Reinforcement Learning and Adversarial Attacks on Player Model with Doodle Jump

Amulya Raveendra Katti*
M.S. in Computer Science
University of Southern California
Los Angeles, USA
akatti@usc.edu

Anamay Sarkar*
M.S. in Computer Science
University of Southern California
Los Angeles, USA
anamaysa@usc.edu

Pranav Mallikarjuna Swamy*
M.S. in Computer Science
University of Southern California
Los Angeles, USA
mallikar@usc.edu

Riya Kothari*
M.S. in Computer Science
University of Southern California
Los Angeles, USA
rskothar@usc.edu

Shenoy Pratik Gurudatt*
M.S. in Computer Science
University of Southern California
Los Angeles, USA
gurudatt@usc.edu

Abstract—**In this paper, we introduce an AI player model for the Doodle Jump game and then implement adversarial attacks to thereafter sabotage the modelling. The AI player model which we call our game agent, interacts with the game and allows the 'Doodler' to climb up the platforms without any human intervention and also attempts to beat human high scores. The agent is trained using different reinforcement learning algorithms. We have also outlined details on past work in this field, the different reinforcement learning algorithms used and details on the game environment that we have used in order to train and build our AI agent. Furthermore, we plan to use the trained game agent to perform various types of evasion based adversarial attacks that would deceive the agent and confuse it to give a false output.**

Index Terms—**Reinforcement Learning, Adversarial Attacks, Deep Learning, Pygame**

I. Introduction

A. Reinforcement Learning

Deep Reinforcement Learning is an area of Machine Learning(ML) that has been widely used to build game agents and bots to challenge and beat human players. It has been widely used in the recent years ever since it's introduction in 2013 [1] which showed how reinforcement learning (Q-learning) can be combined with a Convolution Neural Network(CNN) to learn to play Atari games from game image inputs. Most popular and common ML algorithms are trained with a set of inputs called features/attributes and a target variable. The system then tries to learn from this information and attempts to make predictions of the target based on new inputs. In our case we do not know what the best outcome/action is at every step of the game and hence this approach, which is also called supervised learning will not be effective. On the other hand, reinforcement learning is an approach of training machine learning models to make a sequence of decisions. Reinforcement learning starts by building an agent that learns from an environment by interacting with the environment through trail and error. The system also learns by receiving rewards which serves as feedback for performing an action.

Our goal in this project is to build an agent that learns to play the Doodle Jump game using Deep Reinforcement learning algorithms. We have explored the different Deep Reinforcement learning algorithms like the Deep Q-learning(DQN) and Deep Recurrent Q-learning(DRQN) algorithms and trained our agent on both these models. We have also experimented with different reward functions and hyperparameters to choose a model that gives us a good performance.

B. Adversarial Attacks

Once we have a good agent that is able to play the Doodle Jump game satisfactorily, we further wish to explore the topic of Adversarial Attacks. Adversarial attacking is a technique in machine learning that attempts to fool models by supplying deceptive/misleading input that looks similar to the human eye. This part of our work will mainly look into possibilities of eliminating players using game bots to deceive Anti-cheat engines. We will be researching on using various evasion based attacking models that would manipulate the input image, sent to reinforcement learning system to confuse the game bot.

C. Doodle Jump

Doodle Jump is a very popular video game created by Igor and Marko Pušenjak and published by American studio Lima Sky. The game is available on all major platforms. When it was released, Doodle Jump skyrocketed to fame, accounting for over 25,000 copies of the game being sold every day for 4 months. The game is so popular that it has been developed into a video redemption game at video arcades.

The main aim of the game is to propel 'The Doodler', the main character of the game which is a four-legged creature, up a never-ending series of platforms, without falling from

*All authors have equal contributions

them. The left end of the playing field connects to the right end to help the doodler stay within the bounds of the screen. The doodler can get a boost in score and height from springs attached to some platforms. There are monsters on some platforms that the doodler must avoid otherwise it will get killed on contact with the monster. The game ends when the doodler falls from a platform or when it hits a monster.

II. Related Works

The relation between machine learning and playing games goes back to very early days of Artificial Intelligence [2] [3] where several machine learning techniques and game playing techniques were described over a game of checkers. These fields have grown considerably and research combining the two fields has been ongoing, with each research yielding some groundbreaking results. Building AI bots/agents that are used to learn several aspects of the game including playing the game has gained a lot of popularity. Applying machine learning to game applications include player modeling, learning about the game, understanding players and their behaviours, etc.

