Modeling Decisions in Games Using Reinforcement Learning

Conference Paper · October 2017

DOI: 10.1109/MLDS.2017.13

3 authors, including:

Palvi Aggarwal
Indian Institute of Technology Mandi
12 PUBLICATIONS **12** CITATIONS
SEE PROFILE

Varun Dutt
Indian Institute of Technology Mandi
117 PUBLICATIONS **672** CITATIONS
SEE PROFILE

Modeling Decisions in Games Using Reinforcement Learning

Himanshu Singal, Palvi Aggarwal, Dr. Varun Dutt
Applied Cognitive Science Lab
Indian Institute of Technology Mandi, India
singalh2@gmail.com, palvi_aggarwal@students.iitmandi.ac.in, varun@iitmandi.ac.in

***Abstract*— Reinforcement-learning (RL) algorithms have been used to model human decisions in different decision-making tasks. Recently, certain deep RL algorithms have been proposed; however, there is little research that compares deep RL algorithms with traditional RL algorithms in accounting for human decisions. The primary objective of this paper is to compare deep and traditional RL algorithms in a virtual environment concerning their performance, learning speed, ability to account for human decisions, and ability to extract features from the decision environment. We implemented traditional RL algorithms like imitation learning, Q-Learning, and a deep RL algorithm, DeepQ Learning, to train an agent for playing a platform jumper game. For model comparison, we collected human data from 15 human players on the platform jumper game. As part of our evaluation, we also increased the speed of the moving platform in the jumper game to test how humans and model agents responded to the changing game conditions. Results showed that DeepQ approach took more training episodes than the traditional RL algorithms to learn the gameplay. However, the DeepQ algorithm could extract features directly from images of gameplay; whereas, other algorithms had to be fed the extracted features. Furthermore, conventional algorithms performed more human-like in a slow version of the game; however, the DeepQ algorithm performed more human-like in the fast version of the game.**

Keywords—reinforcement learning; deep reinforcement learning; imitation learning; Q-learning; Neural Networks; DeepQ learning

I. INTRODUCTION

Machine learning deals with getting computers to act without being explicitly programmed [1]. In the past decades, machine learning has been widely used in a wide variety of applications: generating recommendations on e-commerce websites, driving cars, content filtering, speech recognition, effective web search, and an improved understanding of the human genome [1].

Although machine learning has a broad area of applications; however, traditional machine-learning techniques are unable to process datasets in their raw formats. To overcome this limitation, representational-learning algorithms have been used along with conventional machine-learning techniques [1]. Deep learning removed this barrier of manual feature extraction: it is possible to directly feed raw data to deep learning algorithms and obtain the desired results [1]. In deep learning, between input and output layer, there are one or more hidden layers where each layer transforms the data received from a previous layer to a more abstract level [1]. Thus, deep neural networks can automatically find compact low-dimensional representations (features) of high-dimensional data (e.g., images, text and audio) [2].

Reinforcement Learning is a form of machine-learning technique that is goal-oriented [3]. In the reinforcement-learning technique called Q-learning, an agent learns by interacting with the environment across several time steps [3]. At each time, the agent chooses an action from the set of available actions, the environment moves to a new state, and the reward associated with the transition is determined. The goal of RL agent is to maximize her rewards over time [3].

Deep learning, involving learning directly from raw pixels, could be applied to reinforcement learning too. Mnih V. et al. [4] proved that such an approach is possible and can yield improved results: Mnih V. et al. [4] showed how a DeepQ approach could be used to train an agent to play several Atari games at a superhuman level from image pixel. The DeepQ algorithm used layers of convolutional networks to understand the image provided as raw input and used the output of the convolutional layers to generate Q-values for different actions. This additional intelligence of self-extracting the features from raw inputs comes at an extra computation cost [2]. Thus, one likely needs a number of training episodes in DeepQ algorithm compared to traditional RL technique like Q-learning.

Although several deep RL algorithms have been proposed; however, currently little is known about how deep RL techniques compared to traditional RL techniques for accounting for human decisions in different decision tasks. The main research questions that are less investigated include: What is the learning rate of these algorithms compared to human learning? How do these algorithms respond to changes in the decision environment? And, what approach (deep or traditional) should one follow in different simple or complex decision tasks?

To answer these questions, in this paper, we tested deep and traditional RL techniques in a virtual environment. Our testing environment is a jumper game, where the goal is to reach the final state from the initial state by smartly jumping on different platforms. First, we made several human players play multiple rounds in simple and complex versions of the jumper game and collected human data. Next, we trained several traditional and deep RL algorithms to account for human decisions in the jumper game. The algorithms trained by us included: imitation learning [5], Q-learning [6] and DeepQ learning [4], [7].

In what follows, we first detail background literature on different RL algorithms. This background literature is followed by a description of the platform jumper game, human data collection, and the working of various RL algorithms. Next, we

detail results from different RL algorithms in their ability to account for human decisions in the jumper game. Finally, we close this paper by discussing the implications of our findings for using traditional and deep RL algorithms for accounting for human decisions in simple and complex decision tasks.

II. Background

One way to test an Artificial Intelligence (AI) algorithm is likely via simulation games [8]. A real-world scenario may seem a more natural setting; however, a virtual environment seems more practical [8].

