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. Our project also focused on an analysis on the exploration and exploitation trade-offs for these methods and we expermiented with 3 exploration functions. We also have performed an analysis on different reward functions and described the results for each of them. We also worked on some game modifications to help the models learn faster. Another part of our project looks into policy gradient methods of Reinforcement learning like the Advantage Actor Critic(A2C) and Proximal Policy Optimization(PPO) algorithms.

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

*All authors have equal contributions

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 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. Applying machine learning to game applications include player modeling, learning about the game, understanding players and their behaviours, etc.

Player modeling became popular with the chess system by Deep Blue, that was developed at IBM Research during the mid-1990s. [4]. 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 [5]. 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. 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. [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.

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.

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

Game development and simulation has never been easier than it is currently. Pygame is a 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. 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 and this is achieved by introducing recurrence in the Q-network.

Deep Q-learning is a value-based RL algorithm. We estimate the Q values and the action corresponding to the highest Q value(maximum expected future reward) at each state is performed. 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. Both value-based and policy-based methods have their own drawbacks. 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 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. The Actor Critic methods have proven to perform well and yield state of the art results. [19] talks about Asynchronous Methods of Reinforcement learning using actor-critic model. Advantage Actor Critic and Proximal Policy Optimization are two of the algorithms that show promising results. [20] speaks about the PPO algorithm and the introduction to the clipped surrogate objective function to learning the cost function faster without much deviations from the previous values.

Even though deep neural networks show human level accuracy in areas of vision, speech and language, they are vulnerable to small perturbations in inputs. These perturbation based attacks are called adversarial attacks. Initial work on adversarial attacks was done on by [11]. The authors introduced new way of deceiving deep networks with the Fast Gradient Sign Method(FGSM). This was the first attempt to expose the linearity of neural networks with adversarial perturbations in input images. In [12] authors propose a more robust attack which is universal and based on "first order adversary", called Projected Gradient Descent(PGD). The PGD attack is further

enhanced by adding different restarts and more number of iterations to perform perturbations. Despite PGD being one of the best methods for creating adversarial attacks, the different parameters needed for the attack like step size, epsilon value, number of restarts and number of iterations made it difficult to compare different defense techniques. This led to proposal of Autoattack by [13]. Autoattack is an ensemble of 4 attacks; 2 white-box attacks which are step-free variations of PGD and differ in terms of loss functions, the other 2 black-box attacks targeted FAB [14] and Square Attack [15]. For some time now this attack has been considered the academia standard. Later, claiming similar robustness in adversarial training as PGD, [16] claimed improvement on FGSM with random initialization. This method was time and compute efficient as compared to previous techniques. We use this improvement on FGSM to implement our attacks on off-policy based methods.

Deep Reinforcement Learning techniques inherited the flaws of deep learning models and became equally susceptible to adversarial attacks. [17] studied the affects of adversarial attack in policy based methods. The authors also showed how RL methods see a significant drop in accuracy with both white box and black box attacks. A recent work [18] focuses on effects of attacks on multi-agent RL systems. In our work we focus on single agent policy based methods to perform adversarial attacks.

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

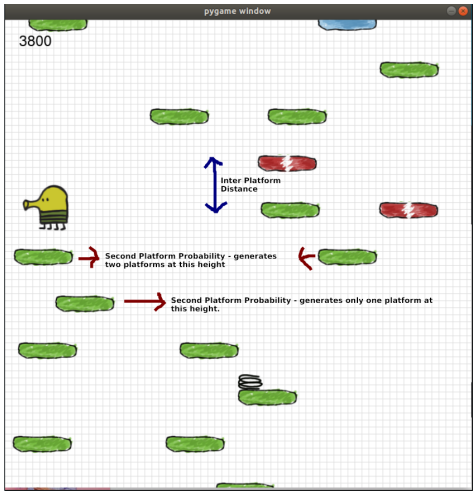

Fig. 1. Explaining inter-platform distance and second platform probability.

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 explored multiple Reinforcement learning algorithms instead of a supervised ML approach.

Reinforcement learning algorithms can be broadly classified into value based and policy based.

