Reinforcement Learning to play Atari Games

Aritra Chowdhury (ac79277), Casey Copeland (cmc6793),
Rohitashwa Chakraborty (rc47878), Soumik Choudhuri (sc64856)

Introduction

Reinforcement Learning (RL), or approximate dynamic programming, is a machine learning technique which enables an agent to learn and discover the optimal behavior in an interactive environment via trial and error. The reinforcement learning algorithm discovers the optimal policy via its previous actions, experiences, and subsequent rewards. The RL agent must explore new states while simultaneously maximizing its overall reward, balancing exploration vs exploitation. Reinforcement Learning is used to solve highly complex dynamic programming problems where the distribution of randomness is unknown, large state spaces and action spaces, as well as problems that have multiple optimal solutions. Many real-world scenarios we want to automate involve these issues. Reinforcement learning applications include robotics, self-driving cars, manufacturing automation, portfolio construction, and traffic light management.

Our video game company is interested in the use of Reinforcement Learning to automate opponents' actions. Many AI players use rule-based machine learning which identifies rules to store and apply to win the game. Reinforcement learning can be used to create more robust game players, as RL is always updating its actions and learning to execute optimal decisions.

In this report, we explore multiple model structures and algorithms to build a neural network that trains itself by playing games repeatedly. Based on a combination of experimentation and intuition, we built neural networks to play 2 games - Pong and Assault. Our final goal of the project is to build the models such that they can play the above games properly and hopefully, win!

Reinforcement Learning Algorithms

We explore two popular Reinforcement Learning Methods, Policy Gradients and Actor Critic, to compare and evaluate which method produces the best model. The methods take a different approach in finding the optimal solution to the dynamic program, approximating the value function via feedback and interaction to set up the optimal policy.

Policy Gradients

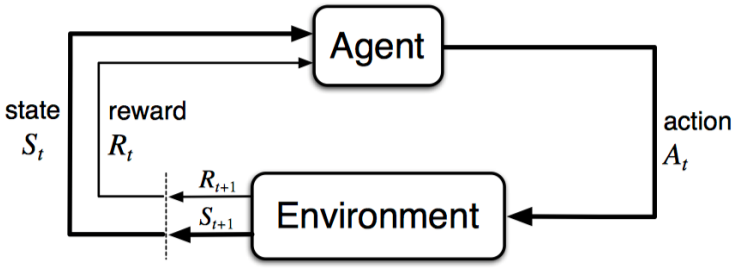


Figure 1: Structure of Policy Gradients

Policy gradients pose the problem not as a value function to be optimized, but rather a classification problem aimed at choosing optional actions. The classification model takes state variables as inputs, in our case the various game image frames, and outputs a probabilistic distribution of which available actions are best based on the current state. As the model trains, it improves the classification of which action is best to maximize game score at each state.

Actor-Critic

Actor Critic is an RL ensemble method that combines Policy Gradients (described above) and deep Q learning (DQN). DQN is an older reinforcement learning method that uses neural networks to approximate the value function of each possible action. Actor Critic uses policy gradients to pick the action and evaluate the chosen actions using deep Q learning. PG is the actor making the decision, DQN is the critic determining if the choice was good or bad. The neural networks train simultaneously allowing acting and learning to be more focused, fixing biases associated with each method and hopefully, improving the RL performance.

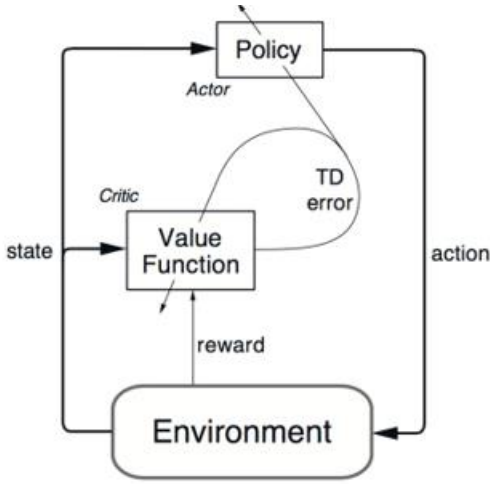


Figure 2: Actor-Critic Structure

Game I: Pong

Game Brief

Pong is a two player, two-dimensional game that simulates playing ping-pong. The player controls an in-game paddle by moving it vertically across the left or right side of the screen to hit the pong ball. They can compete against a computer controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth and the first player to 21 points wins the game.

