ENGINEERING DESIGN DOCUMENT - CSCI527

TEAM DOODLE JUMP

Team Members
1. Amulya Raveendra Katti (6237870039)
2. Anamay Sarkar (6849143870)
3. Pranav Mallikarjuna Swamy (9564754972)
4. Riya Kothari (8465528522)
5. Shenoy Pratik Gurudatt (7435908983)

Submitted to : Prof. Mike Zyda

March 16, 2021

Contents

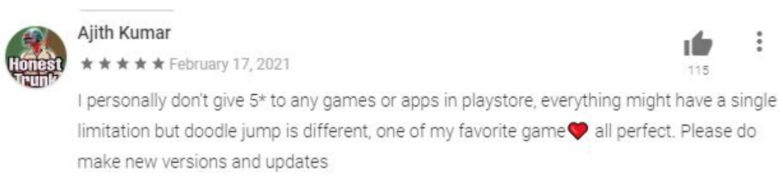
1 Introduction and Project Goal

1.1 Introduction

Deep Reinforcement Learning is one of the areas of Machine Learning that has been widely used to build game agents and bots to challenge and beat human players. It has gained a lot of attention 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 to learn to play Atari games from game image inputs.

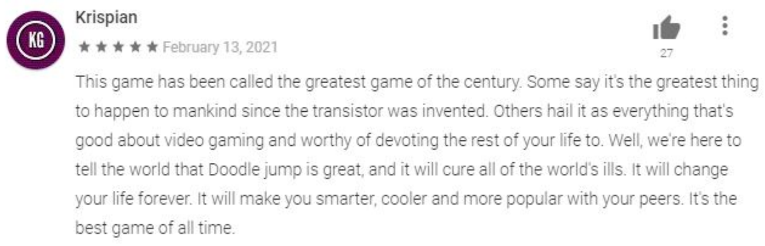
Doodle Jump is a popular video game by American studio Lima Sky, created by Igor and Marko Pušenjak[2], available on all major platforms. When it was released, Doodle Jump skyrocketed to fame, accounting for over 25,000 copies of the game being sold every day for 4 months. The game is so popular that it has been developed into a video redemption game at video arcades[2].

1.2 Latest Game Reviews



Ajith Kumar
★★★★★ February 17, 2021
115

I personally don't give 5* to any games or apps in playstore, everything might have a single limitation but doodle jump is different, one of my favorite game❤️ all perfect. Please do make new versions and updates



Krispian
★★★★★ February 13, 2021
27

This game has been called the greatest game of the century. Some say it's the greatest thing to happen to mankind since the transistor was invented. Others hail it as everything that's good about video gaming and worthy of devoting the rest of your life to. Well, we're here to tell the world that Doodle jump is great, and it will cure all of the world's ills. It will change your life forever. It will make you smarter, cooler and more popular with your peers. It's the best game of all time.

1.3 Goal

Through the project, we aim to train an agent to play a python coded version of the Doodle jump game by exploring different reward functions and different learning models like Deep Q-Networks, Deep Recurrent Q-Networks, Proximal Policy Optimization and others.

2 Background and Game Overview

2.1 Background

Doodle Jump is a platforming video game developed and published by American studio Lima Sky, for Windows Phone, iPhone OS, BlackBerry, Android, Java Mobile (J2ME), Nokia Symbian, and Xbox 360 for the Kinect platform. [3] The game has gained widespread popularity since its initial release and is now currently available on 9 platforms.

2.2 Game Play

2.2.1 Basic Doodle Jump

The Doodle Jump game has many variations across the different distributors. The basic game play has a Doodler - a four legged creature, which tries to get on top of platforms to get scores. The higher the doodler reaches, the more score he gets. The doodler constantly jumps in its position, making the game control simple, allowing the player to move the doodle either left or right. The basic game snapshot includes the doodler, platforms and a score. The platforms keep appearing endlessly as the doodler climbs higher. The game ends when the doodler falls off a platform. The left end of the screen connects to the right end, making it continuous in the horizontal plane.

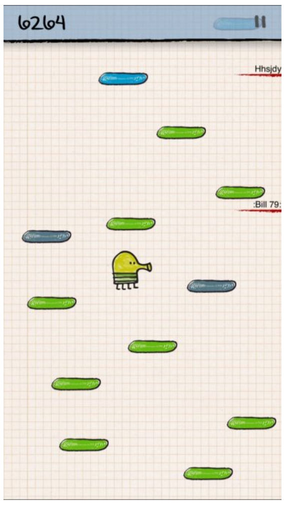
The game can have many dynamic elements. Springs provide boost to the jump. Jetpacks and rockets aid in the doodler's journey by providing vertical lift to the doodler. UFO's pull the doodler towards it which eventually kills the doodler and ends the game. Monsters appear on some of the platforms, touching which freezes the doodler and kills it.