About 15-20 years back, the AI for board games like chess and checkers was being developed [9]. The rules of such games were known in advance. These rules were used to produce game trees [9]. Some outstanding programs like *Deep Blue* for *Chess* [10] and *Chinook* for *Checkers* [11], [12], were developed on the same approach. Overall, these programs generated enormous game trees and carried out on some optimized search over these trees to create successful game bots.

In today's time, with the increase in complexity of video games and the game rules not being entirely known in advance, the game tree approaches cannot be used. For example, consider a platform jumper game where the task is to jump on several static or moving platforms and reach a goal. A user may likely know the goal state; however, she might not know how to go from the start state to the goal state and how to overcome obstacles in between. Thus, generation of game trees becomes a difficulty in such situations. The player may learn either by imitation [5] or by experience [3].

Imitation learning is the act of following actions of a trained player in the game [13]. It can be used by recording how a trained player performed in a situation while playing the game. From this data, the bot tries to find the closest state to its current state and imitates the action taken by the trained player in that state.

Imitation learning has been used to teach robots various tasks such as flying vehicles [14], [15] and drive self-driving cars by imitating an expert driver [16]. Also, it has been used to create an AI for first person shooter games [13].

Beyond imitation learning, an alternative learning approach could be learning from experience [17]. Learning from experience means a player learn about rewarding and punishing actions through positive and negative outcomes from the environment [18]. Reinforcement Learning [3] is a classic machine learning technique where an agent learns by experience gained by interacting with the environment. It is a kind of action selection method which is governed by a feedback mechanism [3]. Unlike imitation learning, it does not need to be supervised with data. Q-learning is a model-free reinforcement learning technique [6]. An agent tries an action in a state, evaluates the action taken based on reward given and estimates their Q-values. By doing it repeatedly for all actions in all states, it can learn the best action in each state, determined by a long-term discounted reward.

Reinforcement learning has been used in robotics, e.g., training a robot to return table tennis ball [19], two degrees of freedom crawling motions [20], teaching a caterpillar robot to move forward [21], and path planning a mobile robot [22]. Q-learning have been used in video games too where AI becomes smarter as it keeps on playing [23]. To name a few, RL-DOT was designed using Q-Learning for an Unreal Tournament domination game [24], behaviors such as overtaking in The Open Racing Car Simulator (TORCS) were learned using Q-learning [25], [26].

Traditionally, Q-values for different states are stored in a 2D array or similar structure. The dimensions of the array will be (no of states * no of actions). Q-learning can also be implemented using a single-layered neural network [27]. The input for this neural network can be one hot encoded vector of length equal to the no of states. Thus, each bit of the input represents a state. The output of this network will be of length equal to the number of actions, and each output node will represent the Q-value of an action. Thus, inputting the state into the neural net will yield the Q-values computed. Such an approach was used by B.Q. Huang et al. [27] for a mobile robot to avoid obstacles.

Is it possible to extend this one layered neural network to yield better results? Though Q-learning performs reasonably well, it faces a problem in the form of extracting states. While playing a video game, the natural input is in the form of a video. Learning to control agents directly from the video input had been a challenge for reinforcement-learning algorithms. In Q-learning, the states need to be defined. One cannot simply give the algorithm a video feed and expect it to take action.

This problem was solved with DeepQ algorithms [4]. The revolutionary paper by Mnih V. et al. [4] demonstrated the power of DeepQ algorithms by training it to superhuman levels in Atari games. It was made possible by using convolutional neural networks along with Q-learning. Later, DeepMind released another paper which used dueling to yield better and more stable architecture [7].

In this paper, we investigate certain deep and traditional reinforcement-learning algorithms that perform in a most human-like fashion. To test these algorithms, we implemented a platform jumper game, where the agent had to reach a final state from a start state. Human participants were made to play the game, and their data was recorded. Furthermore, we test an imitation learning model, Q-learning model, and a DeepQ model play the jumper game to evaluate their performance against human participants.

III. The Jumper Game Environment

The jumper game was developed by following the tutorials on Arcade Games with pygame and Python [28]. Their game was modified to a single screen moving platform jumper game where the motive is to reach from the start to the final state of the game. During the game, there are five actions possible which are listed in Table I.

TABLE I. Possible Actions

Actions	Outcome

Left	Moves the player to its left.
Right	Moves the player to its right.
Jumpl	Jumps the player to its left.
Jumpr	Jumps the player to its right.
Stay	Player takes no action

The game environment is shown in figure 1. The red rectangular block is the player. The green rectangular blocks are the platforms on which the player can jump. Let's call the bottommost platform P1, the middle platform P2, and the topmost platform P3. G refers to the goal point, where the player needs to reach to end the game. P2 is a moving platform. The figure 1a shows the player at the start state and figure 1d shows the player at the goal state (figure 1b and 1c show intermediate player positions).

