Reinforcement Learning based 2^{nd} player for Atari Pong: An Attempt

Rutvik Vijay Patel
Department of Computer Science
The University of Western Ontario
London, Canada
rpate492@uwo.ca

***Abstract*—Reinforcement Learning is slowly but steadily becoming one of the state-of-the-art Machine learning techniques, greatly expanding the domain where ML can be used. This report talks about an attempt to create a Reinforcement Learning (Q-learning) based 2^{nd} player to replace the hard-coded game AI in Atari Pong. The Neural network model used here is a Convoluitional Neural Network which uses game state and feedback information available in the game environment. The input to the network is an image of resolution 100×100 pixels and previous action predicted by the model (feedback). The performance of two similar loss functions is discussed along with the techniques used.**

***Index Terms*—Machine Learning, Reinforcement Learning. Deep Q-learning. Atari Pong. Convoluitional Neural Network. Non-Player Character.**

I. Introduction

One of the long-standing problems in the field of Machine Learning and Artificial Intelligence is using image, video or any sensory data to control an agent and take actions based on rewards from previous actions. Reinforcement Learning (RL) is a relatively new domain of Artificial Intelligence, when compared with the likes of Supervised Learning or Clustering. Reinforcement Learning has particularly become popular after the advent of self-driving cars and the always increasing interest in Robotics.

Advancements in Convolutional Neural Networks over the past decade has made it easier and feasible to get closer to approach this problem. Reinforcement Learning based agents trained with visual and/or auditory inputs, can contribute to the development of better Non-Player Characters (NPCs) in open world games and online multi-player games. This know-how can also be used for the development of self-driving vehicles, pharmaceuticals, nuclear reactions, or any other application that can benefit from Reinforcement Learning. Further, its indirect application and the benefits it can bring to other industries can be compared to the advancements in Game Engine and how it greatly helped with movie graphics.

Even with the recent developments, commercially deployable Reinforcement Learning based Non-Player Characters which learn completely from sensory inputs like image, video or sound, has been a fantasy, with only futuristic movies made on this topic. One of the major challenges to the use of Reinforcement Learning algorithms include the lack of labelled data and the use of reward signal with is often noisy. Further, there is a need to account for previous actions and rewards while determining a future action. Also, most deep learning models assume that the data samples are independent and there should not be a significant effect on the training outcome based on the order in which the training samples are passed. The Data distribution is also expected to change with training as the model tries to learn the behaviour.

This report demonstrates and talks about an attempt to create a version of Atari Pong from scratch, and use Convolutional neural networks, Multi-Layered Perceptrons (MLP), and various relevant techniques. This network considers the difference between current and previous frames and previous action to predict the current action, and reward from the current action for training. The training model shares its weights with target model, which is also used to generate the training data. The weights are updated with stochastic gradient descent policy. Goal of this project is to demonstrate the use of Deep Reinforcement Learning and attempt creating an agent capable of predicting actions to maximize future rewards. Atari Pong was considered here due to its simplicity, compared to other Atari 2600 games.

The two main approaches of Reinforcement Learning are model-based and model free. Here, model-free or model based do not refer to a neural network model. Rather, model-based implies that the learning agent has information or resources to ask the ”model” for next reward or a distribution curve to help the learning with predictions. The ”model” can be aware about the rules of the task or game.

Algorithms which consider only experience, like in the case of State-Action-Reward-State-Action (SARSA), Q-learning or Monte-Carlo Control are considered ”model-free”. These approaches do not have access to a ”model” or external information which might facilitate or help them with predictions.

In this project, a model-free Reinforcement Learning approach which is based on Q-learning is used. The term ”model” used hereafter in this report refers to a Deep Neural

Network.

The main objective of this project is to attempt to create and learn about the use of Reinforcement learning algorithms to create an adaptive Non-Player Character (NPC) or 2_{nd} player.

II. Related Work

One of the most popular Reinforcement learning projects of the last decade is perhaps Deepmind's AlphaGo, which defeated a Go professional, and can be regarded as being ahead of its time. Further development of this project lead to AlphaZero capable of teaching itself board games like Go and chess, without any data on human gameplay. The next generation of the program called MuZero is capable of learning the gameplay without any prior information on its rules.

Another popular example of Reinforcement Learning in a board game include TD-gammon. It was a RL based program which learned to play backgammon, entirely by learning through self-play. It achieved human-level playing capabilities, which was very significant at the time as this research was done during 1990s. TD-gammon used a single hidden layer Multi-layered perceptron network which approximated the value function, like Q-learning. Later attempts to use a similar method with games with Chess or Go were not successful, due to huge state space and lack of an element which seemed to introduce stochasticity, thus enabling the algorithm to explore more states in the state space.

Since 1992, when Q-learning was first introduced by Chris Watkins by providing the first convergence proof of the same. There has been a lot of research and experiments based on this algorithm in the last decades. Q-learning is based on the Bellman equation, which breaks a larger problem into a series of sub-problems based on Bellman's "principle of optimality". This equation is used to update values Q-values in Q-table.

Some of the most popular papers about Arcade games, specifically Atari games being played by a Reinforcement Learning Agent were published by Deep Mind in early and mid 2010s. These papers brought Deep Q-learning and related approaches to spotlight and there have been many significant advances in this field. Atari games and their hard-coded AIs were considered state-of-the-art at the time of their release and years after that. Between 1970 and 1990, Atari games were the most popular games in the market and "Arcade" became a part of the culture in that era. These games were played on computers or Arcade devices with only a few KBs or RAM and a limited display resolution. But, early attempts to play Atari 2600 games using a RL based bots were not successful. But, this started changing with advancements in the field and the use of a Deep Q-learning algorithm by Deep Mind to demonstrate playing Atari 2600 games with a Reinforcement learning Agent.

