TEXAS McCombs | MASTER OF SCIENCE IN BUSINESS ANALYTICS

Reinforcement Learning

REINFORCEMENT LEARNING REPORT

Project 3

as part of **Optimization II**

Prepared By

Group 11:	**EID**
Ian Arzt	iga246
Brett Bartol	bjb3972
Mahika Bansal	mb62835
Prakhar Bansal	pb25834

Submitted to

Daniel Mitchell

May 4, 2022

Table of Contents

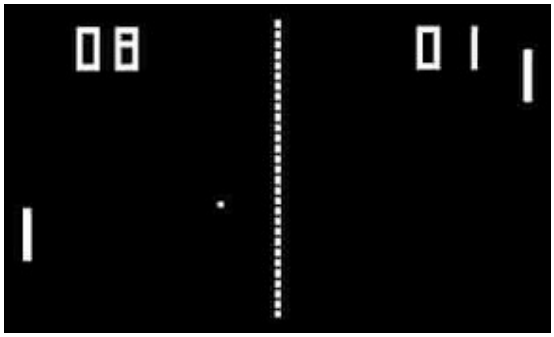
1. Background

Reinforcement learning has become a popular technique in the machine learning realm, alongside the other two paradigms of supervised and unsupervised learning. Popularized in the gaming industry, specifically with the classic game of Atari Pong, Reinforcement Learning has expanded to other use cases as well including robotics, finance, and traffic control. The concept of reinforcement learning deals with maximizing a specified reward through desirable actions while avoiding undesirable actions. This is accomplished by first initializing a random policy that takes an action. This action is then implemented into the environment. The next component in reinforcement learning is evaluation of the action taken. This evaluation is measured in a cumulative reward in which the action taken is attempting to maximize. Eventually, we want to train our actions to perform more positive actions than negative actions in order to effectively maximize the reward.

There are several popular methodologies utilized in Reinforcement Learning, each with their own unique way to approach a problem. The first is Q-Learning where the input into the neural network is the state and the output is the value function for each actions taken. The action with the highest value function is then chosen. Deep Q-Learning is known as a State Action Reward State Action, SARSA, algorithm. The next intriguing methodology are Policy Gradients. Instead of explicitly optimizing the value function done through a regression neural network, Policy Gradients use a classification model inputting frames and outputting different probabilities for each action being the optimal one. An issue with Q-Learning and Policy Gradients is the multicollinearity of the states and inputs into the neural network. To solve this problem, memory buffers were created where many games are kept in memory and the model will use random assortments of frames from a variety of games in memory. The final methodology to discuss is those that use an Actor-Critic strategy. The Actor-Critic method combines Q-Learning and Policy Gradients. It uses Policy Gradients to choose the action while it uses Q-Learning to evaluate whether the action chosen is positive or negative. Each of these methods have their own advantages and disadvantages. As seen in this project, we implemented a variety of these strategies for the two environments.

2. Project Overview

We work for a video game company, that is thinking of utilising reinforcement learning to automate opponents' actions, i.e., computer response against the player. For this project, we'll first try working with RL for 2 classic games: Atari Pong and Activision Skiing. The images below display a sample of each game.



Atari Pong

Activision Skiing

Our team's hope after the project is to implement reinforcement learning in a variety of games for our company. We will be able to implement similar strategies as we have utilized in these two games for future projects that our company wishes to pursue. Throughout this report, we will dive into our implemented strategies for both Pong and Skiing.

3. Game 1 – Pong

About the game

Pong is one of the earliest two-dimensional video games and is based on table tennis. The idea of pong was first conceived by Atari founder Nolan Bushnell and later developed by Allan Alcorn in 1972. In this, a player controls an in-game paddle to hit the ball by moving it across the screen.

The game has 3 controls which is NOOP, Up and Down.