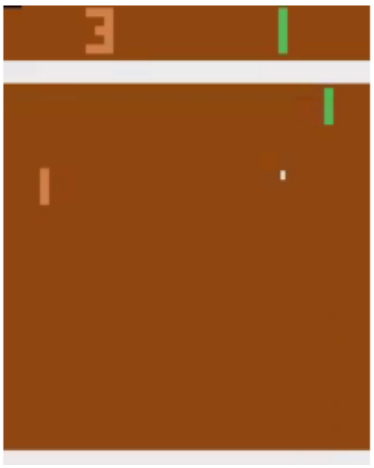


Figure 3: A game of Pong

There are a total of 6 action spaces available to us. While we model the algorithm to account for all 6, we only care about the following three.

- *NOOP:* No Action (Skip a Chance)
- *RIGHT:* In the context of Pong, move downwards.
- *LEFT:* In the context of Pong, move upwards.

Model Structure

Our Actor-Critic method provides better results in playing the game. Our algorithm structure is explained below.

Pre-Processing

There are a lot of redundant sections in the frame, such as the space left or right of the two paddles, and the space recording the scores. These pixels will not be helpful in training the model so we crop those out. We also convert the pixels to grayscale to speed up training. The final frame is of the size 80x80, with 4 frames stacked on top of each other.

```python
def prepro(self, frame):
    # cropping frame to 80x80 size
    frame_cropped = frame[35:195:2, ::2,:]
    if frame_cropped.shape[0] != self.COLS or frame_cropped.shape[1] != self.ROWS:
        frame_cropped = cv2.resize(frame, (self.COLS, self.ROWS), interpolation=cv2.INTER_CUBIC)

    # converting to black and white for faster training
    frame_rgb = 0.299*frame_cropped[:,:,0] + 0.587*frame_cropped[:,:,1] + 0.114*frame_cropped[:,:,2]
    frame_rgb[frame_rgb < 100] = 0
    frame_rgb[frame_rgb >= 100] = 255

    new_frame = np.array(frame_rgb).astype(np.float32) / 255.0
    self.frame_buffer = np.roll(self.frame_buffer, 1, axis = 0)
    self.frame_buffer[0,:,:] = new_frame

    return np.expand_dims(self.frame_buffer, axis=0)
```

Figure 4: Code for Pong Pre-processing

Neural Network

The Actor and Critic are at the heart of the algorithm and are defined by dense neural networks. Both are composed of a single dense later with an ELU activation function. The Actor model outputs probabilities of taking an action using the Softmax activation function. The Critic model is a single output layer that evaluates the weights through the MSE loss function. The model summaries are shown below.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 4, 80, 80)]	0
flatten (Flatten)	(None, 25600)	0
dense (Dense)	(None, 512)	13107712
dense_1 (Dense)	(None, 6)	3078

Total params: 13,110,790
Trainable params: 13,110,790
Non-trainable params: 0

Figure 5: Actor Model Summary

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 4, 80, 80)]	0
flatten (Flatten)	(None, 25600)	0
dense (Dense)	(None, 512)	13107712
dense_2 (Dense)	(None, 1)	513

Total params: 13,108,225
Trainable params: 13,108,225
Non-trainable params: 0

Figure 6: Critic Model Summary

Game Algorithm

Deciding an Action

The Actor model predicts probabilities for each action based on the input frame. The action is then decided by a random choice, weighted by the probabilities predicted by the Actor.

```python
def decide_action(self, feed):
    prediction = self.Actor.predict(feed)[0]
    action = np.random.choice(self.action_size, p=prediction)
    return action
```

Figure 7: Code for Deciding an Action

Taking a Step

Each step taken gives us the next frame, the reward, along with further information and whether the game is complete. The reward is +1 every time we win a point, -1 every time the opponent wins a point, and 0 at all other times. The new frame is then processed again for the next step.

```
def take_step(self, action):
    pix_new, reward, done, info = self.env.step(action)
    pix = self.prepro(pix_new)
    return pix, reward, done, info
```

