Machine Learning in Games: Exploring CNNs, LSTMs and Deep Q Learning for Fun Run 3

Kishan Gajera
University of Southern California
kgajera@usc.edu

Krunaal Tavkar
University of Southern California
tavkar@usc.edu

Masira Mulla
University of Southern California
mmulla@usc.edu

Param Patel
University of Southern California
parampat@usc.edu

Rutvik Kakadiya
University of Southern California
rkakadiy@usc.edu

***Abstract*—Machine Learning and Reinforcement Learning have recently been pushing the boundaries in terms of intelligent systems learning to play Video Games and obtaining "super-human" performance. The aim of this paper is to summarize our work and preliminary results, while exploring the options to develop an AI agent to play the game Fun Run 3, a multiplayer racing game. In this paper, we explore various state of the art algorithms and techniques, such as a combination of Convolutional Neural Networks (CNNs) for Object Detection and Long Short-Term Memory (LSTMs) cells, to train the agent to play the game by observing and learning from the gameplay of a human. We also look at a more robust learning mechanism, Deep Q Learning, which has been a very popular alternative to collecting massive data of human gameplay and instead have the agent learn to play the game via the trial and error method and learn on its own, to improve over time.**

***Index Terms*—Machine Learning, Deep Learning, Deep Q Learning, LSTMs, CNNs, Video Games, Fun Run 3, Reinforcement Learning**

I. Introduction

Fun Run 3 is a multiplayer online racing game. Players are expected to race while collecting power ups, tackling obstacles and overtaking other players. The game is setup in a side scrolling environment with dynamic terrains and unique obstacles and challenges, that add to the fun and also make the levels challenging with random spawning of lightening strikes to knock your racer down. The game consists of two modes, the Quick Race and the Arena Challenge modes. In the Quick Race mode, the player's goal is to finish the game as fast as possible, whereas in the Arena mode is set in a race against the clock format, where the last player at each given interval of time is eliminated from the game. The Arena mode has the top 3 survivors declared as champions of the race.

Our goal in this endeavor is to train an autonomous AI agent to succeed at the game, playing in any terrain level or mode of the game. The agent will learn to navigate the game space by observing screenshots of the game and has to choose between one of two moves, Jump, Slide. The additional move that the agent can make is to use a power-up, if available.

The agent will use CNNs for object detection to understand the game environment, by extracting a feature map from screenshots of the game and identify key elements of the game such as obstacles, power-ups and opponents. The agent will then learn to take actions in the environment based on trail and error.

The current road map is to have the agent learn and give a good performance in the Quick Race mode of the game and then using Transfer Learning, we would like to observe how the agent fairs in the Arena mode of the game, having never played in that setup. This is currently a stretch goal for this project.

II. Related Work

A. Super Mario

Liao et al. applied Reinforcement Learning to design an automatic agent to play Super Mario Bros [1]. By abstracting the game environment into a state vector and using Q learning — an algorithm oblivious to transitional probabilities — we achieve tractable computation time and fast convergence. After training for 5000 iterations, our agent is able to win about 90% of the time. We also compare and analyze the choice of different learning rate α and discount factor γ.

Lei et al. [2] discussed using Long Short-Term Memory (LSTM), an artificial recurrent neural network (RNN) architecture used in the field of deep learning, to train the computer agent to play the game Super Mario Kart. In this they found the agent sometimes did not realize it was driving in the wrong direction or the agent got permanently stuck in the corner. and suggested improving the model using minimap positions, which may help the network differentiate between two areas of a track that are otherwise indistinguishable.

B. DeepMind and Atari

Mnih et al. [3] gave the world the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model introduced a new variant of Q Learning, alongside a convolutional neural network, whose input was raw pixels from the game and the output was a value function estimating future rewards. Tested on seven Atari 2600 games from the Arcade Learning Environment, and no adjustment of the architecture or learning algorithm, their model outperformed all previous approaches on six games and surpassed a human expert on three of them.

C. Image Captioning, Image Localization and Attention

Yang Z. et al. [4] presented a multi-model neural network method derived to mimic the human visual system which automatically learns to describe the content of images. The said model contained an object detection and localization model, to extract information of objects and their spatial relationship in images respectively. A deep recurrent neural network (RNN) based on long short-term memory (LSTM) units with attention mechanism were used for sentences generation. Although the application was for a Natural Language Generation task, they provided a base algorithm to solve many object localization and attention problems in the field of Deep Learning.

