. Fig. 2.17 shows the performance increase over the Double DQN [35].

2.2.4 Deep Recurrent Q-Networks

DQNs [22], DDQNs [35], and Dueling DQNs [37] allow the agent to have full access to the information of an environment. In other words, our system has full observability in that it is given full state information of the entire game state. For example, grid worlds and Atari 2600 games have all the information of the world in a single state (no scrolling game play or worlds). However most real world problems will not give an agent full observability. Assume an agent does not have access to all the information in

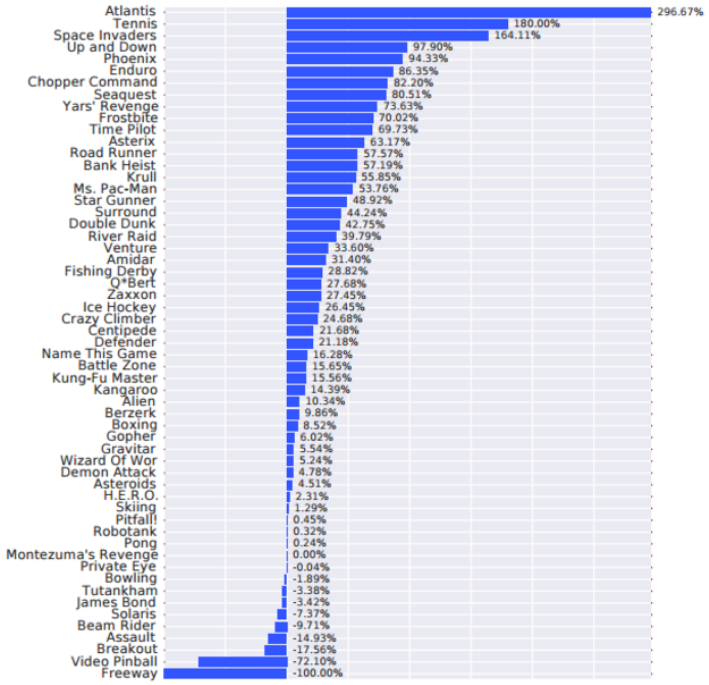


Figure 2.17: Dueling Deep Q-Network results on Atari 2600 games when compared to Double DQN [37].

a world (partial observability), traditional DQN methods will not be able to converge. For example, in cases where there are walls or doors an agent will not know the states that are on the other side. Also, while spatial limitations exist, temporal information is also crucial. Often, in single image inputs motion, speed, and direction can be lost to an agent. Even in DQN architectures, only 4 frames are used as an input, therefore in environments where past information is even more crucial these 4 frames greatly limit what can be learned. Environments where all information is not available to an agent are called Partially Observable Markov Decision Processes (POMDPs).

Hausknecht et al. [11] found that the performance of DQNs decline when trying to learn a POMDP and could be better learned using recurrent neural networks. This created the Deep Recurrent Q-Network (DRQN) architecture. Instead of passing in a series of images as input to the network, DRQNs take in a single image. The difference lies in the first fully-connected layer being replaced with a recurrent LSTM layer. This change can be seen in Fig. 2.18 [11].

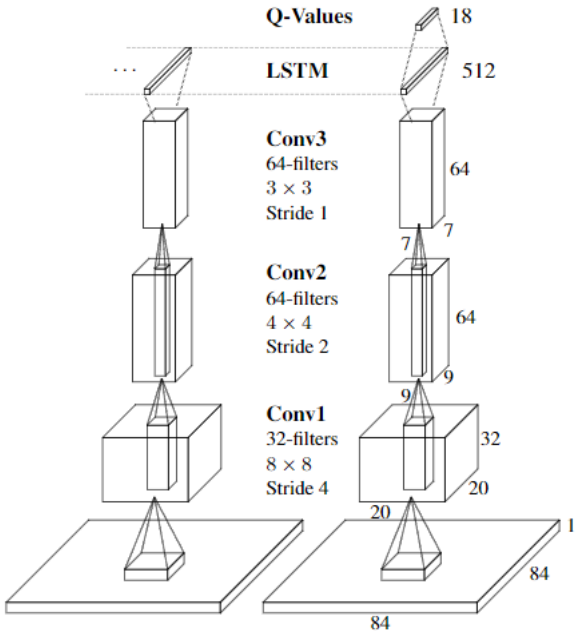


Figure 2.18: DRQN Architecture [11].

As shown, the LSTM layer is inserted just before the value and advantage layers are calculated the same as in dueling DRQNs. Another change in this network is in the experience replay. Instead of selecting a random batch of experiences, DRQNs train on random batches of sequence of experiences at a set length. This is important so that sequences can be learned which is crucial to the recurrent nature of DRQNs. Using this network with the Atari 2600 framework, DRQNs showed improvement on POMDP style game play as shown in Fig. 2.19 [11]. As shown, while the DRQN did well in some games, the original DQN architecture still beat it on many Atari games. The DRQN performed the best on POMDP style games such as Frostbite and Double Dunk. However, one issue with using a recurrent layer is that it breaks the MDP rule that reinforcement learning follows; the next state only depends on the current state. However, DRQNs make next state decisions based on previous states and actions, causing it not to adhere to this rule.

Lample et al. [16] made a modification to this network to encourage only accurate gradients were being propagated through the network. Using the replay memory in sequences at a set trace length, only half of the errors of

Game	DRQN $\pm std$	DQN $\pm std$ Mnih et al.
Asteroids	1020 (±312)	1629 (±542)
Beam Rider	3269 (±1167)	6846 (±1619)
Bowling	62 (±5.9)	42 (±88)
Centipede	3534 (±1601)	8309 (±5237)
Chopper Cmd	2070 (±875)	6687 (±2916)
Double Dunk	**-2** (±7.8)	-18.1 (±2.6)
Frostbite	**2875** (±535)	328.3 (±250.5)
Ice Hockey	-4.4 (±1.6)	-1.6 (±2.5)
Ms. Pacman	2048 (±653)	2311 (±525)

Figure 2.19: DRQN results on Atari 2600 games when compared to the DQN architecture [11].

the that sequence are propagated back through the network. More specifically, the first half of the trace errors are not used. This was shown to cut-down on non-accurate information from updating the network. Often the first set of moves in a trace are random, where as the last moves in a trace are more prone to be going on the right track. By using this method, it was shown that the Atari game Doom was played at a much better rate than traditional DRQN and Dueling DQN [16]. However, the size of the trace, number of sampled episodes, and percentage of trace used in updates are hyper-parameters that need to be adjusted for each application.

2.3 Apprenticeship Learning

Apprenticeship Learning is a subset of inverse reinforcement learning. Inverse reinforcement learning is the process of learning a cost/reward function that explains a set of demonstrations. In these cases, the system observes and attempts to learn from another system that performs some task. Unlike reinforcement learning, the cost-function (reward) does not need to be known. Therefore, this type of system takes in an expert's policy (through demonstration) and learns a cost function based on its own state features [13]. Apprenticeship learning is the process of learning a policy which can model and generalize observed demonstrations. Often this is obtained through the learned reward function from inverse reinforcement

learning. Both methods utilize and learn from a set of expert demonstrations. Often these terms are interchangeable, however the primary difference is that inverse reinforcement learning's end goal is to learn a reward and apprenticeship learning's end goal is to learn a policy.

Hamahata et al. [10] create an imitation learning (inverse reinforcement learning) algorithm that is used in two ways, one with supervised learning and one with reward shaping. In supervised learning, the imitator observes a demonstrator's motion and attempts to model it. One issue with this is that this is a black box learning technique. The imitator can observe the final state of the demonstrator, but it has no knowledge of hidden states and actions used to create the motion. This can be mitigated if the imitator knows the inverse control model of the demonstrator system. The inverse control model would allow the imitator to accurately replicate the the motions of the demonstrator to get to the final state. Hamahata et al. [10], assumes that the imitator obtains the estimated actions over a discrete time from the demonstrator. This then allows the imitator to find the optimal control to imitate the demonstrator using least squares or ridge regression. In reward shaping a more implicit approach to imitation learning is used. This method attempts to create a reward function from the observations of the demonstrator. In many cases, changing a rewards function also changes the optimal policy of the system. However, the rewards function found is designed to be included as an aggregate term to the underlying rewards function. The additional rewards function is defined as (2.8) [10].

$$r^{\text{sub}}{}_{\text{t+1}} = \gamma\phi(x_{t+1}) - \phi(x_t) \tag{2.8}$$

Where x is the state, t is the time step, γ is the discount factor, and r^{sub} is the rewards function. The value of this is added to the value of the underlying rewards function. In the additional rewards function, the ϕ term is defined when it is equal to the optimal value function. Using this shaping method the imitator has a faster learning rate [10].

One challenge of inverse reinforcement learning is limited demonstrations and not performing the exact task as the demonstrator. In these cases,

traditional imitation learning is not a feasible option. In [6] a method is proposed to learn an optimal policy rather than just a reward. This means that the system learns at the task level rather than just matching patterns which is useful for cases where the exact motion of the demonstrator is not learned. This paper explores the inverse pendulum problem, and attempts to solve it using reinforcement learning. This approach proved to be limited by intricately complex movements, however it did demonstrate a sense of learned movement from demonstration [6].

2.3.1 Bayesian Inverse Reinforcement Learning

One method of solving an inverse reinforcement learning problem is using a Bayesian process. Ramachandran et al. [29] created a Bayesian Inverse Reinforcement Learning (BIRL) algorithm that allowed for imperfect and incomplete expert data. By using a probability distribution the uncertainty present in expert data sets can be modeled. This allows for complicated tasks to be learned since the transitions from state to state are not only stochastic but may be due to a complex reward function. To do this, a posterior distribution is derived for the rewards obtained from a prior distribution and a probabilistic model of the expert's actions given by the reward function. Given a set of observations (O), from an expert (X), two observations can be made: the expert is attempting to maximize its total accumulated reward, R, and the expert executes on a fixed policy that does not change over time or based on a decision. Therefore, the Bayesian probability of k observations given the reward can be assumed independent as shown in (2.9) [29].

$$Pr_X(O_X|R) = Pr_X(O_1|R)Pr_X(O_2|R)...Pr_X(O_k|R) \tag{2.9}$$

In terms of the expert, the goal is to maximize the accumulated reward which is equivalent to performing the action which causes a states Q^* value to be maximum. Using this, the more confident we are in the expert's ability to choose a good action (denoted as α_X) the likelihood increases on the prediction that the expert would choose action a in state s. This is shown in (2.10) [29].

$$Pr_X(O_X|R) = \frac{1}{Z} e^{\alpha_X \sum_i Q^*(s_i,a_i,R)} \tag{2.10}$$

Where, $\sum_i Q^*(s_i, a_i, R)$ is the expected value of the observations (s_i, a_i) using R. Z is a normalizing constant. To compute the posterior probability of the reward function Bayes theorem is used as shown in (2.11) [29].

$$Pr_X(R|O_X) = \frac{Pr_X(O_X|R)Pr(R)}{Pr(O_X)} = \frac{1}{Z'} e^{\alpha_X \sum_i Q^*(s_i,a_i,R)} * Pr(R) \tag{2.11}$$

To use this in apprenticeship learning tasks a policy loss function must be defined. The loss function (2.12) [29] for learning a policy π is based on some norm (p) of the difference between the optimal values for each state achieved using the optimal policy for R ($V^*(R)$) and the values for each state achieved using the learned policy for R_π ($V^\pi(R)$). The goal is to find the π that will minimize the expected policy loss over the posterior distribution of R.

$$L^p_{policy}(R|\pi) = \|V^*(R) - V^\pi(R)\|_p \tag{2.12}$$

One challenge lies in the computation of a posterior distribution for R at a specific point. This is because this calculation relies on calculating the optimal Q-function which is not efficient. Therefore a modification is made to release this constraint. As the algorithm learns it keeps track of the policy (π) that is optimal for the current infered reward (R). Using a uniform sampling ($\tilde{R}$) from the neigbors of R the Q-values for π are computed for all state-action pairs (s, a), $Q^\pi(s, a, \tilde{R})$. If the Q-value for a state using π, $(Q^\pi(s, \pi(s), \tilde{R}))$, is less than the Q-value for a state with any action, $(Q^\pi(s, a, \tilde{R}))$, then, a new $\tilde{\pi}$ is found by updating the Q-values using policy iteration using expert data, $\tilde{R}$, and π. Then with probability of $\frac{f(\tilde{R},\tilde{\pi})}{f(R,\pi)}$, R is set to $\tilde{R}$ and π is set to $\tilde{\pi}$, otherwise only R is set to $\tilde{R}$ with probability $\frac{f(\tilde{R},\tilde{\pi})}{f(R,\pi)}$. This algorithm returns the final calculated R after all iterations are complete [29].