Figure 8: Code for Taking a Step

Discounted Rewards

In general, only the most recent action gives us the reward. However, actions prior to the most recent one should also be regarded as positive actions and thus we introduce the concept of reward discounting. Each action is discounted by a factor of delta (0.99) from the most recent award. This way, the most recent action gets the highest reward, with the earlier actions mapped to lower rewards.

```
def discount_rewards(self, reward):
    # Compute the delta-discounted rewards over an game
    delt = 0.99 # discount factor
    nr = r.shape[0]
    discounted_r = np.zeros(nr)
    for t in range(nr):
        # start at the end
        if r[nr-t-1] > 0:
            discounted_r[nr-t-1] = 1
        elif r[nr-t-1] < 0:
            discounted_r[nr-t-1] = -1
        elif t==0:
            discounted_r[nr-t-1] = 0
        elif discounted_r[nr-t-1] == 0:
            discounted_r[nr-t-1] = delt*discounted_r[nr-t]
    return discounted_r
```

Figure 9: Code for Computing Discounted Rewards

Model Training

Once a game is complete, the rewards are stored and discounted. The Critic model then makes predictions on the feed and the weights are computed from calculating the difference between its predictions and the discounted rewards (the truth). The Actor model is then trained with the action array as inputs and the calculated weights as the sample weight. The Critic model is trained with the discounted rewards as the sample weights. Thus, the model is updated, and we move to the next game.

```
def play_a_game(self):
    # reshape memory to appropriate shape for training
    feed = np.vstack(self.frames)
    actions = np.vstack(self.actions)
    # Compute discounted rewards
    discounted_r = self.discount_rewards(self.rewards)
    # Get Critic network predictions
    predictions = self.Critic.predict(feed)[:, 0]
    # Compute weights (advantages)
    weights = discounted_r - predictions

    # training Actor and Critic networks
    self.Actor.fit(feed, actions, sample_weight=weights, epochs=1, verbose=0)
    self.Critic.fit(feed, discounted_r, epochs=1, verbose=0)
    # save models
    self.Actor.save('NN_WinPong_AC2_Actor_temp.tf')
    self.Critic.save('NN_WinPong_AC2_Critic_temp.tf')
    # reset training memory
    self.frames, self.actions, self.rewards = [], [], []
```

Figure 10: Code for Playing the Game

Gameplay Summary

For every game in training, we first decide the action and take the next step in the game using the same. We then move to the next frame and decide on a new action and take the new step. The process continues until the game is done and we have a computed score. Using the score and stored frames, we update the weights of the model and finally move on to the next game.

```
def run(self):
    for e in range(self.train_games):
        current_frame = self.reset_environment()
        done = False
        score = 0
        while not done:
            # Actor picks an action
            action = self.decide_action(current_frame)
            # Retrieve new state, reward, and whether the game is done
            future_frame, reward, done, _ = self.take_step(action)
            # Memorize (state, action, reward) for training
            self.store_feed(current_frame, action, reward)
            # Update current state
            current_frame = future_frame
            score += reward
            if done:
                average = self.save_scores(score, e)
                self.play_a_game()
    self.env.close()
```

Figure 11: Code for Gameplay Summary

Tuning Considerations

Memory Buffer and Warm Up Games

We initially play several warmup games to build up a memory of frames. When we begin training the model, we sample frames from different games to feed into our model. This way, we theoretically remove the correlation between states. However, while faster, that tends to perform poorly when compared to a simple model that moves frame to frame. We did not need warm up games as we initialized the model weights as uniform and allowed the network to update itself organically.

Averaging 4 frames into 1

Since adjacent frames are extremely correlated, we could combine a set of 4 frames into and averaging the rewards at the time. However, while adding this made the model faster, it also took the model a larger

number of games to start performing well. We considered the tradeoff and decided on lesser games rather than lesser training speed.

Results

Training

We trained the Actor and Critic models for a total of 5000 games. We plot the training scores, along with a 50-game average, below. The model starts off strong but then the learning rate reduces. It nears a positive average score by the end of 3000 games with winning a fair chunk of games towards the end.