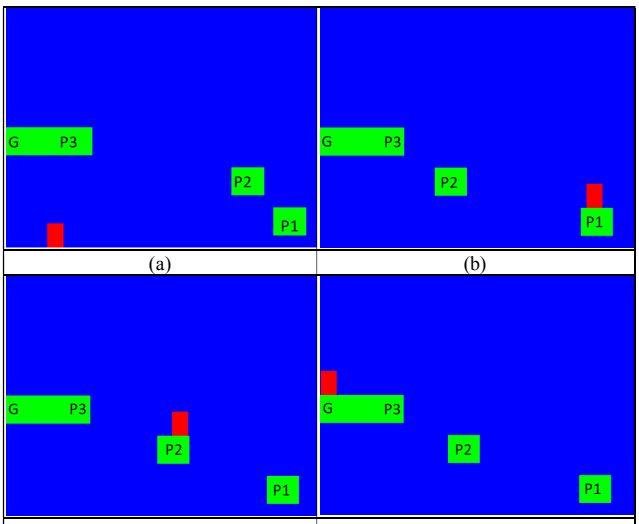

Fig. 1. The Jumper Game, where the player is in red color. (a) Shows the initial position of the player and the platforms P1, P2, P3 and the goal state G. (b) Shows the player waiting for moving platform, P2. (c) Shows the player on moving platform, P2. (d) Shows the goal or the final state G of the player.

During the gameplay, the agent has two targets; one is the global target, i.e., the goal state, and the other is the local target, i.e., the intermediate next platform to reach the goal state. There are four local targets, first three being the platforms and the last one being the final goal. A local target for a player can be any value from 1, 2, 3, 4, where the higher local target means closer to the goal. Precisely speaking, the local target 1 is at P1, 2 is at P2, 3 is at P3, and 4 is at G. For example, in figure 1a, the agent first should reach the platform P1 to reach its ultimate target G so; its local target is 1. Similarly, in figure 1b, the new local target for an agent is 2, and in figure 1c, the new local target for an agent is 3. In figure 1d, to reach the goal state, the local target of the agent is 4 which is same as its global target.

For each action, the game provides a reward. Consider the player wishes to take an action *A*. Before taking action, let his local target was L, and his distance from L was distL. As the player has taken action *A*, it may or may not have changed his local target. The local target changes if the player jumps on a platform from the ground, or jumps on a new platform or jumps to the ground from a platform. Let's call his new local target L' and his distance from L' is distL'. Now, the reward provided by the game is a function of action A, L, L', distL and distL'.

If an agent chooses such an action which results in its change of local target and brings him closer to the global target state, i.e., L < L', the game gives a +400 reward. However, if the new local target is away from the previous target, i.e., L' < L, the agent will get -400. The L < L' means the agent is now closer to its global goal.

If the local target does not change, i.e., L==L', and the local target is not 2, i.e., platform 2, then the reward is the difference in the distances, (distL-distL') – Penalty of action A. The difference in distL and distL' ensures that moving closer to the local target gives a positive reward and moving farther gives a negative reward. A penalty is also associated with keeping the same local target even after an action. This penalty is action dependent where 0.5 for stay action and 5 for other actions. The stay action is given a less penalty because stay action contributes less in score calculation as shown in equation 2. The lesser the score, the better, thus the game gives priority to stay action by giving it less penalty than others.

In case if L==L'==2, the game does not include the difference in the distance while giving the reward. The reason being 2 is a moving platform. Thus, the distance changes irrespective of what action the agent takes.

Reward (A, L, L', distL, distL')

1. if(L==L'=2)
3. Reward = Penalty(A)
4. else if (L==L')
5. Reward = (distL – distL') – Penalty(A)
6. else if (L < L')
7. Reward = +400
8. else (L > L')
9. Reward = -400

Penalty (A)

1. if (A=stay)
3. Penalty = 0.5
4. if (A€{left, right, jumpl, jumpr})
5. Penalty = 5

The jumper game has been implemented in two versions: Slow and Fast. During the slow jumper game, the moving platform P2 moves 1 pixel per frame. However, for fast jumper game, the moving platform P2 moves 4 pixels per frame. Total 15 rounds of the jumper game were played, first 5 being the slow rounds and last 10 being fast.

The action vector for a round i is defined as:

Action Vector = [N(left), N(right), N(jumpl), N(jumpr), N(stay)] (1)

where N(A) means the count of action A from start to goal state. Action Vector is needed to calculate the score and Mean Squared Deviation. The score for a round is calculated as:

*Score = N(left) + N(right) + N(jumpl) + N(jumpr) + 0.3*N(stay)* (2)

The lesser the score, the better the performance. The human players and different models were made to play the jumper game. To evaluate the performance of models against human players, we choose Mean Squared Deviation (MSD) as a performance measure. MSD is the average squared difference between human actions and model actions. The MSD_{All} (over all the rounds) is calculated for the Action Vector (Equation 1) with respect to humans over all the cycles as

$$MSD_{All,M} = \sum_{i=1}^{15} \frac{\sum_{j=1}^{5}(AV_{M,i,j} - AV_{H,i,j})^2}{15} \quad (3)$$

Where $MSD_{All,\,M}$ is mean squared deviation over all the rounds of model M. $AV_{M,\,i}$ is the action vector of model M at i_{th} round and $AV_{H,\,i}$ is the action vector of human player at i_{th} round. Furthermore, we calculated MSD separately for slow and fast rounds given as

$$MSD_{Slow,M} = \sum_{i=1}^{5} \frac{\sum_{j=1}^{5}(AV_{M,i,j} - AV_{H,i,j})^2}{5} \quad (4)$$

$$MSD_{Fast,M} = \sum_{i=6}^{15} \frac{\sum_{j=1}^{5}(AV_{M,i,j} - AV_{H,i,j})^2}{10} \quad (5)$$

IV. Human Data

Fifteen participants in the age group of 20-22 years (Average = 21.07 years and standard deviation = 0.58 years) participated in the jumper game. All the participants were males. They were from an engineering background, studying Computer Science and Engineering, Mechanical Engineering or Electrical Engineering.