2.2.2 Doodle Jump Modes

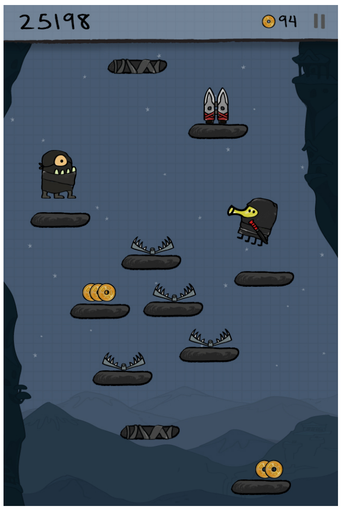
The game can have different look and feel, supporting different modes.

Original

The original version of the game has monsters, springs and platforms. Some implementations have UFOs, some have jetpacks, while some give the doodler, the ability to shoot at the enemies.



(a) Doodle Jump Original

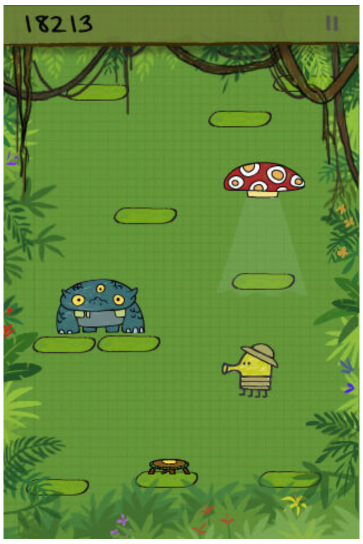


(b) Ninja Doodle Jump

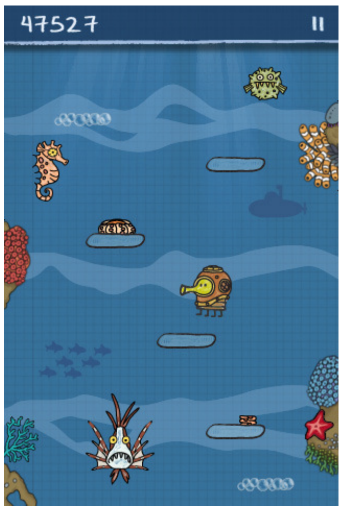
Figure 2.1: Game Modes

Ninja

In the ninja mode of the game, the doodler wears a ninja costume. He can collect knives to shoot at the monsters. The donuts acts as springs. More the donuts, higher the doodler jumps. It uses a dark theme.



(a) Rainforest theme



(b) Underwater theme

Figure 2.2: Game Modes

Rainforest

In the rainforest mode of the game, the doodler wears a safari costume. The animals act as monsters. Trampolines act as springs. It uses a jungle theme.

Underwater

In the underwater mode of the game, the doodler wears a scuba costume. The underwater creatures act as monsters. This variation also introduces flying monsters as seahorses. The doodler can collect shells to shoot at the monsters. It uses an ocean theme.

2.2.3 Game Implementation

Starter code

For this course we have chosen the original mode of the game. A basic version of the game is forked from a github repository. (https://github.com/f-prime/DoodleJump). In this version the doodler could jump over static green platforms. It had springs as the only dynamic element, touching which would provide a boost in the jump height.

Game Improvements

Added 2 more types of platforms - blue platforms which goes back and forth in the horizontal plane and red platforms which breaks when the doodler jumps on it. To be on the blue platforms, the doodler constantly needs to move with the platform, failing which the doodler falls down. The red platforms can only be used once for jumping. These introduce dynamic difficulty to the game. We can control the proportion of each of these platforms in the game.

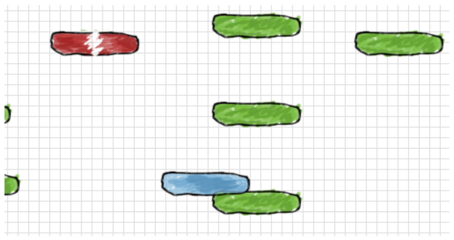


Figure 2.3: Multiple platform types

Adjusted the jump height of the springs, to prevent the doodler to go out of the screen view while jumping.

Added monsters as another dynamic element. The monsters appear on some of the green stable platforms. The doodler must avoid them as touching them kills the doodler and resets the game. We have also added a red effect on the touch. The monster turns red and for a brief interval doodler figure changes to a dead doodler character.