The concept of player modeling became popular with the chess system by Deep Blue, that was developed at IBM Research during the mid-1990s. [4]. Deep Blue gained popularity in 1997 after defeating the then World Chess Champion Garry Kasparov in a six-game match. Ever since the success of Deep Blue, there have been multiple attempts to develop player models for a variety of games. With the development of newer games and as the complexity of games increased, using Reinforcement learning to learn about the game environments to build player models gave a new avenue for research.

The most successful game agent to use reinforcement learning is TD-gammon, which is an agent that plays backgammon. This agent learnt entirely using reinforcement learning and achieved very high levels of play. [5]. As game complexities further increased, the growth of Deep Reinforcement learning became widespread with the use of Q-learning with Neural Networks. The first experiments on Deep Reinforcement learning was performed by Google Deepmind on a set of seven Atari 2600 games from the Arcade Learning Environment. [1]. The model called the Deep Q-learning(DQN) model was a convolution neural network, that was trained with a variant of Q-learning. This model took screenshots/raw pixels of games as input and returned a value function that estimated future rewards. The model outperformed all previous approaches on six of the Atari games and was also able to beat human experts for three of the games.

[6] by Deepmind also describes a new approach to the Go game by using deep neural networks that are trained by supervised learning from human games with tree search, and by reinforcement learning, from self-play games. This agent played against a professional human player in the game of Go and defeated the human player. The game of Go was one of the most challenging games then, due to it's large search space and complexities involving board positions and actions. [7] also talks about how riddles can be solved with the help of deep distributed recurrent q-networks.

[8] talks about extending the capabilities of DQNs to improve performances in complex games. These Recurrent Deep Q-Networks use a Long Short Term Memory (LSTM) and deep Q-network thereby enabling the possibility of learning from observations that might have occurred much earlier in the learning phase.

[9] explores DRQN on a set of games like Q*bert, Seaquest. This paper also examines attention with DRQN and also evaluate its usefulness.

The Atari games were implemented on the Arcade Learning Environment(ALE) [10]. ALE is an object-oriented framework that aids development of AI agents for Atari 2600 games. This framework was made on top of the Atari 2600 emulator, Stella. The details of emulation and agent design are kept separate.

With reinforcement learning being on top for developing game agents, another toolkit called the Open AI gym gained a lot of importance. OpenAI Gym is a toolkit that aids developing and evaluating reinforcement learning algorithms. It provides an open-source library with gives access to a standard set of environments. OpenAI Gym provides 59 Atari 2600 games as environments. The gym also comes prepackaged with many environments like super mario bros, etc. There are also additional third party environments that do not come prepacked with the gym that can be utilized to implement Reinforcement learning algorithms.

Gym Retro is another platform for reinforcement learning research on games.

Game development and simulation has never been easier than it is currently. There are many libraries that help develop games. Pygame is one such library based on Python that we have used in our game development for Doodle Jump. We have outlined more details on the game and the simulation environment in the next section of this paper.

We have used the DQN and DRQN models in our game. We have also explored another variant of the DQN model called the Action Specific Deep Recurrent Q-learning(ADRQN). Most of the research on Deep Q-learning focuses on fully observable environments. The DRQN aims to provide a workarounds to overcome partial observability. However, we need a mechanism for keeping track of the history of observations while estimating the Q function with a neural network. This can be achieved by introducing recurrence in the Q-network.

Q-learning and Deep Q-learning are value-based reinforcement learning algorithms. We estimate the Q values and the action corresponding to the highest Q value(maximum expected future reward) at each state is performed. There are other algorithms called the policy-based reinforcement learning algorithms. In policy-based methods, we learn a policy function which is a mapping of the state to action instead of a value function that gives the expected sum of rewards for each action at a particular state. We have explored a hybrid method that combines value-based and policy-based algorithms called the Actor-Critic. The 'Critic' provides an estimation of the value function which could be the action-value/Q value or state-value/V value. The 'Actor' updates the

