Optimization 2 - Project 3

Reinforcement Learning

Pong & Asteroid Games

Analyzed By:

Rohan Garg (rg47856)

Connor Gilmore (cmg5829)

Mario Gonzalez (mxg76)

Brandt Green (bwg537)

Overview:

The goal of this project is to explore Reinforcement Learning (RL) on some simple Atari games as the company we work for is considering using RL to automate opponents' actions when playing against the person playing the game. We first started by developing an RL strategy to play pong, and then afterwards we chose to automate the opponent's actions for the Asteroid game.

Outline:

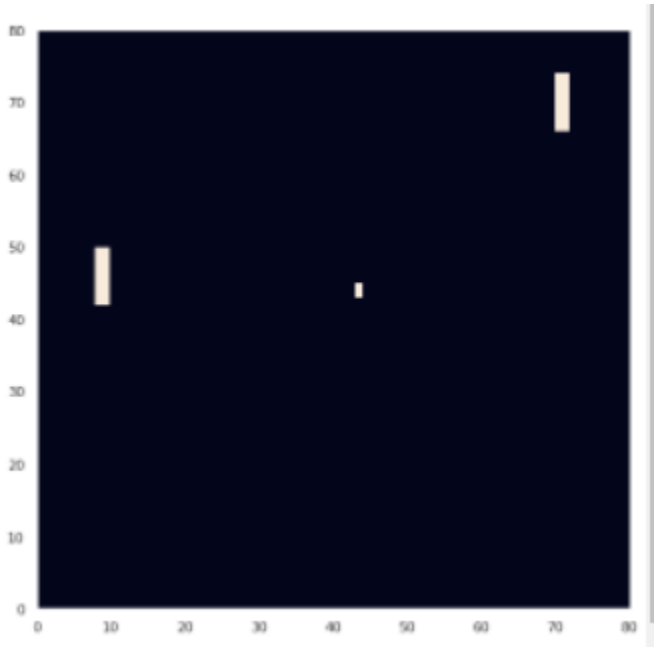
First, we began by building a reinforcement learning player that plays pong and tries to perform as well as possible during training. We had the options of using several strategies to improve our performance including actor/critic, memory buffer, removing frames without a ball, playing the same action for 4 frames in a row, or using a linear annealed strategy. Whatever strategy we used, we needed to continue training the RL algorithm until it got as close as possible to obtaining a win. Afterwards, we then used the trained RL algorithm to play 200 games and calculated our average score from those 200 games (which needed to be a winning score).

Moving on, we then trained another RL algorithm for no more than 5,000 games using the same network structure as earlier, and subsequently played 200 games with that neural network. Again, we calculated our average score over those 200 games.

Once we were finished doing this for the pong game, we moved on to creating a different reinforcement algorithm that can play the Asteroid game. With this, we again had the option of modifying the reward, but this time, the reward is different since it is a different game altogether. At the very end, we were asked by our boss to write a report on the effectiveness of Reinforcement Learning, as he is considering using RL to automate players.

Pong Approach:

Pixel representation of a game of pong



This game is simple enough: we just need to get the ball past the opponent's paddle 21 times before they do the same to us!

There are 3 different possible controls or operations of the Pong game that are listed as follows:

1.) Up
2.) Down
3.) NOOP

There are a variety of techniques and methodologies that can be used to train a reinforcement learning algorithm, but for this game, we decided to use the policy gradient method. The policy gradient method differs from Deep Q-learning as it does not use a value function. Instead, it creates a choice between the various game buttons as a classification problem. Overall, the policy gradient method involves building a neural network whose input is several frames worth of pixels on the screen, and the output layer has 3 probabilities, corresponding to the probabilities for each of the 3 operations: up, down, or do nothing. Because we are dealing with probabilities, we need a softmax on the last layer of the neural network. Also, since this is technically a classification problem, we would normally need to know what the truth is to evaluate how our neural network is performing via a confusion matrix (truth table). However, to

counteract this, we fiddle with the categorical, cross-entropy objective function such that we don't actually need to know what the best button is.