A. Experimental Setup

Participants could either choose no action for the game or could press one of the arrow keys. The keys were mapped to the actions as shown in Table II.

TABLE II. Key Action Mapping

Key	Action
Left Arrow	left
Right Arrow	right
Up Arrow	jumpr
Down Arrow	jumpl

The players were only informed that they had to use these four keys without telling the mapping to actions. If they were inactive for more than 20 frames, it was counted as a stay.

Each participant first played 5 rounds of a slow game, then advanced to the faster game in which the platform moved at a 4x speed. For each round, the total number of actions and scores were recorded in a file Also, for each action taken by participants, the experience tuple *<action number, state, action, new state>* was also stored in the file for an elaborate record. A ***state*** was defined as a vector ***[X, Y, dis1, dis2, dis3],*** where X, Y referred to the X, Y coordinates of the player, dis1 was the distance from the platform P1, dis2 was the distance from the platform P2, and dis3 was the distance from the platform P3.

B. Results

Figure 2 presents the number of actions and score for each round averaged over all the participants. Initially, due to lack of experience, the score is high. However, the score starts decreasing till round five. It further increases in the 6th round due to transition from the slower version of the game to the faster version of the game. After that, the score starts decreasing again as the players adapt themselves to the changed environment.

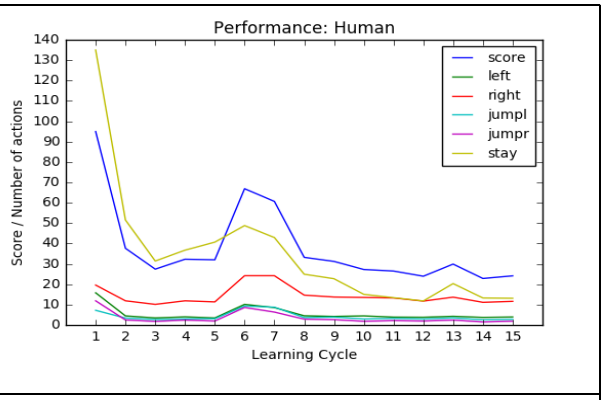

Fig. 2. The performance of humans in the Jumper game. The x-axis denotes the round number. Y-axis denotes the score or number of actions (left, right, jumpl, jumpr) taken to reach the goal state. The First five cycles are for the slow game, and the last ten cycles are for the fast game.

V. Models

Three models were selected to play the jumper game. These models are imitation learning, Q learning, and DeepQ network (DQN) model.

A. Imitation Learning

In a state, the model calculates Euclidean distance between the model state and the human states using the human state-action pairs. If the distance is less than a pre-defined threshold, then that state is called the relevant state. Next, the probability of each action in the model state is calculated. Each relevant state has an action mapped to it. Say the x number of relevant states have an action left. Then the probability of choosing a

left action is given by P(left) = x/(total number of relevant states). Similarly, probability for other actions is calculated.

The action is taken based on the probabilities calculated as mentioned above. The imitation learning algorithm was made to play the jumper game for 15 rounds. Model played first 5 rounds of slow games followed by 10 rounds of fast games. Ten simulated players participated in the jumper game.

B. Q Learning

Q-Learning [6] is a type of model-free reinforcement learning algorithm. The objective of this algorithm is to select optimal actions for a given Markov Decision process. Q learning algorithm maintains a Q-table that consists of states and actions. It calculates the rewards associated with each action in a state. Initially, all the rewards are instantiated as zero. The Q-table gets updated using the following Bellman equation (see equation 6 below). The Q value of an action *At* in a state *St* is determined by equation 6:

$$Q_{St,At} = Q_{St,At} + \alpha(r_t + \Upsilon(\max_A Q_{St+1,A}) - Q_{St,At}) \quad (6)$$

Where, $Q_{St,At}$ is the Q value for a given state *St* and an action *At* at time t. The r_t is the reward observed on taking action *At*. The α is the learning rate, and γ is the discount factor. The discount factor allows us to decide how important the possible future rewards are compared to the present reward. For a given state *St* at time t, the action with highest Q value is most suitable.

For the jumper game, the set of actions includes At ϵ {left, , right, jumpl, jumpr}, the learning parameter *(α)* is set to 0.01, and the discount factor *(γ)* is set to 0.85. For exploration in the jumper game, we use another parameter, i.e., the probability *e*. The value of *e* is set to be high (0.15) for initial rounds and low (0.03) for later rounds for both slow and fast variants of the jumper game (these values were obtained using trial-and-error). Thus, for a state St at time t, we choose a completely random action with the probability *e* or the action with highest Q value with probability (1-*e*). The jumper game was divided into 33 states, and a state structure was maintained as *<X, Y> coordinate Dis1, Dis2, Dis, Q-Value of Actions = {'left':v1, 'right':v2, 'jumpl':v3, 'jumpr':v4,"stay":v5}*. Here, *<X, Y>* are the state's coordinates. Dis1 refers to the distance from platform P1, Dis2 from platform P2 and Dis3 from platform P3.