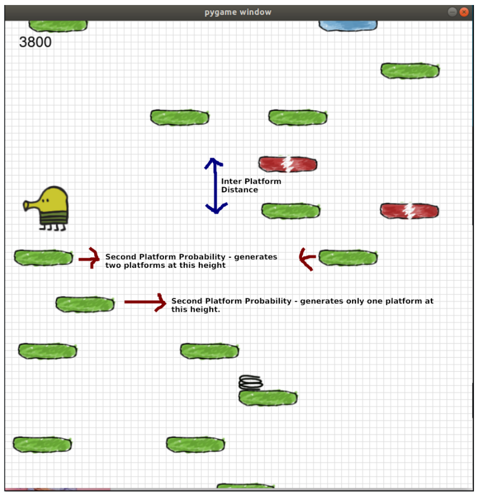
pygame window
3800
Inter Platform Distance
Second Platform Probability - generates two platforms at this height
Second Platform Probability - generates only one platform at this height.

Fig. 1. Explaining inter-platform distance and second platform probability.

policy distribution as directed by the Critic. Both the Actor and the Critic use Neural Networks to learn. This method is extremely useful in environments with a large action space.

We will now take a look at the details of our game implementation and discuss the approaches we have used.

III. Data and Environments

A. Simulation Environment

Pygame is a module developed for creating video games natively in Python. With extra functionality added over the SDL library, pygame is one of the most popular game development environments for Python, allowing us to build feature-rich games. Some of the advantages of pygame is that it is free, simple to use, highly portable, modular and easy to maintain.

For our project, using pygame was very beneficial since it is very light on the CPU, hence enabling quicker training and cutting down development costs. The game we have used is a modification of the Doodle Jump game developed by Frankie [1]. The game runs on a 800x800 window, where there are multiple platforms upon which the doodler jumps to move up the game. In our modification to the game, we have increased the complexity of gameplay by adding regular platforms, moving platforms and broken platforms. We have also introduced three levels to the game, "EASY", "MEDIUM", and "HARD", which can be initiated through constructor injection. The difficulty of the game can be manipulated through two parameters: the inter-platform distance - distance between two consecutive platforms that are at a different height from each other, and the second platform probability - the probability with which a second platform is generated along with a single platform at a particular height (Fig. 1). Through manual experimentation, the threshold values for both of these parameters for all the three levels of the game were set.

[1]https://github.com/f-prime/DoodleJump

B. Action Space

The doodler is treated as a single point during the simulation and it has a discrete action behaviour. At every point in the game, the doodler has three action choices - it can either do nothing, or go left, or go right.

TABLE I
Action Spaces

No.	Action	Description
1	No Action	Doodler does nothing, jumps in place.
2	Left	Doodler turns left and goes left.
3	Right	Doodler turns right and goes right.

In the learning process, the vector representing the action choice of the doodler has 3 dimensions. For prediction, the vector element of the selected action is 1, and all others are 0, i.e., one-hot encoding is performed for the predicted vector.

C. Reinforcement Learning System

In our project, we have used the DQN algorithm instead of a supervised ML approach.

Our reinforcement learning system consists majorly of the environment and agent (Fig. 2). The agent is the Q-learning agent that fetches the rewards and states from the environment and passes it to the model. The model is a Deep Q-learning CNN model that fetches the current state from the agent and generates action that the agent has to perform. The environment which is the Doodle Jump game powered by the pygame engine interacts with the agent and rewards the agent either positively or negatively based on an action that the agent performs. The nature of the reward also depends on the quality of the action, i.e., how good or bad the action is for a particular state. The action is a boolean array comprising of 3 values Left Jump, No Action, Right Jump for the doodler.

The reward is a combination of long term (expected reward) & short term reward (current game score).

The goal of the agent is to maximize the reward for each state of the game by learning what action to perform. In our scenario, the state is a screenshot of the current game that is fetched from pygame engine's display component. The agent receives the states/observations at each iteration from the environment.

In Reinforcement learning terms, the decision-making process used by the agent can be termed as a policy. The policy is a mapping from the state space into the action space (the

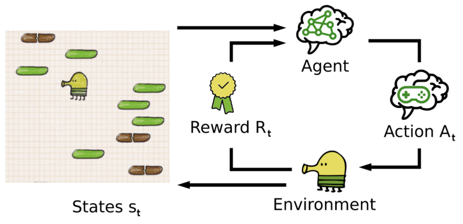

Fig. 2. Doodle Jump RL System

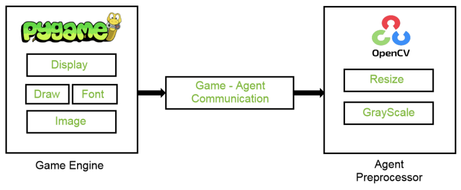

Fig. 3. Pre-processing pipeline

set of actions an agent can take, in our case NO ACTION, LEFT or RIGHT as mentioned previously).