2.3.2 Gaussian Process Inverse Reinforcement Learning

An issue with many inverse reinforcement learning algorithms is that they assume a linear reward function. This limits the extent of what inverse reinforcement learning can learn from an expert. For example, in a highway driving model used in [17], an agent needs to learn how to avoid cars going a fixed speed, choose the speed it is going, and make sure it is not speeding when it is within two car-lengths from a police vehicle. The connection between the speed and proximity to a police vehicle makes the underlying reward function nonlinear. To learn nonlinear rewards, Gaussian processes are used with inverse reinforcement learning, which is referred to as Gaussian Process Inverse Reinforcement Learning (GPIRL) [28] [17]. A key to GPIRL's success is its ability to combine the probabilistic reasoning of stochastic expert behavior with the ability to learn a nonlinear function of features for the reward [17]. As the name suggests, GPIRL is modeled as a Gaussian process whose structure is determined by a kernel function. In Gaussian process regression, the noisy observations y of the true outputs u are used. GPIRL learns u which in turn represents the rewards that are associated with the feature coordinates X_u. These coordinates can be the feature values of all states present or a subset of all states. Any state not present in the observations have their reward inferred by Gaussian process. The goal of GPIRL is to learn the kernel hyper-parameters θ that reveals the structure of the reward (r). Therefore, the values of u and θ are found by maximizing their probability with respect to the expert demonstrations D as shown in (2.13) [17].

$$P(u, \theta|D, X_u) \propto P(D, u, \theta|X_u) = [\int_r P(D|r)P(r|u, \theta, X_u)dr]P(u, \theta|X_u) \tag{2.13}$$

Where $P(D|r)$ is the IRL term, $P(r|u, \theta, X_u)$ is the Gaussian process posterior, and $P(u, \theta|X_u)$ is the Gaussian process probability. The Gaussian posterior represents the probability of the reward function with respect to u and θ. The Gaussian process probability is the prior probability of an assignment to u and θ. The log of this function gives the Gaussian process

log marginal likelihood which will favor simple kernel functions and those values of u that support the current kernel matrix. Using the automatic relevance detection (ARD) kernel with $\theta = \{\beta, \Lambda\}$ states with different highly-weighted features will take on different reward values whereas states with similar highly-weighted features will take on similar reward values. In this case β is the overall variance and Λ represents the weights on each feature. The process of learning Λ gives features that are less relevant low weights, and features that are very relevant high weights. One advantage to GPIRL is that while X_u needs to cover the space of feature values, fewer points are needed than are states. This is advantageous with problems with large state-spaces. In [17] X_u contained feature values for all states visited in D, as well as additional random states to add to the overall count of X_u.

Levine et al. [17] compared GPIRL performance to FIRL, MaxEnt, and MaxEnt/Lp. In the object world experiments it was shown that GPIRL needed significantly less expert demonstrations to converge to a solution when compared to the other algorithms. It was also shown that when novel states were presented, GPIRL learned more accurate rewards than the other algorithms. Using the highway driving model, an experiment with nonlinear rewards tough on many IRL algorithms, GPIRL was able to successfully learn the reward structure [17]. Qiao et al. [28] also saw similar success with GPIRL. In experiments, they compared GPIRL with Linear Inverse Reinforcement Learning (LIRL) and Convex Programming Inverse Reinforcement Learning (CPIRL). In the GridWorld experiment, each algorithm converged, however, GPIRL converged much faster than LIRL (2 times slower) and CPIRL (1.8 times slower). They also used a discrete version of the hill climb problem where environment information was skipped (partially observable). GPIRL was able to successfully build the reward structure despite not having all of the information. Therefore, GPIRL was shown to be able to find the reward structure with fewer observations than LIRL and CPIRL [28].

2.3.3 Maximum Entropy Inverse Reinforcement Learning

In many imitation learning problems, modeling sequential decision-making behavior is often very difficult due to the lack of foresight in these systems.

Ziebart et al. [41] developed an inverse reinforcement technique based on the principle of maximum entropy theory to counter this. They tackled modeling real-world navigation and driving behaviors by sampling noisy and imperfect data from driving "experts". Their approach was able to model route preferences of drivers as well as infer destinations and routes based on partial trajectories using GPS data of taxi-cab driving. The environment they tested in was a known-fixed structure of the world (such as a road network) with known actions characterized by various road features such as speed limit and number of lanes.

The idea of inverse reinforcement learning revolves on an agent optimizing a function that linearly maps features of each state (s_j) to a reward value. Therefore, the reward value of an "expert" data sample, or trajectory, is a sum of state rewards which can be simplified to the reward weights, (θ), applied to the feature counts in a trajectory/path (ζ) (2.14) as shown in (2.15) [41].

$$f_\zeta = \sum_{s_j \in \zeta} f_{s_j} \tag{2.14}$$

$$reward(f_\zeta) = \theta \cdot f_\zeta = \sum_{s_j \in \zeta} \theta \cdot f_{s_j} \tag{2.15}$$

A major problem of using "expert" trajectories is that an inverse reinforcement algorithm may find a preference for one path over others. This poses a problem that an agent will only learn a path and not how features of a state can help in path decision making. Using maximum entropy this problem is solved by choosing a distribution that does not exhibit a path preference. Using this probabilistic model for deterministic (no randomness) path distributions, trajectories with the same final reward will be treated equally, where a preference will only be given for trajectories with higher reward. Trajectories with a higher reward will have an exponentially higher probability. For example, given the diagram in Fig. 2.20, assume that from $A \rightarrow B$ each path provides the same reward. It can be seen that with an action-oriented model that path 3 will have a 50% probability of being chosen and paths 1 and 2 will have a 25% probability of being chosen. With a

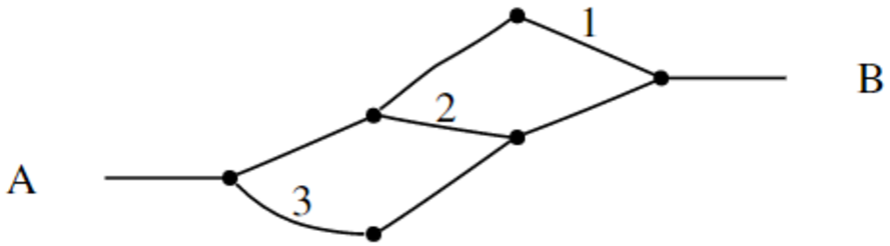


Figure 2.20: Maximum Entropy versus Action-based selection diagram [41]. See text for description.

Maximum Entropy model each path would have equal probability because they produce the same reward. This is shown in (2.16) [41], where *Z(θ)* is the partition function which will always converge given trajectories that reach a reward-giving goal in a finite number of steps.

$$reward(P(\zeta_{\mathrm{i}})|\theta) = \frac{1}{Z(\theta)} e^{\theta \cdot f_{\zeta_i}} = \frac{1}{Z(\theta)} e^{\sum_{s_j \in \zeta_i} \theta \cdot f_{s_j}} \tag{2.16}$$

For non-deterministic (randomness) path distributions, (2.16) must be altered to take randomness in path distributions into account. Most Markov Decision Processes (MDP) that relate to real-world environments or dynamics will have non-deterministic transitions between states, meaning that an action that is supposed to transition to one state may not do so with some probability ϵ. Therefore, this distribution over paths produces a stochastic policy where the probability of an action is weighted by the expected rewards of all paths that begin with that action as shown in (2.17) [41].

$$reward(P(action|\theta, T) \propto \sum_{\zeta : action \in \zeta_{t=0}} P(\zeta|\theta, T) \tag{2.17}$$

Therefore, to find the optimal weights the likelihood of the observed data is maximized under the maximum entropy distribution (2.18) [41]. Since this function will be convex for deterministic MDPs the optima is found using the gradient which is defined by the difference between the expected feature counts from the expert and the learner's expected feature counts represented in terms of expected state visitation frequencies, D_{si}. The gradient descent for the weights is shown in (2.19) [41]. At the optima, the feature

expectations will match which will show that the learner performs equivalently to the demonstrated behavior even if the reward weights found are not the same as the ground truth.

$$\theta^* = argmax_\theta L(\theta) = argmax_\theta * \sum_{trajectories} \log\Big(\tilde{\zeta}|\theta, T\Big) \tag{2.18}$$

$$\bigtriangledown L(\theta) = \tilde{f} - \sum_{\zeta} P(\zeta|\theta, T) f_\zeta = \tilde{f} - \sum_{s_i} D_{s_i} F_{s_i} \tag{2.19}$$

This paper [41] uses Maximum Entropy IRL to recover a reward function for predicting driving behavior and route recommendation. The model of this problem contained over 300,000 states (road segments) and 900,000 actions. All expert trajectories were assumed to reach a goal while optimizing time, safety, stress, fuel costs, and maintenance costs. This is used as the cost function for this system, whereas the destination for all trajectories is the state where no additional cost is incurred. The expert trajectories were GPS traces from 25 taxi drivers which resulted in over 100,000 miles of collected data over 3,000 hours of driving. Each state (road segment) was characterized by: road type, speed, lanes, and transitions. Using the Maximum Entropy IRL model, this paper [41] achieved state-of-the-art results for its time for path matching.

