RM 294 – Optimization II
Project 3 – Reinforcement Learning
Group 3

Oluwaseun Ibitoye, Shawn Kalish, Safiuddin Mohammed, Shiyong Liu

Overview

The purpose of this report is to assess the viability of using reinforcement learning to automate correct actions when playing against humans in Atari video games. The goal is to determine the appropriate playing strategy and then train a model by tuning multiple parameters to learn how to win against an opponent. For the models built, optimizing strategies have been explored to improve the performance. Those methods include adding a memory buffer, adjusting learning rates, reconfiguring the neural network structure, removing some parts of the frame where the game is not in play, implementing the actor-critic method, etc..

Background

Reinforcement learning has a useful optimization strategy where players learn to attain optimal strategy through interaction to the environment. It has been widely applied to different fields such as robotics, manufacturing automation, portfolio management and traffic light management where computers are trained to do well in those tasks through reinforcement learning where little human intervention is needed.

There have been many popular reinforcement learning methods such as Q-learning and policy gradient to optimize the game. In terms of Q- learning, we approximate the value function using a neural network and output a value function that matches each possible action that has been taken. Then, among those value functions, we pick the one action that yields the highest value function. As for policy gradient, we pick the optimal action as a classification problem using a neural network. The output of the model will give action probability as indication of performance. This report will mostly focus on the policy gradient approach for Pong and a form of Q-Learning called Deep-Q Learning for Video Pinball .

Methods

Pong

Approach 1

Pong is a table-tennis type game where one player hits the ball with a paddle to another player and they continue to hit it until one person misses and the opponent wins a point. In pong, we get a reward of +1 if we score against the opponent and -1, if they score against us. The goal of the models is to increase the average number of points per game. The first model was developed using Andrej Karpathays' (Head of AI at Tesla) article titled **Deep Reinforcement Learning: Pong from Pixels**. The main steps for constructing this first policy gradient model are:

1. **Building the Neural Network (NN)**
   This modified version of the code uses keras on tensorflow to build the sequential dense layer network with 200 nodes with the relu activation function, and a 1 node output layer with sigmoid activation that will give the probability of the action being between 0 and 1. The closer the probability is to 1, the more likely the action was 'up' and the closer it is to 0, the more likely it was "down". The Adam optimizer was used instead of RMSprop and the learning rate was adjusted to 0.001.

2. **Preprocessing the Image**
   The input image from the game was initially downscaled to 80x80. However, upon inspection, more downscaling could be done to prevent the NN from having to learn where the ball is not playing. The final preprocessed frame dimension was 75x80. Additionally, the difference between the frames was calculated to capture motion and then fed into the NN.

3. **Defining the Discount Reward**
   The discount reward, also known as the value function, allowed us to adjust the weights in the neural network to put a higher weight on the action that led to a success and a lower weight on the action that did not, so the network is learning which action provides the best results when it is given a frame to assess.

4. **Defining how to take an Action**
   To decide the action, the model uses the NN and a random generator (uniformly distributed) to generate a probability of the actions "down" and "up". At first, it is exploration, but as the NN gets better, exploitation becomes higher and can cause the model to get stuck at a local optima.

After training this first model for 2,410 games, the average score was -18. This model took almost 48 hours to run and did not improve very much from the starting average score of -21. With more processing speed and time, this model could have probably been trained to eventually beat the game. Because of the poor performance, we decided not to play any games with this model.

Approach 2

The second policy gradient model is adapted from Daniel Mitchells' Reinforcement Learning with Memory Buffers lecture. The discounted rewards and the preprocessing steps are the same as the first policy. The framework for the model is as follows:

1. A NN with four hidden layers, two convolutional layers (activation = relu), a flatten layer and one dense layer. The output layer is also a dense layer with three nodes and a softmax activation function. Unlike the previous model, there are three actions here, "up", "down" and "noop" hence the three nodes in the output layer. The optimizer is RMSprop and the learning rate is set to 1e-3.