D. Pre-Processing Pipeline

For our Doodle Jump AI agent, we have a pre-processing pipeline set in place before we train the agent on our models (Fig. 3). We first have the game engine which is powered by pygame. The display renderer from pygame is used to fetch a screenshot of the game which gives us the current state of the game as an image. This is an image of size 800x800.

The next component is the Game-Agent communication wherein the Game provides the state to the agent. The agent learns which action to perform based on the state and communicates the action back to the game. We have also used OpenCV in order to preprocess the current state image/input image prior to sending it to our model for learning. The 800x800 image is resized to an 80x80 image. This is further converted to a single channel image in grayscale and given as input to the model. This pre-processing is done in order to reduce the network parameters of our model and also reduce the training time per iteration.

E. Reward Functions

Reward Functions are important to any Reinforcement Learning system because they determine how quickly agent can learn the objective. They also make sure the agent learns the given task at hand without circumventing the given constraints in game engine/environment. For the purpose of training our game agent we use 4 different types of reward functions. Each of these are explained in detail in the experiments section.

F. Agent Modelling

We have the following (Fig. 4) schema in place for training our agent. Prior to passing the parameters to train our model, another important consideration in any Reinforcement learning task is the Exploration/Exploitation trade-off. Exploration is the task of exploring the environment by allowing the agent to try out random actions from the action space to understand the environment thoroughly. On the other hand, exploitation refers to the task of using/exploiting the known information to maximize the reward. In order to be able to achieve a good cumulative reward, we need to balance how much we explore the environment and how much we exploit. For this purpose, we have an arbitrary variable epsilon set to a certain value. The value of epsilon helps us determine whether the agent should explore the environment or exploit known information.

The agent fetches the current state image from the environment. Once the agent chooses to explore a random move or exploit using the known model output, the action is performed. Based on the action, the next state of the game is fetched from the environment. We then perform a short training on the model and store the rewards and the image frames for a particular action. Once these set of operations are complete, we check if the Doodler is still alive, i.e., whether the game is active or has ended. If the game has been terminated after an action, we reset the game parameters and we train the long memory based on the sequence of states that have been stored for every action until the game was terminated. We also save the model that was trained until this point. If the game was not terminated, we loop back to fetching the current state of the game and perform the above operations in sequence.

If we give sequential inputs of states to the model/network, there might be cases wherein the model forgets previous interactions and experiences as new ones are learned. In order to overcome this issue, we make use of a replay buffer. This buffer stores details of the experiences for every interaction or experience that the model learns in the form of tuples. At every step in the long training, we take a random subset of this buffer and train the model. This ensures that the model not only learns from current experiences but also from the past experiences.

IV. Methods

We have used two very popular and widely used Reinforcement learning models in order to train the agent. The two models that we have used till date are DQN and DRQN. We have performed experiments on both these models for different reward functions and different parameters. We will elaborate more on our experiments in the next section of the paper.

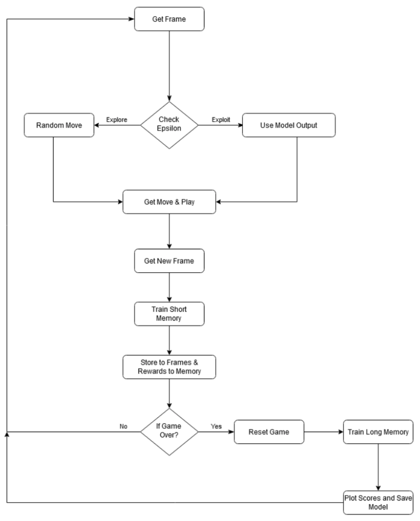

Fig. 4. Agent Modelling Schema

Q-learning is one the first RL algorithms that has been widely used. It is an off-policy value-based method that uses a temporal difference approach to train an action-value function. This action value function, also called the Q function is used to identify the value of being at a state and performing a specific action. The Q represents the Quality of an action (how good or bad an action is) at a particular state. Q learning uses a Q table to update the Q values at each iteration. Using a Q table to update Q values becomes very complex when the state space of the environment becomes very large. In order to overcome these complexities, we use DQN, which uses a Neural Network to approximate Q values for every action based on the state.