2.3.4 IRL using DQN

One modification to inverse reinforcement is using a DQN (such as the one in section 2.2.1) in the value iteration step to determine the Q-values. Sharifzadeh et al. [31] used this type of network to teach a simulation to drive in a highway environment. The agent was placed in a 3 lane highway and expert data was used to teach it to merge into lanes and not hit cars using three actions (forward, right, left). The inputs to the system were values of 13 sensors that were discretized into 16 bins of visibility that indicated whether or not their was an obstacle. This resulted in 208 binary features which allowed for 2^{52} states. The feature weights obtained from the IRL

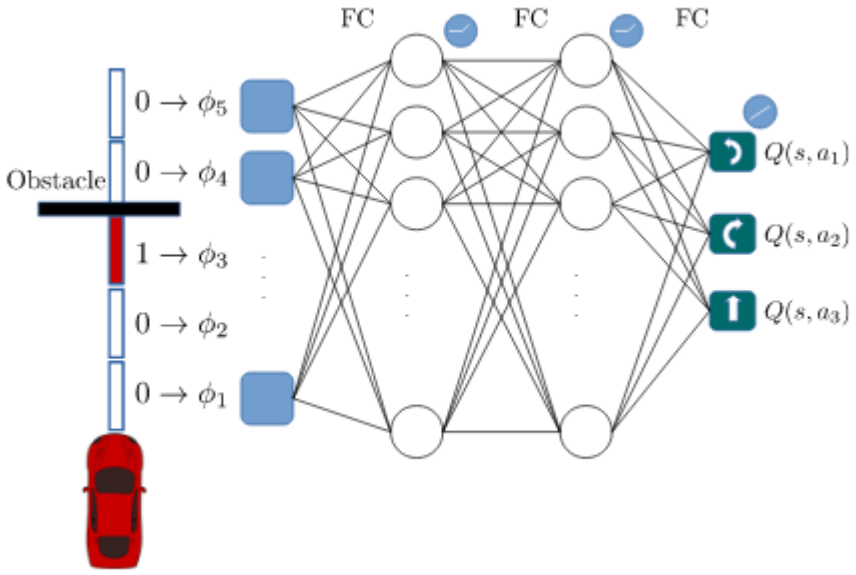


Figure 2.21: Deep Q-Network Architecture in IRL [31]. See text for description.

step were fed into the DQN to produce an output value for each action, or policy. The DQN step is shown in Fig. 2.21. Through experimentation it was shown that the reward was successfully able to be extracted from expert demonstrations in large state spaces.

2.4 Deep Inverse Reinforcement Learning

A relatively new field that has come into the realm of reinforcement learning is Deep Inverse Reinforcement Learning (Deep IRL) or Deep Apprenticeship Learning. This is the idea of using expert data to teach systems complex tasks. Just as in reinforcement learning, the Deep IRL networks take in complex data types, such as images. However, just as in regular IRL, these networks do not find a policy, but instead find a reward structure based on learning a policy from an expert. There are many challenges to this, in that the reward function an expert uses is assumed to not be known. The expert is assumed to make optimal and perfect moves, however, ideally many of these datasets come from experts behaving in a stochastic manner. Expert data that is not stochastic brings about an issue of the expert not exploring

its entire world, meaning when the system encounters a state that it did not learn from the expert it is unable to generalize well.

Unlike in deep reinforcement learning, research using raw pixel images in this area is limited. Much of the expert data is displayed as trajectory movements or direct observations from an expert. While this works, it does not generalize well to tasks in the real world where only a series of videos may be available to learn from. The issue with pixel data is that the system must first understand the task it is observing, extract the policy from the expert, then create a generalized reward system for an agent to follow. This makes it a much more complex task than reinforcement learning, but its a task that is needed to be solved to expand deep learning to agents in the real world.

2.4.1 Deep Gaussian Process IRL

Jin et al. [14] took the successful GPIRL architecture from Section 2.3.2 and modified it to work in a deep network. This furthers the advantages of Gaussian process by allowing to learn even more complex reward structures across very large state spaces with limited expert demonstrations. Another advantage of using a deep network for GPIRL is that is not reliant on predefined features like its non-deep counterpart. This allows for learning more complex reward structures from complex tasks where features may not be obvious. Deep GPIRL uses a deep belief network with Gaussian process and Gaussian process latent variable model layers. Its goal is to learn an abstract reward structure using small data sets which mimics a human's ability to perform inductive reasoning after a few experiences. The Deep GPIRL architecture is shown in Fig. 2.22. As shown, W stands for inducing points for the first Gaussian process layer which are small set of data points that are learned or selected directly. V is the learned distribution, or selection, of those points. Z and f act in a similar fashion as W and V for the second Gaussian process layer. B and D are the latent and noisy outputs from the first Gaussian process layer. X is the state feature representation, r is the learned latent reward that is learned from X to explain the demonstrations M. Using the Deep GPIRL in experiments, particularly ObjectWorld and BinaryWorld, it performs better than traditional IRL algorithms using fewer

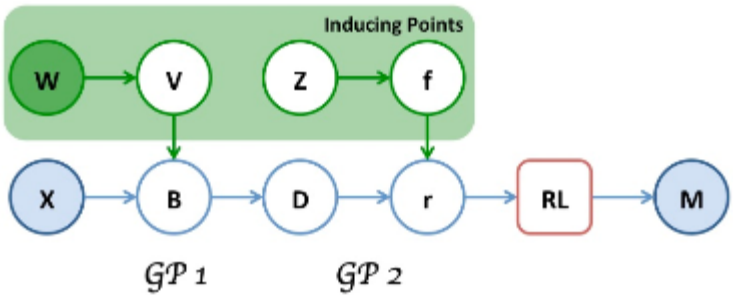


Figure 2.22: Deep GPIRL Architecture [14].

demonstrations.

2.4.2 Deep Maximum Entropy IRL

Wulfmeier et al. [39] used multi-layer neural networks to use Maximum Entropy IRL (from Section 2.3.3) without the need to hand-craft features. Using similar grid environments as [41], a neural network (specifically a Fully Convolutional Neural Network) with wide convolutional layers (width one) is used to learn spatial features based on raw input.

The first tests of this network used a regular neural network structure without any convolution layers, therefore the inputs were expert trajectories. The output of this neural network is used to estimate the reward function as is done in traditional inverse reinforcement algorithms. The input of this network is expert demonstrations (μ_D^a) of state (s), action (a), reward pairings. The high level algorithm deals with iterating over *n* epochs which represent the amount of gradient descent iterations. First, randomly intialized weights (α) are forward propagated in the neural network. The reward (r^n) is then extracted from this model and used to find an approximate policy (π^n) using value iteration. This policy is then used to find the expected state visitation frequency ($E|\mu^n|$) for each state (s) as defined in Section 2.3.3. The maximum entropy loss is found by (2.20) [39] using maximum entropy theory. The gradient is computed by taking the gradient of (2.20) with respect to the reward (r^n) which is equivalent to the difference of the expected state visitation frequency of the expert (μ_D) and learner expected state visitation

frequency ($E|\mu^n|$) as shown in (2.21) [39]. This is the same gradient calculation as found in [41]. This gradient is then back-propagated through the network to update nodes. The weights (α) are then also updated with this gradient and the algorithm repeats until the number of epochs is reached.

$$L_D^n = log(\pi^n) \times \mu_D^a \tag{2.20}$$

$$\frac{\delta L_D^n}{\delta r^n} = \mu_D - E|\mu^n| \tag{2.21}$$

Wulfmeier et al. [39] implemented and analyzed their DeepIRL solution on two types of environments, Objectworld and Binaryworld. An objectworld consists of an $M \times M$ grid representing M^2 possible states. There are five possible actions for an agent to move (up, down, left, right, stay in place). The state features are defined as the minimum distance to colored object. The objects can be one of C colors. The reward is defined as positive for grid cells which are within a distance of three units of color one and a distance two units of color two. The reward is defined as negative for grid cells which are only within distance of three units of color one, and zero otherwise.

The binary world consists of states being randomly assigned blue or red. The feature vector for each state is a binary vector of length nine which encoded the color of each cell in a 3×3 neighborhood. The reward is positive if four out of nine neighboring states are blue, negative if exactly five are blue, and zero otherwise. The binary world relies on a direct relationship between states which makes it a unique, and complex problem to solve.

For both worlds the DeepIRL network produced state of the art results. Inclusion of convolutional layers, Fully Convolutional Neural Network (FCNN) with width one, allowed for the input of raw data without the need of hand-crafted features (such as what is needed for object world and binary world). The raw input is the entire state-space with what each state occupies (objects for object world, color for binary world). This allows the network to not only learn an appropriate reward function but also spatial features of the input environment in terms of the expert trajectory data. One drawback

of this is that more expert samples are needed for training to match performance of DeepIRL network without a CNN, however, this extends this architecture to take in raw image data to learn complex tasks in difficult environments [39].

2.4.3 Deep Apprenticeship Learning

Markovikj [18] used a deep architecture to perform IRL on various games using raw pixel inputs. These games were Atari games; Freeway, Space Invaders, and Seaquest. This was done through the creation of a multi-part architecture known as Deep Apprenticeship Learning (DAL). The two parts of this architecture are Deep Apprenticeship Q-Network (DAQN) and Deep Apprenticeship Reward Network (DARN).

2.4.3.1 Deep Apprenticeship Q-Network

The DAQN [18] is used to take in raw inputs from expert game play (D_E) to extract the policy the expert is using. This is used to learn a reward (or score) function which rates actions for states, much like the Q-function. Therefore, training this network will learn the expert's policy. This network was created using convolutional layers, similar to the Deep Q-Network [22]. The inputs to this network are 83×83 raw pixel images representing states. Tests were done with feeding in one-four frames per input. The network was updated using batch processing of 32 samples. The structure is shown in Fig. 2.23. As shown, this architecture outputs a softmax prediction between three possible actions (stay, up, down). Therefore, for every state (raw pixel input) the network predicts the next action to be taken. The loss function of this network is shown in (2.22) [18].

$$J(w) = \sum_{a} [q_w(s,a) - \hat{q}(s,a)]^2 \tag{2.22}$$

Where, w are the learned weights, $q_w(s,a)$ is the softmax output of the DAQN and $\hat{q}(s,a)$ is the actual action taken by the expert represented by a 1-hot array. Therefore, for inputs ($s = s_n, a = a_n \in D_E$), the array $\hat{q}$ is 1 if $a = a_n$ and 0 otherwise. The network is updated using AdaGrad where

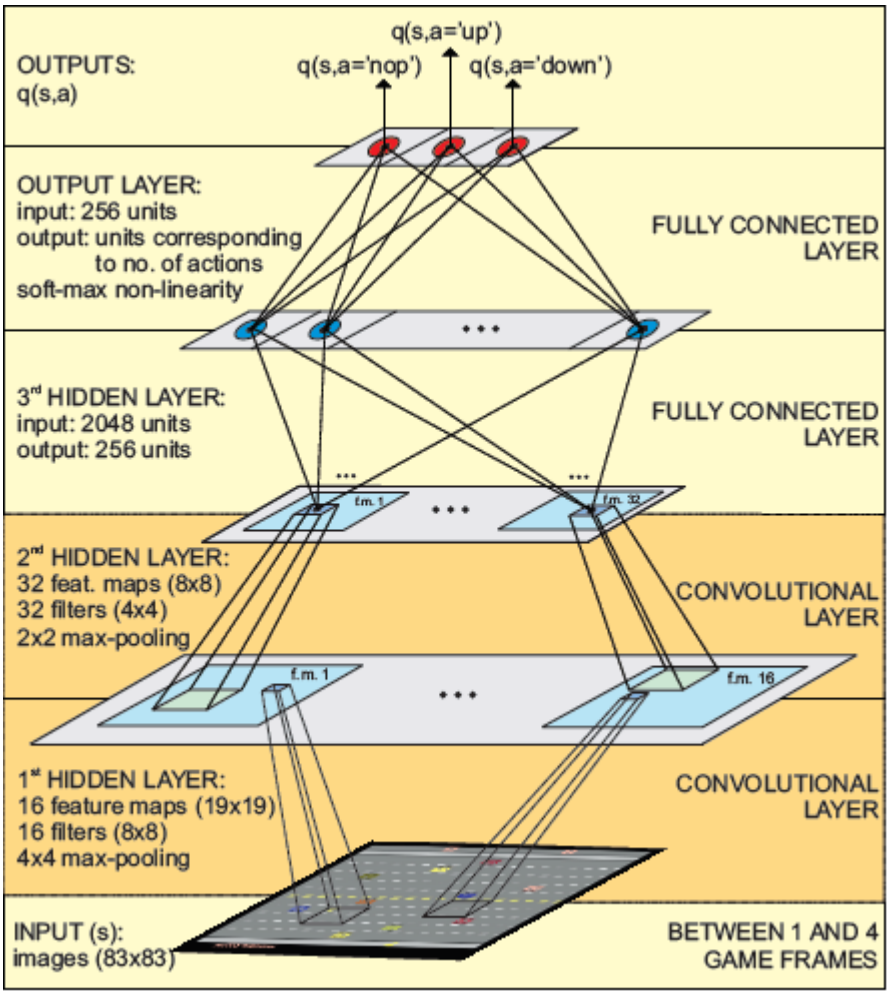


Figure 2.23: DAQN Architecture [18].

each parameter has its own learning rate, η learned over time.