2. Instead of the difference between the frames, four frames of 75x80 dimensions are fed into the NN. The total parameters for this model are 371,539, almost 100,000 parameters

less than the model presented in the original code. The expectation was that the reduction would make the NN learn faster.

3. The action to take is determined by the NN producing the probability used in randomly choosing the actions.

4. A memory buffer was added to decrease the correlation between the states. The parameters set were to keep 200,000 frames in the buffer. The idea is that when we train, we will grab an assortment of frames from different games in the memory and then once the 200,000 limit is reached it will periodically remove some old games from the memory.

After training this model for 5000 games, the average score was -21. This model took about 6 hours to run and did not win a single game or score a single point as shown in **Figure 1.** Despite adding a convolutional neural network, reducing the size of the frames, and including a memory buffer, the model did not learn anything and therefore we decided not to play any games with this model.

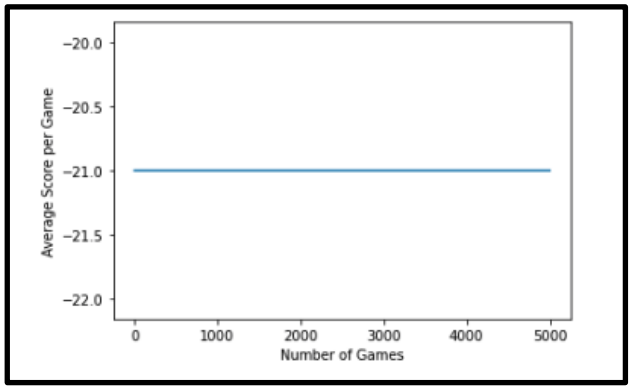

Figure 1: Change in average score per game using memory buffers.

Approach 3

The third policy gradient model is also another version of Andrej Karpathays' model, however the changes made include using a xavier initializer, since it prevents the gradient from exploding or vanishing, instead of the default glorot uniform. We used the Adam optimizer and changed the batch size to 32 instead of 1. We also downsized the image to 75x80 (6000 pixels as the observation size). The learning rate was increased to 0.001 from 0.0006 and at the end, the rewards are standardized so that the state, action, and standardized rewards can be trained on the NN.

After training this model for 10,000 games, which took approximately 16 hours, the average score per game was about -5 (**Figure 2**). This means the trained agent would lose each game but not by much. It still scored an average of 16 points per game. Clearly, with more time and training, this model could potentially win more games than the computer. At some points during training, the agent actually won a few times as shown in **Figure 3**.

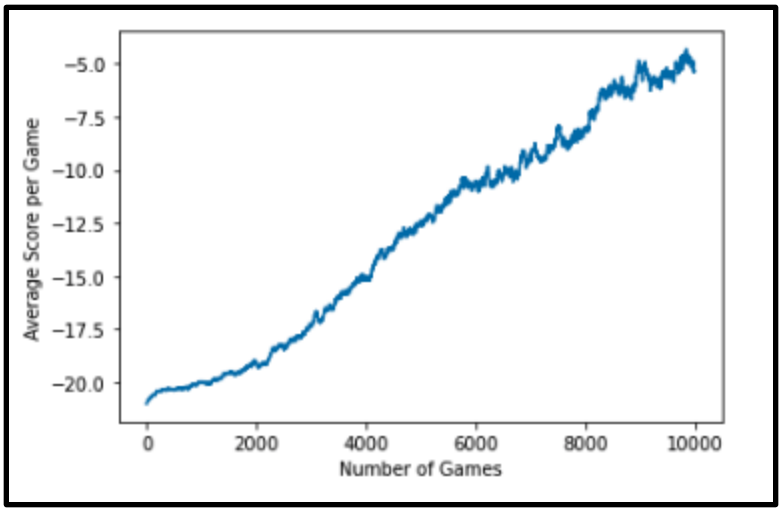

Figure 2: Change in average score per game using third policy gradient model and 10,000 games.

```
Episode no: 9737,  Game round finished, Reward: 1.000000, Total Reward: 3.000000
Total Reward was 3.000000; running mean reward is -4.945768
```

```
Episode no: 9738,  Game round finished, Reward: 1.000000, Total Reward: 7.000000
Total Reward was 7.000000; running mean reward is -4.826310
```

Figure 3: Evidence of the trained agent winning some games during training.