1) Value Based Approaches: In the first part of the project, we have looked into value based approaches of reinforcement learning and we have experimented with the Deep Q-learning and Deep Recurrent Q-learning algorithms. Our RL system for the value based approaches 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).

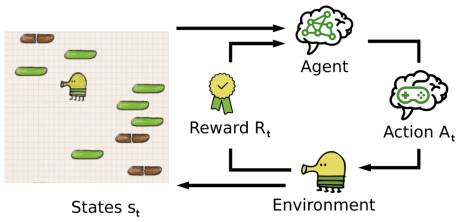

Fig. 2. DQN and DRQN RL System

2) Combined Value and Policy Based (Hybrid) Approaches: In this part of our project, we have explored several policy gradient approaches of reinforcement learning. In such methods, we try to optimize the policy without a value function. But policy methods have their own disadvantages and hence we have looked at hybrid approaches. We particularly look at Actor-Critic methods, which are a combination of both policy based and value based methods. We have experimented with the Advantage Actor Critic(A2C) and Proximal Policy Optimization(PPO) algorithms. The RL system for actor-critic based methods is very similar to our previous RL system except that the agent now will use 2 networks to interact with the game(Fig. 3). Our actor takes the state, gives an action and receives a new state and reward. Critic computes the q value of taking that action at that state. The Actor then updates its policy parameters based on the q value and chooses the next action to take. The Critic then updates its value parameters.

The Advantage function in advantage actor critic model tries to calculate the prediction error (Temporal-Difference Error). The TD error is calculated as the difference between the TD target and the value of the next state as calculated by the critic model. This TD target is nothing but the predicted value of all the future rewards generated from the current state. The value of Advantage function helps in determining if the agent should be encouraged to take more of such actions or not.

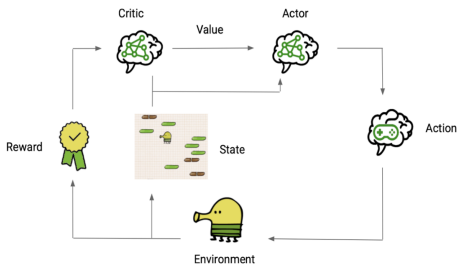

Fig. 3. Actor Critic RL System

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 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. 4). 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.

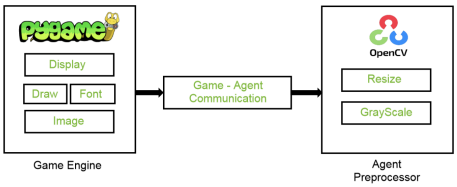

Fig. 4. Pre-processing pipeline

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 6 different types of reward functions. Each of these are explained in detail in the experiments section.

F. Agent Modelling

The schema for training our agent is described below. An important consideration in any RL task is the Exploration/Exploitation trade-off. Exploration is the task of allowing the agent to try out random actions from the action

space to understand the environment thoroughly. On the other hand, exploitation is 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. To achieve this we use an arbitrary epsilon value that 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. We then check if the game is active or has ended. If the game has been terminated after an action, we reset the game parameters and train the long memory based on the sequence of states that have been stored for every action until the game was terminated. If the game was not terminated, we loop back to fetching the current state of the game and perform the above operations in sequence.

Sequential inputs of states to the model/network, can cause the model to forget previous interactions and experiences as new ones are learned. We make use of a replay buffer that stores details of the experiences for every interaction or experience that the model learns. 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.

G. Adversarial Attack

In this section we focus on certain pitfalls of game bots trained using RL techniques. We implement [16] that manipulates the input image sent to reinforcement learning system by adding some gradient based perturbations. However, these perturbations look like white noise and can be easily ignored by the human eye. (Fig. 5) shows different game states before and after perturbation. We notice that it has a huge impact on the RL agent and results in the agent performing poorly. We generate these white box attacks in the testing phase after the model is trained as shown in (Fig. 6).