There are countless articles and YouTube videos available which talk about Reinforcement Learning and Q-learning. Many such articles and videos were reviewed for the purposes of this project. Many of the videos referred talk about creating an agent play pong, against the hard-coded game AI. Most of them used OpenAI gym library to train and interface with the model. One of the approaches used included NEAT (NeuroEvolution of Augmenting Topologies) algorithm. It was first mentioned in a paper published in 2002. It was inspired by the evolution characteristic of biological life. Similar to biological evolution, the network starts with a simple model which grows more complex with training and generations.

III. Game Background and Introduction to the Approach used

The game used for this project was written using pygame library due to the unavailability of the feature in OpenAI gym to replace the built-in hard-coded 2^{nd} player game AI with a user defined version of the same. This version of Pong was created considering the OpenAI gym version and designed to follow similar game rules. Some rules were modified to make the training possible, with limited computing power and time resources, and to prevent infinite loops.

The approach considered here is based on a combination of Convolutional neural Network based models and Q-learning. The approach used in this project does not use a specific Q-table or a Q-function. Instead, the Q-table is replaced by a Deep Neural Network, which predicts the future action based on current state and previous action. The Deep Q-Learning based method is used for the purposes of experiments in this project. The approach is discussed further in this report and mentioned in Fig. 6.

A. Modified version of the game

The modifications include, setting the $y - velocity$ of the ball to be non-zero and prevent the game from getting stuck in an infinite loop where the ball bounces with $y - velocity$ 0, from both the paddles and there is no increment in any of the player's score, and thus, creating an infinite loop. In case, the ball's $y - velocity$ is 0 at any time, the ball is initialized again from the centre of the screen, with a random $y-velocity$ and the same $x-velocity$.

The Atari Pong game involves two paddles (P_1 and P_2) and a ball. The paddles are places at the opposite ends of the screen, usually right (P_1) and left (P_2). Both the paddles can be allowed 3 actions: up, down or stay. The ball bounces at an angle based on the distance between the center of the paddle and the ball's point of contact with the paddle. The ball bounces off all the other edges by simply inverting the x

or y velocities respectively. If the ball hits P_1 but not P_2, and instead hits the left edge, P_1 scores 1 $point$ and vice-versa for P_2. With regards to the game used in this project, the player to score more than 20 first wins. When a player wins, the game is reset.

B. Computing-power related modifications

Considering the limited computing-power available on a personal computer and the display related requirements of PyGame, the model had to be trained on a local CPU. This type of problem can be said to have virtually infinite sate-space due to the number of positions of paddles and ball locations possible, which can be very large. The paddle locations in horizontal direction was restricted to a small number, in this case, 5. Further, ball speed is increased to decrease number of possible states. These modifications help limit the state-space in the game. Further, classic game graphics like a dotted line in the middle of the window, individual player scores within the game window, colored elements and effects, etc. were not used to limit the frame processing required. The display size used here is 600×600 pixels and the paddle velocity is set to 120 pixels per step and ball velocity to 20 pixels per step.

C. Hard-coded game Artificial Intelligence

The hard-coded agent which is one of the players (can be P_2 in our case) tracks the position of the ball and P_2 is moved accordingly. To make it possible for the other player, P_1 to win, the P_2 agent is limited to be able to look at the ball only if it is in 80% of the window width, from the right edge.

IV. Deep Learning Model Architecture and Experiments

A. Game Environment

For the given pong game environment E_{pong}, the following information is made available to the agent, while training

1) **Current game state**, the difference between current and previous game frames
2) **Action** taken by the agent in the previous state
3) **Reward** from action taken in the current state

The following information is made available while predicting next action

1) **Current game state**, the difference between current and previous game frames
2) **Action** taken by the agent in the previous state

The state information can be defined as following

- The game episode generates n frames $F = \{f_0, f_1, f_2, f_3, ..., f_{n-1}\}$
- The states of the game are defined as $S = \{s_1, s_2, s_3, ..., s_{n-1}\}$, where s_n is the difference between f_n and f_{n-1}
- The game data includes $n-1$ states, $n-1$ rewards and $n-1$ actions performed in each game episode.
  $S = \{f_1 - f_0, f_2 - f_1, f_3 - f_2, ..., f_{n-1} - f_{n-2}\}$,
  $R = \{r_1, r_2, r_3, ..., r_{n-1}\}$,
  $A = \{a_1, a_2, a_3, ..., a_{n-1}\}$

1) Game frame and state images: Some examples of image frames in the game are shown in Fig. 1. The resized and normalized difference image between the n^{th} and $(n-1)^{th}$ frame is considered as an input to our model. The difference images or state images are shown in Fig. 2.

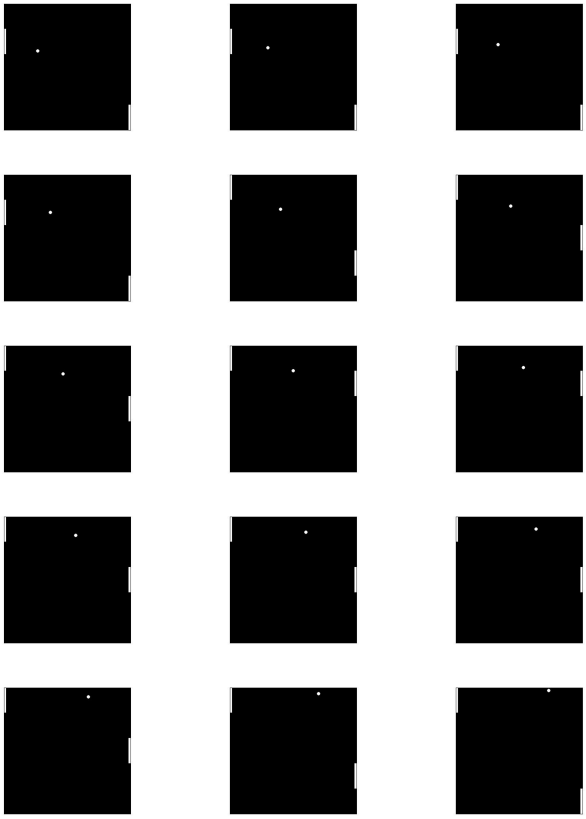

Fig. 1: Game Frames

It can be seen that since the difference is considered, the paddle location is not available to the agent or model if the position of that particular paddle is the same in both f_n and f_{n-1}. This limits the information available to the the learning agent. Also, because of the image difference, the ball movement direction becomes available to the agent. In case there is no paddle movement between two frames and the ball hits one of the paddles, there is a possibility that no information is available in the state image. Similarly, it is also possible that all 3 objects, ball and the two paddles