When the player is in a state, the most similar state (out of the 33 initialized) is retrieved using *similarity*. The *similarity* is calculated based on the Euclidean distance between the player's coordinates and state's coordinates. The lesser is the distance, the more the similarity. If more than one state is fetched, then we use second similarity method. Let the distance of player from platform P1 be dis1', from P2 be dis2' and from P3 be dis3'. Euclidean distance between the distance of retrieved states from platform <dis1, dis2, dis3> and the distance between player from platform <dis1', dis2', dis3'> is calculated. The lesser the distance, the more the similarity hence the action associated with the more similar state is chosen.

C. DeepQ Network

Deep Q Network (DQN) [4] is a first deep reinforcement learning algorithm to demonstrate how an AI agent can learn to play games by just observing the screen without any prior information about those games. In *DQN* algorithm, a neural network is used to approximate the reward based on the state. The model agent should be able to process game's screen output. The use of convolutional network helps to consider the image regions instead of independent pixels. Unlike the state-based algorithms, where the states should be initialized at proper places, here we just provide the images of the jumper game to the algorithm. The implementation was done in python using Tensor Flow library [29].

1) Preprocessing and Model Architecture

Working with the raw frames of the jumper game, which was 800 x 600 pixels image, can be computationally complex. So, we applied basic preprocessing and compressed the pixels to a lower pixel size (84, 84, 3). DQN model uses four convolutional layers, which downsamples the data. This down-sampling is followed by one fully connected layer. We downsample the data using strides and "VALID" padding. Figure 3 shows the architecture of our DQN with different assumptions. The blocks indicate the feature maps; the arrows in between them indicate the weights.

The compressed image was passed through the four convolutional layers as shown in the figure. The output of the convolutional layers is flattened to an array of dimension 512*1 which is then split into advantage and value feeds [13]. The feeds are added to get the *Q values*. Let Q(s, a) be the Q value of an action *a*, in the state *S*. It can be broken into *value* and *advantage*. Value, i.e., V(s) is the measure of how good a particular state is, Advantage, i.e., A(s, a) tells how good a particular action *a* is in given state *s*.

$$Q(s,a) = A(s,a) + V(s)$$

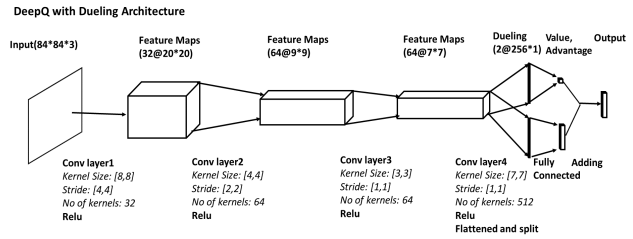

Fig. 3. DeepQ Architecture

The reason for splitting Q-value into two parts is that a state might be good even if it is not advantageous to carry out any action. Thus, it gives a more accurate representation of how good a state is. It helps to achieve greater performance and stability. For updating the Q-values, we used back-propagation where the goal was to minimize the squared loss between the predicted and targeted Q-values. Targeted Q values are found by using the Bellman equation (6). We use the experience replay [4] method to update the weights. The idea behind this

concept is storing the experience which the agent observes during its gameplay and then randomly choosing a batch of the previous experiences to update the weights [4]. It prevents the agent to learn just what it is immediately happening, but instead learn from past experiences too. The experiences are stored as a tuple of <*state, action, reward, new state*>
We chose a batch size of 32 tuples, and the update frequency was set at 4. Thus, the weights were updated after every four actions by using 32 random experiences from our experience tuple store. Also, we used a separate target network [4], [7]. Instead of having just one network, we had two networks, one main network, and another target network. The weights of the target networks are slowly updated to that of the main network. The Q-values from the target Q-Network is used to calculate the loss and update Q-values in the main Q network [4]. This stabilizes the learning procedure. The reason being the Q-values are updates at every training step. Thus, the values may spiral out. Adding one more network which is updated slowly stabilized the learning [4].

VI. Results

A. *Imitation learning*

Figure 4 represents the performance of imitation learning over the 15 rounds against human participants. The initial score is high, and it starts decreasing till the 5th round. As the model reaches 6th round, there is a transition from the slower version of the game to the faster version of the game. The score increases due to a new environment, and it again starts decreasing as the model adapts itself to the changed environment. Other actions were also compared against human data. The overall performance of the model was like human participants. The MSDs obtained for all the rounds, slow rounds and fast rounds were very low as shown in Table III. The reason good similarity is learning through human data. In this model, the strategy of human was fed to the model. Thus, imitation learning is a good method to build human-like agents.