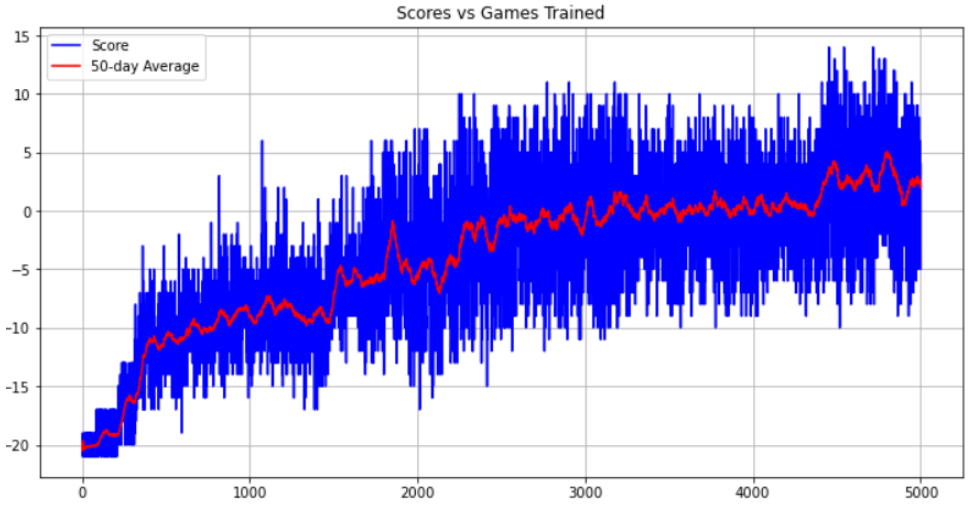


Figure 12: Training Scores with 50-game Average

Test Results

Post training, we played the game of Pong 200 times using the trained Actor Model. The distribution of scores is shown on the right.

We achieve an average score of -1 with a standard deviation of 3.
The model 64 out of 200 games.

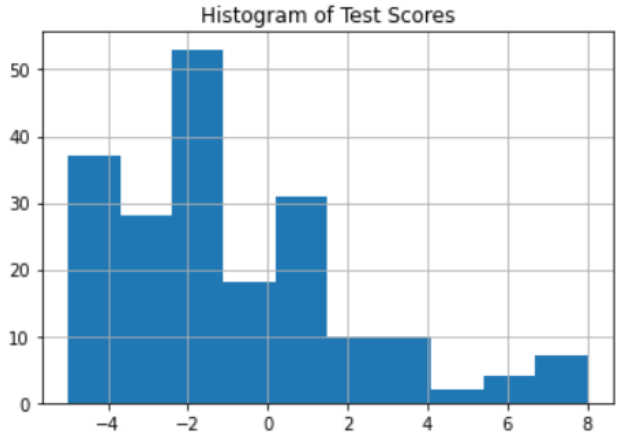


Figure 13: Histogram of Test Scores for 200 Games

Game 2: Assault

Game Brief

Assault is a popular fixed shooter video game developed. It requires the player to control a spaceship fixed at the bottom of the screen, to shoot projectiles towards an enemy mothership that deploys smaller ships to attack the player. The player must also dodge enemy projectiles, missiles, and bombs from touching their ship.

Figure 14: A Game of Assault

The objective of the game is to kill as many children ships as possible.
The constraints are:

- You have four lives (Number of lives is indicated at the bottom left of the screen)
- Shooting heats up the cannon. Overheating the Cannon costs you a life (The heat bar is on the bottom right of the screen)

The action space available to the player is:

- 'NOOP' : No-Operation (Skip a chance)
- 'FIRE' : Fire Upwards towards the enemy
- 'RIGHT' : Move Right
- 'LEFT' : Move Left
- 'RIGHTFIRE': Fire to the Right
- 'LEFTFIRE' : Fire to the left

Modeling Assumptions

In Assault, the nature and difficulty of the environments and enemy ships increase as the levels progress; with teleporting enemy ships, bombs, missiles, etc. to name a few. Since we have limited computational power and our models are relatively light, we sought to limit the scope of our RL agent. Though the game allows for firing on either side (to counter bombs which feature in Level 7 and beyond), we chose not to allow those moves, making our game a little easier to learn.

Model Structure

We modeled several variations of Neural Nets, of varying complexity, in an attempt to beat the game. From shallow Convolutional Neural Networks (CNNs) to deep CNNs and hybrid models with CNNs augmented with numeric inputs. Below are a few of our most successful models:

Pre-Processing

In comparison to Pong, Assault has a more complex gaming environment. Different ships spawn and fire randomly, the enemy ships and their projectiles change shape and color at each level, and there is no set threshold for winning the game.