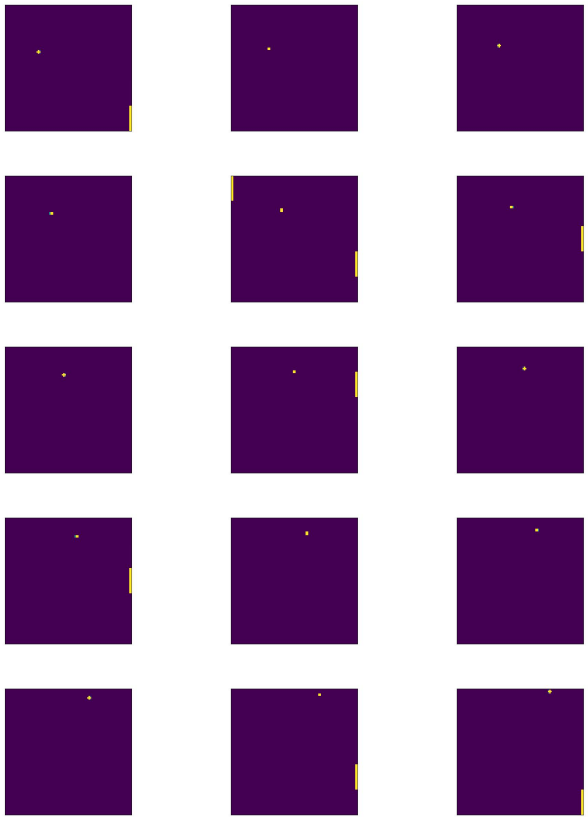

Fig. 2: Game States

make a movement and are visible in the state image.

2) Rewards: Rewards from previous states are considered to calculate loss, with a custom loss function.

The reward for a specific game frame is based on the interaction between the ball and the paddles. Say, the right paddle (P_1), the Reinforcement Learning based Agent, is placed such that the ball is able to bounce off the paddle and reach the other end of the game frame, in the opposite $x - direction$. Here, the hard-coded agent (left paddle, P_2), which has access to ball-coordinates if the x-coordinate of the ball is below a certain threshold, for the scope of this project, this threshold is set to 0.8 times the window width, as mentioned earlier in this document. The raw rewards or score are assigned based on the following factors:

1) In the event that P_2 is unable to be placed such that the ball bounces off P_2, P_1 is awarded 1 point and the reward for that frame is set as $+1$
2) In the event that P_1 is unable to be placed such that the ball bounces off P_1, P_2 is awarded 1 point and the reward for that frame is set as -1
3) In case that the ball (which bounced off the right edge of the game window and was missed by P_2) is unable to bounce off P_1, but rather bounces off the left edge, P_1 is awarded -1 reward and an increment is made in P_2 total score
4) In all other cases like the ball bouncing off both P_1 and P_2 in any order, or bouncing off the top and bottom edges of the frame, the reward is 0

Rewards are processed after completion of each episode. In case the reward is non-zero, a decaying reward value is propagated to previous array indices to replace zero reward values in the reward array. The decay rate is set to 0.9, in this case. Later, the rewards array is normalized using mean and standard deviation, and the resulting processed rewards array is used to calculate loss.

Decaying rewards help with propagating the $-ve$ or $+ve$ result of an action to preceding actions or frames, since it is a sequence of previous actions that determine the present reward, and to weight the current result on previous actions reward propagation is used. Further, to exaggerate the propagated rewards, the array is normalized. Hereafter, $rewards_{normalized}$ is refereed to as $reward$ or r

The rewards are propagated using Algorithm 1, and later normalized.

Algorithm 1 Reward Propagation

```
const decay ← 0.9
t ← 0
i ← length(rewards)
for i ≥ 0 do
    if rewards[i] = 0 then
        t ← t × decay
    else
        t ← rewards[i]
    end if
    rewards_processed[i] ← t
    i ← i − 1
end for
```

An example of rewards (raw values, propagated and normalized) from one of the episodes in shown in Fig. 3.

3) Actions allowed: There are three legal paddle actions allowed by the agent $A_{allowed} = \{\{0, 1, 2\}$, where 0, 1 and 2 correspond to "up", "no movement" and "down" respectively. The paddles are limited to the screen by restricting the movement beyond the screen in the vertical direction.

B. Information available to the Learning Agent and Challenges involved

The agent interacts with the given environment E_{pong}, with the limited information available, and for each time-step predicts or selects an action a_t, at time t, from the set of legal actions Allowed. This action value is passed to a method in the game class, to update game state and generate reward value. Information like paddle or ball co-ordinates and speed is not observed by the agent. It can be said that E_{pong} can be stochastic. The data available to the agent does not include game rules or any other information other than state, reward