We chose this method primarily because it makes the most intuitive sense to us, and we feel this learning method will be the easiest to explain to other business stakeholders. The general idea of using the policy gradient is: play a bunch of games, then match up actions and the predicted best action. Minimize the difference between these two items, but weight the loss function in proportion to the reward that was received and the distance between the reward and the action.

$$\min \sum_i w_i loss(actual\ action_i, predicted_i)$$

To implement this, we need to generate the weights involved in the formula above. To do so, we utilized the below function. The below function takes a vector of "rewards" which in our case will be a vector of -1s, 0s, and 1s. It then applies a discount factor equal to discount^n where n is the distance between the current frame and the next point scored. This is the vector of weights that we can then provide to our NN in the training process.

```
def discount_rewards(r,delt):
    """Send in an array 'r' of scores and a discount factor 'delt'. This function will return and array of discounted rewards."""
    # take 1D float array of rewards and compute discounted reward
    # gym returns a reward with every single frame.  most of those rewards are 0
    # sometimes they're 1 or -1 if we win or lose a point in that specific frame
    # we want non-0 rewards for every frame.
    # so take each frame, figure out if we eventually won the corresponding point or not
    # if so make the reward positive, if not negative
    # but more recent actions (relative to the frame where the point is awarded) are more
    # impactful to the score that frames a long time ago, so discount rewards...

    nr = len(r)
    # we want to change all those zeros into discounted values of the next reward (this is the value function!)
    discounted_r = [0.0]*nr

    for t in range(nr):
        # start at the end
        if r[nr-t-1] > 0: # if you won a point in this frame we want a good reward
            discounted_r[nr-t-1] = 1
        elif r[nr-t-1] < 0: # if we lost the point we want a bad reward
            discounted_r[nr-t-1] = -1
        elif t==0: # this is just for error catching...at t==0 r[nr-t-1] should have already been + or -...
            discounted_r[nr-t-1] = 0
        elif discounted_r[nr-t-1] == 0: # otherwise you want to look at the next reward value and discount it
            discounted_r[nr-t-1] = delt*discounted_r[nr-t]
    return discounted_r
```

The below few sections explain some of the enhancements that were attempted in the training process.

Memory Buffer: We implemented a memory buffer as a part of our training process. A memory buffer is a technique that is used as a way to adjust for the fact that much of the typical training data for this problem is highly correlated. This occurs because our training data consists of sequential sets of frames. But one set of frames will be highly correlated with the next set of frames, as the game setting will only have changed a marginal amount! To account for this, we do not update our neural network using just the frames from the most recent game, but we take a random subset of frames from the past few games. In our case, this means we store the 30,000 most recent sets of frames across recent games, and pick from this group when fitting the neural network. While we would have liked to have had a larger buffer than 30,000, we ran into memory constraints and so settled on this number.

Ball Tracking: We made an attempt to incorporate "ball tracking" into our training process. Ball tracking simply means that our paddle will choose the action that moves the paddle closest to the ball. Originally, we set our training so that our model will choose to track the ball for the first 10% of games, and afterwards, will continue normal training. Our hope was that by training on a strategy that we know is partially successful, the model can jump start the training process. Unfortunately, this approach did not work well at all, and it appears that this strategy leads to the model becoming stuck in a poor state. We tried several different iterations of this strategy, and with each attempt, our ending strategy resulted in a model that consistently produced scores of -21(The worst possible score). Because of this, we decided to eliminate ball tracking from our training process.

Ball tracking function below

```
def move_toward_ball(pixels:np.array) -> int:
    """This function returns the action necessary to move towards the ball."""

    r_pad_y = np.where(pixels[:,71])[0].mean() # Where is our paddle
    ball_p_range = pixels[:,10:70] # What is the range of pixels the ball could be in

    if ball_p_range.sum() < 1:
        action = 0 # If the ball isn't in our range, we don't do anything
    else:
        ball_col = np.where(ball_p_range.sum(axis=0))[0][0]
        ball_y = np.where(ball_p_range[:,ball_col])[0].mean()

        if ball_y > r_pad_y:
            action = 3
        elif ball_y < r_pad_y:
            action = 2
        else:
            action = 0

    return action
```

Linear Annealing: For a time, we utilized a decreasing epsilon approach for model training. This means that we chose a probability with which to possibly choose a random action at each decision point. In theory, this facilitates exploration by the model to allow you to explore new strategies and find larger rewards. We chose an epsilon of 80% for the first 5% of games, which decreased rapidly to a lower level of 1% by the time 20% of total games were played. Using this strategy did not seem to work any better than not using this approach and therefore we dropped it for the sake of model simplicity.

Architectures: We experimented significantly with a variety of different NN architectures. There are many hyper parameters that can be tuned, but the key ones we focused on were: number of layers, learning rate, regularizers, activation functions, and the inclusion or absence of convolution layers (along with max pooling). The final model we chose is pictured below, but the primary reason this model was chosen was because of its feasibility: none of the other models made it past 500 games without crashing.