Model Structure

Our team was able to try a few different models to see what would perform the best. We started out testing a model that calculated all the values that we though could be gained by directly inputting frames in. This model took ball height, ball position, previous ball height, previous ball position, ball direction, ball speed, paddle height, previous paddle height, paddle direction, opponents paddle height, opponents previous paddle height, opponents paddle direction. We tried a introducing a few different layers and regularization parameters, but nothing seemed to work well. We then moved on to using frames directly as an input. Every frame we inputted into the model was run through the prepro function which changes the ball and paddle values to a 1 and everything else to a 0. It also reduces the frame size to 80x80 and when we directly inputted frames into our model, we always fed the frames through prepro first. We tried using the 4 most previous frames as in input to a few of our models which would be a 4x80x80 input.

```
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)
    out0 = Dense(3,activation = 'softmax')(mid)
    model = Model(imp, out0)

    model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-4), loss = 'sparse_categorical_crossentropy')

    return model
```

We also tried summing up the 3 previous frames. The way we did this was to take the current frame, multiply the previous frame by .5, and then take the frame before that multiply it by .25 and sum all three together.

```
feed = np.array(frame_array[-1]) + .5 * np.array(frame_array[-2]) + .25 * np.array(frame_array[-3])
feed = feed.reshape(-1, 1, 80, 80)
```

This model input size is 1x80x80. We did this so that the model input could have the same amount of information in one third of the data points. We multiplied the previous frames by constants so that the model can still differentiate between the different frames. This is the input that we ended up going with.

```
def create_model():
    imp = Input(shape = (1, 80, 80))
    mid = Flatten()(imp)
    mid = Dense(32, activation = 'relu')(mid)
    mid = Dense(32, activation = 'relu', kernel_regularizer = tf.keras.regularizers.L1(0.005))(mid)
    mid = Dense(16, activation = 'relu', kernel_regularizer = tf.keras.regularizers.L1(0.005))(mid)
    out0 = Dense(3, activation = 'softmax')(mid)
    model = Model(imp, out0)

    model.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 1e-4), loss = 'sparse_categorical_crossentropy')

    return model
```

While we tried a few different model architectures, the one we ended up going with was to immediately flatten the input and feed it through 4 dense layers. The number of nodes on the layers was 32, 32, 16, 3. All dense layers used a ReLu activation function except for the last one which used a SoftMax.

Training

We used a few different methods for training our model. To start our training, we played 300 games using the up/down strategy described in homework 3.

```
ball_height = np.where(pix[:, 10:70] == 1)[0].mean()
padel_height = np.where(pix[:, 71] == 1)[0].mean()

action = 0
if ball_height != np.nan:
    if ball_height > padel_height:
        action = 2
    elif ball_height < padel_height:
        action = 1
```

We fed the results of these games into our model with the reasoning being that we want the model to see games where we scored a few points so it could potentially learn faster. We then set a degrading epsilon. At game 301, we started with an epsilon of .6 and after each game reduced it by .001 until it reached .1. We left epsilon at .1 until game 3000 and then turned it off.

```
if game < 300:
    use_model = False
    epsilon = 0
else:
    use_model = True
    epsilon = max(.6 - .001 * (game - 300), .1)

if game > 3500:
    epsilon = 0
```

The epsilon allows our model to see some sort of randomness and learn new things. We also tried a few variations of adding rewards. We tried adding a positive .5 reward for hitting the ball and a negative .5 reward for the opponent returning the ball. We tried turning them on and off at different intervals, but they did not seem to be much help.

We used a few different methods for training our model. To start our training, we played 300 games using the up/down strategy described in homework 3. We fed the results of these games into our model with the reasoning being that we want the model to see games where we scored a few points so it could potentially learn faster. We then set a degrading epsilon. At game 301, we started with an epsilon of .6 and after each game reduced it by .001 until it reached .1. We left epsilon at .1 until game 3000 and then turned it off. The epsilon allows our model to see some sort of randomness and learn new things. We also tried a few variations of adding rewards. We tried adding a positive .5 reward for hitting the ball and a negative .5 reward for the opponent returning the ball. We tried turning them on and off at different intervals, but they did not seem to be much help.

Initially we tried just training the model on each game we played but quickly switched to using a memory buffer. We adjusted the memory buffer code that we were given by changing the probability of choosing a frame with a positive reward to 25 from 5. This means that when randomly choosing frames from our memory buffer, the positive

reward frames are 25 times more likely to be chosen than frames with negative rewards. We upped this value because of the initial large disparity of negative scores relative to positive scores. We also multiplied the positive rewards by a factor of 5 before we fed our rewards into the sample weight parameter so that positive rewards would hold more weight when training our model.

```
for grab in range(size_grab):
    rewards[grab] = buffer['rewards'][which_choose[grab]]
    actions[grab] = buffer['actions'][which_choose[grab]]
    frames[grab] = list(buffer['frames'][which_choose[grab]] + .5 * buffer['frames'][which_choose[grab] - 1] + .25 * buffer['frames'][which_choose[grab] - 2])

current_frames = current_frames.reshape(-1, 1, 80, 80)
rewards = np.array([5 * x if x > 0 else x for x in rewards])
mod.fit(current_frames, actions, epochs = 1, steps_per_epoch = nbatch, verbose = 0, sample_weight = rewards, use_multiprocessing = True)
```