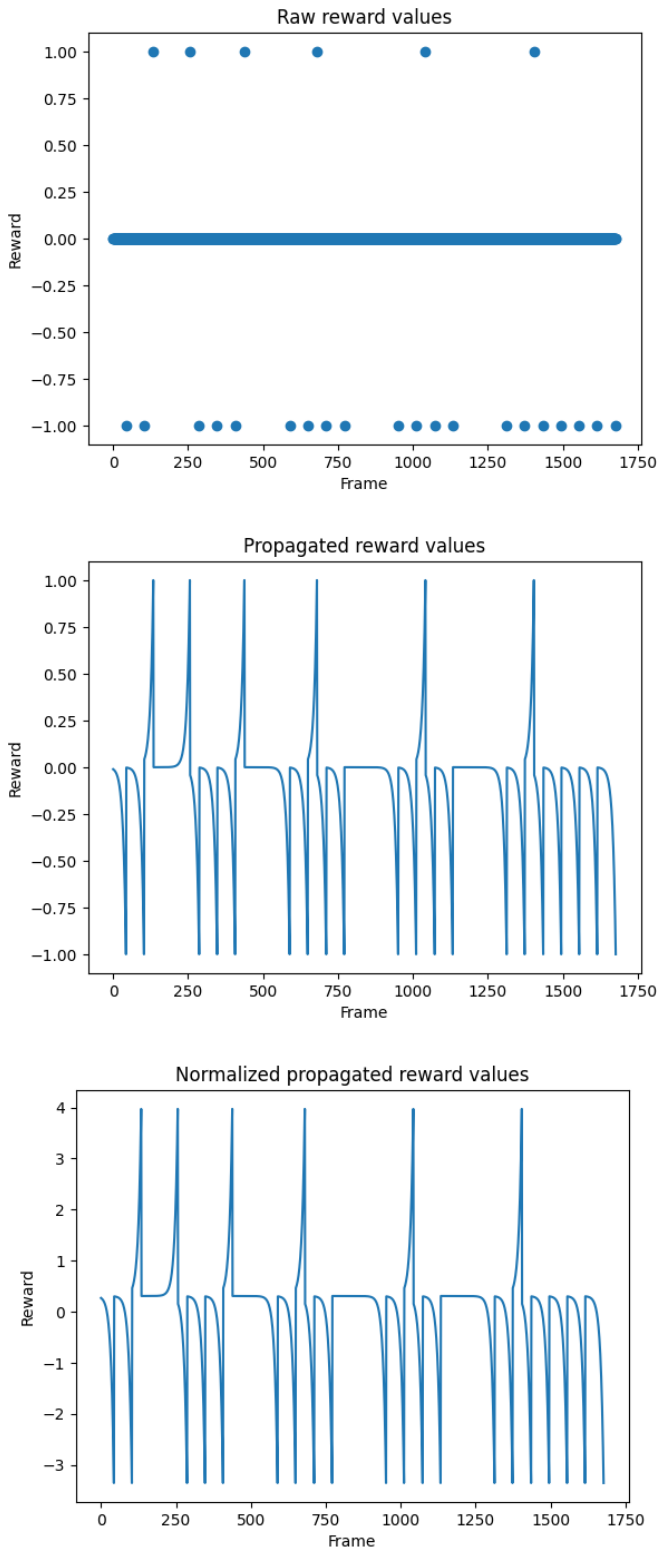

Fig. 3: Reward Array: (i) Raw Reward values, (ii) Propagated Reward values, (iii) Processed Propagated Reward values

and previous action feedback. It can be said that the reward or feedback about current action prediction would be available only after hundreds, if not thousands, of game steps or frame updates. This makes it difficult for the agent to "realise" the true consequences or effects of its actions, whether single or particular. In addition to this, due to the nature of this type of problem, time is also a factor in the problem, classification problems where Convolutional neural Networks are generally used. Therefore, the model has to incorporate or learn the behave to consider an additional dimension, which is the order of occurrence of the game state in this case.

Due to the limited information available to the agent, the task is not completely observable and it is possible that multiple states can be aliased or indistinguishable after the state image is processed and is propagated through the network. This issue can be said to be analogous to n-body problem on physics, where the final solution can greatly depend on the initial state of the agent. Since the action is predicted based on probability likelihood instead of selecting the action with maximum probability, here $numpy.random.choice()$ is used to predict the action and maintain a stochastic environment. This helps with increasing the observation space available to the agent for training. Also, this contributes to helping avoid an n-body problem like situation, so that over a number of episodes, the results are similar irrespective of the initial random state.

The primary goal of the agent is to maximize rewards, which is done by minimizing loss value and selecting a larger proportion of "correct" actions.

C. Experiments with Loss functions

The Loss function required for this type of problem needs to accommodate $reward_{normalized}$ as well as the difference in $y_{predicted}$ and y_{true}. Initial experimentation involved $sigmoid$ activation in the output layer and appropriate loss function l_0, and $softmax$ activation with l_1 and l_2. The addition of a constant in l_0, "1" in this case, a higher conceptual complexity, in combination with no improvement in model training over l_1 and l_2, were the reasons for not considering l_0 for further experimentation .

$$sigmoid(x_i) = \frac{1}{1 + e^{-(x_i)}} \tag{1}$$

$$softmax(x_i) = \frac{e^(x_i)}{\sum_j e^(x_j)} \tag{2}$$

$$l_0(y_p, y_t, r) = sum(SE(y_p, y_t) \times (1 + SE(M(y_p, y_t), r))) \tag{3}$$

$$M(y_p, y_t) = \begin{cases} 1, & if\ max(y_p) = y_t \\ 0, & otherwise \end{cases} \tag{4}$$

$$SE(x_1, x_2) = (x_1 - x_2)^2 \tag{5}$$

Considering that a learning agent of this kind works based on probability distribution or likelihood, l_1 and l_2 are considered. The initial loss function used with this strategy,

l_1 consists of a negative product of $Cross - EntropyLoss$ between $y_{predicted}$ and y_{true}, and $reward$.

In later experimentation, it was found that a loss function l_2 which considers only $Cross - entropyloss$ and not its product with $reward$, where $reward$ values are near 0, seemed to make it more stochastic at an earlier stage in training. This can be seen in the loss function plots comparing l_1 and l_2.

