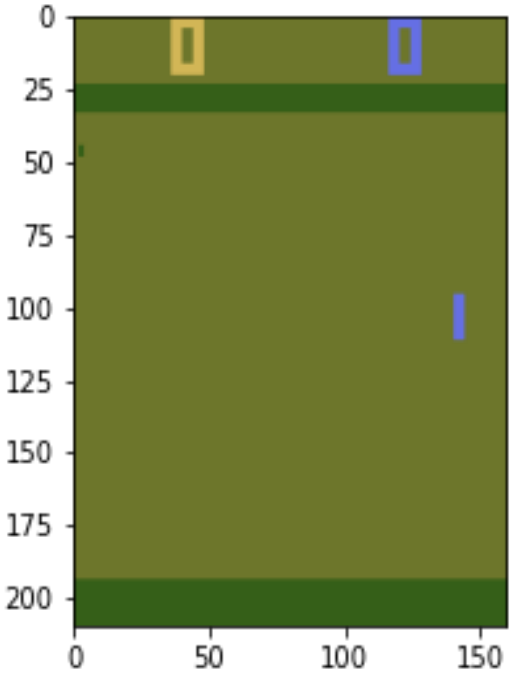
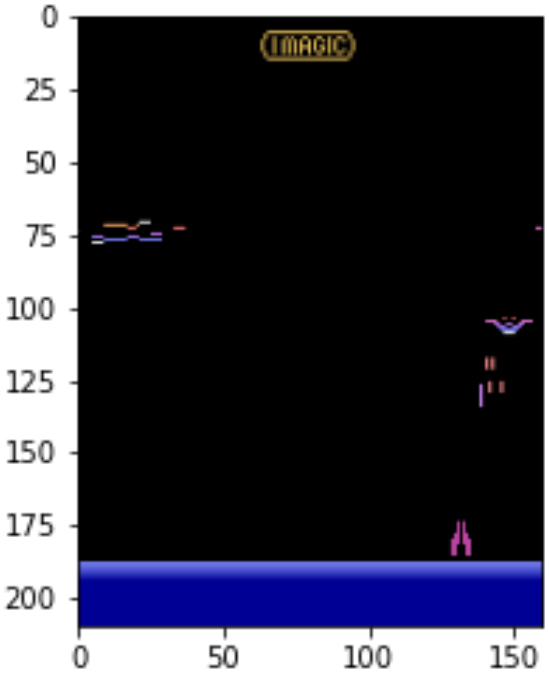
Optimization II – Project 3

Playing Atari games with Reinforcement Learning

Team Members: Andre Han, Anant Gupta, Vishal Gupta, Aatheep Gurubaran



Pong



Demon Attack

Introduction

Playing games has always been fun, whether it's older, simpler games developed by Atari in the 70s to more advanced games like Call of Duty, Grand Theft Auto, etc. Despite the stark difference in complexity and graphics, the high-level objective of all the games stays the same, more or less, that is taking a bunch of actions from all possible actions to maximize some kind of reward, in simpler games like Pong that reward is basically the total points collected. In recent years reinforcement learning has made a lot of headway in understanding such games and taking the best possible decision to maximize the total rewards and finally winning the game. Reinforcement learning is just an area of machine learning that tries to mimic the action that could be taken by the best player in the game to maximize the total cumulative reward, it tries to reward desired actions, while simultaneously punishing undesired actions. This is helpful in creating a formidable AI that can play complex games and even beat humans in most instances.

Problem Description

The goal of the project is to evaluate the use of a reinforcement learning algorithm to automate opponents' actions when playing against the person playing the game. Instead of jumping directly into creating an algorithm to play complex games developed by our organization, we will start first by developing the algorithm that can play 2 Atari games - Pong and Demon Attack. Based on the performance of the said developed algorithm we can understand and identify the future scope of Reinforcement learning to automate opponents' actions for complex games. If successful, this can be developed further to play more advanced games that will challenge human players and increase the average playtime on our platform. Furthermore, this will also help in creating AI bots for humans to play newer games with a low user base.