III. Project Framework

After going over the literature and related works, we had initially planned to go with the method of using CNNs for Object Detection and recording the gameplay of humans playing the game, along with their keystrokes. Then, with the help of multiple LSTMs, we would map keystrokes to predicted output moves and have the agent learn this way.

We quickly realised that this would be a highly cumbersome process, as we would need a lot of annotated gameplay data, in order to obtain decent results. Another drawback of this method was the fact that the agent would learn to play the game by "mimicking" a human. This went against the idea that humans learn from their mistakes. From trial and error, and not by observing another human. This led us to move pursue Deep Q Learning as the learning algorithm for our agent.

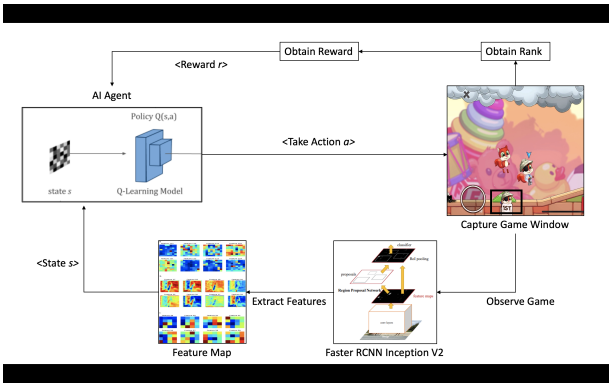

Fig. 1. Project Architecture

Figure [1] is a detailed view of our Project Architecture. The general idea of how our agent learns is that we capture a screenshot of the game window, pass that though an object detection and feature extraction layer, in order to obtain the state of the game. We then pass this state to the Deep Q Learning module, which then performs actions like $\langle slide, multi_slide, jump, multi_jump \rangle$ at random. Next, we obtain the rank of the agent in the race, and pass that back to the Deep Q Learning module as the reward for the current Policy: $\langle state, action, reward, nextstate \rangle$ and the process recursively continues for several games during training. We will dive deeper into each section of this framework in the next section.

IV. Methodology

A. The Dataset

We recorded 10 games, each being a video of approximately 90 seconds. These videos were then split into frames at a rate of 5 frames per second using Adobe Premiere Pro. The entire dataset consists of 4000 frames.

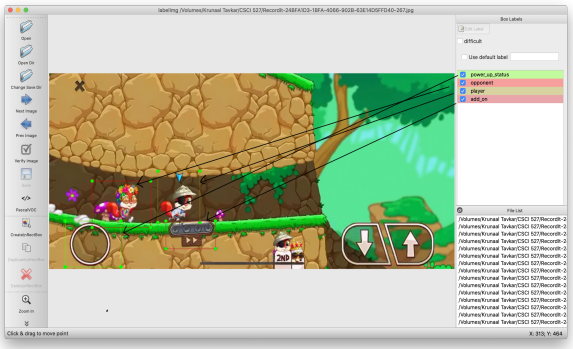

Fig. 2. Labelled Data using LabelImg

B. Annotating Training and Testing Data

Next, we labelled the frames in the dataset using LabelImg. LabelImg is a graphical image annotation tool written in Python and it uses Qt for its graphical interface. Annotations are saved as XML files in PASCAL VOC format, the format used by ImageNet. Besides, it also supports YOLO format. The labels we used are as follows:

- Player
- Opponent
- Power Up Box
- Power Up Status
- Add On
- Obstacle

C. Training the CNN for Object Detection

Once we obtained enough labelled data, we began training our Object Detection model. We trained a Faster Region Based Convolutional Neural Network with Inception v2 (Faster RCNN) as a feature extractor on 3000 images and tested the network on 1000 images. The training batch size was 1 and we trained the network for 12039 iterations. We initially had a model trained on 3811 iterations, as part of our initial development process. The results of that model were riddled with misclassification and also failed to isolate the background from the foreground and objects like the player and the opponent. This new model, fared much better in testing and overcame the drawbacks of the previous model. We also implemented Non-Max Suppression (NMS) on top of the Object Detection module, to eliminate multiple classes and bounding boxes being assigned for each detected object.