The loss function l_2 used in this project does not consider reward values where $reward \in (-1, 1)$. Considering the more stochastic nature of l_2, this loss function is considered for training.
Cross Entropy loss is defined as:

$$CE = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c}) \tag{6}$$

Here,
M - number of classes
log - the natural log
y - binary indicator (0 or 1) if class label c is the correct classification for observation o
p - predicted probability observation o is of class c

$$l_1(y_p, y_t, r) = -1 \times CE(y_p, y_t) \times r \tag{7}$$

$$l_2(y_p, y_t, r) = \begin{cases} CE(y_p, y_t), & if\ r \in (-1, 1) \\ -r \times CE(y_p, y_t), & otherwise \end{cases} \tag{8}$$

Here, y_p, y_t and r correspond to $y_{predicted}$, y_{true} and $reward$ arrays respectively. CE is the cross-entropy loss function defined in Keras library, it takes $y_{predicted}$, y_{true} as its parameters. The custom loss functions, l_1 and l_2, were used for training for 250 episodes each and the mean-centered moving average (window size 5) of plots for cumulative loss in each episode of training is mentioned in Fig. 4. Towards the end of limited training, moving averages for both the loss functions seem to correlate.

The custom loss function is based on the thought that if reward value is greater than 0, the effective loss value should be low. Similarly, if reward value is less than 0, the effective loss value should be high. If the reward value is in the range $(-1, 1)$, only cross entropy loss is used, without multiplying it with the reward value for that action-state.

D. Model

Since, the model is trained on a local computer, the pre-processing steps involve resizing the input image to shape 100×100, this is in addition to reducing the image to Grayscale.

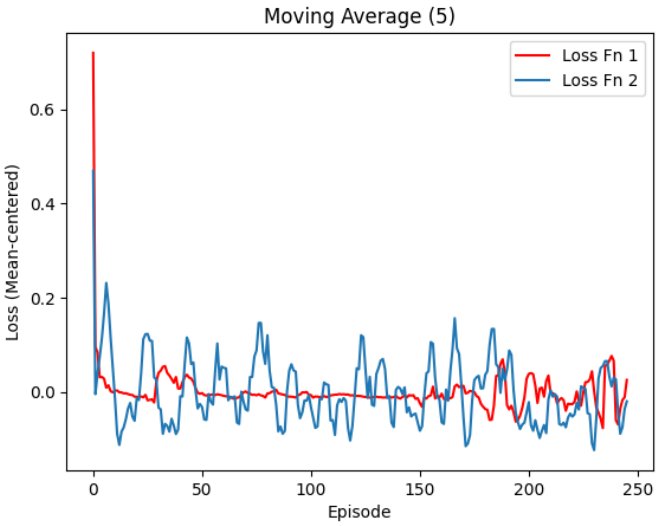

Fig. 4: l_1 and l_2 losses for 250 training game episodes

The model consists of Convolutional layers, with ReLU activation, Max Pooling and Batch Normalization. The model takes two inputs, the game state (difference between game frame n and frame $n-1$) and previous action. First, features are extracted from the game state image with shape $(100, 100, 1)$, with an array of Convolution, MaxPooling and Batch Normalization layers. The first layer convolves 16 filters with kernel size 3×3 and stride 1. After the ReLU activation is applied, MaxPooling (2×2), and Batch Normalization is used. The second layer convolves 32 filters with kernel size 3×3 and stride 1. Similar to the first layer, ReLU activation is applied, MaxPooling (2×2), and Batch Normalization is used. The third layer only consists of a 2D Convolution layer with 64 filters as output, and kernel size 3×3 and stride 1. The fourth and last convolution layer convolves 64 filters with kernel size 3×3 and stride 1. Followed by ReLU activation, MaxPooling (2×2), and Batch Normalization.

The output from the Batch Normalization Layer is "Flattened" into a 1-Dimensional vector and the 1-Dimensional "previous action" input vector with 3 rows is concatenated. The new concatenated vector is fed to the succeeding fully connected layers. The first fully connected hidden layer consists of of 8192 units with ReLU activation. The consecutive second layer contains 2048 units with ReLU activation. Dropout normalization is applied after this layer to help avoid overfitting, with dropout rate 0.2. The third hidden layer consists of 256 hidden layers with ReLU activation. The final layer of the model consists of 3 units, since there are three possible legal actions, $A_{allowed}$. Softmax Activation is used in the output layer. The model architecture is graphically described in Fig. 5.