A shallow CNN model requires a simple input, therefore we stripped away the superfluous and redundant game information to structure the neural network. The game environment was broken down into 5 parts and processed individually as follows:

- **(Segment 1)** Scoreboard : This section from the top was ***discarded***
- **(Segment 2)** Mothership : The mothership was ***compressed*** into a one-dimensional array signifying the width of the mothership
- **(Segment 3)** Child Ships : This region contains the positional information of all the children, missiles, etc. We ***down sampled*** the region by a factor of 2, while attempting to retain as much information as possible
- **(Segment 4)** Agent / Our Ship : All we care about is the agent's location on the X axis. Therefore, the region has been ***compressed*** into a 1-D array too
- **(Segment 5)** Lives : We processed this segment and ***calculated the number of lives*** the agent has remaining.
- **(Segment 6)** Heat Percentage: We processed the image and ***calculated the heat percentage*** of our cannon. A higher value means higher heat and a value of 1 leads to loss of a life.

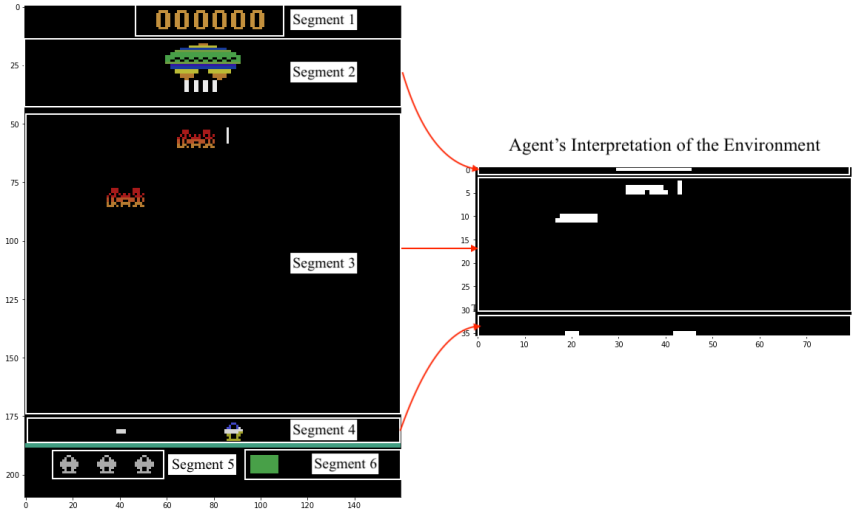


Figure 15: Agent's Interpretation of the Environment

Model

Assault, as mentioned before, is a non-deterministic game. Therefore, even random play, on average, can give us a score of around 400. This sets the bar for our model very high. We initially tried a couple of variations of a simple CNN, feeding in the entire environment and letting the model train. However, we realized the model was unable to learn effectively. Especially, modeling the relation between heat level of the gun and lives would be difficult to learn in a shallow network. Therefore, to reduce computation and model complexity, we created an augmented CNN model which takes two inputs:

- Game Environment: The processed environment (as shown above) is passed to a 2-layer convolutional neural network and is subsequently flattened
- Heath metrics: Number of lives and heat percentage of the gun are passed as input and concatenated to the flattened output of the CNN. This lets the Dense network learn the interaction between health, heat, and the board easily.

Model: "dual_input_model"

Layer (type)	Output Shape	Param #	Connected to
board_input (InputLayer)	[(None, 36, 80, 4)]	0	[]
conv1 (Conv2D)	(None, 17, 39, 32)	2080	['board_input[0][0]']
health_input (InputLayer)	[(None, 1, 2, 4)]	0	[]
conv2 (Conv2D)	(None, 4, 9, 16)	12816	['conv1[0][0]']
health_conv1 (Conv2D)	(None, 1, 2, 5)	25	['health_input[0][0]']
flatten (Flatten)	(None, 576)	0	['conv2[0][0]']
flatten_1 (Flatten)	(None, 10)	0	['health_conv1[0][0]']
tf.concat (TFOpLambda)	(None, 586)	0	['flatten[0][0]', 'flatten_1[0][0]']
dense1 (Dense)	(None, 256)	150272	['tf.concat[0][0]']
output (Dense)	(None, 5)	1285	['dense1[0][0]']