Results

Overall, we tried quite a few methods for learning Pong, and none of these results were able to produce any meaningful results. Surprisingly, none of the models were able to score any points after watching a few hundred games of using the move towards the ball strategy, and the only times that the model was able to score a point can be attributed to the epsilon parameter adding enough randomness so the model would score by luck. If we had created a model that showed any promise in working, we would have been able to try messing around with changing up the reward system. Early on in training we would have liked to add a half a point reward for hitting the ball, and then removing the rewards after our model learned to consistently score points. After the model trained to that point, we wanted to add in a negative half point reward for when the opponent returned the ball. The thought behind this was that if our model gained the ability to hit the ball consistently, it would then learn to hit the ball in ways where the opponent could not return it. The theory is at that point, our model would change from not trying to lose to trying to win the game.

4. Game 2 – Skiing

About the game

Skiing is another one of the popular Atari games developed by Activison. The open AI has provided an environment through gym package in python to apply Reinforcement Learning and train a model which can effectively learn to play and eventually win Skiing.

It is a single player game where the task is to ski down the slope and pass between the poles on the path. The player must ensure that they don't get stuck in the trees and obstacles in the path. The winner is a player who passes all the poles in the least amount of time.

There are three actions in the game: NOOP, Right and Left which corresponds to 0,1,2 using gym environment. The original image in the game is of (210,160,3) size.

```
env = gym.make("Skiing-v0")
```

```
env.unwrapped.get_action_meanings()
```

```
['NOOP', 'RIGHT', 'LEFT']
```

```
list(range(env.action_space.n))
```

```
[0, 1, 2]
```

```
raw_pixels = env.reset()
raw_pixels.shape
```

```
(210, 160, 3)
```

Model Structure

As we iterated through the results of Pong, we started directly with training the model using the input image. We created multiple iterations of model to effectively train the Skiing model with the highest score.

To produce an input for the game, we used the same pre-processing structure of reducing the image size to 80X80 which speeds up the training process. For Skiing however, we didn't erase the background completely to ensure there is differentiation between poles and trees that the model should understand with successive iterations.

We also tried processing an image with all the three RGB channels, but it took the model too long to train and didn't give any satisfactory results.

```
mod.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 80, 80, 6)]       0

 conv2d_2 (Conv2D)           (None, 19, 19, 16)        6160

 conv2d_3 (Conv2D)           (None, 8, 8, 32)          8224

 flatten_1 (Flatten)         (None, 2048)              0

 dense_3 (Dense)             (None, 256)               524544

 dense_4 (Dense)             (None, 64)                16448

 dense_5 (Dense)             (None, 3)                 195

=================================================================
Total params: 555,571
Trainable params: 555,571
Non-trainable params: 0
```

To create the network architecture, we decided to go with CNN network with more layers because of the complexity of the game involved. We iterated with more convolutional layers, adding extra kernels with different strides. We also added more dense layers and decreased the units in logarithmic scale to ensure model learn all essentials of the game.

After multiple iterations and looking at an architecture which is not too complex i.e. it does not take too much time to train and has sufficient power to train the model was selected. We added one extra layer of Feed Forward layer with 64 neurons. The final model has 555,571 trainable parameters.

Model Code

```
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)
    mid = Dense(64,activation='relu')(mid)
    out0 = Dense(3,activation='softmax')(mid)
    model = Model(imp,out0)


    return model
```

Training

With the final model architecture selected, we focussed on training the model with different iterations. We iterated with total frames that will be used in training. From 2,4 6, 10 we tried different variations. Expected the best results were from 10 and then 8 but these values were causing a lot of time to train the model for 5000 games. To optimize for the best results and least amount of time to train, we selected 6 frames for training.

To prevent the model from crashing due to memory leak issue in TensorFlow, we bought Colab Pro to enable it to run GOY with High Ram. Also, we made changes to code structure wherein only compiling the model after creating it and creating another dummy model to add weights and biases to it. We also cleared the session using Keras backend package.