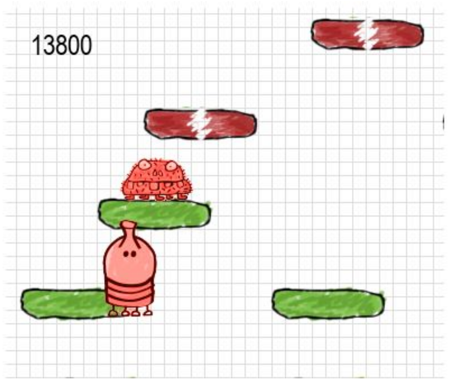


Figure 2.4: Monster

Added support for multiple platforms on the same level. Parameterized the vertical inter-platform distance.

Enabled three game modes - easy, medium and hard based on the number of monsters, number of springs, the proportion of three types of platforms, inter-platform distance and number of platform in the same level.

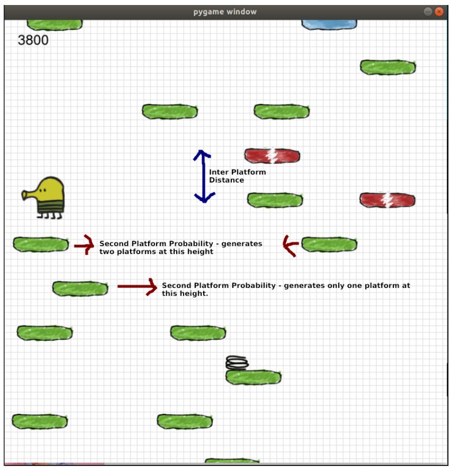

pygame window
3800
Inter Platform Distance
Second Platform Probability - generates two platforms at this height
Second Platform Probability - generates only one platform at this height.

Figure 2.5: Game Design

Added a kick start code as a function, where we generate platforms one over the other till the doodle reaches a specific initial score.

2.2.4 Game Agent

The purpose of the game agent is to connect the game and the model that we use for prediction.

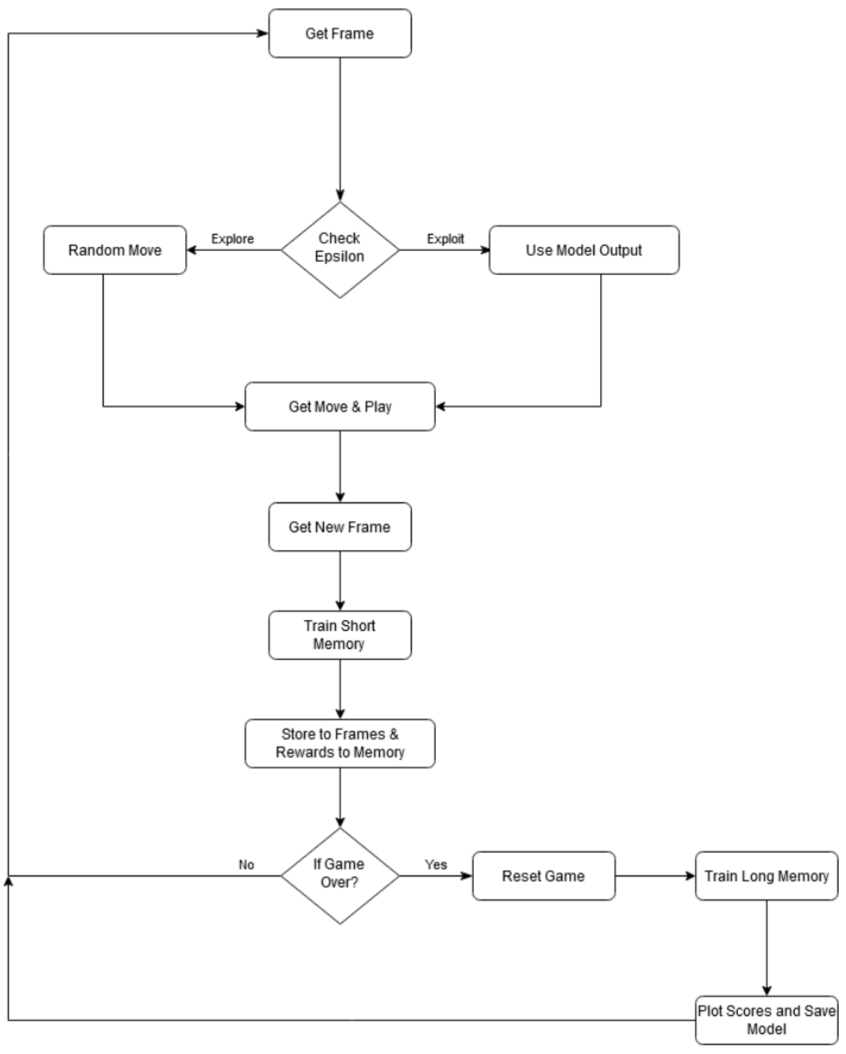


Figure 2.6: Agent Flowchart

The agent receives the game state from the game. The state is received in the form of current frame. The agent checks the epsilon value to determine whether to perform exploration or exploitation on the game states. A larger epsilon value favors exploration, where the doodle performs more random moves in order to attain the optimal states, whereas a smaller epsilon value favors exploitation where the doodler is heavily dependent on the model for the next states. A proper balance between the two is desired.