The Q function can be approximated using the Bellman equation as a linear combination of the rewards (Fig. 5).

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a)$$

Fig. 5. Q-function formula

A. *Deep Q-learning*

Our DQN architecture (Fig. 6) consists of a series of steps. We have 3 convolution layers, 3 max-pool layers, and finally a fully-connected network.

For every time step, the model receives a set of values (state, action, reward, new_state). The state to the network is an image of size 80x80 as input. This is the image that has been preprocessed, i.e., resized and converted to gray scale. This passes through the layers of the network, and outputs a one dimensional array of Q-values for each action for a particular state. We then take the largest Q-value of this array to find the suitable action.

In our case, the frames are processed by three convolution layers so that we are able to exploit spatial relationships in image. Each convolution layer is using RELU as an activation function

We first pass our image through a convolution layer with filter size of 8x8x32, stride 4 and padding 2. This is then downsampled using maxpooling and passed through another convolution layer and the same process is continued for the next 2 convolution and max pool layers. Finally, we are using a Fully Connected Layer(FCL) with RELU activation function and an output layer which is a FCL with linear activation to give the Q-value estimate for each action. In our case it produces as 1x3 output with estimates for 3 actions - moving left, do nothing or move right.

B. *Deep Recurrent Q-learning*

The DRQN architecture (Fig. 7) is very similar to the DQN model, but with an additional Recurrent Neural Network(RNN) being using in the model between the convolution and fully connected layers. DRQN is a combination of an RNN and DQN. We have used the Gated Recurrent Unit(GRU) in our network. The DRQN retains useful information for longer due to the presence of RNNs [8]. This is expected to help the agent perform actions that require it to remember states that might have occurred much earlier during training.

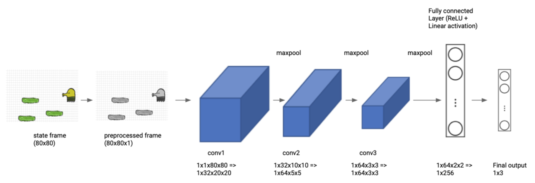

Fig. 6. Deep Q-learning architecture

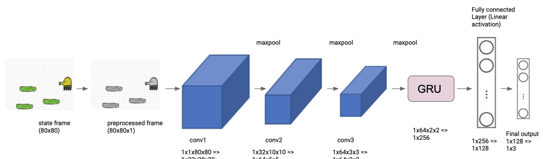

Fig. 7. Deep Recurrent Q-learning architecture

V. Experiments, Results and Analysis

RL based game agents use a lot of hyper-parameters that are needed to be fine-tuned, to make sure the agent learns well. These comprise of namely batch size, learning rate, number of game episodes, discount factor for the Q function i.e. gamma, memory queue for storing previous games and reward formulations. We optimized each of them with different ablation studies based on short number of game episodes.

A. *Reward Formulations*

We used 4 different type of reward functions to see how agent reacts to different factors of the game.

1) Type 1: The game agent is given a reward of +3 if the score is increased. If the agent dies or gets stuck it is penalized with a -2 reward. Whenever agent doesn't increase the score i.e. if doodle doesn't jump up on a new upper platform the agent is rewarded with 0.

2) Type 2: The game agent is penalized for staying on the same platform with reward of -1. The rest of the reward remain same as Type 1.

3) Type 3: The game agent is rewarded +3 if it jumps on spring and is penalized -4 if it touches monster. The rest of the rewards remain same as Type 2.

4) Type 4: We add a living reward based on the score earned by the agent in the game. Adding the game score to the reward function makes sure game agent tries to live longer. The rest of the rewards remain same as Type 3.

We ran our initial reward based set of experiments for 300 game episodes, gamma of 0.9, learning rate of 1e-3, batch size of 1k and max memory of 10k. The exploration was random till first 40 games there on continuously training the Deep Q-Network model.

An interesting observation in Type 1 reward experiment was agent getting stuck in the game and not jumping on

TABLE II
REWARD EXPERIMENT RESULTS

Reward Type	High Score	Mean Score
Type 1	8700	1181.3
Type 2	**8800**	1491.0
Type 3	**8800**	1340.0
Type 4	7400	**1543.0**

neighboring platforms after some iterations. The main point that giving the agent with a non negative reward by default encouraged it to stay there in the same position. From Table II on the basis of mean score, we infer that Type 4 reward works the best for our game agent. Hereafter, we restrict all our experiments to the type 4 reward.