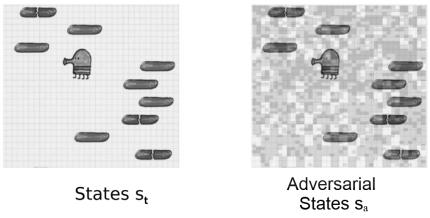

Fig. 5. The left image is the actual game state where model predicts Jump Left, Right image is the perturbed state where model predicts No Action

IV. Methods

We have used 4 popular and widely used Reinforcement learning models in order to train the agent. The two value

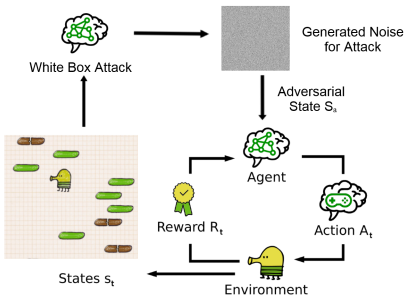

Fig. 6. Test Time Adversarial Attack

based models that we have used are DQN and DRQN. On the other hand we have also implemented hybrid methods like A2C and PPO. We have performed experiments on all these models for different reward functions and different parameters. We will elaborate more on our experiments in the next section of the paper.

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. 7).

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

Fig. 7. Q-function formula

A. Deep Q-learning

Our DQN architecture (Fig. 8) 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.

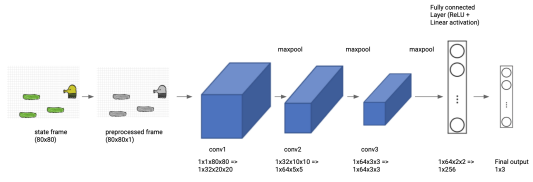

Fig. 8. Deep Q-learning architecture

B. Deep Recurrent Q-learning

The DRQN architecture (Fig. 9) 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.

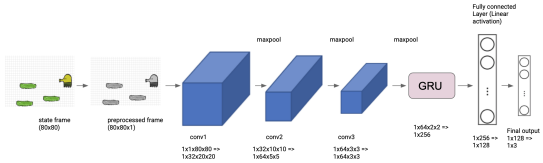

Fig. 9. Deep Recurrent Q-learning architecture

C. Advantage Actor Critic

The A2C architecture (Fig. 10) is very similar to the baseline DQN model, but we make use of 2 networks for the Actor and the Critic. The preprocessing steps remain the same for hybrid approaches. Finally, the input is an 80x80 image which has been preprocessed and converted to gray scale. This goes through multiple convolution layers and the flattened result is sent to the actor and critic networks which are dense layers. The actor outputs the action probabilities of the policy (generated from the neural network). This is the policy part of the model. The other output channel is the value estimation from the critic which is a scalar output which is the predicted value of the current state. The actor initially starts off with some random action and the critic evaluates how good the action is. This process continues as the agent learns to go ahead in the game while trying to maximize the reward.

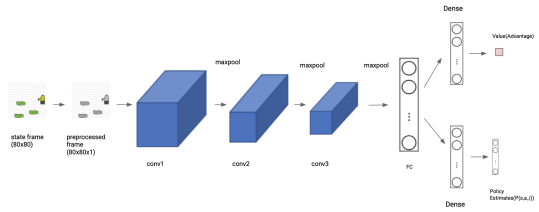

Fig. 10. A2C/PPO architecture

D. Proximal Policy Optimization

The PPO architecture (Fig. 10) is the exact replica of our A2C architecture. However, PPO is said to improve the stability of the Actor training by limiting the policy update at each training step. The major difference in the implementation is w.r.t to the way we optimize the cost function. In PPO, we use a clipped surrogate objective function to compute an update at each step that minimizes the cost function while ensuring the deviation from the previous policy is relatively small.

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. Exploration vs Exploitation

In any Reinforcement learning algorithm, it is important to have a balance between exploration and exploitation. Exploration helps the model perform random moves and search for global maxima/minima, whereas exploitation helps the model learn from the training. A large exploration could lead to random results whereas a large exploitation could lead to overfitting. There are several algorithms that address this. We have researched three such algorithms:

1) Type 1 - Epsilon Greedy with fixed epsilon: In this algorithm, we fix the epsilon value (e) at the start of the game to a probability. At each step, we generate a random number (r) between 0 and 1. If $r > e$, we perform exploration, else we perform exploitation.