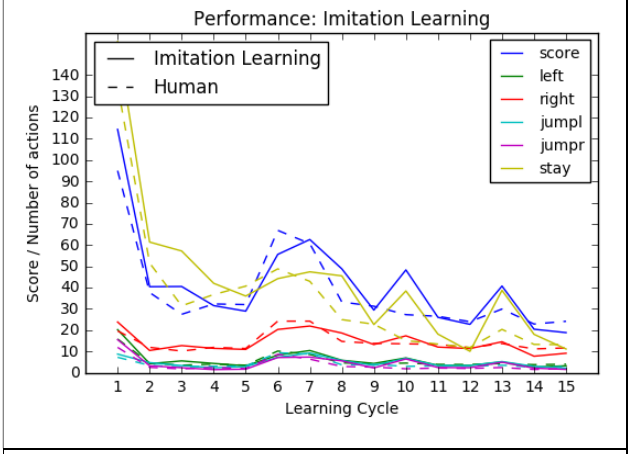

Fig. 4. Shows the performance of Imitation Learning in the Jumper Game, the x-axis denotes the round number. Y-axis denotes the score or number of actions (left, right, jumpl, jumpr) taken to reach the goal state.

TABLE III. Mean Squared Deviation, Imitation Learning

Method	Value
MSD_{All}	190.88
MSD_{Slow}	262.31
MSD_{Fast}	155.17

B. *Q-Learning*

Figure 5 shows the performance of Q-learning model against human participants. The model was not able to perform like humans during the slow rounds. The scores obtained by the model player were higher compared to human participants. However, the Q-learning model could learn the fast version of the jumper game. The stay action by the model was not able to follow the trend of human data. The MSDs obtained for all rounds, slow rounds and fast rounds, were high as shown in Table IV.

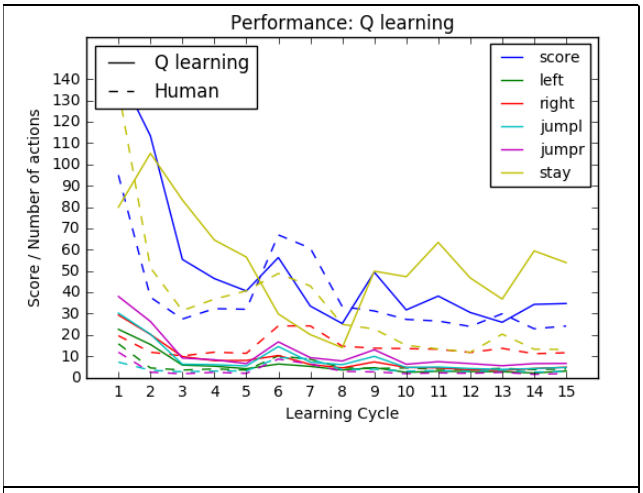

Fig. 5. Shows the performance of Q Learning in the Jumper Game, the x-axis denotes the cycle number. First five cycles are for the slow game, last ten cycles are fast. Y-axis denotes the number of actions taken to reach from initial to final position or score.

TABLE IV. Mean Squared Deviation, Q Learning

Method	Value
MSD_{All}	1620.13
MSD_{Slow}	2439.97
MSD_{Fast}	1210.21

C. *DeepQ Network*

Unlike other models, DQN model required more number of rounds of training to perform like human participants. DQN took 45 rounds to learn a slow version of the jumper game and 55 rounds for a faster version of the jumper game. We ran DQN model for 45 slow episodes followed by 55 fast episodes, 10 times. The average plot thus obtained is shown in figure 6.

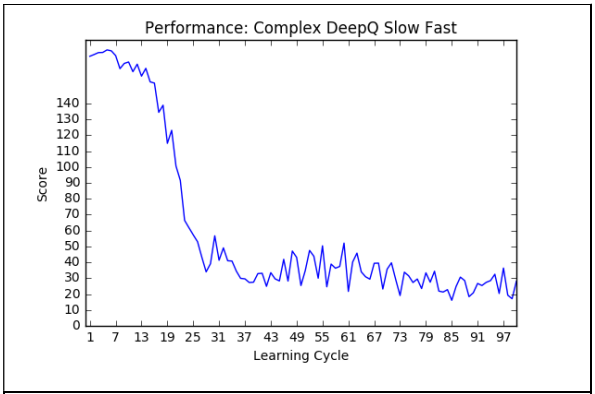

Fig. 6. A plot of score obtained by complex DeepQ network. The x-axis shows the episode number; the y-axis shows the score. First 45 cycles are of the slow platform, last 55 cycles are for the fast platform.

To map these 100 rounds to 15 rounds, we blocked few rounds. First 45 rounds were blocked for every 9 rounds to convert them in 5 rounds for a slow version of the jumper game. Last 55 rounds were blocked for every 6 rounds to convert them into 10 rounds. As shown in figure 7, the DQN model can capture the human-like trends. However, scores initially obtained by DQN model are much higher compared to the human sores. During the slow rounds of the game, the model performed poorly, and the MSD_{Slow} is also very high. The DQN model could perform well for a faster version of the jumper game. The reason behind such performance is learning, which is more during the later rounds of the game compared to earlier rounds. The MSDs obtained during the slow and fast rounds are shown in Table V.