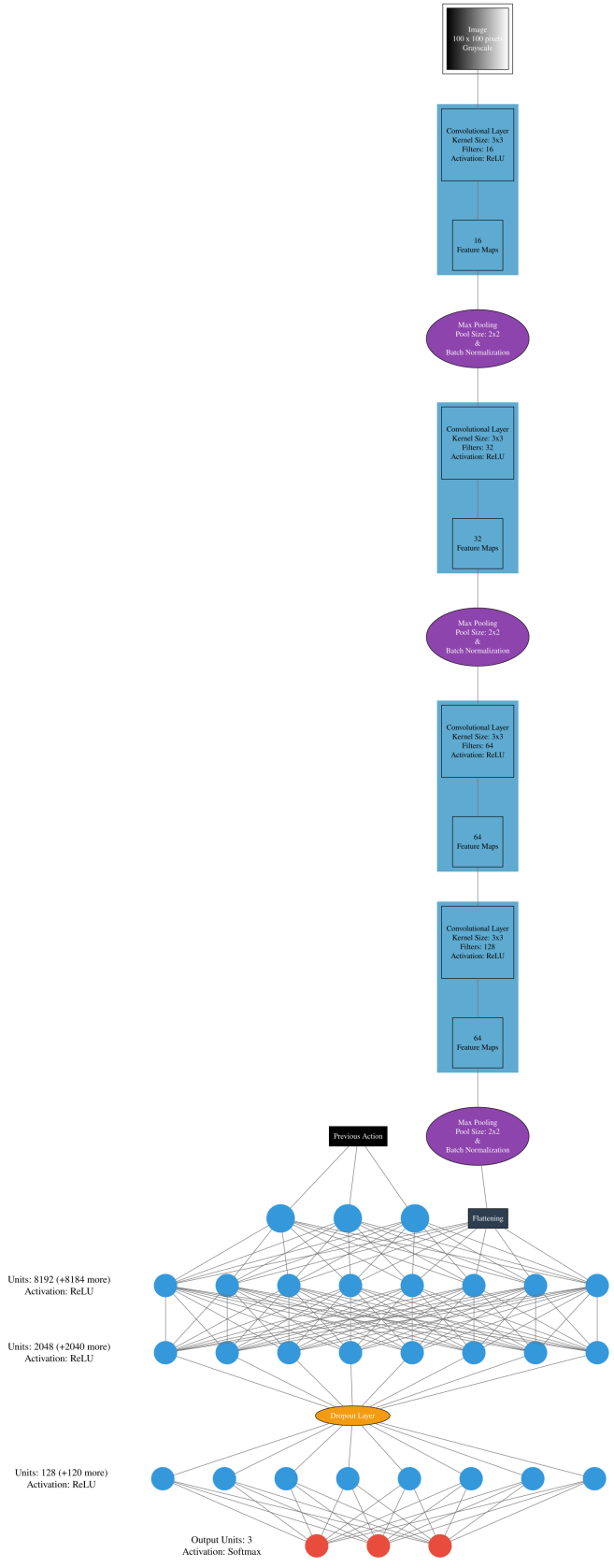

Fig. 5: Model Architecture

A total of two models, $target - net$ and $policy - net$ are used for training and experimentation. $target - net$ is used to predict actions while training and $policy - net$ is used for is used for training. The two models share all model parameters including weights. Since, $target_net$ is not used to fit the model, it is not compiled. Whereas, $policy - net$ is compiled because it is used for model fitting and updating weights. The custom loss function l_2 is used to compile $policy - net$ model. For optimization, Adam optimizer is used for $policy - net$.

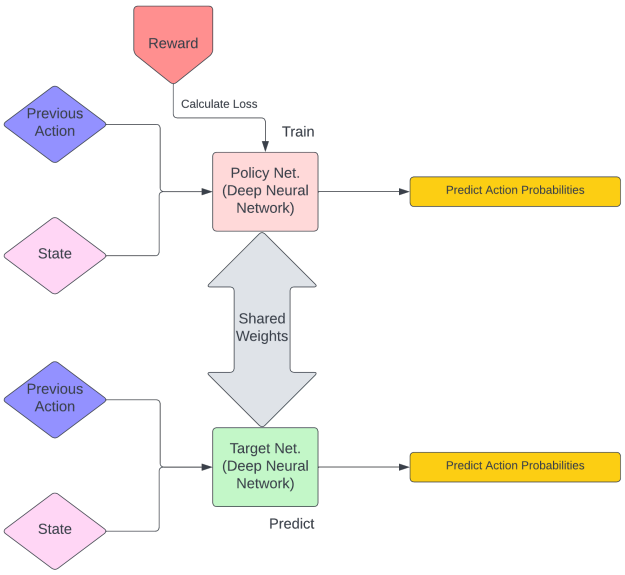

Fig. 6: Deep Q-Learning based Approach used in this project

Due to the nature of the problem, each episode can have a different number of states. This depends on the number of frames it takes for either P_1 or P_2 to achieve a score of 20. A batch size of 64 is used for fitting or training the model. For each episode, game information like game frame, reward, action, total scores is stores in a list, to be further used for training and visualization. The model is trained once an episode is complete, frames are processed, and game states, rewards and action list is available.

During the episode generation process, it can happen that the game is stuck in an infinite loop, i.e., P_1 and P_2 play against each other such that the ball bounces off both the paddles one after the other, usually reflecting through either top or bottom edge of the game window. A very frequent case of this type of problem which involved ball travelling with $0y - velocity$ hitting both the paddles in a straight line, was avoided by conditioning the ball's $y - velocity$ to be non-zero. In case, the ball's $y - velocity$ turns 0, the ball is "thrown" from the center with a random $y - velocity$ in the same $x - direction$, as mentioned earlier in this report. To prevent such loops, a threshold of 5000 frames is used to terminate the game if either P_1 or P_2 is unable to win. During experimentation with l_1 and l_2 it was found that episodes where the game had to be terminated due to an

infinite loop occurred more frequently with model using l_1 loss as compared to when using l_2.

E. Training and Results

To enable the algorithm for larger state-space exploration, the Epsilon-Greedy Algorithm was used (Algorithm 2). This strategy results in a random action to be taken when $\epsilon_{threshold}$ value is less than a generated random number. This strategy is based on the idea that if the output of some of the initial frames is skewed towards one specific class of output, the model would consider it to be correct and keep selecting the same class label after training, since the original training data was skewed. The model fails to explore other game-states or generate enough examples of other class labels. To prevent this type of scenario Epsilon-Greedy strategy can be helpful.

Algorithm 2 Epsilon-Greedy Algorithm

$const\ \epsilon_{start} \leftarrow 0.95$
$const\ \epsilon_{end} \leftarrow 0.05$
$const\ \epsilon_{decay} \leftarrow 500$
$s \leftarrow 0$
for each function call **do**
  $sample \leftarrow random(0, 1)$
  $\epsilon_{threshold} \leftarrow \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) * e^{\frac{-1 \times s}{\epsilon_{decay}}}$
  if $\epsilon_{threshold} < sample$ **then**
    return $exploit(params)$
  else
    return $explore(params)$
  end if
  $s \leftarrow s + 1$
end for

Here, the function $exploit(params)$, is used to predict an action based on current game state and previous action. Whereas, the function $explore(params)$ predicts a random action to be taken. $params$ are the required function parameters. For the purposes of this project, the output of $exploit$ and $explore$ functions is a 1-Dimensional row vector with 3 columns.

The training for the two models, $target - net$ and $policy - net$ is done for 250 episodes of the game. Each episode contain a varying number of frames, as the game is terminated when one of the players P_1 or P_2, achieve a score of 20. Here, an increment in the total score of Learning Agent is equivalent to a $+1$ raw reward value. Similarly, an increment in total score of Hard-coded agent is equivalent to -1 raw reward value. The average number of frames in each episode, in the case of both l_1 and l_2 is 1500. The standard deviation in number of frames per episode is 380 and 255 for l_1 and l_2 respectively. The total score for Hard-coded Agent is The plot for number of frames per episode or episode duration is mentioned in Fig. 7.