We then played 200 games with this trained model and got an average score of -3 (**Figure 4**). This means the trained AI player scored an average of 18 points per game.

```
scores =[]
choices = []

for each_game in range (200):
    score = 0
    game_memory =[]
    observation =[]
    environment.reset()
    for cycle in range (games_per_cycle):
        if len(observation)==0:
            action = np.random.randint(0,2)
        else:
            action = np.argmax(model.predict(observation)).numpy()[0]
        choices.append(action)

        new_observation, reward, done, info = environment.step(action)
        game_memory.append([new_observation,action])
        score += reward
        if done:
            break
    scores.append(score)
print("Average Score", (sum(scores)/len(scores)))
```

```
Average Score -3.0
```

Figure 4: Average score after playing 200 games using an AI agent trained with 10,000 games.
The third policy gradient was run again but for 5,000 games which took approximately 8 hours. At the end of training, the average score was about -14 (**Figure 5**). This means that on average,

the trained agent was losing 7 to 21 points. Again, there is evidence that it is learning so with more time and training, it would beat the game.

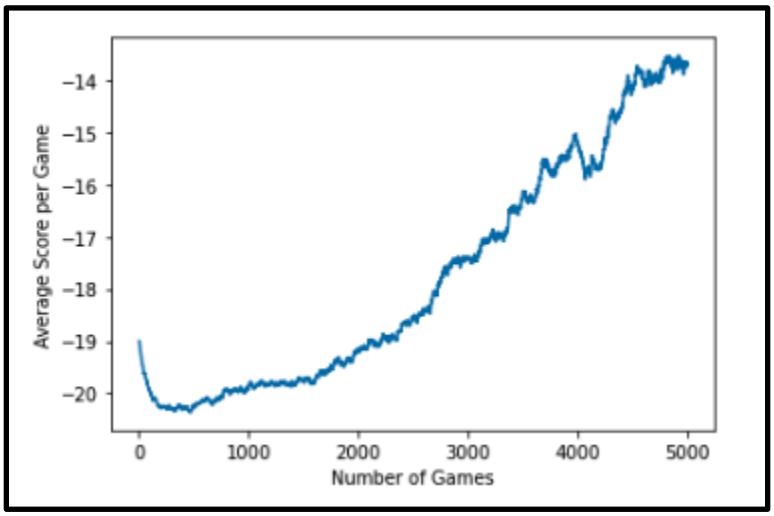

Figure 5: Change in average score per game using third policy gradient model and 5,000 games.

We also played 200 games with this trained model and got an average score of about -9. (**Figure 6**). This means the trained AI player scored an average of roughly 12 points per game.

```
scores =[]
choices = []

for each_game in range (200):
    score = 0
    game_memory =[]
    observation =[]
    environment.reset()
    for cycle in range (games_per_cycle):
        if len(observation)==0:
            action = np.random.randint(0,2)
        else:
            action = np.argmax(model.predict(observation)).numpy()[0]
        choices.append(action)

        new_observation, reward, done, info = environment.step(action)
        game_memory.append([new_observation,action])
        score += reward
        if done:
            break
    scores.append(score)
print("Average Score", (sum(scores)/len(scores)))
```

```
Average Score -9.21
```

Figure 6: Average score after playing 200 games using an AI agent trained with 5,000 games.

Video Pinball

The next game we attempted to learn is Video Pinball (**Figure 7**). Video Pinball was a game released for the Atari 2600 video game console in 1980. The goal is to score as many points as possible. Similar to the pong game, we use the actions available through the gym package in python.