2.4.3.2 Deep Apprenticeship Reward Network

Once the DAQN is trained, the DARN [18] is used to extract the reward function from the learned expert policy in the DAQN. The DARN has the same architecture as the DAQN (shown in Fig. 2.23). However, it uses the output from the DAQN before the softmax in its loss function. The input to DARN is a (s, a, s') pairing where s is an 83×83 image representing a state, a is the action taken at that state, and s' is the next state after taking action a from state s. The loss function for DARN uses an L2-norm and is shown in (2.23) [18].

$$J_r(w) = \|r_w(s, a) - \hat{r}(s, a)\|_2 \tag{2.23}$$

Where, w are the learned weights, $r_w(s, a)$ is the output of the DARN and $\hat{r}(s, a)$ is a Bellman representation of the DAQN (2.24) [18].

$$\hat{r}(s, a) = DAQN^{PS}(s, a) - \gamma \max_{a'} DAQN^{PS}(s', a') \tag{2.24}$$

Where, $DAQN^{PS}(s, a)$ are the presoft values of the DAQN with inputs (s, a) and $\max_{a'} DAQN^{PS}(s', a')$ is the maximum presoft value of the DAQN with inputs (s', a'). γ is the discount factor on how much to weight future states. Therefore, the expanded loss function for the DAQN is shown in (2.25) [18]. The overall training architecture for DARN is shown in Fig. 2.24.

$$J_r(w) = \|r_w(s, a) - (DAQN^{PS}(s, a) - \gamma \max_{a'} DAQN^{PS}(s', a'))\|_2 \tag{2.25}$$

The DARN is trained using random state transitions contained in a separate dataset from the expert data, D_g. This allows the system to generalize to all states, even ones not explored by the expert.

2.4.3.3 Results

Once training on the DARN is complete, the system can now be used to predict next actions given a state. Due to the complexity of Space Invaders, this

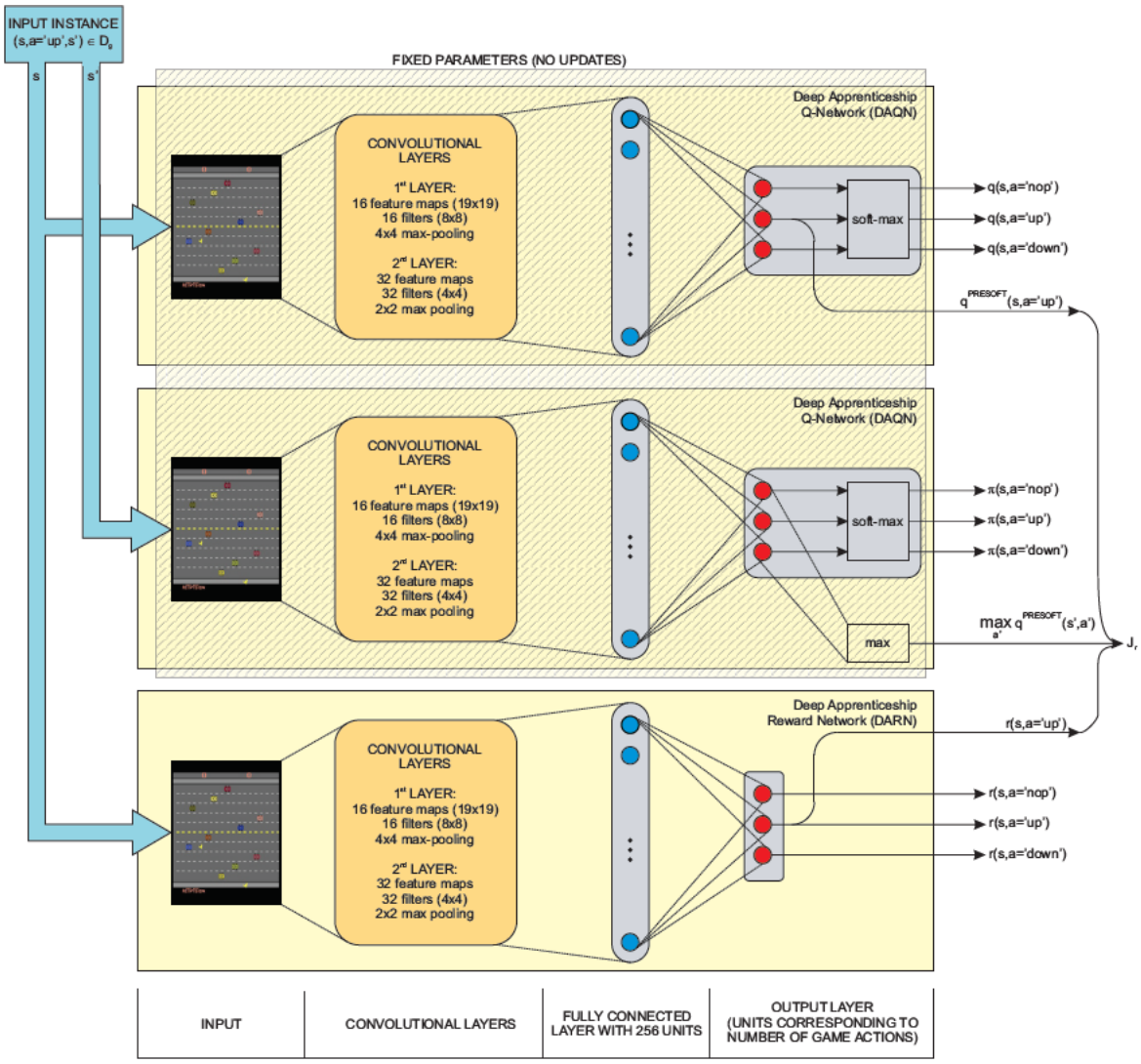


Figure 2.24: DAL Architecture [18].

Method	Reported score
random agent	0
human player	32
Sarsa	11
DAL	17

Figure 2.25: DAL results for Atari 2600 Freeway compared to a random agent, human player, and Sarsa [18].

type of network did not generalize to its game play. However, it performed better on the simple game of Freeway, obtaining a score higher than other non-deep IRL algorithms (Sarsa) but much lower than humans, as shown in Fig. 2.25 [18].

2.4.4 Deep Q-Learning from Demonstrations

Hester et al. [12] undertook the challenge of bringing deep reinforcement learning to real world tasks. They recognized, that while exploration in traditional reinforcement learning works well in simulation, it is not practical in physical environments. Instead, they propose an algorithm that extends traditional DQNs to train with expert data using transfer learning theory called Deep Q-Learning from Demonstrations (DQfD). They use important aspects of DQN such as Double DQN and experience replay. To update the Q-values of the network with values from the target network the cost function in (2.26) [12] is used.

$$J_{DQ}(Q) = (r(s,a) + \gamma Q(s_{t+1}, a_{t+1}^{max}; \theta') - Q(s,a;\theta))^2 \tag{2.26}$$

Where θ' are the parameters of the target network, and θ are the parameters of the current network [35]. However, due to the supervised learning step of using expert demonstrations, the cost function needs to be updated to include (2.27) [12].

$$J_E(Q) = max_{a \in A}[Q(s,a) + l(s, a_E, a)] - Q(s, a_E) \tag{2.27}$$

Where a_E is the action the expert takes in state s. $l(s, a_E, a)$ is a margin function where it equates to 0 when $a = a_E$ and a positive number (usually 1) otherwise. This will allow for the expert's decisions to be weighted higher. An L2 regularization term is also used and applied to the weights and biases of the network to prevent from over-fitting on small expert datasets. Therefore, the entire loss function is (2.28) [12].

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_E(Q) + \lambda_2 J_{L2}(Q) \tag{2.28}$$

Where λ_1 and λ_2 are weights for each of the losses.

For experience replay, with probability p an expert experience $(s, a, s', reward)$ will be chosen from the entire set of expert demonstrations and with probability $1 - p$ a random demonstration will be chosen from the set of agent's experience held in a limited buffer to batch update the network weights. When the buffer of agent experiences is full, older experiences will be written over with new ones.

The algorithm for DQfD is shown in Algorithm 3. Using the Atari game dataset and a custom-made game called Catch the DQfD algorithm was compared to DQN. In most cases, DQfD learned faster due to the transfer learning step and out performed DQN and Double DQN. When compared to the Double DQN [35] the DQfD algorithm performed better on many Atari 2600 games and comparable on others as shown in Fig. 2.26.

Algorithm 3 DQfD [12]

for steps t $\in$ epochs **do**
  Get batch of n transitions from expert data
  Calculate loss J(Q) in (2.28)
  Update θ using gradient descent

for steps t $\in$ epochs **do**
  Sample action from network given a state
  Play action and observe (s',reward)
  Store experience into replay buffer for agent experience (s,a,s',reward)
  Get batch of n transitions from expert experience (p) and agent experience ($1 - p$)
  Calculate loss J(Q) in (2.28)
  Update θ using gradient descent
  At time interval τ update θ' with θ

Game	DQfD	Double DQN	Imitation
Alien	**577.1**	280.1	473.9
Amidar	**250.4**	76.3	175.0
Assault	1017.4	**1384.9**	634.4
Asterix	2353.3	**4715.4**	279.9
Asteroids	**2507.8**	914.6	1267.3
Atlantis	**17647.0**	13494.8	12736.6
Bank Heist	**106.2**	8.7	95.2
Battle Zone	12486.1	3456.4	**14402.4**
Beam Rider	464.3	**748.8**	365.9
Bowling	46.5	28.5	**92.6**
Boxing	**89.1**	85.2	7.5
Breakout	**95.8**	5.3	3.5
Chopper Command	**2989.3**	2582.3	2485.7
Crazy Climber	103980.9	**108450.5**	14051.0
Defender	**7607.6**	3505.7	3819.1
Demon Attack	186.0	**405.1**	147.5
Double Dunk	**-16.9**	-20.2	-21.4
Enduro	624.8	**736.8**	134.8
Fishing Derby	-18.0	**-13.5**	-74.4
Freeway	**30.9**	28.5	22.7
Gopher	**9079.5**	4909.8	1142.6
Gravitar	245.1	35.6	**248.0**
Hero	**20428.2**	5373.0	5903.3
Ice Hockey	-9.8	**-4.7**	-13.5
James Bond	145.5	6.5	**262.1**
Kangaroo	1311.2	**1779.2**	917.3
Krull	1054.8	1880.2	**2216.6**
Kung Fu Master	**12328.6**	6677.8	556.7
Montezuma's Revenge	**780.9**	0.0	576.3
Ms Pacman	680.6	308.8	**692.4**
Name This Game	**4376.5**	4171.5	3745.3
Pitfall	-124.6	-26.3	**182.8**
Pong	**15.2**	13.6	-20.4
Private Eye	38280.5	-111.3	**42749.6**
Q-bert	2211.6	245.9	**5133.8**
River Raid	2368.0	**3202.6**	2148.5
Road Runner	38041.5	**39988.2**	8794.9
Seaquest	181.2	**1113.9**	195.6
Solaris	3107.9	221.8	**3589.6**
Up N Down	**10265.1**	8522.9	1816.7
Video Pinball	**10926.2**	7135.5	10655.5
Yars' Revenge	4764.3	**5731.8**	4225.8

Figure 2.26: DQfD results for Atari 2600 games compared to Double DQN and Imitation Learning (supervised learning) [12].