Non-Max Suppression focuses on combining all bounding boxes for a given size or area of the image and returns a single label and bounding box for an object, which has the highest confidence. Faster RCNN is used to place a bounding box on the objects discussed previously. It uses Inception v2, a deep neural network, as a feature extractor. The network automatically selects the most relevant features namely colour, shape, texture, contours, and lines.

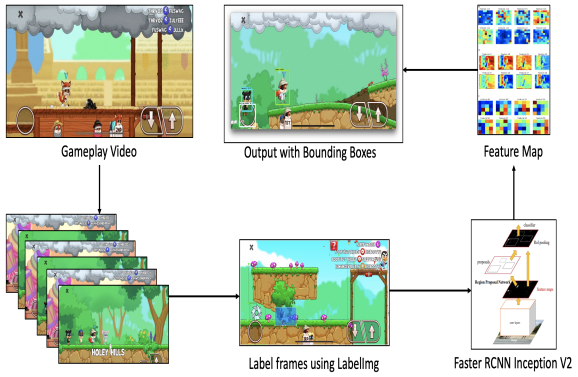

Fig. 3. Object Detection Module

Figure [4] (3811 Iterations) and Figure [5] (12039 Iterations) depict the loss functions and learning rate of the 2 models. Figure [6] indicates the "objectness loss", which helps the model discern the foreground object from the background of each game frame.

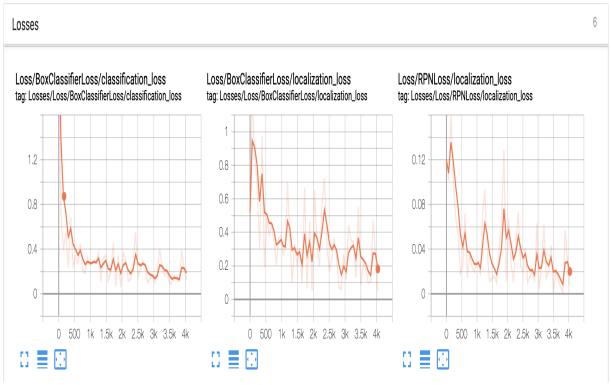

Fig. 4. Loss Functions - 1 (3811 Iterations)

D. The Faster RCNN Architecture

The architecture of Faster R-CNN is complex because it has several moving parts. It all starts with an image, from which we want to obtain:

- A list of bounding boxes
- A label assigned to each bounding box
- A probability for each label and bounding box.

The input images are represented as Height×Width×Depth tensors (multidimensional arrays), which are passed through Inception v2 up until an intermediate layer, ending up with a convolutional feature map. We use this as a feature extractor for the next part.

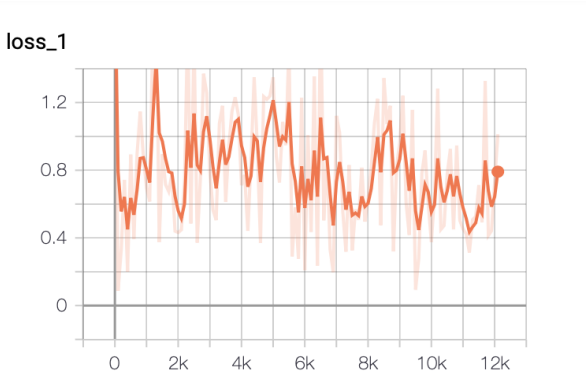

Fig. 5. Loss Functions - 2 (12039 Iterations)

Fig. 6. Loss Functions - 3 (12039 Iterations)

The next part is a Region Proposal Network (RPN). Using the features extracted by Inception v2 we apply Region of Interest (RoI) Pooling and extract those features which would correspond to the relevant objects into a new tensor.

The next is the R-CNN module which uses the previously obtained information to classify the content in the bounding box and adjust the bounding box coordinates.

E. Deep Q Agent

Deep Q Learning is an extension of Q Learning, a reward based method for Reinforcement Learning. In Q Learning, the agent interacts with the environment iteratively, by taking an action. The environment responds by informing the agent of

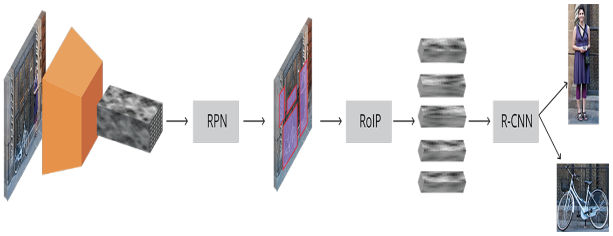

Fig. 7. Faster RCNN Architecture

the reward from that action, and advancing to the next state. This happens continuously until the environment is solved.