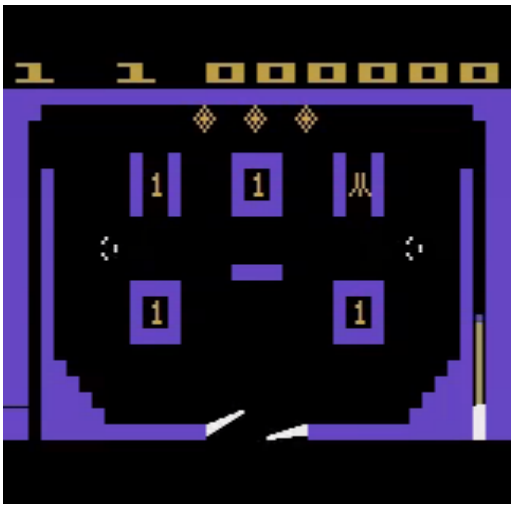

Figure 7: GIF generated from random actions

While deciding how to train the model, we struggled with several techniques and algorithms. Treating this as a learning exercise, we executed the following algorithms:

1. DQN
2. DDQN
3. Advantage Actor Critic (A2C) algorithm - stable_baselines3 package
4. Self-Defined Model with Convolution Layers

Before we started building the models, we cropped the image, and reduced the resolution to focus on the area near the paddles (**Figure 8 and Figure 9)**.

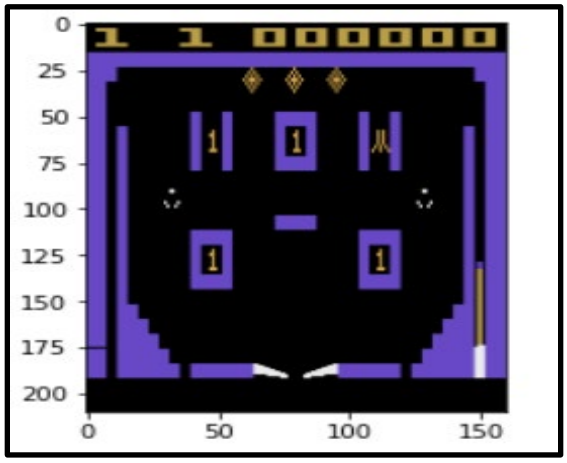

Figure 8: Original Image

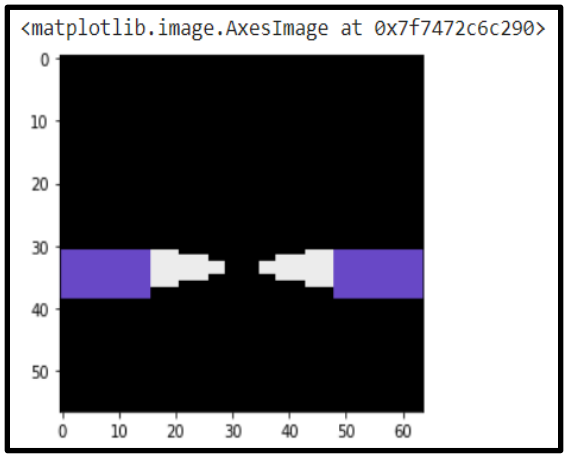

Figure 9: Cropped and Preprocessed Image

There are a total of 9 actions available (**Figure 10**). Of the 9 actions listed below we focus on actions [0,3,4]. Action 1 is used to Fire the ball, and begin the game.

```
env.unwrapped.get_action_meanings()

['NOOP',
 'FIRE',
 'UP',
 'RIGHT',
 'LEFT',
 'DOWN',
 'UPFIRE',
 'RIGHTFIRE',
 'LEFTFIRE']
```

Figure 10: Actions to play in Video Pinball.

Approach 1

We first start off by using the experience replay memory for training our DQN. It stores the transitions that the agent observes, allowing us to reuse this data later. By sampling from it randomly, the transitions that build up a batch are decorrelated. It has been shown that this greatly stabilizes and improves the DQN training procedure.