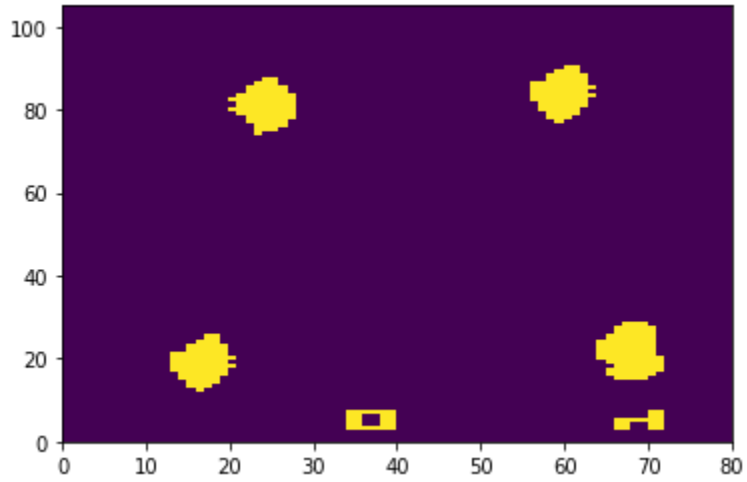
Final, chosen architecture:

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

    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=1e-6),loss='sparse_categorical_crossentropy')

    return model
```

Frame Chunks & Momentum Decision Making: We made one final enhancement to our training process: frame chunking with momentum decision making. Frame chunking means that instead of treating overlapping sets of 4 frames as a data point, we only grabbed the frames in chunks of four. This is another approach that helps to reduce the correlation in training data. At the same time, we decided that when an action is chosen, it will persist for 4 frames, hence the model has a 4-frame momentum. This makes the paddle more human-like and less jerky. To implement these changes, we simply took a snapshot every four frames that capture the action that is made for the following four frames and the four frame input that was used to make that decision. Additionally, we needed to associate a reward with this input-action combo so we sum up the rewards for the next four states and link that to the snapshot. This combination of frame chunks, actions, and total reward is what is fed into our NN for training.

Training: The policy gradient approach, while intuitive, is incredibly slow to train. Our model was trained for approximately 30,000 games. The training process began at a rapid pace, with each game and the follow up model tuning taking an average of 1.2 seconds per iteration, but as the model grew more sophisticated, and perhaps because of memory leaks, each iteration began taking longer. By the end, each loop was taking an average of 10 seconds. Clearly, scaling the model is an issue. This approach is certainly not the most efficient way to accomplish our task.

Asteroids:



For the Asteroids atari game, we utilized one of the more popular methodologies for Reinforcement Learning: the Deep Q-learning approach. This method approximates the value function using a neural network whose input is several frames worth of pixels on the screen, and whose output layer contains “x” nodes, corresponding to the value functions for each of the “x” operations of the particular game.

For this problem, our specific neural network was built with 2 convolution layers followed by a flattening layer and 14 dense layers, corresponding to the 14 different possible operations. The convolution layers were used so that the model could understand that the multiple asteroids on screen were the same type of objects.

Below is a screenshot of the neural network we created:

```
def create_model(height,width,channels):

    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(1,activation='linear',name='out0')(mid)
    out1 = Dense(1,activation='linear',name='out1')(mid)
    out2 = Dense(1,activation='linear',name='out2')(mid)
    out3 = Dense(1,activation='linear',name='out3')(mid)
    out4 = Dense(1,activation='linear',name='out4')(mid)
    out5 = Dense(1,activation='linear',name='out5')(mid)
    out6 = Dense(1,activation='linear',name='out6')(mid)
    out7 = Dense(1,activation='linear',name='out7')(mid)
    out8 = Dense(1,activation='linear',name='out8')(mid)
    out9 = Dense(1,activation='linear',name='out9')(mid)
    out10 = Dense(1,activation='linear',name='out10')(mid)
    out11 = Dense(1,activation='linear',name='out11')(mid)
    out12 = Dense(1,activation='linear',name='out12')(mid)
    out13 = Dense(1,activation='linear',name='out13')(mid)
    model = Model(imp,[out0,out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out1

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

    return model
```

The 14 different possible controls or operations of the Asteroid game are listed as follows:

4.) NOOP	6.) DOWN	11.) LEFTFIRE
5.) FIRE	7.) UPRIGHT	12.) DOWNFIRE
6.) UP	8.) UPLEFT	13.) UPRIGHTFIRE
7.) RIGHT	9.) UPFIRE	14.) UPLEFTFIRE
8.) LEFT	10.) RIGHT FIRE	

When it comes to Deep Q-Learning, by default, the model wants to take the predicted value for each output node, the truth for each of those output nodes, subtract them, and then square them.

TF treats the objective as

- $\sum_{data} \sum_{output\ nodes} (predicted - truth)^2$

However, we never actually play all 14 buttons at once; instead we only play 1 button at a time, corresponding to 1 node. So, to account for this, in the objective function, we can change the sum of squared errors to the weighted sum of squared errors.

- $\sum_{data} \sum_{output\ nodes} w_{d,o}(predicted\ - truth)^2$

In this case, for a particular output of a particular frame, if we actually played that 1 action / pushed that 1 button, then we will multiply that specific weight by 1, whereas for the actions/buttons that we didn't push, their weight will be multiplied by 0.