The given changes were beneficial in terms of processing time. From an average of 7 seconds to train one game, we were able to train it within 2 seconds.

```
frames_to_net = 6          # how many previous frames will we feed the NN
possible_actions = [0,1,2]
mod = create_model(80,80,frames_to_net)
mod.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-6),loss='sparse_categorical_crossentropy')
mod.call = tf.function(mod.call,experimental_relax_shapes=True)
```

To create a dataset for training model, we initially let the game play for 50 games without any training. Like Pong, we also initially selected the reward from 5 to 25 to see if any improvement is done. But there was no improvement by changin the reward value.

```
    if game >= warmupgames:
        prob = np.ones(len_buff)
        prob[np.array(buffer['rewards']) > 0] = 5.0
        prob /= np.sum(prob)
        which_choose = np.random.choice(len_buff,size=nframes,replace=False,p=prob)

        for grab in range(nframes):
            rewards[grab] = buffer['rewards'][which_choose[grab]]
            actions[grab] = buffer['actions'][which_choose[grab]]
            for f in range(frames_to_net):
                if grab-f > 0:
                    current_frames[grab,:,:,f] = buffer['frames'][which_choose[grab]-f].copy()
```

Results

With all the changes, we trained the model for 1k, 2.5k and 5k iterations. The results did not improve with the changes or number of games. Also because of memory leak issue even with colab pro, we could never train the model for a full 5k iterations.

The below results show a snapshot of first 10 games while training the model. The scoring in Pong Atari is done in a such a way that every missed pole is awarded some negative points and taking too long to complete the game is awarded a huge negative score. We can see from the results that the score was not improving with different iterations.

```
0 -30000.0 8.665698766708374 6008
1 -30000.0 8.597562551498413 12030
2 -30000.0 8.33570647239685 18050
3 -30000.0 8.868283748626709 24103
4 -30000.0 8.595168113708496 30087
5 -30000.0 8.313992023468018 36064
6 -30000.0 8.86296272277832 42079
7 -30000.0 8.26844048500061 48100
8 -30000.0 8.222976922988892 54097
9 -30000.0 8.288319826126099 60114
10 -30000.0 8.535723209381104 66125
11 -30000.0 8.339668989181519 72144
12 -30000.0 8.359014987945557 78137
13 -30000.0 8.308887720108032 84179
14 -30000.0 8.346257448196411 90201
15 -30000.0 8.520092725753784 96213
16 -30000.0 8.36451768875122 102242
17 -30000.0 8.34120774269104 108235
18 -30000.0 8.379099607467651 114254
19 -30000.0 8.439885377883911 120268
20 -30000.0 8.385426521301?7 126285
```

To check what was happening, we ran the model for a random 100 iterations and saved the video for of the game to check what is happening. The investigation revealed that the game is not able to learn going forward down the slope and gets stuck in one art of the screen which is why the score doesn't improve with the games.

Time Stamp 1

Time Stamp 2

Time Stamp 3

5. Conclusion & Recommendations

For both Pong and Skiing, our team attempted many different strategies and methods. Our team focused on refining the neural network to better capture the optimal actions. We attempted several types of neural networks including ones that do not include convolution layers as well as different optimizers and learning rates. Once we started to

see slight improvements within our games, we decided to move into the actual strategy of how and what to train the model with. With Pong, we adjusted the reward strategy by adding a reward for hitting the ball with the paddle. We also used a weighted equation, which took the most recent frame, the previous frame, and the next previous frame into account. We also trained our Neural Network using a ball tracking strategy for the first 300 games in hopes of the agent being able to extract ball patterns and positive actions early on. After a variety of different trial and errors runs, which were emphasized in the sections above, we could not obtain a winning score in either game.

Our recommendations for the future of this project and objective is to attempt a different method of reinforcement learning entirely. Of note, we used a memory buffer in both of our games. Perhaps, we would like to try an Actor-Critic method to combine Q-Learning and Policy Gradients. As we saw throughout our process in this project, we struggled with Google Colaboratory in general with RAM issues, runtime timeouts, data leaks, and GPU usage errors. Our team would like to avoid these by having a more sought-after plan of what to attempt and how to attempt it. Instead, we had a lot of different ideas on how to play and win these two different games. Our team decided to try all these strategies and methods instead of coming up with a proper plan. We recommend against this as it cost us time on the backend as our models and methods did not perform as intended. As is though, our team believes that there are significant results from our trial and error. We now comprehend what does not work and given another task associated with these games, our team believes we would accomplish our goals swiftly.

Further, this project has helped us understand what avenues RL can be applied to, and using these models, we can try and develop other games, after the required modifications.