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output.

Steps involved in reinforcement learning using deep Q-learning networks (DQN):

1. All the past experience is stored by the user in memory
2. The next action is determined by the maximum output of the Q-network
3. The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived from the Bellman equation we have:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

The model is defined with three Dense layers acting as Q layers (**Figure 11**). In the forward pass these layers are passed through several relu activation layers (**Figure 12**). Performs the forward pass on a batch of states to generate the action probabilities. This returns a Q-value tensor of shape [batch_size, num_actions], where each row is a probability distribution over actions for each state.

```
hidden_sz1 = 256
hidden_sz2 = 128

self.Q_1 = tf.keras.layers.Dense(hidden_sz1)
self.Q_2 = tf.keras.layers.Dense(hidden_sz2)
self.Q_3 = tf.keras.layers.Dense(self.num_actions)
```

Figure 11: Layers in the Q-Learning Network

```
l1 = tf.nn.relu(self.Q_1(states))
l2 = tf.nn.relu(self.Q_2(l1))
qVals = self.Q_3(l2)
return qVals
```

Figure 12: Set of Relu activation functions.

We trained this architecture for 200 games, and further played 100 games on this.

Highest Score Achieved	52248
Average Score Achieved (Last 5 games)	29743.1

Table 1: Results from DQN model.

Approach 2

Double DQN uses two identical neural network models. One learns during the experience replay, just like DQN does, and the other one is a copy of the last episode of the first model. The Q-value is actually computed with this second model.

That is the way Q-value gets calculated in Double DQN. We find the index of the highest Q-value from the main model and use that index to obtain the action from the second model. And the rest is history.

Highest Score Achieved	34210
Average Score Achieved (Last 5 games)	18716.4

Table 2: Results from Double DQN model.

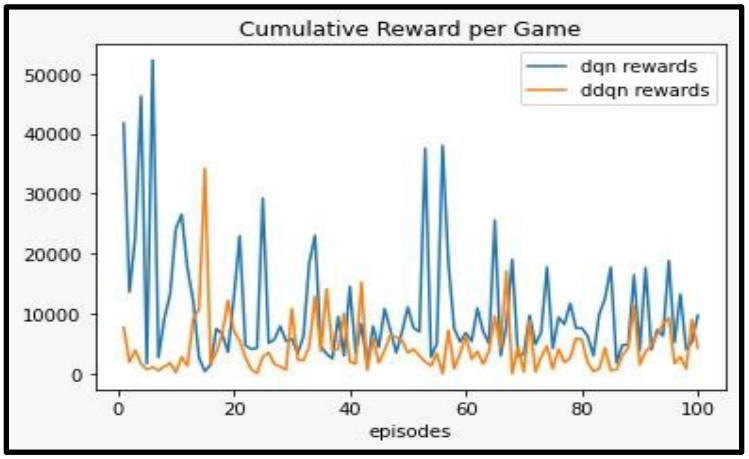

Figure 13: Comparison of Rewards for DQN and DDQN.

As we can see, the rewards are better in the case of DQN, rather than DDQN **(Figure 13**). We believe that this could be because DDQN may require more training as compared to DQN network.

Approach 3

We utilize the stable_baselines3 package, to use the built in A2C architecture to train the model. The Advantage Actor Critic (A2C) algorithm combines two types of Reinforcement Learning algorithms (Policy Based and Value Based) together. Policy Based agents directly learn a policy (a probability distribution of actions) mapping input states to output actions. Value Based algorithms learn to select actions based on the predicted value of the input state or action.

The actor critic algorithm consists of two networks (the actor and the critic) working together to solve a particular problem. At a high level, the Advantage Function calculates the agent's TD Error or Prediction Error. The actor network chooses an action at each time step and the critic network evaluates the quality or the Q-value of a given input state. As the critic network learns which states are better or worse, the actor uses this information to teach the agent to seek out good states and avoid bad states.