Strategy - Creating RL opponent

The goal of a general RL algorithm is to come up with a strategy to earn as many points as possible while restricting the opponent from scoring. This can be accomplished by providing the model with a reward for actions that lead to a point scored while discouraging the actions that were taken leading to a point for the opposition. Since there is randomness involved in the

process, that is, we can't know what decision our opponent is going to make, we have to interact with the game environment to constantly understand the outcome (reward) of the action taken by the RL opponent. To identify the decision to take among all the possible decisions, we will try to create a neural network model that can identify the decision to make based on the past behavior and rewards that were earned based on the decision made. We also have to keep updating this model as we interact with the game environment more and more to improve upon the neural network capability to pick the optimal action. Initially, we will make a lot of mistakes and lose a lot of points and games, but as the NN model continues to interact with the environment, it will keep on learning and improving. The high-level approach of how we will leverage RL to create an opponent to play the two Atari games will be as below -

1. Initialize the environment for the Atari game (Pong or Demon Attack)
2. Initialize the neural network model that can tell us the optimal decision to make
3. Play the first game, take action based on the NN model and record the reward and all the frames in the game (frames can be thought of as the state, and reward is the outcome when we make the decision based on the NN model)
4. Using the states and rewards for each step in the game, update the weights of the NN model in an effort to accurately identify the right decision to be taken for each state
5. Keep doing steps 1-4 for a lot of games till the NN model is able to accurately identify the right decision

Developing RL opponent for Pong

About Pong

Pong is a table tennis-themed twitch arcade sports video game, featuring two-dimensional graphics, manufactured by Atari and originally released in 1972. The player controls an in-game paddle by moving it vertically across the left or right side of the screen. The possible actions in pong are - move up, move down, and do nothing. They can compete against another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth. The goal is for each player to reach 21 points before the opponent; points are earned when one fails to return the ball to the other.

Model Specifics

Creating the perfect RL opponent involves making a lot of decisions about choosing the right RL approach, neural network architecture, defining the state to be fed into the neural network, etc. Since there is no way of knowing which combination of these “hyperparameters” will work best for us, we have tried a few combinations to arrive at the best possible approach for us. Broadly the decision or “hyperparameter” choices that we have are as follows -

1. Optimal Decision - Reinforcement Learning Algorithm

RL algorithm is used to arrive at the optimal decision to be made for each state (or frame). The most common algorithm in the RL space that has been used to do a similar task is *Deep Q-Learning* and *Policy gradient*.

Deep Q-Learning tries to approximate the value function using a neural network. The value function represents the cumulative discounted reward that can be obtained if a certain decision is taken, hence at each state, we will have a different value function associated with each possible decision. Once the approximated value function is known we can take the optimal action.

Policy gradient, on the other hand, goes a step further and tries to identify the best decision right away. In policy gradient, we will also train a neural network, but the neural network gives the probability of the best decision that should be taken to maximize the cumulative reward from the current state to the end of the game.

2. Value function approximation (DQL) or Best Action Classification (PG) - Neural network Architecture

In both Deep Q-Learning and Policy gradient, we have to create a neural network model, although one is regression and another is for classification. Network architecture plays a huge role in affecting the performance of the overall approach, as we will see in the results. Since we will be using the frames of the game, the NN model will have convolution layers followed by dense layers, the set of hyperparameters that will be changed are - loss function, number of filters, and number of layers. The two network architectures that are tried are as below -

Original Architecture

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, 80, 80, 4)]       0         
                                                                 
 conv2d (Conv2D)             (None, 19, 19, 16)        4112      
                                                                 
 conv2d_1 (Conv2D)           (None, 8, 8, 32)          8224      
                                                                 
 flatten (Flatten)           (None, 2048)              0         
                                                                 
 dense (Dense)               (None, 256)               524544    
                                                                 
 out0 (Dense)                (None, 1)                 257       
                                                                 
 out1 (Dense)                (None, 1)                 257       
                                                                 
 out2 (Dense)                (None, 1)                 257       
                                                                 
=================================================================
Total params: 537,651
Trainable params: 537,651
Non-trainable params: 0
_________________________________________________________________
```

Modified Architecture

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, 80, 80, 4)]       0         
                                                                 
 conv2d (Conv2D)             (None, 19, 19, 32)        8224      
                                                                 
 conv2d_1 (Conv2D)           (None, 8, 8, 64)          32832     
                                                                 
 conv2d_2 (Conv2D)           (None, 6, 6, 64)          36928     
                                                                 
 flatten (Flatten)           (None, 2304)              0         
                                                                 
 dense (Dense)               (None, 512)               1180160   
                                                                 
 dense_1 (Dense)             (None, 256)               131328    
                                                                 
 out0 (Dense)                (None, 1)                 257       
                                                                 
 out1 (Dense)                (None, 1)                 257       
                                                                 
 out2 (Dense)                (None, 1)                 257       
                                                                 
=================================================================
Total params: 1,390,243
Trainable params: 1,390,243
Non-trainable params: 0
_________________________________________________________________
```

The above model architecture is for Deep Q-Learning, in Policy Gradient we will only need one output layer that can give the probability for the best action, the rest remains the same. This will not change the number of parameters as the architecture is still the same.