This gives the next move the doodler is supposed to play.

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. 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. The agent then sends the game action to the game, which performs the action and a resulting frame is returned to the agent.

One particular game step information with regards to initial state, final state, action and reward is used for short term memory training. The current game information is saved and when the doodler dies, the entire game information is used for long term memory training.

Rewards

The doodler receives rewards for its actions. The reward function that we have explored [11] are as follows:

- **Type 1 (Baseline)** - A reward of +3 awarded to the doodler if the score increases, -2 if the doodler dies, -2 if the doodler is stuck in the current frame for over 100 seconds and 0 reward otherwise.
- **Type 2** - A reward of +3 awarded to the doodler if the score increases, -2 if the doodler dies, -2 if the doodler is stuck in the current frame for over 100 seconds and -1 reward otherwise.
- **Type 3** - A reward of +3 awarded to the doodler if the score increases + (+3) if the agent touches a spring and (-4) if the doodler touches a monster, -2 if the doodler dies from falling, -2 if the doodler is stuck in the current frame for over 100 seconds and -1 reward otherwise.
- **Type 4** - A reward of +3 awarded to the doodler if the score increases + (+3) if the agent touches a spring and (-4) if the doodler touches a monster + log(current score), -2 if the doodler dies from falling, -2 if the doodler is stuck in the current frame for over 100 seconds and -1 reward otherwise. This reward function is continuous while all others are discrete.

3 Related Work

3.1 Reinforcement Learning System

Reinforcement Learning is a Machine Learning technique which resorts to learning through attempts, failures and the associated rewards. For our game, training of the doodler happens via rewards and punishments, and the doodler is expected to move forward in game by maximizing this reward. To do this learning, the agent evaluates the current situation, takes an action and then based on the new state of the environment, receives the feedback (reward).

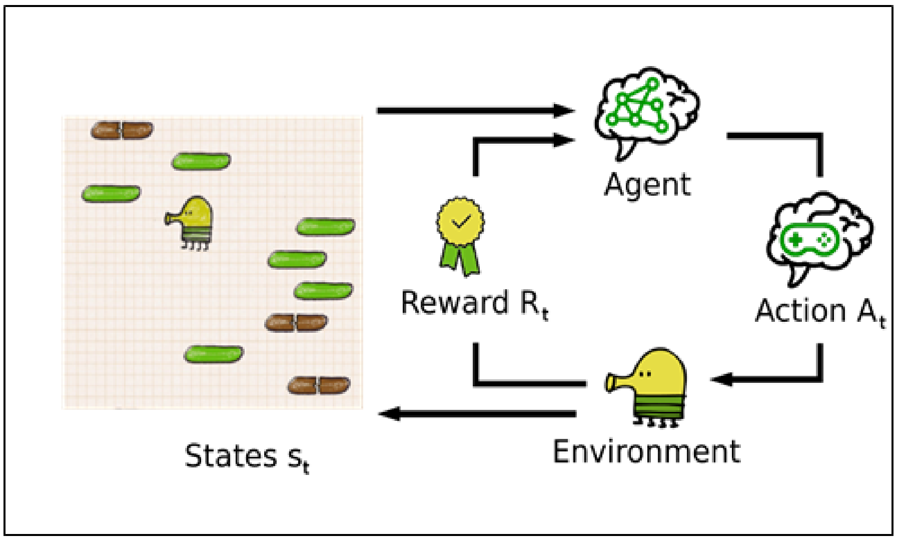


Figure 3.1: A reinforcement learning system

3.2 Q-Learning

Q-Learning is an off-policy RL algorithm that uses temporal difference approach to find the best action from a current state in order to maximize the reward. The q-learning function holds the capacity to learn from actions which lie outside the current policy, similar to performing a random move, and are therefore referred to as off-policy algorithms [8]. The Q here stands for the quality of the action's usefulness in gaining future reward. Here, a q-table is created that stores for all state and action pairs, their q-values and updates them as the agent plays. Thus Q-value is the maximum total reward generated from the sequence of actions, given that we know the expected reward for each action at every step. This Q-value is a recursive formula as follows:

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

Our environment consists of dynamic parts like platforms, monsters and springs. This can result in infinite number of state and action pairs. As Q-Learning maintains a table of reward for each step, this would spiral out quickly and we won't be able to infer Q-values of new states from already explored states. So instead of directly determining these values, we try to approximate the Q-value function by the use of neural networks.