Chapter 3

Dataset and Technologies

3.1 Maze World

Maze World is a variation of Grid World, without negative valued states. An agent is placed inside a binary maze where white spaces are open and black spaces are walls the agent cannot go through. The objective of the world is for the agent to navigate through the maze to a goal. An example of this world is shown in Fig. 3.1. As shown in the figure, the world is composed of 100×100 pixels but only represents a 10×10 maze. These mazes were generated in Python using example code from [3] which creates an $m \times n$ grided maze in an $M \times N$ pixel image, where m and n are factors of M and N, respectively. For example, a 100 pixel by 100 pixel image (M=100, N=100) would hold a 10 by 10 gridded maze (m=10, n=10). At any white space in the maze there exists a path to any other white space on the maze that is unobstructed by a black space. This allows the goal to be put at any open white space in the maze. There is also only one optimal path to the

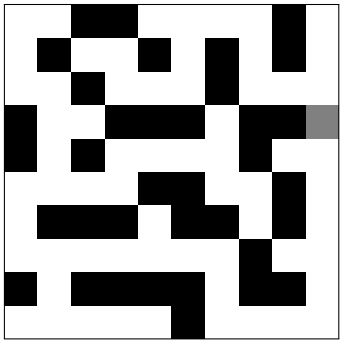


Figure 3.1: Maze World Example: White spaces are open, Black spaces are blocked, and the Gray space is the goal.

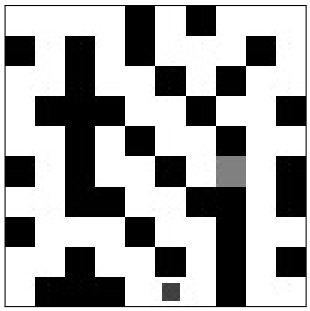


Figure 3.2: Maze World with Agent: small, dark gray square represents agent.

goal from any white space. To make the dataset simpler, the mazes were converted into a single-channel 8-bit integer image representations.

An agent is placed in the maze randomly in any white space. The agent is represented by a smaller dark gray square as shown in Fig. 3.2. The goal of the agent is to make it to the goal in the least number of steps. The agent has four possible actions; Move Right, Move Down, Move Left, Move Up. Actions are one-hot enconded. If the agent hits a wall (black) it loses, if it hits the goal it wins. At each step, the agent moves a full grid step. In the back-end, the maze is treated like a $m \times n$ grid, and Python's PIL framework is used to generate the graphics used for simulation and dataset generation.

3.1.1 Expert Data

Expert data is needed to teach an agent through demonstration. To generate expert data, a maze is created and an agent is placed randomly inside of the maze at any valid space. Then using Python's networkx library, the 2D grided maze is converted into a graph. Then using the networkx function $single\_source\_shortest\_path()$, every path from a valid point to the goal is computed using Dijkstra's algorithm. These paths are then used to allow the agent to traverse through the maze to the goal in the least amount of steps possible.

3.1.2 Random Data

Random data is needed to ensure that the agent is able to have experience at every space in the maze. Expert data will be able to hit all of these spaces,

but it will only ever choose one action (the optimal action) in these spaces. To generate random data, a maze is created and an agent is placed randomly inside of the maze at any valid space. The agent then takes any valid action randomly to move to another valid space, this action does not necessarily have to be optimal. This data does not continue after this, only the original state of the agent, the action taken, and state after action of the agent is stored.

3.1.3 Datasets

Expert data and random data are stored as GIFs. To do this, at each movement of the agent an image is generated (such as in Fig. 3.2) to represent the current state of the agent in the maze. For expert data these images are collected until the agent reaches the goal. The images are then combined into a GIF and saved for later use. Each action the agent takes is saved into a text file to be used later. The same is done for the random data, except that the GIFs were only two images total. To obtain the two images, the agent is placed randomly in the maze which accounts for one of the images, the agent then takes a random action to end in another state, this accounts for the second image.

Two types of datasets were created for expert data and random data; same and random.

3.1.3.1 Same

"Same" means that the maze the agent is in does not change. Therefore, the expert data and random data use the same maze, however the placement of the agent will change. This dataset is used to show whether or not a system can learn a single task without generalizing what the task is.

3.1.3.2 Random

"Random" means that the agent will be placed in mazes varying in structure. Therefore, the expert data and random data use different randomly generated mazes, and the placement of the agent changes in each maze. This dataset

is used to show whether or not a system can learn a single task while also generalizing what the task is.

3.1.4 Simulation

Given a maze, current placement of the agent, and the action to be performed, a simulation of the maze task can be completed. This is used in testing as well as in training to show how well the agent is learning and to gather data. At each step, the simulation environment tells the system the new state of the agent (image of maze with agent), whether or not the agent has reached the goal, and whether or not the agent has crashed into a wall. If the simulation is fed an initial maze image, it will keep that maze for the rest of the simulation (in cases where the same maze is being tested) otherwise it will use a randomly generated maze. If the simulation is given no initial state, it will randomly place the agent in the maze.

3.1.5 Processing Data

To make the saved GIFs and action-list text files usable to the system architectures, some pre-processing needs to be done. First, the GIFs are read in frame by frame. These frames are then converted to the data size needed using Python Image Library's (PIL) image resize function. These are then saved into a numpy array of shape $[?, datasize, datasize, 1]$. As shown, all the data is square. The actions are read out of the action-list and converted into one-hot encoded vectors stored into a numpy array of shape $[?, numberofactions]$. These arrays are then split into test, train, and reward datasets for the image data and action data. These split arrays are then stored into an h5py file so that it can be accessed faster and more efficiently when the system is run again.

3.2 Tools and Technology

3.2.1 Python

Python 3.5.2 was used in this research using Anaconda. Python served as a quick development environment, and allowed for a variety of debugging

strategies. Much of the research in deep learning or reinforcement learning is done using Python, so running example code from others' research was made easy by using this scripting language.

3.2.2 TensorFlow

TensorFlow 1.0 [5] was used for this research. TensorFlow is an open source library for Python developed by Google. It is mostly used for numerical computation using data flow graphs, and is primarily used to build a variety of neural networks. This framework allows users to easily implement deep architectures and run a variety of analysis and training tools.

3.2.3 Python Imaging Library

Python Imaging Library (PIL) adds image processing and creation tools to Python. The PIL library was used in many ways in this research including dataset generation, dataset processing, and result visualization.

3.2.4 Numpy

Numpy is a Python library that allows for complex data management and mathematical operations, especially in the linear algebra space. TensorFlow heavily relies on numpy for its powerful mathematical operations. Numpy was also used in this research for data storage and manipulation.

3.2.5 h5py

h5py is a Python package to implement the HDF5 binary data format. This format allows for the storage of large amounts of data to be loaded efficiently at a later time. In this research, the datasets were often large and a huge bottleneck for the system, however, after converting the data and storing it with h5py, reading the data was done in seconds. This severely cut-down on run-time (from 6 hours for data reading to 30 seconds).

Chapter 4

Proposed Methodologies

4.1 Deep Apprenticeship Learning Network Modifications

4.1.1 No Pooling Layers

One issue with the Deep Apprenticeship Learning (DAL) architecture [18] in section 2.4.3, is its use of pooling layers. Pooling does well in classification to increase performance, however it removes many fine details that are necessary in task completion and understanding. Removing the pooling after each convolution layer results in slower training, but allows the network to learn from smaller details within the images given. For example, using Maze World, pooling layers would remove the goal and the agent from existing after the first layer, causing all of the states going through the network to look almost identical. This was shown in testing the DAL network when it would predict the same action be taken for each state presented.

4.1.2 Transfer Learning

Another change to the DAL architecture [18] is the use of transfer learning. In the current implementation of DAL, the DAQN and DARN are trained seperatley. This means that, while they use the same network architecture, the DARN network starts with random weights and biases to learn from the DAQN network. Instead, using a methodology called transfer learning, the DARN network is initialized with the same weights as the trained DAQN network. This allows the DARN network to have a good starting point in understanding the structure of the states so that its main focus is on updating its Q-values and not processing the image. This also allows it to learn faster,

and with a lower cost.

4.1.3 Using Q-Learning

On top of removing pooling layers and including transfer learning, methodologies from Q-learning were also implemented. In its current form, the DAQN network in DAL [18] does not use Q-updates to compute its loss. Instead it uses the softmax of the output of the network to compare it to the action it is supposed to be. While a method like this would work in classification, it is unstable in task learning because it leads to large jumps in updating the weights of the network. Instead the loss function was changed to be more representative of Q-learning for both the DAQN and DARN networks as shown in (4.1).

$$J(Q) = (R(s,a) + \gamma * max_{a'} Q(s',a')) - Q(s,a))^2 \tag{4.1}$$

4.1.3.1 Reward Abstraction

An inherit issue with using Q-learning is the need for the reward R. The purpose of apprenticeship learning is to find the reward model from given expert demonstrations, which means that the reward for each step an agent or expert takes is not known. By abstracting the reward function from step-level to task-level, it serves as a task completion modifier for the loss. For example, for each step taken by the expert if the task is not complete the reward is 0, if the task is complete the reward is $+100$, and if the task is failed the reward is -100. Since the expert data used is all successful, only the last transition for the expert into the goal is given a reward of $+1$. For the DARN network, the same loss function is used, but instead of using random data (as in section 3.1.2) the network "plays against itself" to accrue more experience. Therefore, the agent starts in a maze in a random location and performs a step based on the network output from the state it is in. If the step is into the goal the reward is $+100$, if it hits a wall or takes too long it is -100 (and the maze resets), otherwise it is 0.

4.1.3.2 Experience Replay

Upon creating experiences using the DARN network, the network will be updated through mini-batches of experiences in the form of (s, a, s', r) once its replay buffer is full. As the agent gains more experiences the older experiences are overwritten by the newer ones. This allows the network to learn only from the most recent experiences to weed out "bad habits" from earlier experiences.

4.2 Deep Q-Network Implementations

Many success have been seen in the realm of Deep Q-Learning and its modifications. While this does extremely well in reinforcement learning, its methodologies have not been used much in deep apprenticeship learning. Two methodologies from this space include shared experience and dueling deep Q-networks. Both of these network modifications use the Deep Q-Network architecture [22] which is fully convolutional, and has been proven successful in task completion environments in the Atari world.