While we were able to understand this model to a certain extent, the model took too long to train and we were not able to achieve results for this model.

Approach 4

The last approach we used was a self-defined model with convolutional layers. We attempted to follow the approach used for "pong" with differences in layers and actions to be played. These are the details of the approach, and **Figure 14** is the architecture followed to create the model.

```
def create_model(height,width,channels):
    # we cannot simply have 3 output nodes because we want to put a weight on each node's impact to the objective
    # that is different for each data point.  the only way to achieve this is to have 3 output layers, each having 1 node
    # the effect is the same, just the way TF/keras handles weights is different
    imp = Input(shape=(height,width,channels))
    mid = Conv2D(16,(8,8),strides=4,activation='relu')(imp)
    mid = Conv2D(32,(4,4),strides=2,activation='relu')(mid)
    mid = Flatten()(mid)
    mid = Dense(256,activation='relu')(mid)
    mid = Dense(128,activation='relu')(mid)
    out0 = Dense(1,activation='linear',name='out0')(mid)
    out1 = Dense(1,activation='linear',name='out1')(mid)
    out2 = Dense(1,activation='linear',name='out2')(mid)
    model = Model(imp,[out0,out1,out2])


    return model
```

Figure 14: Convolutional layers network.

We trained this model for over 1000 games, but this is not enough. Training the model further caused the environment to crash. We wish we were able to train this approach further before the deadline. We continue to train this model while we write this report, and hope to get better results. In the meantime, testing this model for 200 games further, resulted in almost all games with a score of 0.

Highest Score Achieved	58
Average Score Achieved (Last 5 games)	Not Stored due to Programming Error

Table 3: Results from Convolutional layers network.

Recommendations & Further Steps

Experimenting with non-dedicated computers can be a time-wasting endeavor. Therefore, our 1st recommendation is to ensure that a dedicated and powerful computer is on hand that can complete these tasks without interfering with productivity. Sometimes, all progress is lost because of unforeseen circumstances and errors, and if other work was on pause to simply wait for these results, the loss would be doubly as bad. As you can see in the image below, things certainly don't always go as planned. This 10,000-game training session crashed at 9,774, in dramatic fashion (**Figure 15**)

Figure 15: Technical difficulties experienced while running 10,000 games.

Our second recommendation is to allow us to provide documentation outlining a beginning approach or base model that everyone can use to build upon. This will lend itself to more consistent uniformity and lead to less time spent overall trying to fix unnecessary problems and conflicting styles.

Our third recommendation is to ensure that our teams train each model without rendering the environment. This will allow training to run much faster, and will save a lot of time. Of course, at times, viewing the environment is necessary to better understand issues that may arise, but it should really not be done otherwise.

Finally, our fourth recommendation is to allocate enough time for us to get the results that are required. Allowing the dedicated machine to train the model for as long as possible could make an impactful difference, and cannot be ignored. According to our research, there are times when

there is a “top of the hill” moment during training. That means that for a long time it could seem like the code is not working as desired, but suddenly, when the training process reaches the “top of the hill”, the desired results start rolling in and improvements increase tenfold. The main point here is that giving up too soon could be more costly than not.

Moving forward, we are excited to discover how we can further apply what we’ve learned together during this project. The approaches and strategies seem limitless when attempting to teach a computer how to play a game. We are now more aware than ever of the countless applications of this concept, and the clear implications it has for the future of our company. Furthermore, we would recommend that the company invest in hiring a RL expert to develop these models further to realize its true benefits and potential.

Links to References

1. https://towardsdatascience.com/intro-to-reinforcement-learning-pong-92a94aa0f84d
2. http://karpathy.github.io/2016/05/31/rl/
3. https://github.com/omerbsezer/PolicyGradient_PongGame
4. asmitnayak/pinheads at 05b415b971478b955de7e1692f40603ab0603772
5. hltung/double-deep-q-learning