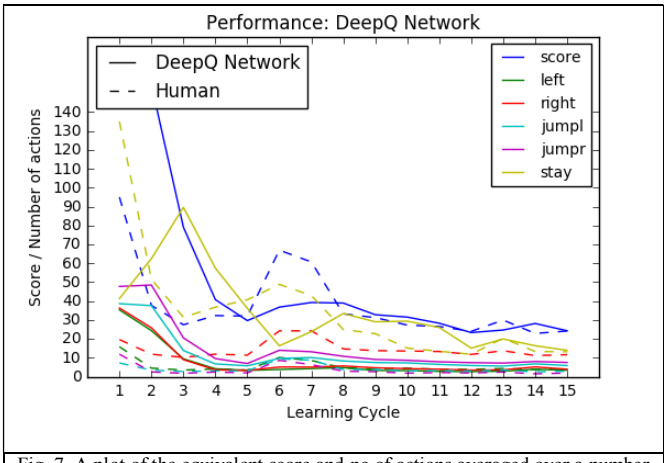

Fig. 7. A plot of the equivalent score and no of actions averaged over a number of players for each cycle. The x-axis shows the episode number; the y-axis shows the equivalent number. First 5 cycles are the slow platform, last 10 cycles are for the fast platform.

TABLE V. MEAN SQUARED DEVIATION, DEEPQ

Method	Value
MSD_{All}	1605.898
MSD_{Slow}	4055.22
MSD_{Fast}	381.08

VII. MODELS COMPARISON

In this section, we compare all the models to identify the most human-like model. Figure 8 summaries all the models and compares the scores obtained from different models and human participants over the 15 rounds.

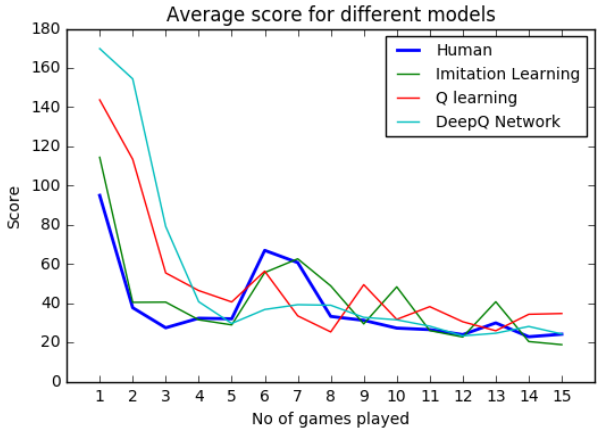

Fig. 8. Score (y-axis) scored by humans and different models in different learning cycles(x-axis) for Jumper Game. The score at a particular learning cycle is averaged over the number of times the algorithm played the game. 1-5 cycles were run with the slower platform and 6—15 with the faster platform. The blue curve is the human score. This graph can be used to compare the performance of different algorithms with respect to humans. For DeepQ Network, the equivalent score is plotted.

As we can see in figure 8, the imitation learning model behaves like human participants. Q learning and DQN model obtained higher scores compared to humans and imitation learning model. The high score values were for slower rounds of the game. The MSD over all the rounds for different models is shown in Table VI. It is evident from Table VI, the MSD value for imitation learning is lowest. MSD for slower rounds of Q learning and DQN models is very high compared to faster rounds.

TABLE VI. MEAN SQUARED DEVIATION

Method	MSD_{All}	MSD_{Slow}	MSD_{Fast}
Imitation Learning	190.88	262.31	155.17
Q Learning	1620.13	2439.97	1210.21
DeepQ	1605.898	4055.22	381.08

VIII. CONCLUSION

In this paper, we used a jumper game to compare traditional and deep reinforcement learning algorithms. Fifteen human players were made to play the game, and their data was recorded for comparison for both the slow and fast versions of the game. It is evident from our results that there was a decrement in performance when the game speed changed from the 6th round. This decrement was best captured by imitation learning as its actions were derived from human data. Thus, it is a very good technique if we have the actual data of the gameplay of expert human players. It helps to create a bot which is very human-like.

Beyond imitation learning, we tried the traditional Q-learning and a DeepQ architecture. If we consider only slow cycles, then the DeepQ architecture gives the highest performance. The reason for this result is that it has to understand both the gameplay images and the correct actions. Humans have an advantage for identifying images, and they were given information about which is the player and which is the platform. Also, humans have previous experiences which help them recognize such things faster; whereas, the DeepQ architecture was pretty much like a newborn baby who had to learn everything on its own from scratch. When we considered the fast cycles, the DeepQ also performs exceptionally well. One likely reason could be that it had now learned the image and was able to extract better states than the ones provided.

If we see the overall picture, traditional algorithms tend to perform good and learn at a pace like humans; however, they need the states to be pre-initialized by a human user. Thus, we are giving some sort of input based on our experience of the game. When we consider the DeepQ architectures, we find that they are really good once the network has been trained and they have the least score. They learn directly from images of the gameplay, and the states need not be pre-initialized. The features are extracted automatically from the image that is provided. Thus, we can say that they are more intelligent. That intelligence comes at a cost as they need a high number of episodes to train and cannot match the pace at which humans learn. Also, they need very high computational power for training. Overall, there exists a trade-off between how intelligent the learning model is versus how much time and computational power you can give. If time, learning pace, and computational power is not a constraint, then DeepQ architectures are better than the traditional approaches. If that is not the case, conventional algorithms would be the one we would recommend.

ACKNOWLEDGMENT

We would like to thank our guide, Varun Dutt for introducing us to the field of Cognitive Modeling, Artificial Intelligence, and Deep Learning and motivating me to further work in this area. He is a much-disciplined person and a very supportive mentor. We are very grateful to him for his time and attention.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” Nature, 2015, vol. 521, pp. 436–444.