Apart from the vast number of state-action pairs, learning involves decoding complex pattern of the environment. Additionally, it also requires that the doodler understands which type of pattern yields positive rewards, and which type will result in negative rewards or failures.

For learning this complex architecture and to overcome the obstacle faced by simple Q-Learning, we employed the use of a Deep Learning network. The usage of Deep Learning models enables us to take advantage of the advancements happened in this field, which makes the recognition of platforms, monsters and springs quite easy compared to non-deep learning methods.

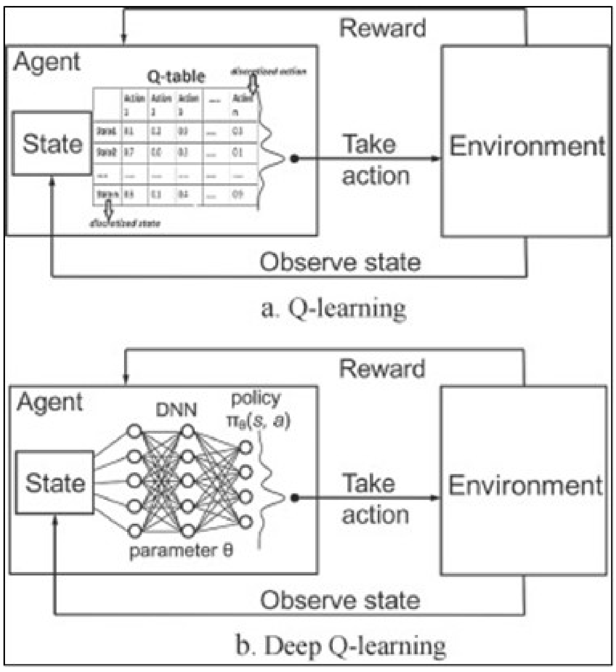


Figure 3.2: Q-Learning vs Deep Q-Learning

3.3 Actor-Critic Models

The two main classes of RL methods are Value based and Policy based methods. The former tries to approximate the most favourable value function, which is a mapping between an action and a value, while the latter tries to find the optimal policy directly. Both have associated pros and cons. Policy based RL models perform better for environments that are continuous and stochastic while Value based perform better in more steady and sample efficient environments. Researchers have developed models that merge these two categories and thereby benefit from both of their pros. Actor-Critic [7] methods are a result of this development.

Principally, the actor-critic model divides the model into two parts. One which computes an action based on the current state, while the other determines the Q-value of the action. The actor part takes in a state, learns the optimal policy and outputs the most suitable action. The critic evaluates this action and computes the value function. These two models interact with each other while getting better at their job with time [6].

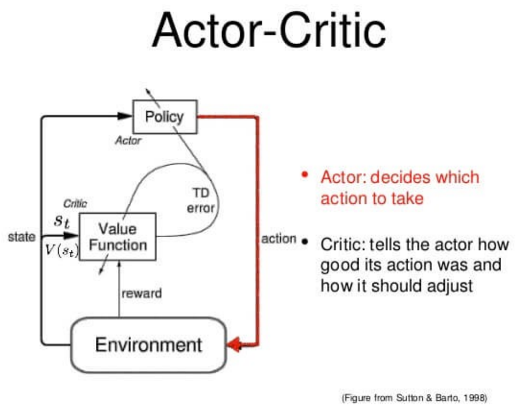

Actor-Critic
Policy
Actor
Critic
TD error
s_t
state
$V(s_t)$
Value Function
reward
action
Environment
• Actor: decides which action to take
• Critic: tells the actor how good its action was and how it should adjust
(Figure from Sutton & Barto, 1998)

Figure 3.3: Actor Critic model overview [6]

Both the models can be approximated by neural networks, but their training is performed separately and gradient ascent is used to update their weights [6]. With time, the actor gets better in producing actions while the critic gets better in evaluating those actions. These have proven successful in complex games such as Super Mario and so we plan to apply this method in our Doodle Jump game as well, in the future.

There are two variants to Actor-Critic models. The first is Advantage Actor-Critic or A2C, where instead of the Q-values, the critic learns the Advantage values. The second is Asynchronous Advantage Actor-Critic or A3C [13], which has multiple independent agents who are trained in parallel with different weights for faster exploration. We aim to try these models for our game as well.