2) Type 2 - Epsilon Greedy with exponential decaying epsilon: In this algorithm, we choose epsilon value (e) as 1 at the beginning of the game. At eachstep, we decay the epsilon value. $e = e^{'}(1-b)x$, where e is the final epsilon value, $e^{'}$ is the initial epsilon value, b is the decay factor and x is the number of steps.

3) Type 3 - Epsilon Greedy with exponential curve decaying epsilon: In this algorithm, we again choose epsilon value (e) as 1 at the beginning of the game. $e = \lambda\ (\mathbf{e})^{-x}$, where e is the final epsilon value, **e** is the Euler's number λ is the decay factor and x is the number of steps.

TABLE II
EXPLORATION VS EXPLOITATION EXPERIMENT RESULTS

Algorithm Type	High Score	Mean Score
Type 1	8700	1715.3
Type 2	9100	**3115.0**
Type 3	**12200**	2837.3

Table II refers to the high score and mean score noted during these experiments. From these observations we infer that Type 2 exploration type - Epsilon greedy with exponential decaying epsilon worked best for our agent and have restricted our experiments to the type 2 exploration type.

B. Reward Formulations

We used 6 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.

5) Type 5: The reward remains similar to Type 4, except that it is not penalized if it stays on the same platform.

6) Type 6: The reward is similar to Type 5, but the penalty for dying is very high: -20, to force the agent to learn well.

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.

TABLE III
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**

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

TABLE IV
REWARD EXPERIMENT RESULTS WITH EXPLORATION VS EXPLOITATION

Reward Type	High Score	Mean Score
Type 4	7400	1543.0
Type 5	**18100**	**2110.3**
Type 6	7100	395.33

The results above in Table IV are obtained with the best exploration type - *Epsilon greedy with exponential decaying epsilon*.

C. Kickstarting the Doodler

The doodler is only rewarded when it moves up and there is an increase in the score. For the purpose of training, a kickstart was given to the doodler at the beginning of the game, where it would jump a series of platforms and reach an initial score of 500 before being left to exploit the search space on its own. The intuition behind this move was to give the agent a slight push in the right direction with easy rewards at the beginning of the game, to reduce the exploitation space.

In addition to that, to improve the mean score of the agent and to prioritize learning of navigation through the platforms, we modified the game so that the moving blue platforms would appear after a score of 10,000 was reached, and the red broken platforms and monsters would appear only after a score of 25,000 was reached. However, we did not see any improvement in the mean score after adding making this modification.

D. 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 V, we note that learning rate of 1e-3 gives the best mean reward and mean score.

TABLE V
LEARNING RATE 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 VI shows the results of these experiments.

TABLE VI
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 VII

TABLE VII
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**

E. Final Training Results and Analysis

The table below in Table VIII summarizes the final results of training for all our models, with each model trained for 2000 game episodes. The DQN and DRQN models were trained with a learning rate of 1e-3, gamma 0.9, game memory of 10000 and batch size of 1000. We can see a very good performance improvement in our DQN and DRQN models with the addition of the type 2 decaying epsilon parameter(epsilon_g_decay_exp) for the exploration phase. Both the DQN and DRQN performed the best with reward type 4 in terms of all the 3 metrics of our agent - mean score, mean reward and high score. We also notice that A2C and PPO models give good results in terms of both the mean score and mean reward. The mean reward is the lowest in comparison to other models for these methods. Overall, we notice a significant improvement and agent performs well. We see that the agent has also reached very high scores. The A2C gave the best results with reward type 5 and PPO performed best with reward type 4. The actor and critic networks both used a learning rate of 4e-4 for both A2C and PPO.