4.2.1 Using Shared Experience Replay

Experience replay [22] has been shown to lead to better learning in networks by using past experience. However, up until now, in traditional reinforcement learning only the agent's experience was used with this method. While this worked for algorithms that relied on exploration and exploitation, it did not always produce great results if the experiences were not successful. This may cause the network to hit a local minima where the optimal action selection will never be reached. In apprenticeship learning, experiences from an expert are given and are guaranteed to be correct. Using these experiences along with the agent's learned experiences would allow for better learning. Hester et al. [12] introduced this in their DQfD architecture, and showed that just using 10% of expert data in each batch update of experiences gave way to better learning, and ensured that the agent was not going to keep relearning from bad experiences. This also ensures that the network can get out of a local minima and continue learning despite the "bad" learned

experiences.

4.2.1.1 Scheduled Shared Experience Replay

A modification to shared experiences is the idea of scheduled shared experience. Where the percent of experiences from the expert in agent training is high at first but then slowly decays over time. This serves as a guide for agent training in its early stages, and ensures that the training does not hit a local minima early.

4.2.2 Target Q-Network

Mnih et al. [22] also introduced the concept of using a separate target Q-network. This was due to the fact that the Q-network architecture is unstable along sequential updates. By updating the weights of the Q-network by small amounts every iteration, the network is more likely to diverge due to tightly coupled correlations. A target Q-network is used outside of the main Q network. The main Q-network is updated at each iteration using values from the target Q-network. After τ iterations, the target Q-network's weights are updated with the main Q-network's weights.

4.2.3 Using Dueling DQN

Another modification made was using the Dueling Deep Q-Network architecture (section 2.2.3) [37]. Splitting the value and advantage functions from the main Q-network, allows for the separation of state estimation and action selection. By decoupling these two, the network can more accurately predict the best action to take regardless of the state that was predicted. For the dataset used (Maze World), this allows for part of the network to determine the goodness of the current state of the maze, and the other part of the network to determine best action to take.

4.2.4 Using Deep Recurrent Q-Networks

As explained in Section 2.2.4 [11] [16] DRQNs perform well with sequential based Atari games. Using the same concept of training our apprenticeship learning architecture with sequential data and an LSTM, it can be

shown that sequential tasks can also be learned. However, the dataset processing algorithm first needs to be modified to account for this change in experience replay storage. One problem is that the amount of steps in each episode can vary, which means that having a set trace number may cause a problem when randomly sampling from episodes. Another problem follows the first problem in that there may not be enough episodes at a certain trace length or higher, although this is rare in large datasets. To solve both problems, the episodes for the agent and expert were stored with their total lengths. Then, the hyper-parameters for maximum trace length and maximum episode number were chosen. During training on expert data, a unique trace length was randomly chosen from the trace lengths stored for the expert. If the trace length was smaller than the max it would be chosen as the trace length for that epoch, otherwise the maximum trace length was chosen. The number of episodes meeting the trace length requirement was then found. If the number of episodes found were less than the maximum number of episodes that number would be chosen as the episode number for that epoch, otherwise the max episode number would be used. If the max episode number is used, then a random sampling of eligible episodes with the specified trace length was used for training that epoch. For each chosen episode, a random point step in that episode was chosen as the start of the trace and each step after that up to the trace length was stored. The same was done when collecting and using agent replay experience. The stored steps were then fed to the DRQN to update, which used only the first half of each trace as described in [16].

Another modification made to the DRQN-AL algorithm was the addition of agent stochascity. It was found that the since the expert always displayed optimal movements the network could not generalize well when faced with non-explored states when using a recurrent model for sequential actions. Therefore, using a decaying ϵ-greedy method [9], a percentage of the agent's moves are randomly chosen. The ϵ value would start high ($\epsilon = 0.9$), meaning that the agent would prefer choosing random actions. As the agent training progressed the ϵ value would decrease meaning the agent would favor an action learned by the network. This added stochascity which prevented the network from getting stuck in a local minima, and also

allowed it to update the Q-values for all actions at each state rather than just one or two. This gives a larger range of Q-values between actions for each state helping the agent to learn faster.

Chapter 5

Implementation

5.1 Architecture Details

A number of architectures were created and tested as described in Chapter 4. The descriptions of each network tested are as follows:

- *DAL* [18]: The Deep Apprenticeship Learning architecture from [18] without any modifications.
- *DAL* [18] (no pooling): The Deep Apprenticeship Learning architecture from [18] without any pooling layers. This is a novel modification to this architecture.
- *DAL with Transfer Learning*: The Deep Apprenticeship Learning architecture from [18] where the trained DAQN network weights are used to initialize the weights of the DARN. This is a novel modification to this architecture.
- *DAL with Bellman*: The Deep Apprenticeship Learning architecture from [18] using Bellman Q-value updates for both the DAQN and DARN, as well as experience replay. This is a novel modification to this architecture.
- *DQN-AL*: Deep Q-Network for Apprenticeship Learning which uses the DQN architecture and shared experience replay. This architecture is modeled after [12].
- *DQN-AL with Scheduled Shared Experience*: Same as DQN-AL except that the expert experience for shared experience replay starts at a

high percentage and decays over time. The introduction of scheduled shared experience is a novel modification to this architecture.

- *Dueling DQN-AL*: Dueling Deep Q-Network for Apprenticeship Learning which uses the Dueling DQN architecture (with Target Q-Network) from [37] and shared experience replay. The use of Dueling Deep Q-Network in the realm of apprenticeship learning is a novel use of this architecture.

- *Dueling DQN-AL with Scheduled Shared Experience*: Same as Dueling DQN-AL except that the expert experience for shared experience replay starts at a high percentage and decays over time. The introduction of scheduled shared experience is a novel modification to this architecture.

- *DRQN-AL*: Deep Recurrent Q-Network for Apprenticeship Learning which uses the DRQN architecture from [11] with experience replay modifications explained in Section 4.2.4. The use of DRQN in the realm of apprenticeship learning is a novel use of this architecture.

The hyper-parameters for each of these architectures were tuned to ensure best results for the networks. These hyper-parameters include:

- Dataset Split: Number of training data, number of testing data, and number of random data.

- Data size: Size of input data.

- Epochs: Number of epochs for training from expert, number of epochs for training agent.

- Learning Rate (α): Rate at which network weights update.

- Discount Factor (γ): Weight of future state values.

- Reward Value: Arbitrary value for maximum and minimum award.

- Shared Experience Ratio (SER): Percentage of expert experience used when training agent.

- Shared Experience Decay (SED): Value at which SER would decrease over time.
- Shared Experience Decay Frequency (SEDF): Rate (number of epochs) at which shared experience ratio would be updated,
  $SER_{update} \leftarrow SER - SED$.
- Replay Buffer: Size of experience memory.

For the DQN implementations (DQN-AL, Dueling DQN-AL, DRQN-AL), parameters for the loss function needed to be defined:

- λ_1: Weight for temporal difference learning.
- λ_2: Weight for cross softmax entropy.

For the DRQN-AL implementation extra parameters needed to be defined. These include:

- Maximum Episode Number: Maximum number of episodes from dataset to sample from.
- Maximum Trace Length: Maximum length of trace to extract from each episode.
- ϵ-greedy Value: Starting value for ϵ for agent randomness.
- ϵ Decay: Value at which ϵ would decrease over time.
- ϵ Decay Frequency: Rate (number of epochs) at which ϵ would be updated,
  $\epsilon_{update} \leftarrow \epsilon - \epsilon_{decay}$.
- Step Punishment Value: Value of punishment the agent would experience at every step, to help ensure the agent will continue to go forward in an environment and not oscillate between states.

For each network the discount factor (γ) was kept constant at 0.9. The dataset split for task completion for all DAL [18] networks was 800 training, 200 testing, and 1,000 random with a data size of $83 \times 83 \times 1$. These were

Table 5.1: Architecture Hyper-Parameters for Task Completion.

Network	***Epochs (Expert,Agent)***	α	***SER***	***SED***	***SEDF***	λ_1	λ_2
DAL [18]	50000,100000	0.01	-	-	-	-	-
DAL [18] (no pooling)	50000,100000	0.01	-	-	-	-	-
DAL with Transfer Learning	10000,10000	0.01	-	-	-	-	-
DAL with Bellman	10000,10000	0.01	-	-	-	-	-
DQN-AL	10000,10000	0.001	0.1	-	-	0.5	0.5
DQN-AL with Scheduled Shared Experience Replay	10000,10000	0.001	0.9	0.05	500	0.75	0.25
Dueling DQN-AL	5000,10000	0.001	0.1	-	-	0.25	0.75
Dueling DQN-AL with Scheduled Shared Experience Replay	5000,10000	0.001	0.9	0.05	500	0.25	0.75
DRQN-AL	10000,100000	0.001	0.125	-	-	1.0	0.0

trained with a batch size of 50. For DQN networks it was 1,000 train (the testing and random came from self generated experiences) with a data size of $84 \times 84 \times 1$. These were trained with a batch size of 32. The dataset split for task understanding for all DAL networks was 8,000 training, 2,000 testing, and 10,000 random. For DQN networks it was 10,000 train (the testing and random came from self generated experiences). Also, for Dueling DQN the target Q-network update frequency was set for four epochs for task completion and four epochs for task understanding. The DQN implementations used a replay buffer of size 500. For the DRQN implementation, the maximum episode number was set to eight and the maximum trace length was set to 32 to make an overall count of 256 data inputs per epoch. The DRQN used a replay buffer of size 500 and an ϵ-greedy value of 0.7 which would decay by 0.1 every 10,000 epochs to a minimum of 0.0. The DRQN also employed a -1 punishment value for each step to help the network break cyclical action choices. The hyper-parameters for task completion and task understanding for each network are shown in Tables 5.1 and 5.2.

Table 5.2: Architecture Hyper-Parameters for Task Understanding.

Network	***Epochs (Expert,Agent)***	α	***SER***	***SED***	***SEDF***	λ_1	λ_2
DAL [18]	50000,100000	0.01	-	-	-	-	-
DAL [18] (no pooling)	50000,100000	0.01	-	-	-	-	-
DAL with Transfer Learning	10000,10000	0.01	-	-	-	-	-
DAL with Bellman	10000,10000	0.01	-	-	-	-	-
DQN-AL	10000,10000	0.001	0.1	-	-	0.75	0.25
DQN-AL with Scheduled Shared Experience Replay	10000,10000	0.001	0.9	0.05	500	0.75	0.25
Dueling DQN-AL	10000,10000	0.1	0.1	-	-	0.75	0.25
Dueling DQN-AL with Scheduled Shared Experience Replay	10000,10000	0.1	0.9	0.05	500	0.6	0.4
DRQN-AL	10000,100000	0.001	0.125	-	-	1.0	0.0

5.2 Algorithms

5.2.1 Deep Apprenticeship Learning Networks

The algorithm utilized for the Deep Apprenticeship Learning (DAL) networks follows what is described in Section 2.4.3 [18]. They utilize expert data to first train the DAQN to extract the expert policy. It then relies on stochastic random data to train the DARN to model the underlying reward function.