4 Methodology

Before implementing the model and its communication to the game, we set up the game engine and the agent preprocessor. The game engine consists of Python's Pygame display renderer which fetches the current state as an image. For preprocessing the state, we used the OpenCV library. This preprocessing includes resizing of an 800x800 image to 80x80 image. Post resizing, the image is also being converted to single channel in grayscale. All this helps in reducing the network parameters while keeping the required information.

So far, we have implemented two Reinforcement Q-Learning based methods.

4.1 Deep Q-Networks

In Deep Q-Network [10], we are using a neural network to approximate the Q-value function. The input to this network is state image and output is the Q-value for each action in that state. We are using a epsilon-greedy exploration scheme and are probabilistically choosing between a random action and the action with highest Q-value.

In DQN, we are using a replay buffer to store samples with the current policy. Then from this buffer, we randomly sample batches of experiences (experience replay) and use them to update the Q-network. This is the whole cycle that happens repeatedly. The use of experience replay ensures that the data being fed to the network is not correlated and is independently distributed. If this step is not performed, the data has temporal correlation and might result in significant oscillations and divergence in our model [12].

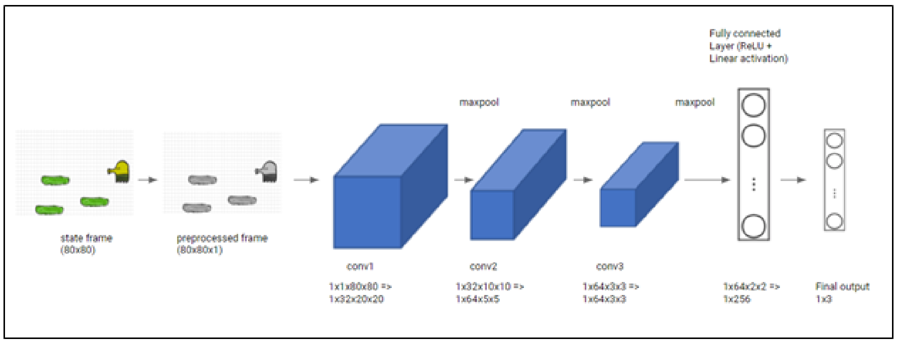
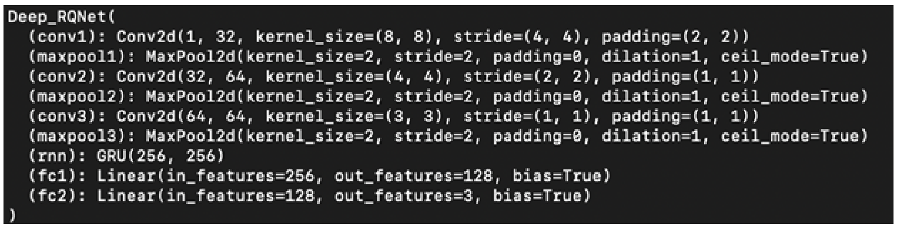


Figure 4.1: DQN Model architecture

The Q-network is updated by minimizing the mean squared error loss between the target Q-value which is obtained by using the Bellman equation and our current Q-output.

The underlying architecture we chose for our Doodle-jump game consists of three convolutional layers, each followed by a maxpool layer. Each convolutional layer has RELU [15] as an activation function. These layers are then followed by two fully connected layers where we have used RELU activations, again. These finally give out an output of dimension 1x3, corresponding to the three types of actions, namely move left, do nothing or move right.



```
Deep_RQNet(
  (conv1): Conv2d(1, 32, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=True)
  (conv2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=True)
  (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (maxpool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=True)
  (rnn): GRU(256, 256)
  (fc1): Linear(in_features=256, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=3, bias=True)
)
```

Figure 4.2: Input/Output dimensions for each layer of DQN

Figure 4.3 gives a detailed description about the specifications like padding, stride and filter size, of each layer in the Deep Recurrent Q-Network.

4.2 Deep Recurrrent Q-Networks

GRUs are improved version of recurrent neural networks, which address the vanishing gradient problem of standard RNNs using two gates, namely update and reset gates. These can retain information from long ago using the gates, and that is something we require for our player to perform better over time. Our doodler needs to learn from the moves that it performed long ago, instead of only focusing on the recent moves [9].