The environment for the agent consists of the captured game window from which the state would be defined. The agent will then have to learn a strategy to find an optimal action for that state. The possible actions for the state are:

- Jump
- Slide
- MultiJump
- MultiSlide

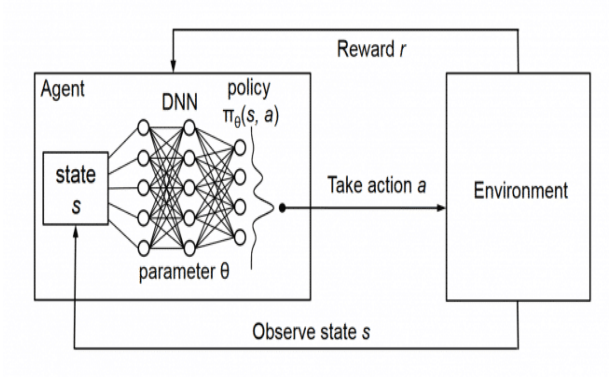

Fig. 8. Deep Q Learning Architecture

Training the Deep Q Network follows the following 4 step framework:

- Estimate the Q-value from the previous state
- Estimate the Q-value of the new state
- Calculate the target Q-value for each action using the known associated rewards
- Train the model with input = (old state) and output = (target Q-value)

We formulated the following Policy for our agent to learn from and generated our own reward system for the agent, based on the agent's rank in the game. The tricky part here was that if the agent got stuck at an obstacle and did not move for more than 15 seconds, the game would automatically end and throw the agent out of the game. Hence, we had to assign a high negative reward for the agent if it failed to complete a race.

- States: Game screenshots processed through the Faster RCNN Inception V2 network. We obtain a (1, 307220) flattened feature map
- Actions: $\langle slide, jump, multiSlide, multiJump \rangle$
- Rewards:
  - Unfinished Race: -10
  - Gain in Rank: +1
  - Loss in Rank: -1
  - FinalReward = (5-FinishRank)*3
- Policy: Two-layered Dense Network that takes feature map as input and predicts total final reward for all 4 actions.

V. Results

For the Object Detection Module, we used Open CV and the Tensorflow Object Detection API, to draw inferences from our pre-trained model. We were able to obtain a highly accurate set of objects being recognised by the model and the model made few to no errors in detecting the player, opponents and various obstacles.

After the addition of Non-Max Suppression, the model also made significant improvements and generated a single bounding box for most objects. Hence, the feature map generated by the Avg.Pool Layer of this model contained all the information we needed to depict a game state for Deep Q Learning.

Figures [9] and [10] show the difference between the predictions of Model 2 before and after implementation of Non-Max Suppression respectively.

Fig. 9. Results before Non-Max Suppression(Left Predictions/Right Ground Truth)

Fig. 10. Results after Non-Max Suppression(Left Predictions/Right Ground Truth)

For the Deep Q Learning Module, we developed two agents. One of the agents was trained to automatically navigate through the game options and begin a race online, with human opponents. This agent was trained for 89 epochs (races or games). Within these 89 epochs, since the agent competed with human opponents online, the agent faces the following challenges:

- The terrain of the race was chosen at random by the human opponents. Hence, since Fun Run 3 has 10 different terrains, it was very difficult for the agent to play in an environment and get prolonged exposure to the environment, in order to lean which action to take

TABLE I
MODEL 1 VS MODEL 2: GAMES COMPLETED

Model	Epochs	Completed	Percentage
Model 1	89	63	70.78
Model 2	50	36	72.00

TABLE II
MODEL 1 VS MODEL 2: FINISHING RANKS

Model	Rank 1	Rank 2	Rank 3	Rank 4
Model 1	4	2	18	39
Model 2	1	2	9	24

- The change in terrains also saw a direct correlation with the increase in the number of games the agent failed to complete, as the agent would get stuck at an obstacle for more than 15 seconds.