Total params: 166,478
Trainable params: 166,478
Non-trainable params: 0

Figure 16: Summary of the Model

Game Algorithm

Playing a Game

```
while not done:
    if render == True: # do you actually want to visualize the playing?
        env0.render()
        time.sleep(slow)

    # skipping every 4 frames
    if frames_this_game%4 == 0:
        # epsilon-greedy
        if np.random.random() < ep:
            action = np.random.choice(action_space)
        else:
            vf = model([feed, health_feed ],training=False).numpy()[0]
            action = np.random.choice(action_space,p=vf)

    if num_bombs > 0: # if we have a bomb, we want to fire left/right
        action = np.random.choice([0,1], p=[0.5, 0.5]) + action_space

    pix_new, reward, done, info = env0.step(action)
    pix, num_lives, percent_heat = segment_env(pix_new)
    health = np.array([num_lives, percent_heat]).reshape((1,2))

    frame_array.append(pix)
    action_array.append(action)
    reward_array.append(reward)
    health_array.append(health)
    frames_this_game += 1

    for f in range(1,frames_to_net):
        feed[0,:,:,frames_to_net-f] = feed[0,:,:,frames_to_net-f-1].copy()
    feed[0,:,:,0] = pix.copy()
    score += reward

    if frames_this_game > 50000:#5000000:
        print("Game is taking too long, breaking")
        done = True
        break
```

Figure 17: Code for Playing the Game

We have adopted an epsilon greedy approach to train the reinforcement learning model. During the initial games, the algorithm favors exploration (epsilon = 1). The Neural Net is only adjusting its weights in this phase. It does not make any predictions. In the later games, the exploration rates decrease gradually, and the neural network's prediction takes over. This allows the neural net to update itself gradually.

To make predictions, the neural network implements a SoftMax normalization in its final output. Since these normalized outputs add up to 1, we approximate them to probabilities. We subsequently choose an action from a random variable, with the probabilities

$$Neural\ Network\ Output\ =\ P(Action \mid Board, Lives, Heat(\%))$$

Where: $Action\ \epsilon\ [NOOP, UP, LEFT, RIGHT, FIRE]$

For now, we have adopted a basic image processing approach to check for bombs in the game. Should there be a bomb, the bot trumps the Neural net's predictions in favor of either left-side fire or right-side-fire (chosen randomly).

```
num_bombs = -1 # at least 1 object in this line...my bot
for i in range(1,pix.shape[1],3):
    if pix[-1,i] == 1 and pix[-1,i-1] == 1:
        num_bombs += 1
```

Figure 18: Code for Checking for Bombs

The other aspects of training the model, like rewards and discounting are similar to the approach implemented in Pong and have been discussed above.

Results

Training

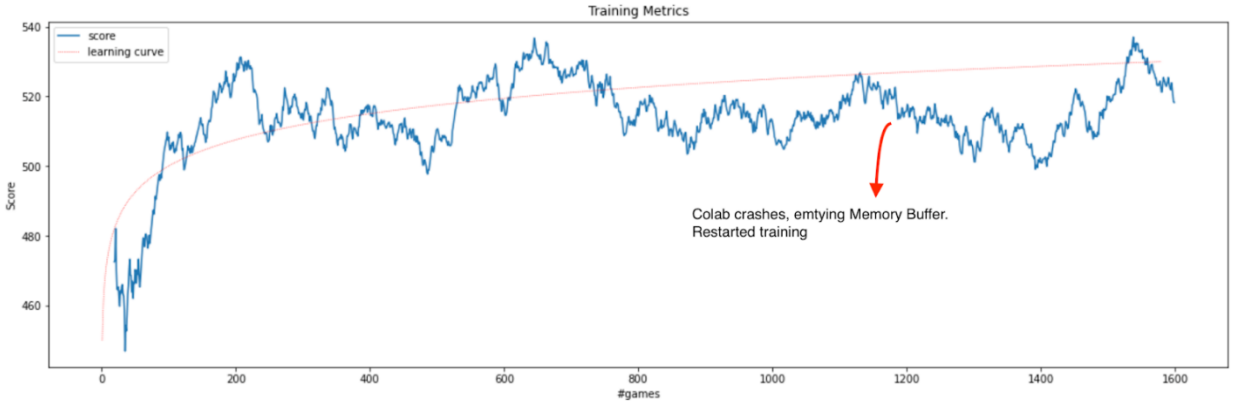
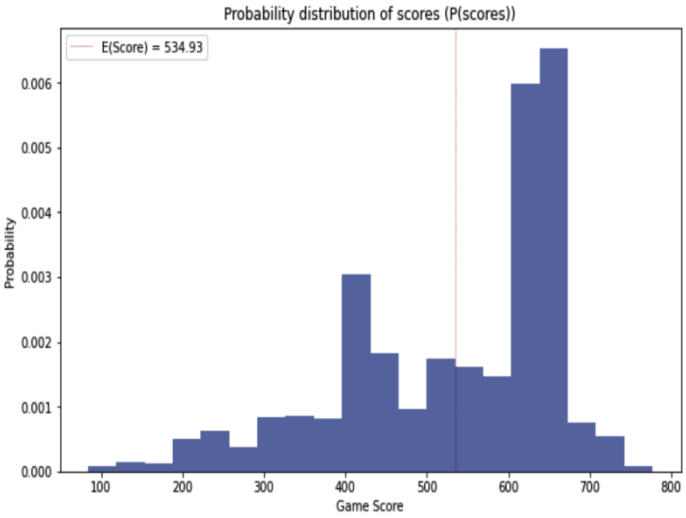