[2] K. Arulkumaran, M.P. Deisenroth, M. Brundage and A.A. Bharath, “A Brief Survey of Deep Reinforcement Learning," arXiv: 1708.05866, 2017.

[3] Richard Sutton and Andrew Barto. Reinforcement Learning: An Introduction. MIT Press, 1998. ISBN 0-262-19398-1.

[4] Mnih V. et al., Playing Atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013).

[5] Schaal S, “Is imitation learning the route to humanoid robots?,” Trends in Cognitive Sciences, 1999, pp. 233–242.

[6] C. J. C. H. Watkins and P. Dayan, “Q-learning,” Machine Learning, vol. 8, no. 3-4, pp. 279–292, May 1992.

[7] Z. Wang et al., "Dueling Network Architectures for Deep Reinforcement Learning," arXiv: 1511.06581 [cs.LG], 2015.

[8] N. Stockton, "This New Atari-Playing AI Wants To Dethrone DeepMind," Wired, 2017 [online]. Available: https://www.wired.com/story/vicarious-schema-networks-artificial-intelligence-atari-demo. [Accessed: 1-Sep-2017]

[9] S. M. Lucas, "Computational Intelligence and AI in Games: A New IEEE Transactions," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 1, pp. 1-3, March 2009.

[10] F. Hsu, *Behind Deep Blue*. Princeton, NJ: Princeton Univ. Press, 2002.

[11] J. Schaeffer et al., “Checkers is solved,” *Science*, 2007, vol. 317, pp. 1518–1522.

[12] J. Schaeffer, “One Jump Ahead: Computer Perfection at Checkers,” Springer-Verlag, New York, 2009.

[13] Thurau, Christian, Christian Bauckhage, and Gerhard Sagerer, "Imitation learning at all levels of game-AI, " *Proceedings of the international conference on computer games, artificial intelligence, design and education*, 2004, Vol. 5.

[14] Claude Sammut et al., “Learning to fly,” Proceedings of the ninth international workshop on Machine learning, 2014, pp. 385–393.

[15] Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y Ng, “An application of reinforcement learning to aerobatic helicopter flight” Advances in neural information processing systems, 2006.

[16] Jiakai Zhang, Kyunghyun Cho, “Query-Efficient Imitation Learning for End-to-End Autonomous Driving, ” arXiv: 605.06450 [cs.LG], May 2016.

[17] V. Dutt and C. Gonzalez, “Accounting for outcome and process measures in dynamic decision-making tasks through model calibration,” Journal of Dynamic Decision Making, 2015. doi: http://dx.doi.org/10.11588/jddm.2015.1.17663

[18] V. Dutt, "Explaining human behavior in dynamic tasks through reinforcement learning," Journal of Advances in Information Technology, 2011. DOI:10.4304/jait.2.3.177-188

[19] Muelling K, Kober J, Kroemer O and Peters J, “Learning to select and generalize striking movements in robot table tennis,” *The International Journal of Robotics Research, 2013.*

[20] Tokic M, Ertel W and Fessler J, “The crawler, a class room demonstrator for reinforcement learning” *International Florida Artificial Intelligence Research Society Conference, 2009*.

[21] R. Yamashina, M. Kuroda, and T. Yabuta, "Caterpillar robot locomotion based on Q-Learning using objective/subjective reward," *IEEE/SICE International Symposium on System Integration (SII)*, Kyoto, 2011, pp. 1311-1316.

[22] H. Çetin and A. Durdu, "Path planning of mobile robots with Q-learning," *22nd Signal Processing and Communications Applications Conference (SIU)*, Trabzon, 2014, pp. 2162-2165.

[23] P. G. Patel, N. Carver and S. Rahimi, "Tuning computer gaming agents using Q-learning," *Federated Conference on Computer Science and Information Systems (FedCSIS)*, Szczecin, 2011, pp. 581-588.

[24] H. Wang, Y. Gao, and X. Chen. RL-DOT, “A Reinforcement Learning NPC Team for Playing Domination Games” IEEE Transactions on Computational Intelligence and AI in Games, 2010, Vol. 2.1, pp. 17-26.

[25] D. Loiacono, A. Prete, P. Lanzi, and L. Cardamone, “Learning to overtake in TORCS using simple reinforcement learning,” IEEE Congress on Evolutionary Computation (CEC), 2010, pp. 1-8

[26] A. Kumar, J. Prakash, V. Dutt, "Understanding Human Driving Behavior through Computational Cognitive Modeling," In: Hsu R.CH., Wang S. (eds) Internet of Vehicles – Technologies and Services, 2014. Lecture Notes in Computer Science, vol 8662. Springer, Cham

[27] Bing-Qiang Huang, Guang-Yi Cao and Min Guo, "Reinforcement Learning Neural Network to the Problem of Autonomous Mobile Robot Obstacle Avoidance," *International Conference on Machine Learning and Cybernetics*, Guangzhou, China, 2005, pp. 85-89.

[28] Simpson College Computer Science, Platform Jumper [online] Available at : http://programarcadegames.com/python_examples/show_file.php?file=platform_jumper.py

[29] *Tensor flow*, [online] Available: https://www.tensorflow.org/