For this reason, we decided to train the a new agent, offline, competing with other AI Bots, designed by the game itself. This gave us the freedom to train the agent on one specific terrain and see if this changed the outcome. We trained this agent for 50 epochs.

Table [1] shows the statistics of the number of races completed by each model vs the number of epochs. Table [2] shows a comparison of the ranks at which the model finished these races.

A. Model 1: Results

Model 1 under testing did not do very well, given the dynamic nature of its training. We did foresee the off-chance that the agent would learn only how to jump and never learn how to slide, given the nature of the maps and inherent bias in the game, where the agent would more likely clear an obstacle by jumping rather than sliding. With more training, say around a 1000 epochs (races) the agent should learn when and where to use the other actions like slide in the game and will complete races and finish at higher ranks more frequently.

The positives, however that we could see from Model 1 was that the agent, among the races it completed in testing, did learn to use $\langle multiJump \rangle$ to overcome tall obstacles that needed multiple taps of the jump button. Overall, among the 10 test races, Model 1 only finished 4 of these 10, finishing 4th in all 4 races.

B. Model 2: Results

Model 2 on the other hand completed 7 out of the 10 races in testing. The agent finished 4th in 4 of these races, 3rd in 2 and 2nd in 1. This was mostly due to the random nature of power-ups, but the agent did perform exceedingly well on sliding and jumping to clear obstacles. Although once again, the agent does tend to statistically try to jump more than slide to get past an obstacle on the map. With more training on the same map, the agent would most definitely finish almost all races in testing. The biggest concern here was that since the size of the feature map is too large (1x307220) the processing time of the feature map is too large. This is something we have left as a future scope of this project, which is to use the MobileNet feature extractor and see the difference in performance and processing time.

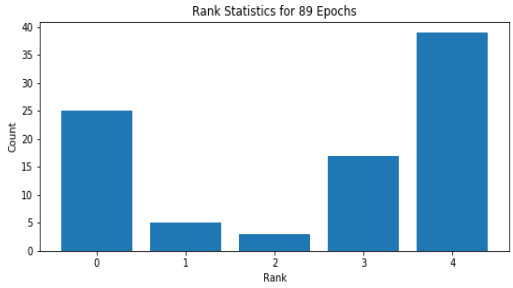

Fig. 11. Model 1 Training Performance

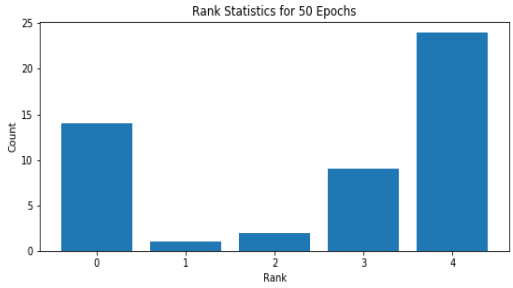

Fig. 12. Model 1 Training Performance

VI. FUTURE WORK

The Future Work of this project consists of efforts to make the agent performance better, not just in one single map or terrain but also across all terrains and levels of the game. The Deep Q Learning Module can benefit from training on multiple 1000s of epochs, will definitely help the agent perform better in the game. The task of the agent learning to use in-game Power-ups optimally, has also been marked as out of the scope of our current efforts, and will be initiated in a future version of this model, where we plan to integrate another neural net to decide when to use a power-up and also modify the reward system to include a reward feedback on clearing each obstacle and obtaining a power-up. Another aspect of this project which we feel might be an interesting application is using Transfer Learning and utilise the learning from the Deep Q Learning Module and having the agent compete in a race, where the last player at each point in 30s of the race is eliminated and see how the agent would perform.

REFERENCES

[1] Y. Liao, K. Yi, and Z. Yang, CS229 Final Report Reinforcement Learning to Play Mario.

[2] J. Lei, M. Zheng, and S. Chen, Using Machine Learning to Play the Game Super Mario Kart, Proceedings on the International Conference on Artificial Intelligence (ICAI), 2019
[3] Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, Playing Atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602 (2013).
[4] Yang Z., Zhang YJ., Rehman S.., Huang Y. (2017) Image Captioning with Object Detection and Localization. In: Zhao Y., Kong X., Taubman D. (eds) Image and Graphics. ICIG 2017.