```
y0 = np.zeros((nframes,1))
y1 = np.zeros((nframes,1))
y2 = np.zeros((nframes,1))
y3 = np.zeros((nframes,1))
y4 = np.zeros((nframes,1))
y5 = np.zeros((nframes,1))
y6 = np.zeros((nframes,1))
y7 = np.zeros((nframes,1))
y8 = np.zeros((nframes,1))
y9 = np.zeros((nframes,1))
y10 = np.zeros((nframes,1))
y11 = np.zeros((nframes,1))
y12 = np.zeros((nframes,1))
y13 = np.zeros((nframes,1))
```

```
weight0 = np.zeros(nframes)
weight1 = np.zeros(nframes)
weight2 = np.zeros(nframes)
weight3 = np.zeros(nframes)
weight4 = np.zeros(nframes)
weight5 = np.zeros(nframes)
weight6 = np.zeros(nframes)
weight7 = np.zeros(nframes)
weight8 = np.zeros(nframes)
weight9 = np.zeros(nframes)
weight10 = np.zeros(nframes)
weight11 = np.zeros(nframes)
weight12 = np.zeros(nframes)
weight13 = np.zeros(nframes)
```

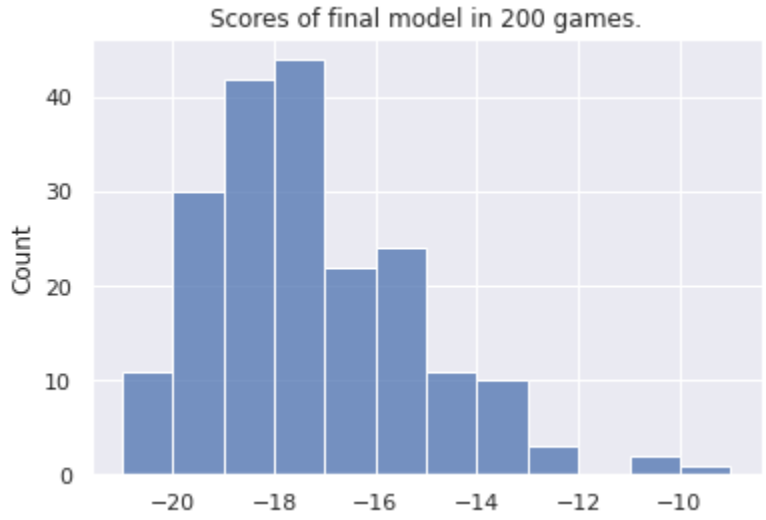
The award was assigned for the number of points scored by destroying an asteroid. Originally, the amount of points scored from destroying an asteroid ranged from 50-200 depending on the size of the asteroid. However, after obtaining sub-optimal results, we tried implementing a different point system in which 1 point is scored if an asteroid was hit regardless of the size, 0 points are scored if no asteroid was hit, and -1 points are scored when the game is lost.

```
if done == False:
  if reward != 0:
    reward_array.append(1)
  else:
    reward_array.append(reward)
else:
  reward_array.append(-1)
```

Final Results:

Pong Results: The Pong Model was tested on 200 games and the results were as follows.

Average score: -17.77
Best score: -9.00



Same Model, Trained on Only 5,000 Games:

We also wanted to test how this model performs with much less training. The model was trained on only 5,000 games and tested again on 200 games. In these 200 games, the average score achieved was -20.48 and the best score was -17. Clearly we can see that our model does improve, but it takes substantial training to get there!

Asteroids Results

While we trained for 10,000 games the performance of the model did not improve by much. By the end of the training the model's best performance was surviving for 11 seconds and scoring 1500 points. This is not as good of results as we would like so we tried to adjust the awarding system. This change in the award system did not create better results as hoped and ended up performing about as well as the last model.

Conclusion:

Our recommendation is that it is beneficial for the company to hire a RL expert to work for the company, if achieving high performance AI in these games will lead to desirable business outcomes. As we have discussed, reinforcement learning can be incredibly successful, but unfortunately there are many difficulties with building models to perform well and the skills necessary to do so are not necessarily the same skills that the typical software expert has. Additionally, with more computational resources to leverage, models can be trained longer to achieve better results, as laid out above. It is clear and obvious that all else equal, the longer the model is trained, the better it will perform. And we were either unable to or limited by time and resources to train any model for longer than about 16 hours. Therefore, we believe an RL expert is necessary to oversee the creation of the models and steer the process.