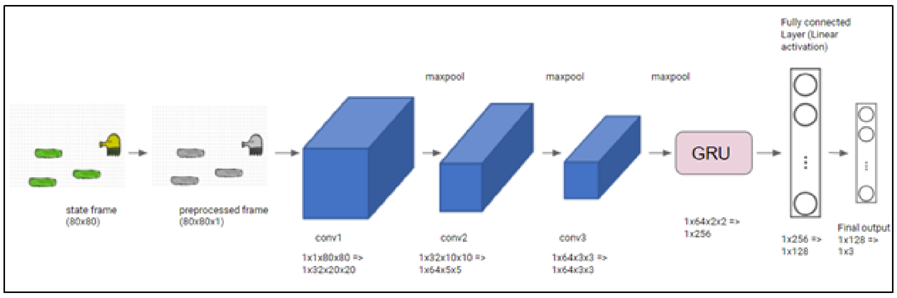
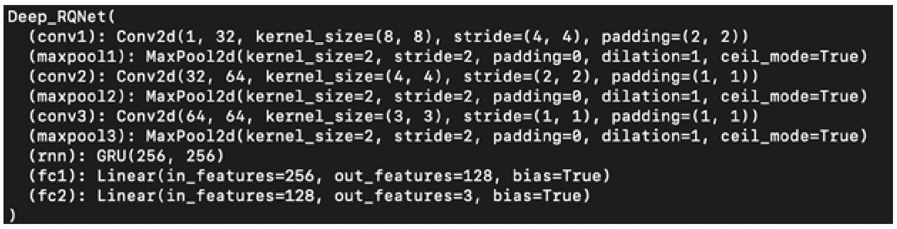


Figure 4.3: DRQN Model architecture

Deep Recurrent Q-Networks or DRQN [14] vary very slightly from DQN, that too in only one of the layers of architecture. In DRQN, we have replaced the first fully connected layer from DQN by a gated recurrent unit (GRU). The motivation behind doing this is that there are noisy and incomplete observations because of partial observability. The GRU helps address this problem by retaining memory of previous observations, enabling it to make better decisions.



```
Deep_RQNet(
  (conv1): Conv2d(1, 32, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=True)
  (conv2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=True)
  (conv3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (maxpool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=True)
  (rnn): GRU(256, 256)
  (fc1): Linear(in_features=256, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=3, bias=True)
)
```

Figure 4.4: Input/Output dimensions for each layer of DRQN

Figure 4.4 gives a detailed description about the specifications of each layer in the Deep Recurrent Q-Network.

5 Results and Experiments

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.

5.1 Rewards Formulation

The 4 different types of rewards that we used for our agent are mentioned in chapter 2.2.4 For each of these reward functions, we ran our agent for 300 game episodes with a learning rate of 1e-3, discount rate of 0.9, batch size of 11 and max memory of 10k. The exploration was random till first 40 games there on continuously training the Deep Q-Network model.

Table 5.1: 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**

Table 5.1 refers to the high score and mean score noted during these experiments. From these observations we infer that Type 4 reward worked best for our agent and have restricted our experiments to the type 4 reward. Interestingly, for the experiment with reward Type 1, our agent was stuck in the game often instead of jumping on neighbouring platforms. It is noteworthy, that this behaviour was caused due to the non-negative reward that the agent received by default for staying at the same position.

5.2 Hyper-parameter Tuning

Hyper-parameter tuning for a RL model is not any different from that of other Deep Learning systems. Though, there is a minor difference in the two, as here we compare the results of the experiments based on mean score and mean reward.

5.2.1 Learning Rate

For tuning the learning rate hyper-parameter, we ran studies with values of learning rate starting from 1e-2 and reducing it to 1e-4. The remaining parameters are the same as that of reward experiments. As seen in Table 5.2, we note that learning rate of 1e-3 gives the best mean reward and mean score.

Table 5.2: 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

5.2.2 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 5.3 shows the results of these experiments.

Table 5.3: 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

5.2.3 Weighted Expected Reward

In Deep Q-Networks, the discount factor (gamma) makes it possible for the agent to give weight to the expected rewards. Expected rewards vary proportionally to the gamma rate. 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 5.4

5.2.4 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,

Table 5.4: 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**

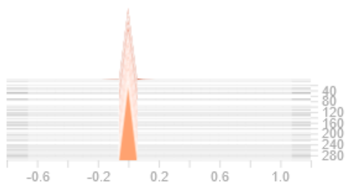
gamma 0.8, game memory of 10k, batch size of 1k. The second set (Agent 2) including learning rate of 1e-3, gamma 0.99, game memory of 25k, batch size of 5k. We run these two experiments for 2000 game episodes to make sure the model is able to achieve good scores consistently for a long duration of time.