5.2.2 Deep Q-Network Apprenticeship Learning

The loss function utilized for the DQN-AL, Dueling DQN-AL, and DRQN-AL is shown in (5.1). As shown in this equation, the loss function for both learning from an expert and teaching an agent is a combination of supervised classification and temporal difference learning.

$$loss = \lambda_1(Q_{target}(s,a) - Q(s,a))^2 + \lambda_2 cse(a^{predicted}, a_E) \tag{5.1}$$

As shown, the temporal difference learning argument is characterized by $Q_{target}(s, a)$ and $Q(s, a)$. $Q(s, a)$ is the output of the network for state s and action a. $Q_target(s, a)$ is found using the Bellman update equation as shown in (5.2).

$$Q_{target}(s, a) = r(s, a) + \gamma * max_{a'} Q(s', a') \tag{5.2}$$

Where, $r(s, a)$ is the reward received for being at state s and taking action a. $max_{a'} Q(s', a')$ is the maximum output of the network for next state s' over all actions a'. The cse is the cross softmax entropy of the action predicted by the network and the actual action taken by the expert. This is to emphasize the correct optimal action an expert will take. Preceding the temporal difference argument and cse argument is λ_1 and λ_2. Both of which are hyper-parameters used to weight each argument. For learning from an expert the values of both hyper-parameters are set, but it was found that weighting the cse higher produced the best results. For agent learning λ_1 was set to 1.0 and λ_2 was set to 0 because the ground truth data was no longer being used.

The algorithm utilized for the DQN-AL, Dueling DQN-AL, and DRQN-AL architectures is shown in Algorithm 4. For the DRQN-AL implementation, the batches sampled are sequences. As shown, the networks are trained on expert data first to give it a starting point. The trained network is then used to generate more samples using an agent, and is updated using a combination of expert and agent samples.

Algorithm 4 Deep Q-Network Apprenticeship Learning

for e $\in$ number of expert training episodes **do**
  Sample a random batch of expert data $[s_E, a_E, s'_E, r_E]$
  Feed-forward s_E to obtain current Q-value (Q) for a_E
  Feed-forward s'_E to obtain the Q-values (Q') for each action
  $Q'_{max} = max(Q')$
  $Q_{target} = r_E(s_E, a_E) + \gamma * Q'_{max}$
  $loss = \lambda_1(Q_{target} - Q)^2 + \lambda_2 cse(a^{predicted}, a_E)$
  Update network with batch $loss$
Initialize environment to get initial state s
for e $\in$ number of agent training episodes **do**
  if s is a terminal state **then**
    Initialize new environment to get initial state s
  Feed-forward state s to get action a
  Play a to get s' and r
  Store $[s, a, s', r]$ in agent replay memory (replace old memories if full)
  $s \leftarrow s'$
  if replay buffer is full **then**
    Sample a random batch of expert data $[s_E, a_E, s'_E, r_E]$
    Sample a random batch of agent data $[s, a, s', r]$
    Combine batches into $[s_T, a_T, s'_T, r_T]$
    Feed-forward s_T to obtain current Q-value (Q) for a_T
    Feed-forward s'_T to obtain the Q-values (Q') for each action
    $Q'_{max} = max(Q')$
    $Q_{target} = r_T(s_T, a_T) + \gamma * Q'_{max}$
    $loss = \lambda_1(Q_{target} - Q)^2$
    Update network with batch $loss$

Chapter 6

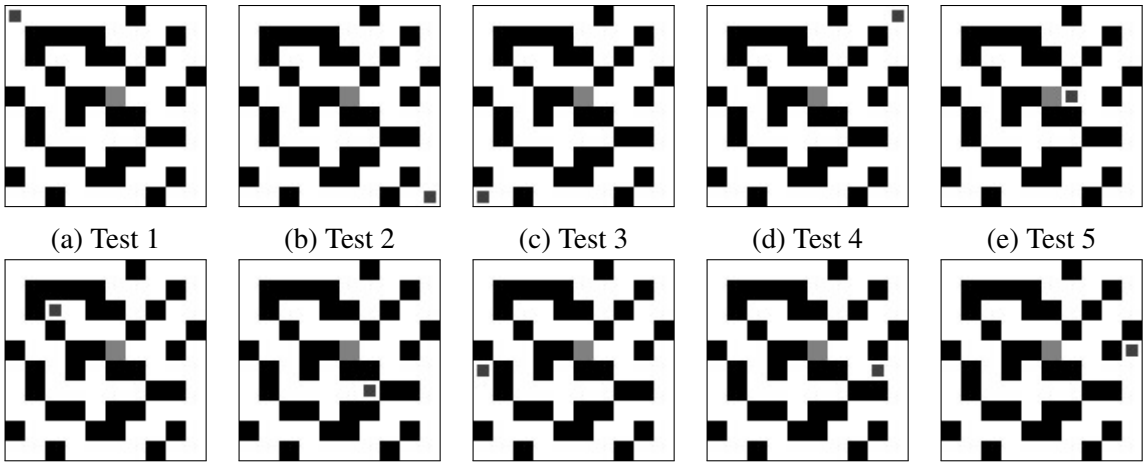
Results and Analysis

6.1 Task Completion and Task Understanding

This research explored two concepts, task completion and task understanding. Task completion is the measure of how well a system can complete a specific task. In this research, task completion was tested using the same maze structure with the agent starting in different locations. This was used to see if the agent could find the shortest path to a goal in a specific environment. This tested the systems ability to mimic the expert, as well as its ability to recognize the agent and goal state. Task understanding is the measure of how well a system can generalize the task needing to be completed. Task understanding was tested in this research using different mazes and different agent starting locations. By feeding the system with expert data from a number of different maze structures it tests the systems ability to generalize the overall task of getting to the goal despite the maze structure. This tests to see if the system was trained well enough to identify walls, unobstructed path, the agent location, goal location, and the best path towards the goal.

6.2 Test Methodology

Each trained network was tested with the same 10 fixed mazes with fixed agent starting locations. For task completion these fixed tests all had the same maze structure. For task understanding these fixed tests all had different maze structures. The agent starting locations for each tests were chosen



(a) Test 1 (b) Test 2 (c) Test 3 (d) Test 4 (e) Test 5

(f) Test 6 (g) Test 7 (h) Test 8 (i) Test 9 (j) Test 10

Figure 6.1: Task Completion Fixed Tests (10) with Agent Starting Location.

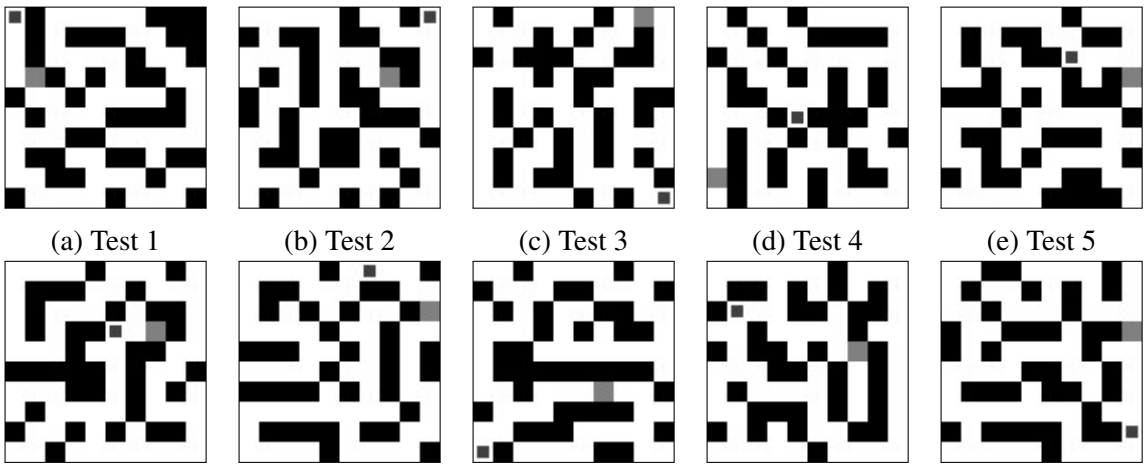
so that there was an even distribution of easy positions (next to goal or linear path to goal) and difficult (far from goal or complex path to goal). The fixed tests with agent starting locations for task completion are shown in Fig. 6.1. The fixed tests with agent starting locations for task understanding are shown in Fig. 6.2.

Success was measured in two ways; tests completed and correct action prediction. Tests completed measures the amount of tests (Fig. 6.1, Fig. 6.2) completed (agent reaches goal). Correct action prediction is a percentage of correct optimal actions predicted for all the tests. This test shows how well the network predicted actions even for difficult agent initial states.

6.3 Proposed Architecture Performances

Each architecture described in section 5.1 was tested for both task completion and task understanding using the described hyper-parameters. The results for the task completion tests are shown in Table 6.1. As shown, the DQN architectures greatly outperform the DAL architectures. Also, the addition of scheduled shared experience replay shows promising results of helping a network learn better and more efficiently.

The results for the task understanding tests are shown in Table 6.2. As shown, all the networks show poor performance for task understanding, as



(a) Test 1 (b) Test 2 (c) Test 3 (d) Test 4 (e) Test 5

(f) Test 6 (g) Test 7 (h) Test 8 (i) Test 9 (j) Test 10

Figure 6.2: Task Understanding Fixed Tests (10) with Agent Starting Location.

Table 6.1: Task Completion Results.

Network	***Completed Tests***	***Correct Action Prediction***
DAL [18]	1/10	6.4% (8/125)
DAL [18] (no pooling)	0/10	5.6% (7/125)
DAL with Transfer Learning (no pooling)	2/10	10.4% (13/125)
DAL with Bellman Implementation	2/10	8.8% (11/125)
DQN-AL	4/10	26.4% (33/125)
DQN-AL with Scheduled Shared Experience Replay	6/10	36.0% (45/125)
Dueling DQN-AL	**10/10**	**100.0%** (125/125)
Dueling DQN-AL with Scheduled Shared Experience Replay	**10/10**	**100.0%** (125/125)
DRQN-AL	2/10	27.2% (34/125)

Table 6.2: Task Understanding Results.

Network	***Completed Tests***	***Correct Action Prediction***
DAL [18]	1/10	8.4% (8/95)
DAL [18] (no pooling)	1/10	14.7% (14/95)
DAL with Transfer Learning (no pooling)	2/10	11.6% (11/95)
DAL with Bellman Implementation	2/10	13.7% (13/95)
DQN-AL	1/10	11.6% (11/95)
DQN-AL with Scheduled Shared Experience Replay	1/10	15.8% (15/95)
Dueling DQN-AL	**3/10**	**23.2%** (22/95)
Dueling DQN-AL with Scheduled Shared Experience Replay	3/10	21.1% (20/95)
DRQN-AL	1/10	13.7% (13/95)

this is a much harder challenge. The DQN architectures do outperform the DAL architectures in terms of correct action prediction.

6.4 Discussion

6.4.1 Task Completion

As shown in Table 6.1, the DAL networks show poor performance in completing the maze task. However, there is a substantial increase in the transfer learning without pooling network (DAL with Transfer Learning), but there does not seem to be any significant or note-worthy performance increases using this network over DQNs. This in part has to do with the simplicity of this network being only two convolution layers deep. Also, this network architecture performed well in reward based environments and was only ever compared to non-deep algorithms [18]. These results show that this type of network is not effective in task completion environments and would not fair well in real world environments.

The DQN architectures, on the other hand, showed much more promising results. The DQN-AL architecture performed a bit better in correct action prediction due to its complex architecture made to understand complexities in different worlds. When paired with scheduled experience replay, the

network performed considerably better in both test completion and correct action prediction. This shows that the edition of the scheduled experience replay gives the system a guide throughout its training. The Dueling DQN-AL architecture performed much better in both test completion and correct action prediction. Much of this is a credit to the stability of the Q-network using a target Q-network as well as the separation of the fully-connected layer in the Dueling DQN architecture. In addition, the Dueling DQN architecture was tested with scheduled shared experience as well. This displayed similar performance showing a promising contribution of scheduled shared experience.