3. Other Modifications - Memory Buffer, Linear Annealing, 4 Frame action update

Memory Buffer - In the default strategy, we are playing one game and then updating the NN model's weight based on the performance of the NN model, so we are just using the last game outcome to update the model. The memory buffer allows us to store all the previous games played and then sample frames from it to update the model weights. With memory buffer, we have also increased the likelihood of playing those games that have a positive reward associated with them. Every time we play a game there are only a handful of frames that have a non-zero reward that slows down the learning of the NN model, but the memory buffer enables us to "up-sample" those frames where we are actually winning a point, leading to faster learning.

Linear Annealing - For each game played, we are not taking the optimal decision, instead we are using an ϵ-greedy approach to ensure that we are exploring as well as exploiting the NN model to win the game. Since we are playing hundred of games to update the NN model, we can't have a fixed ϵ, as we need more exploration early in the simulation and more exploitation when the NN model starts performing better. To have this varying degree of exploration, we use linear annealing that starts with a high value of ϵ and reduces it linearly based on the total number of games.

4-Frame action update - Since each game has thousands of states, before the game is done, playing the game frame by frame for that many states has a higher runtime, as well as it can change the state of the game in an undesirable direction making it difficult to win. To smoothen up the play, we have tried to identify the action every 4 frames and used that action to play the next 4 frames.

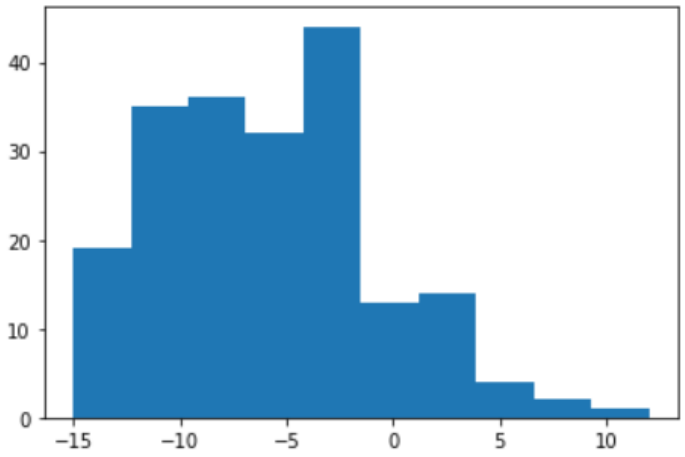
Results

For Pong, we have initially tried to get the best winning score possible for all the games, without any time constraints, later to evaluate the performance, have limited the training up to 5000 games that can help capture the performance improvement and set a benchmark for any future improvement. Once the best specification is identified the same model is applied to Demon Attack. The different specification tried, the outcome of each iteration and comments has been provided in the table below -

Model Specification	**Outcome of the iteration**	**Comment**
Policy Gradient with original architecture, 4-frame action update, linear annealing &	Trained on 5000 games, mean score of -20.72	Most point scored is 5, won 0 games

memory buffer		
Deep Q learning with original architecture, 4-frame action update, linear annealing & memory buffer	Trained on 10000 games, a mean score of -19.39	The maximum points won in any game is 11, won 0 games
Policy Gradient with original architecture, 4-frame action update, linear annealing, memory buffer and saving 4 frame feed for learning	Trained on 15000 games, a mean score of -20.8	The maximum points won in any game is 5, won 0 games
Deep Q learning with modified architecture, 4-frame action update, linear annealing, memory buffer and saving 4 frame feed for learning	Tried training the model multiple times on 5000 games, crashed after 3900+ games. Mean score of -8.13	The maximum number of points won in any game is 21, model started winning the games

Deep Q learning with modified architecture is giving the best performance so far, hence this model is used to play the final 200 games to get a score. The ***mean score over those 200 games is -5.84***, with a ***win rate of 13%.*** The distribution of scores over the 200 games without any further training is as below -



To benchmark the performance of our model with humans we have played ten rounds of Pong to set a benchmark for the average player's skill level. Our human player averaged 17 points, ***winning 40% of the time***. The average round had 37 matches, with each match lasting an average of 7 seconds. The total average playtime for a single round came to 4 minutes and 32 seconds. The detailed score is in the table below -

Game	Me	Opponent
1	10	20
2	12	20
3	17	20
4	16	20
5	20	18
6	20	19
7	15	20
8	17	20
9	20	18
10	20	19

Based on less than 4000 games, we can conclude that this model will perform a lot better than humans if we have the computation power to train further, but due to the computation power limitation, we had to stop under 4000 games. Since it performed the best among all the iterations this model will be used to train on Demon Attack as well.