The results of table 5.5 represent those of training DQN model. We see that Agent 1 of

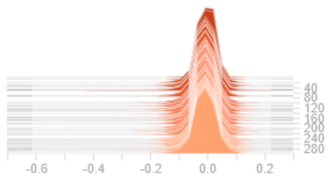
Table 5.5: DQN Agent Results

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

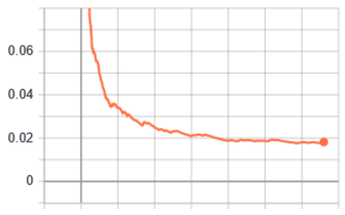
the DQN model achieves the highest high score of 11000 for the game agent. Also, the mean score of Agent 1 is consistent in the range of 1500. This means the model does well invariably over a long period of time. Whereas the Agent 2 is behind Agent 1 in all numbers especially mean score. On the other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



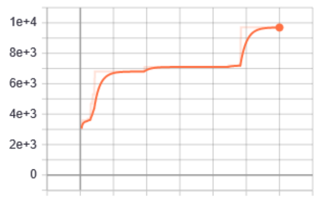
(a) Gradients plot



(b) Weights plot



(c) Loss plot



(d) High score plot

Figure 5.1: Various plots for DQN model

The plots above are from our best DQN agent. These 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.

Table 5.6: 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 in Table5.6 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 among 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.

5.3 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/AC2). In parallel we would like to explore the adversarial attacks: Fast Gradient Sign Method (FGSM)[10] and Projected Gradient Descent (PGD)[16]. Inspired from [17] these attacks would be integrated to the game agent code to sabotage the agent with the manipulated image from one of the above mentioned attacks.

Acknowledgment

We would like to thank "Prof. Micheal Zyda" for his guidance and wonderful teaching in 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 phases of the research project.

Bibliography

[1] Edmonds, Rich, “Doodle Jump hops back onto Windows Phone 8, but drops Xbox Live support”, *Windows Central. Mobile Nations. Retrieved 2013-08-22.*, 2013-08-21

[2] Kafla, Peter, “Meet the App Store Millionaires: The Brothers Behind Doodle Jump”, *allthingsd.com. AllThingsD. Retrieved 29 March 2015.*, 5 April 2010

[3] “Doodle Jump is coming to Kinect”. *Archived from the original on 19 May 2011,Retrieved 22 April 2011.* 15 March 2011.

[4] P. Xiao, W. Qu, H. Qi, Z. Li and Y. Xu, “The SDN controller placement problem for WAN”, *2014 IEEE/CIC International Conference on Communications in China (ICCC)*, Shanghai, 2014, pp. 220-224.

[5] Mnih, Volodymyr, et al. “Playing atari with deep reinforcement learning.” *arXiv preprint arXiv:1312.5602* (2013).

[6] Karagiannakos, Sergios. “The Idea behind Actor-Critics and How A2C and A3C Improve Them.“ AI Summer, Sergios Karagiannakos, 16 Nov. 2018, *theaisummer.com/Actor_critics/.*

[7] Konda, Vijay R., and John N. Tsitsiklis. “Actor-critic algorithms.“ *Advances in neural information processing systems.* 2000.

[8] Violante, Andre. “Simple Reinforcement Learning: Q-Learning.“ Medium, Towards Data Science, 1 July 2019, *towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56.*

[9] Kostadinov, Simeon. “Understanding GRU Networks.“ Medium, Towards Data Science, 10 Nov. 2019, *towardsdatascience.com/understanding-gru-networks-2ef37df6c9be.*

[10] Matthew Hausknecht and Peter Stone. “Deep Recurrent Q-Learning for Partially Observable MDPs.“ *arXiv preprint arXiv:1507.06527,* 2015.

[11] Bonsai. “Deep Reinforcement Learning Models: Tips & Tricks for Writing Reward Functions.“ Medium, Medium, 16 Nov. 2017, *medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0.*

[12] Mnih, V., Kavukcuoglu, K., Silver, D. et al. “Human-level control through deep reinforcement learning.“ *Nature 518, 529–533* (2015). https://doi.org/10.1038/nature14236

[13] Mnih, Volodymyr, et al. “Asynchronous methods for deep reinforcement learning.“ *International conference on machine learning. PMLR,* 2016.

[14] Chen, Clare, Vincent Ying, and Dillon Laird. “Deep q-learning with recurrent neural networks.“ *Stanford Cs229 Course Report 4* (2016): 3.

[15] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,“ Haifa, 2010, pp. 807–814. [Online]. Available: https://dl.acm.org/citation.cfm

[16] Goodfellow, Ian J., Jonathon Shlens, and Christian Szegedy. “Explaining and harnessing adversarial examples.“ *arXiv preprint arXiv:1412.6572* (2014).

[17] Madry, Aleksander, et al. “Towards deep learning models resistant to adversarial attacks.“ *arXiv preprint arXiv:1706.06083* (2017).