B. Hyper-Parameter Tuning

Tuning hyper-parameters of a model in RL system is similar to that of any other Deep Learning system. The only difference is the metric used to judge the best performing parameter. Similar to the reward experiments, we use mean score and mean reward to compare different experimental settings.

1) Learning Rate: We use a wide range of learning rates to run our ablations, we start with 1e-2 and reduce it all the way to 1e-4. The rest of the parameters are the same as that of reward experiments. As seen in Table III, we note that learning rate of 1e-3 gives the best mean reward and mean score.

TABLE III
REWARD EXPERIMENT RESULTS

Learning Rate	Mean Reward	High Score	Mean Score
1e-2	-0.98	6800	1016.7
1e-3	**-0.94**	7400	**1543.0**
1e-4	-0.97	**9400**	1183.7

2) Batch Size & Game Memory: Batch size for both short and long training runs is an important factor. It helps the gradient to be calculated through the a collection of game frames in memory. We use a couple a batch size and game memory parameters starting from batch size of 1k and game memory of 10k as baseline, followed by batch size of 5k & 10k and game memory of 25k & 50k. Table IV shows the results of these experiments.

TABLE IV
BATCH SIZE & GAME MEMORY EXPERIMENT RESULTS

Batch Size, Memory	Mean Reward	High Score	Mean Score
1k, 10k	-0.94	7400	**1543.0**
5k, 25k	**0.01**	**9700**	1326.0
10k, 50k	-0.94	8900	1522.3

3) Weighted Expected Reward: Discount factor (gamma) in Deep Q-Networks enables the game agent to give weightage to the expected rewards. The more discount factor, the more is weight given to the expected rewards. We run our experiments with 3 different values of gamma starting with 0.8 then increasing it to 0.9 and finally having a value of 0.99. The results of experiments with different gamma values are shown in Table V

TABLE V
GAMMA EXPERIMENT RESULTS

Gamma	Mean Reward	High Score	Mean Score
0.8	-0.97	**10600**	1246.3
0.9	-0.94	7400	1543.0
0.99	**-0.92**	8500	**1670.7**

C. Training and Analysing DQN & DRQN Models

Based on the results of the hyper-parameter tuning results, we select two sets of parameters for our long term training. The first one being (Agent 1), learning rate of 1e-4, gamma 0.8, game memory of 10k, batch size of 1k. The second set (Agent 2) including learning rate of 1e-3, gamma 0.99, game memory of 25k, batch size of 5k. We run these two experiments for 2000 game episodes to make sure the model is able to achieve good scores consistently for a long duration of time. Both the agents are for DQN and DRQN architectures each.

TABLE VI
DQN AGENT RESULTS

Agent #	Mean Reward	High Score	Mean Score
Agent 1	**-0.95**	**11000**	**1505.2**
Agent 2	-0.98	9000	1151.8

Above are the results of training DQN model in TableVI. We see that Agent 1 of the DQN model achieves the a good high score of 11000. Also, the mean score of Agent 1 is consistent in the range of 1500. This means the model does well invariably over a long period of time. Whereas the Agent 2 is behind Agent 1 in all numbers especially mean score. On the other hand we don't see good mean reward in both the agents.

TABLE VII
DRQN AGENT RESULTS

Agent #	Mean Reward	High Score	Mean Score
Agent 1	-0.95	10000	**1457.9**
Agent 2	**-0.94**	**11300**	1426.7

The Results below in TableVII show how DRQN models performed. Agent 1 performs well in comparison to both the DQN agents. It reaches the high score of 10000 beating one of the DRQN agents and has a comparable mean score. The Agent 2 of DRQN model achieves the highest high score aamong all game agents played yet. It also has one of the best mean rewards among the 4 agents. The recurrent modelling of the DRQN architecture is one of the reasons why it does better than its DQN counterpart.

VI. Limitations, Conclusion and Future Work

Even though the models achieve consistent scores over a long duration of time, they lack continuous growth in terms of reward earned. The mean reward stays pretty low throughout the experiments. Moreover, Doodle Jump game being non-episodic and having a lot of randomness built-in, makes learning difficult for AI agent.