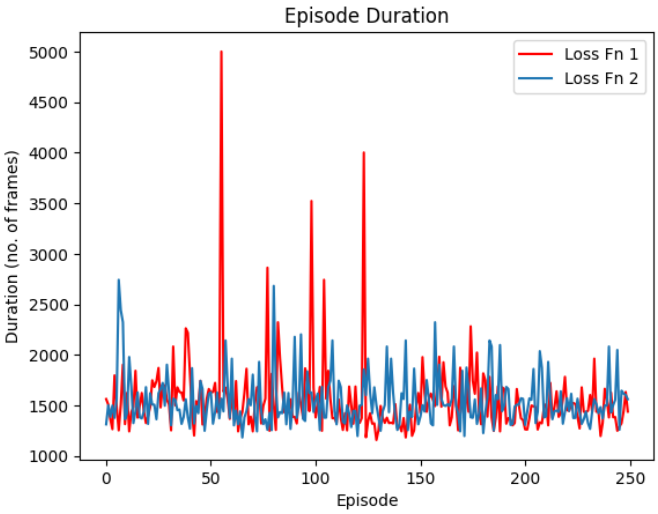

Fig. 7: Number of frames per episode for l_1 and l_2 losses, for 250 episodes

It can be seen in the plots in Fig. 7 that while using l_1 there are larger spikes in the plot for l_1 loss, this shows that the agent is more likely to be trapped in an infinite loop while using l_1. Also, during experimentation it was found that the agent was caught in an infinite loop while using l_2, negligible number of times. The highest number of frames in an episode while using l_1 loss is 5000, which indicates a virtually infinite loop, whereas for l_2 loss this value was 2742.

The total score of the Reinforcement Learning Agent can be used as a metric to check the performance of our approach and l_1 and l_2 loss functions. The plot between Moving Averages (window size 50) of Agent (P_1) score for both l_1 and l_2 loss is shown in Fig. 8.

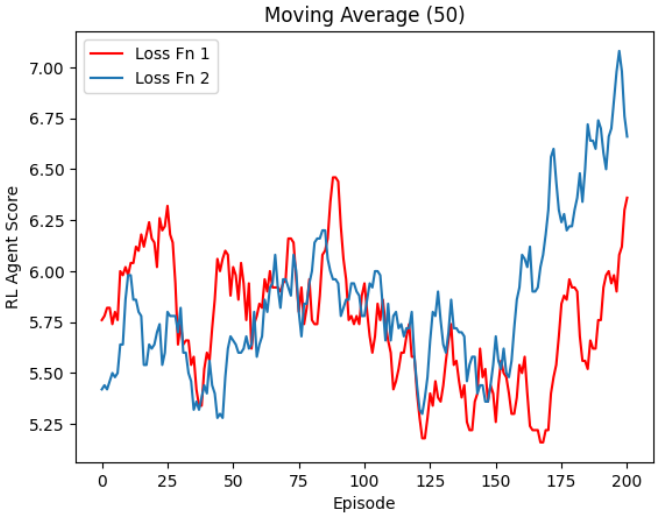

Fig. 8: MA-50 of Agent Scores for 250 episodes

It can be seen in Fig. 8 that when using loss function l_1 the plot, eventually has lower lows for more than a third of total training. Towards the end of training, the plot has higher

lows which might indicate improving performance. In case of l_2, the lowest score (average) occurs near the beginning of training, with score improving near the end of the training. The plot for l_2 seem to make a double bottom like chart pattern, indicating improving performance.

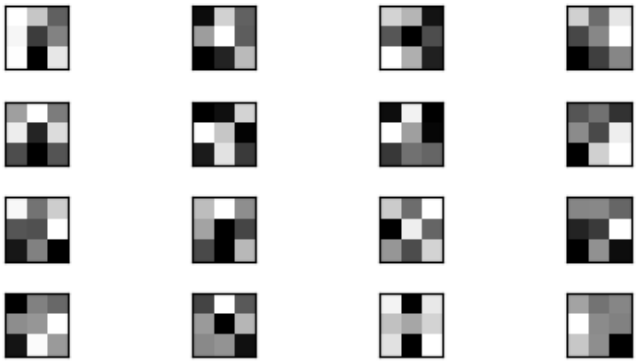

Fig. 9: A Visualization of the First Convolution Layer Filters, after training

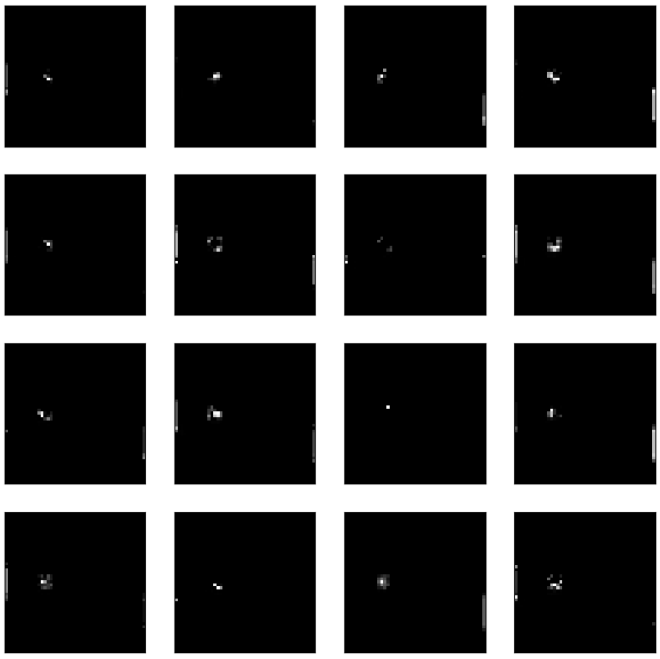

Fig. 10: A Visualization of Feature Maps for a sample game state, after model training, Convolution Layer 2

F. More about the Game

After a model is trained, the Python script asks the user whether they would like to Play the game. There are 3 choices available to the user

1) Player vs. Reinforcement Learning Agent
2) Player vs. Hard-coded Agent
3) Hard-coded Agent vs. Reinforcement Learning Agent

The user can test the original game used in this project by playing against the hard-coded agent or play against the Reinforcement Learning agent which was trained earlier. The user can also watch the recently trained RL agent play against the hard-coded AI.