Developing RL opponent for Demon Attack

About Demon Attack

Demon Attack is another Atari game that involves trying to defend the home base from the attack of demons. Demons spawn from the top sides of the screen and try to harm the player either by shooting weapons or by hitting the player ship by moving towards it. The player has 3 lives, which can be increased up to a maximum of 6 by surviving a wave unscathed. There are 4 possible actions in this game: move left, move right, shoot, and do nothing. The goal of this game is to try and survive as long as possible while shooting enemies to score points.

Model Specifics

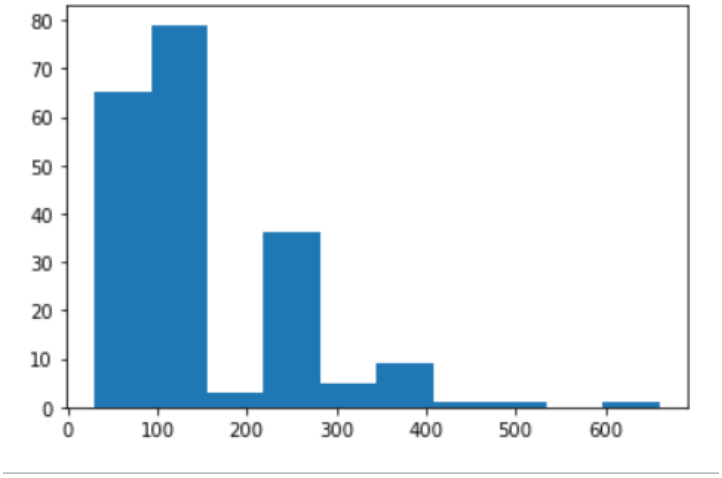
RL models are highly generalizable to solve similar kinds of problems, so to play Demon attack we will be utilizing the same model that performed the best with Pong to evaluate that Demon Attack. This will help us understand the time and resources required to replicate our strategy for creating AI-driven bots for other Atari games, thereby reducing the dependency on the development time of a new model for each unique game. The model will be using Deep Q-Learning to arrive at the optimal decision to make at each state with modified architecture to

predict the value function associated with each action. There will only be a slight modification in the NN architecture as we have 4 actions - fire, move right, move left, and no action, compared to three actions in Pong.

Results

Unlike Pong, Demon Attack doesn't allow one to win the game, the objective, though, is to accumulate as many points as possible. We tried training the model for 5000 games but again due to computational limitations, the machine ran out of memory after 1000 games. Since Demon Attack is more complicated and requires a lot more time to play one game - 45 seconds compared to 3 seconds in the early stage of learning, the resource limit has reached sooner than what we faced for Pong.

Once the model is trained, we played 200 games without any further training to evaluate the performance. The ***mean score over those 200 games is 161.7***. The distribution of scores over the 200 games without any further training is as below -



Since Demon Attack has multiple waves, with increased difficulty, we can see that the distribution is multi-modal, peaking with decreasing frequency as we clear each wave. At best, our model is clearing the first three waves but loses the game in the fourth wave.

We again benchmarked the performance of our model with humans, playing a total of ten rounds of Demon Attack. Our human player averaged 2258 points, lasting an average of 7 waves per game. Each wave took a mean duration of 45 seconds to complete, leading to an

average playtime of 5 minutes and 25 seconds per game. The detailed score is in the table below -

Game	Score
1	640
2	530
3	1340
4	1480
5	2880
6	3250
7	2770
8	3500
9	2960
10	3230

Since we are only able to train the model for less than 1/3rd of games that we are able to achieve with Pong, we can see a significant difference in the score. Although for the first 2 games played by the human, the best performance with the RL opponent gives a better score. This indicates the potential of the RL opponent in the future to actually outperform the human opponent if an appropriate amount of training is done.

Limitations & Conclusion

Based on the performance achieved by the trained RL opponent, we can conclude that there is a lot of opportunity in spending resources to create a universal RL algorithm that can competitively play games with humans. Although it requires a significant investment in computation power, as there is no exact way of knowing what specifications will work better for different games. We could only arrive at the best model by running different combinations of hyperparameters to train the model for a longer period of time to create a good RL opponent.