For conclusion, we note that a continuous reward that increases in a logarithmic way is much more beneficial for RL based systems. The recurrent part enables the model to learn new exploits at the same time retaining memory from previous iterations.

In future, we would like to explore different reinforcement learning architectures like Proximal Policy Optimizations (PPO) and Actor-Critic models (A3C/A2C). In parallel, we would like to explore the vulnerability in RL systems using adversarial attacks: Fast Gradient Sign Method (FGSM) [11] and Projected Gradient Descent (PGD) [12]. Inspired from [13] these attacks would be integrated to the game agent code to sabotage the agent with the manipulated image.

Acknowledgment

We would like to thank "Prof. Micheal Zyda" for his guidance and for the amazing curriculum in the course CSCI 527 at the University of Southern California. We would also like to thank the teaching staff of the course, in helping us through the different phases of the research project.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[2] Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. IBM Journal of research and development, 3(3), 210-229.

[3] Samuel, A. L. (1967). Some studies in machine learning using the game of checkers. II—Recent progress. IBM Journal of research and development, 11(6), 601-617.

[4] Campbell, M., Hoane Jr, A. J., & Hsu, F. H. (2002). Deep blue. Artificial intelligence, 134(1-2), 57-83.

[5] Tesauro, G. (1995). Temporal difference learning and TD-Gammon. Communications of the ACM, 38(3), 58-68.

[6] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. nature, 529(7587), 484-489.

[7] Foerster, J. N., Assael, Y. M., de Freitas, N., & Whiteson, S. (2016). Learning to communicate to solve riddles with deep distributed recurrent q-networks. arXiv preprint arXiv:1602.02672.

[8] Hausknecht, M., & Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. arXiv preprint arXiv:1507.06527.

[9] Chen, C., Ying, V., & Laird, D. (2016). Deep q-learning with recurrent neural networks. Stanford Cs229 Course Report, 4, 3.

[10] Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research, 47, 253-279.

[11] Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572.

[12] Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2018, February). Towards Deep Learning Models Resistant to Adversarial Attacks. In International Conference on Learning Representations.

[13] Gleave, A., Dennis, M., Kant, N., Wild, C., Levine, S., & Russsell, S. (2020). Adversarial Policies: Attacking Deep Reinforcement Learning. In Proc. ICLR-20.

Appendix

Below we have some plots from our best DQN agent. The plots show how gradients, weights, loss and high scores of the model evolved over time. We use tensorboard for plotting these graphs and to monitor different experiment runs.

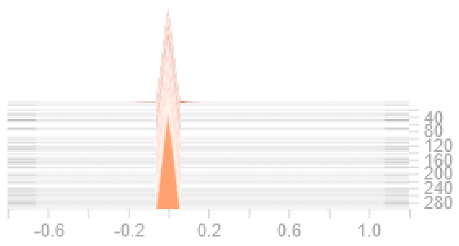

Fig. 8. Gradients plot for DQN model

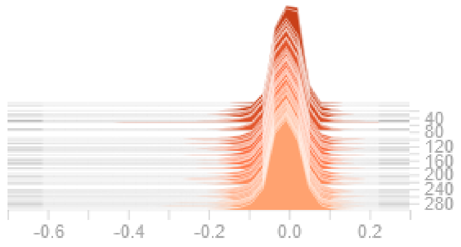

Fig. 9. Weights plot for DQN model

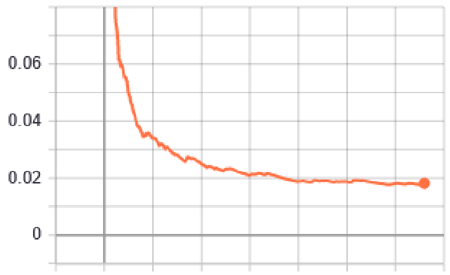

Fig. 10. Loss plot for DQN model

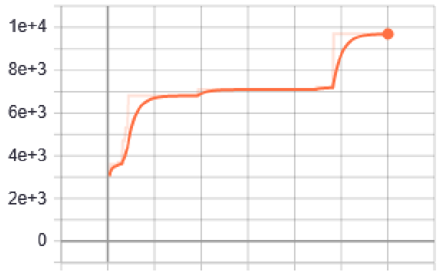

Fig. 11. High score plot for DQN model