The DQN network results also show that the abstract reward values were also a success. By setting the rewards in the Bellman Q-value updater to be descriptive of the completion of the task, abstraction of the reward from the step level to the task level was achieved. With this modification, DQNs were able to be utilized in the apprenticeship learning space where environments have unknown reward structures.

The DRQN implementation did not fair well in the task completion tests. However, much of this may be due to the setting of the hyper-parameters for each application, as explained in Section 2.2.4. Also, because the mazes used only have one solution, the use of a recurrent layer does not make sense for such a simple problem.

6.4.2 Task Understanding

Unfortunately, each of these networks performed poorly when being tested on task understanding, as shown in Table 6.2. The DAL networks performed the same as they did in task completion, further iterating that this type of network architecture is not appropriate for task-based apprenticeship learning. The DQN networks did show much improvement over the DAL networks. Using scheduled shared experience resulted in improved performance. The DRQN implementation also did poorly and shows that a recurrent solution to this problem may not be viable.

Task understanding through raw pixel data is still in its infancy in terms of research. While DQNs have shown to produce superhuman performance on Atari games [22], it does not do well in understanding environments with

unknown reward structures. While the concept of reward abstraction may resemble a reward structure, it is not flexible enough to abstract complex reward structures that may lie in an environment. For example, with the random structured mazes the underlying reward function needing to be learned involved agent proximity to the goal, each wall block, and each open space, it also involved the goal proximity to the agent, each wall block, and each open space. This leads to a very complex, non-linear reward structure that is difficult to decipher using only raw pixel data and no other information. While adding locality information to the data may achieve better results, it does not address the real-world problem of using raw video footage to teach an agent to perform a task.

Chapter 7

Conclusions and Future Work

This research shows promising performance for task completion using deep apprenticeship learning. Each novel contribution introduced allowed for better learning from an expert using an apprenticeship learning model. The key benefits for each contribution are outlined below.

7.1 Remarks on Novel Contributions

7.1.1 Reward Abstraction

An issue with deep apprenticeship learning was the need to learn an underlying reward structure. By bypassing that step by using reward abstraction from step level to task level, common Bellman equations and Q-learning algorithms could be used to train a network. The ability to use these networks and equations allowed for better learning as shown in the results. Giving a large positive reward for completing a task, a large negative reward for failing a task, and a punishment for time to complete a task brought about concepts in reinforcement learning that could be directly tied to apprenticeship learning.

7.1.2 Scheduled Shared Experience Replay

The addition of scheduled experience replay showed the benefits of giving a learning agent a strong guide in its infancy and a weaker guide at its maturity. As the agent started out learning, its memory was overtaken by the memory of the expert. As the agent learned over time, the memory of the expert would slowly fade, leaving only its own memory. As shown in the

results, the scheduled shared experience methodology allowed an agent to learn more efficiently and better than networks that did not use scheduled shared experience.

7.1.3 Dueling Deep Q-Network Architecture

Using the DQN methodology of target Q-networks brought stability to the networks while training, which was lacking in the DAL implementations. Dueling DQN architecture implementations also proved to be the superior network architecture for task completion as its separation of value and advantage modeled the task completion problem well as shown in the results. More research is needed in the problem space of task completion. While scheduled shared experience and DQN implementations proved to be better than the traditional DAL implementations, the novel contributions that succeeded in task completion did benefit the networks in task understanding.

7.1.4 Deep Recurrent Q-Network Architecture

An implementation that may allow for better task understanding is a recurrent layer that can learn sequences over time. While this was shown in the DRQN-AL implementation, the hyper-parameters for maximum episode, maximum trace length, and expert data percentage need to be finely tuned for the problem. However, it was shown that by learning sequences rather than state-action pairs, the agent was able to learn from expert demonstrations, although this task does not need the extra complexity of an LSTM layer. This extra complexity resulted in worse performance.

7.2 Challenges and Future Work

7.2.1 Datasets and Benchmarking

One problem with deep apprenticeship learning with raw pixel data is the unavailability of datasets with both expert data and a simulation environment. This makes it difficult to benchmark against other networks. For this research, a dataset with both these criteria was created, however, no

dataset like this is available for public use. One extension to overcome this is to move from a simulation environment to a physical environment. Using video data from an agent in a physical environment doing a task, that same agent can learn from the expert video data. While this would eliminate the issue with dataset availability, it comes with its own issues in the realm of hardware, video capture, and extra environment variables, such as lighting and camera angle. Another solution is to use existing trained DQNs that have learned Atari games, using the Atari 2600 simulator, at a super-human level to generate expert data. This would allow for a benchmark against existing DQN architectures and to show how well apprenticeship learning measures up. Also, since the Atari 2600 game space has a number of games ranging in difficulty it would allow for testing the robustness of these algorithms.

7.2.2 Overfitting

Expert video data often has limited stochasticity associated with it, as the expert tends to perform optimally. Training on this type of data can make a system over fit to the expert. One solution to overcome this is to introduce stochasticity in the agent to help mitigate this issue. This would allow the agent to generalize over states the expert may not have explored. Another solution to this is to train the agent using apprenticeship learning, then use the trained network as a way to generate expert data to train the agent again. This would introduce stochasticity in the expert data since the trained network will not be perfect. This cycle of "self-learning" can be performed many times until optimal results are achieved.

7.3 Applications

Overall, this research shows the promise of deep apprenticeship learning's extension to unknown reward structures and real-world environments. The possibilities of this technology are endless, and can be used to better understand the learning process for humans. One example application of this research includes autonomous robotic surgery where the system learns proper cutting techniques from video data from surgeons. Another example is in

the realm of factory robots, where they could learn how to work in a heterogeneous environment with humans by watching human interactions.

** All code, datasets, and assets for this research are hosted at:*

- *https://kgcoe-git.rit.edu/aab2210/teaching-agents-with-deep-apprenticeship-learning-thesis*
- *https://github.com/AmarBhatt/RIT_Thesis*

Bibliography

[1] How to implement epsilon greedy strategy / policy. https://junedmunshi.wordpress.com/2012/03/30/how-to-implement-epsilon-greedy-strategy-policy/. Accessed: 2015-11-21.

[2] Q-learning by examples. http://people.revoledu.com/kardi/tutorial/ReinforcementLearning/. Accessed: 2015-11-15.

[3] Random maze generator (python recipe) by fb36 activestate code (http://code.activestate.com/recipes/578356/).

[4] Research areas in social psychology. http://psychology.about.com/od/socialpsychology/p/socialresearch.htm. Accessed: 2015-31-3.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu,

and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] Christopher G Atkeson and Stefan Schaal. Robot learning from demonstration. In *ICML*, volume 97, pages 12–20, 1997.

[7] Casey Bennett. Robotic faces: Exploring dynamical patterns of social interaction between humans and robots. 2015.

[8] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. *arXiv preprint arXiv:1604.06778*, 2016.

[9] Tim Eden, Anthony Knittel, and Raphael van Uffelen. Reinforcement learning. `http://www.cse.unsw.edu.au/~cs9417ml/RL1/index.html`. Accessed: 2015-11-22.

[10] Keita Hamahata, Tadahiro Taniguchi, Kazutoshi Sakakibara, Ikuko Nishikawa, Kazuma Tabuchi, and Tetsuo Sawaragi. Effective integration of imitation learning and reinforcement learning by generating internal reward. In *2008 Eighth International Conference on Intelligent Systems Design and Applications*, volume 3, pages 121–126. IEEE, 2008.

[11] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

[12] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, et al. Learning from demonstrations for real world reinforcement learning. *arXiv preprint arXiv:1704.03732*, 2017.

[13] Rishabh Jangir. Apprenticeship learning using inverse reinforcement learning, Jul 2016.

[14] Ming Jin and Costas Spanos. Inverse reinforcement learning via deep gaussian process. *arXiv preprint arXiv:1512.08065*, 2015.

[15] Henrik Kretzschmar, Markus Spies, Christoph Sprunk, and Wolfram Burgard. Socially compliant mobile robot navigation via inverse reinforcement learning. *The International Journal of Robotics Research*, page 0278364915619772, 2016.

[16] Guillaume Lample and Devendra Singh Chaplot. Playing FPS games with deep reinforcement learning. *CoRR*, abs/1609.05521, 2016.

[17] Sergey Levine, Zoran Popovic, and Vladlen Koltun. Nonlinear inverse reinforcement learning with gaussian processes. In *Advances in Neural Information Processing Systems*, pages 19–27, 2011.

[18] Dejan Markovikj. Deep apprenticeship learning for playing games. Master's thesis, Department of Computer Science, University of Oxford, 2014.

[19] Tambet Matiisen. Guest post (part i): Demystifying deep reinforcement learning - nervana, Dec 2015.

[20] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, 2016.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control

through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[23] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, pages 663–670, 2000.

[24] Vien Ngo and Marc Toussaint. Inverse reinforcement learning.

[25] John P O'Doherty, Peter Dayan, Karl Friston, Hugo Critchley, and Raymond J Dolan. Temporal difference models and reward-related learning in the human brain. *Neuron*, 38(2):329–337, 2003.

[26] Jiansheng Peng. Mobile robot path planning based on improved q learning algorithm. *International Journal of Multimedia and Ubiquitous Engineering*, 10(7):285–294, 2015.

[27] Noé Pérez-Higueras, Rafael Ramón-Vigo, Fernando Caballero, and Luis Merino. Robot local navigation with learned social cost functions. In *Informatics in Control, Automation and Robotics (ICINCO), 2014 11th International Conference on*, volume 2, pages 618–625. IEEE, 2014.

[28] Qifeng Qiao and Peter A Beling. Inverse reinforcement learning with gaussian process. In *American Control Conference (ACC), 2011*, pages 113–118. IEEE, 2011.

[29] Deepak Ramachandran and Eyal Amir. Bayesian inverse reinforcement learning. *Urbana*, 51(61801):1–4, 2007.

[30] Constantin A Rothkopf and Dana H Ballard. Modular inverse reinforcement learning for visuomotor behavior. *Biological cybernetics*, 107(4):477–490, 2013.

[31] Sahand Sharifzadeh, Ioannis Chiotellis, Rudolph Triebel, and Daniel Cremers. Learning to drive using inverse reinforcement learning and deep q-networks. *arXiv preprint arXiv:1612.03653*, 2016.

[32] Nathan Sprague and Dana Ballard. Multiple-goal reinforcement learning with modular sarsa (0). In *IJCAI*, pages 1445–1447. Citeseer, 2003.

[33] Richard S Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in neural information processing systems*, pages 1038–1044, 1996.

[34] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995.

[35] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[36] Dizan Vasquez, Billy Okal, and Kai O Arras. Inverse reinforcement learning algorithms and features for robot navigation in crowds: an experimental comparison. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1341–1346. IEEE, 2014.

[37] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[38] Christopher J. C. H. Watkins and Peter Dayan. Technical note: q-learning. *Mach. Learn.*, 8(3-4):279–292, May 1992.

[39] Markus Wulfmeier, Peter Ondruska, and Ingmar Posner. Maximum entropy deep inverse reinforcement learning. *arXiv preprint arXiv:1507.04888*, 2015.

[40] Chen Xia and Abdelkader El Kamel. Neural inverse reinforcement learning in autonomous navigation. *Robotics and Autonomous Systems*, 2016.

[41] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *AAAI*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008.