Figure 19: Training Scores for Assault

We observe that initially the model performance is acceptable and begins to drop. This could be due to the high level of randomness and exploration (high values of epsilon) in the beginning of gaming. After that, it remains on a similar level of scores.

Test Results




```
1 print('Mean:', scores.mean())
2 print('Mode:', scores.mode()[0])
3 print('Median:', scores.median())
4 print('Std Dev:', round(scores.std(),2))
5 print('Skew:', round(scores.skew(),2))
```

```
Mean: 534.933
Mode: 651.0
Median: 588.0
Std Dev: 137.83
Skew: -0.87
```

Our model scores, on average, 535 ± 270 points 95% of the time. The distribution of scores exhibits a strong negative skew of -0.87, therefore the mode score is much higher, around 651 points. A few games also crossed the staggering 700-point mark.

Limitations

Given our agent's limited action space (no left or right fire), we had expected our model to learn to dodge enemy fire and destroy enemy ships. Our evaluation suggests that while the model can dodge and attack successfully, it still takes risks and ends up losing a life. Thus, we might have to improve the negative reward allocation. Furthermore, the model is defenseless against horizontally incoming bombs, which is why we have a glass-ceiling around 650 points.

Future Scope of Work

Future improvements would include updating the reward function to account for penalties for losing life, overheating the cannon, and wasting fire. We are currently attempting to leverage transfer learning to increase the action space of our model, to include left and right fire. This model is currently not competitive and thus has been excluded from the report.

Conclusion

This project gave us a great, and humbling, introduction into the powerful world of Reinforcement Learning. We have listed our findings, recommendations, and concerns below.

Various issues we faced include the complexity of reinforcement learning and the computation power needed to implement it. Reinforcement Learning is a cutting-edge field that often feels like a black box as we and the algorithm experiment with a lot of trial and error. Hiring a specialist would be necessary for us to reap the benefits in our gaming programs. In our analysis we required long training time and heavy computation power for the RL methods to perform not optimally, but simply satisfactory. We believe reinforcement learning has the potential to give us a competitive edge against other gaming companies, differentiate our products, and catalyze our company bringing sustained success. To reap these benefits, there would be significant up-front investments. Implementation costs include hiring a specialist, the time required to code and discover optimal programs, training existing developers, additional computational power, and bringing everything to production scale. With this, RL would provide our company with more robust and intelligent games. Long term, our team sees substantial potential for the use of reinforcement learning, but we want to be cautious of the possible costs. The major tradeoff this project helped us identify is the large learning curve that comes with the vast potential of reinforcement learning. There are clear long-term competitive advantages, but short-term, up-front investments.

Next steps, we recommend calculating a return on investment and conducting a costs vs benefits analysis for implementing reinforcement learning algorithms. Questions to consider include: Are our current automated opponents sufficient? Will this be a profitable shift for the company, can we quantify it? With this, we do believe there is non-quantifiable strategic and financial leverage associated with implementing reinforcement learning. In the technology space, to stay competitive, you have to push boundaries, stay on top of advancements, and offer consumers the best, cutting edge products. If deemed profitable by the other departments, our analytics team recommends hiring a reinforcement learning specialist to implement this technology in our gaming company.