F. Fast FGSM Attack

We use two of our best models in terms of the mean score, the DQN and DRQN to test their robustness against such attacks. We implement a variation of the FGSM attack which has an epsilon parameter that controls the maximum variation in a pixel. The formulation of FGSM can be seen below. We used an epsilon value of 0.3 to introduce perturbations in input using Fast FGSM method. The Table IX summarizes the result of the FGSM attack on our best DQN and DRQN models. The models were tested for 300 game episodes. We notice that with the generation of attacks our model performance has reduced significantly in terms of the mean score. We notice that DQN has a greater impact on the attack compared to DRQN A reason for robustness of DRQN is its use of recurrent layers to remember temporal information.

TABLE VIII
FINAL RESULTS

Model	Reward Type	Mean Reward	High Score	Mean Score
DQN	4	-0.91053	11800	2854.2
DRQN	4	-0.89775	21100	2079.7
A2C	5	0.0000	11600	2742.1
PPO	4	-0.00034340	9400	1841.3

FGSM Attack Equation:

$$adv_x = x + \epsilon * sign(\nabla_x J(\theta, x, y))$$

TABLE IX
FGSM ATTACK RESULTS

Model	Benign Mean Score	Adversarial Mean Score	Performance Drop %
DQN	2854.2	1464.6	48.6%
DRQN	2079.2	1883.0	9.4%

VI. LIMITATIONS, CONCLUSION AND FUTURE WORK

In our experiments we saw, DQN and DRQN models are prone to overfitting especially when the RL environment is Non-Episodic in nature. In addition, using the imagenet pretrained models did not improve the performance of the off-policy methods. This points to the fact that higher number of parameters and transfer-learning may not always improve metrics especially in an RL system. Finally, Reinforcement Learning models are prone to adversarial attacks just like any deep learning model, even when the environment is made simpler.

For conclusion, we note that a continuous reward that increases in a logarithmic way is much more beneficial for RL based systems. To stop overfitting of DQN & DRQN models, experience replay with decaying epsilon is a good regularization technique. With respect to the policy based models, A2C and PPO, behave more human like as they are cautious in taking steps, this is a result of using critic network working in tandem to judge the output from action model.

In future, we would like to explore other off-policy methods like Action-Specific Deep Recurrent Q Network (ADRQN).

We also look forward to see the effects of adversarial attacks on A2C and PPO models. In parallel, to make models robust we would use adversarial training with the feedback of perturbed images to train Reinforcement Learning System.

ACKNOWLEDGMENT

An endeavour is successful only when it is carried out under proper guidance. We would like to thank "Prof. Micheal Zyda" for his encouragement and guidance throughout the course in completing the project. We would also like to thank Prof. Zyda for introducing the project as part of the curriculum for the course CSCI 527 at the University of Southern California. We would also like to thank the entire teaching staff of the course for 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] Croce, F., & Hein, M. (2020, November). Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks. In International Conference on Machine Learning (pp. 2206-2216). PMLR.

[14] Croce, F., & Hein, M. (2020, November). Minimally distorted adversarial examples with a fast adaptive boundary attack. In International Conference on Machine Learning (pp. 2196-2205). PMLR.

[15] Andriushchenko, M., Croce, F., Flammarion, N., & Hein, M. (2020, August). Square attack: a query-efficient black-box adversarial attack via random search. In European Conference on Computer Vision (pp. 484-501). Springer, Cham.

[16] Wong, E., Rice, L., & Kolter, J. Z. (2020). Fast is better than free: Revisiting adversarial training. arXiv preprint arXiv:2001.03994.

[17] Huang, S., Papernot, N., Goodfellow, I., Duan, Y., & Abbeel, P. (2017). Adversarial attacks on neural network policies. arXiv preprint arXiv:1702.02284.

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

[19] Mnih, V., Adrià Puigdomènech Badia, Mehdi Mirza, A. Graves, T. Lillicrap, Tim Harley, D. Silver and K. Kavukcuoglu. "Asynchronous Methods for Deep Reinforcement Learning." ArXiv abs/1602.01783 (2016): n. pag.

[20] Schulman, John, F. Wolski, Prafulla Dhariwal, A. Radford and Oleg Klimov. "Proximal Policy Optimization Algorithms." ArXiv abs/1707.06347 (2017): n. pag.