V. Discussion and Future Direction

Considering the modern computing capabilities and advancements in Quantum computing, and many other machine learning techniques, it is evident that machine learning of all types, especially reinforcement learning has a long way to go. It has surpassed human capabilities in some specific tasks but there is yet to be an Artificial General Intelligence (AGI) capable of learning any task a human is capable of. There are dozens of AGI projects spread around the world as of today but with a long way to go.

This project demonstrates the use of Artificial Neural Networks with Reinforcement Learning techniques which, with sufficient training and optimization can replace a hard-coded game AI. Although, a Neural Network might have significantly higher computing requirements, the adaptive and less predictable experience it has to offer to the user has the potential to significantly improve the user's overall experience with the game. This can be a significant business opportunity for the Entertainment industry considering the growth in work-from-home culture seen in the post-COVID world.

The performance of the model used in this project can be improved by increased complexity of the neural network, hyper-parameter tuning, use of other reinforcement learning techniques and importantly, more training episodes. Also, additional features can be added to the game state image, like paddle being always always visible to the agent. Also, a better visual indicator of the ball movement direction can be added in the game state. Furthermore, a larger colour gamut, like using two 3 color channels instead of 1, with ball and paddle color representing the speed and direction of movement could be helpful.

VI. Conclusion

To conclude, I would say that there were various different techniques and platforms considered for this project and it was a great learning experience. Although, the objective of creating an AI which can replace a hard-coded AI, was not achieved due to time and computational limitations, some progress was made.

The Reinforcement Learning Agent was able to achieve a 50 game average score of 7 against the hard-coded agent score of 20, which would indicate that the agent was able to converge to the required behaviour to some extent. In future, this project can be extended for use with other Arcade games.

VII. Acknowledgement

I would like to take this opportunity to appreciate all the help and supervision I have been provided by Dr. Katchabaw. Further, I am thankful to all the Professors and Course Instructors who have been a part of my academic journey, and content creators on the internet for their time and help.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. (2015). Playing Atari with Deep Reinforcement Learning. DeepMind Research. doi:https://arxiv.org/pdf/1312.5602.pdf

[2] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., Silver, D. (2019). Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. DeepMind Research. doi:https://arxiv.org/abs/1911.08265

[3] Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. ACM, March 1995 / Vol. 38, No. 3.

[4] Arvidsson, O., Wallgren, L. (2010). Q-Learning for a Simple Board Game. School of Computer Science and Communication, Royal Institute of Technology, Stockholm.

[5] Paszke, A. Reinforcement Learning (DQN) Tutorial. PyTorch tutorials. URL:https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[6] Simonini, T. (2018). An Introduction to Reinforcement Learning. URL:https://www.freecodecamp.org/news/an-introduction-to-reinforcement-learning-4339519de419/

[7] Heidenreich, H. (2019). NEAT: An Awesome Approach to NeuroEvolution. Towards Data Science. URL:https://towardsdatascience.com/neat-an-awesome-approach-to-neuroevolution-3eca5cc7930f

[8] Wikipedia. Q-Learning. URL:https://en.wikipedia.org/wiki/Q-learning

[9] Wikipedia. Bellman Equation. URL:https://en.wikipedia.org/wiki/Bellman_equation

[10] Raval, S. (2016). Pong Neural Network (Live). YouTube. URL:https://youtu.be/Hqf__FlRIzg

[11] Tech with Tim (2022). Python Pong AI Tutorial - Using NEAT. YouTube. URL:https://youtu.be/2f6TmKm7yx0

[12] Tech with Tim (2022). Make Pong With Python!. YouTube. URL:https://youtu.be/vVGTZlnnX3U

[13] Hansha, O. (2019). Atari Pong AI via Reinforcement Learning. URL:https://ozaner.github.io/hw/Spring%202019/Ghost%20in%20Machine/pongpaper/pongpaper.pdf

[14] Trazzi, M. (2018). Spinning Up a Pong AI With Deep Reinforcement Learning. Floydhub. URL:https://blog.floydhub.com/spinning-up-with-deep-reinforcement-learning/

[15] Karpathy, A. (2016). Deep Reinforcement Learning: Pong from Pixels. Github blog. URL:http://karpathy.github.io/2016/05/31/rl/

[16] Wan, A. (2019). Bias-Variance for Deep Reinforcement Learning: How To Build a Bot for Atari with OpenAI Gym. DigitalOcean, UC Berkley. URL:https://www.digitalocean.com/community/tutorials/how-to-build-atari-bot-with-openai-gym

[17] Parmelee, D. (2021). Getting an AI to play atari Pong, with deep reinforcement learning. Towards Data Science. URL:https://towardsdatascience.com/getting-an-ai-to-play-atari-pong-with-deep-reinforcement-learning-47b0c56e78ae

[18] freeCodeCamp.org (2022). Python + PyTorch + Pygame Reinforcement Learning – Train an AI to Play Snake. YouTube. URL:https://youtu.be/L8ypSXwyBds

[19] Tsou, C. (2021). Reinforcement Learning: Deep Q-Learning with Atari games. Towards Data Science. URL:https://medium.com/nerd-for-tech/reinforcement-learning-deep-q-learning-with-atari-games-63f5242440b1

[20] Meriam (2020). Tutorial : AI to play game Pong using reinforcement learning. The Robot Camp. URL:https://therobotcamp.com/2020/04/21/tutorial-teach-ai-to-play-pong-from-scratch-with-reinforcement-learning/

[21] Paudel, P. (2020). Exploring Game Playing AI using Reinforcement Learning Techniques. St. Olaf College. doi:10.13140/RG.2.2.14522.62400