Project Topic

I wanted to take the project in a different direction than the Neural Networks (NN) we have encountered so far. All of the models we have worked with revolved around using a NN to learn some topic and then predict or create information from that topic.

I have been fascinated with the concept of teaching machines to interact with the world and to learn how to navigate an evironment. This project is an attempt to apply a NN to a robotic process problem and then introduce a new type of NN called deep Q learning to see if it can better handle the challenge.

Specifically, I create a really simple graphical game for the computer to play. The ai has 60 seconds to move the mouse up to 5 pixels per 80 ms and try and reach a randomly generated red ball. The ball randomly moves every 5 seconds if it hasn't been reached yet and there is a point scored each time the AI can get the mouse over to the ball before the ball moves. The play space is large enough that often it is impossible to reach across from one side to the other in the 5 seconds.

The next thing I created was an algorithm that can play the game based solely on moving the mouse closer to the target mathematically. This served two purposes, 1. as a theoretical ideal player to compare the NN to and 2. a data generator for the first NN model.

One last thing to note, the scripts in this notebook are copies of the attached python scripts. Jupyter notebook doesn't work with the mouse control required to play this game when the scripts are run in the cell. However there are calls to run the python scripts in the background that can be invoked to show the game playing.

```
import tkinter as tk
import random
import mouse

class CircleGame(tk.Tk):
    def __init__(self, move_decision_callback, train=False):
        super().__init__()
        # Create the game window
        self.title("Circle Game")
        self.geometry("800x600+400+200")
        self.canvas = tk.Canvas(self, bg="white", width=800, height=600)
        self.canvas.pack()
        self.circle = None
        # Train runs don't have a time limit and the circle only moves when it is clicked.
        self.train = train
        self.create_circle()
        # Stop the game with the Escape key
        self.stop_flag = False
        self.score = 0
        self.canvas.bind("<Button-1>", self.on_click)
        self.bind("<Escape>", self.stop)
        # Circle game can be played with a variety of move decision scripts which is defined here
        self.move_decision_callback = move_decision_callback
        if not train:
            # End game after 1 minutes (60000 ms) if not a training session
            self.after(1000 * 60, self.end_game)

    def create_circle(self):
        # Draw Circle on a random spot
        if self.circle:
            self.canvas.delete(self.circle)
        x = random.randint(40, 760)
        y = random.randint(40, 560)
        self.circle = self.canvas.create_oval(x-20, y-20, x+20, y+20, fill="red")
        if not self.train:
            self.move_circle_timeout()

    def on_click(self, event):
        # Mouse clicks occur when the mouse is in the circle at the end of a move
        x, y = event.x, event.y
        self.score += 1
        self.create_circle()
        if not self.train:
            self.canvas.after_cancel(self.move_timeout)
            self.update_title()
    def move_circle_timeout(self):
        # Set a timer to move the circle during play sessions
        self.move_timeout = self.canvas.after(5000, self.create_circle)  # Move circle every 5 seconds
    def update_title(self):
        # Keep Score in the title bar
        self.title(f"Circle Game - Score: {self.score}")

    def end_game(self):
        # End the game after the 60 second timer on play runs
```

```
        self.canvas.unbind('<Button-1>')
        self.canvas.create_text(400, 300, text="Game Over", font=("Arial", 24), fill="black")
        self.event_generate("<Escape>")
    def move_mouse_and_click(self):
        # Looping script that manages the mouse movement and clicking when the mouse gets to the circle
        if not self.stop_flag and self.focus_displayof() == self:
            circle_x, circle_y = self.canvas.coords(self.circle)[:2]
            circle_x += 20  # Adjust for the circle radius
            circle_y += 20  # Adjust for the circle radius

            # Calculate the absolute position of the circle on the screen
            window_x, window_y = self.winfo_rootx(), self.winfo_rooty()
            target = window_x + circle_x, window_y + circle_y

            # Get a new mouse position from the decider
            current_position, new_position, speed_loss = self.move_decision_callback(target)
            new_mouse_x, new_mouse_y = new_position

            # Move the mouse to the new coordinates
            mouse.move(new_mouse_x, new_mouse_y)

            # Click if the mouse is close enough to the circle
            distance_sq = (new_mouse_x - target[0]) ** 2 + (new_mouse_y - target[1]) ** 2
            if distance_sq <= 64:
                mouse.click()

            # If a data gathering training run
            if self.train:
                save_mouse_position_log(current_position, target, new_position, speed_loss)

            self.after(80, self.move_mouse_and_click)  # Call this function every 80 milliseconds

    def stop(self, event):
        self.stop_flag = True

def save_mouse_position_log(current_position, target_position, new_position, speed_loss):
    with open("train data.csv", "a") as file:
        log_entry = f"{current_position[0]},{current_position[1]},
                      {target_position[0]},{target_position[1]},
                      {new_position[0]},{new_position[1]},
                      {speed_loss}\n"
        file.write(log_entry)
```

Algorithm Decision Maker

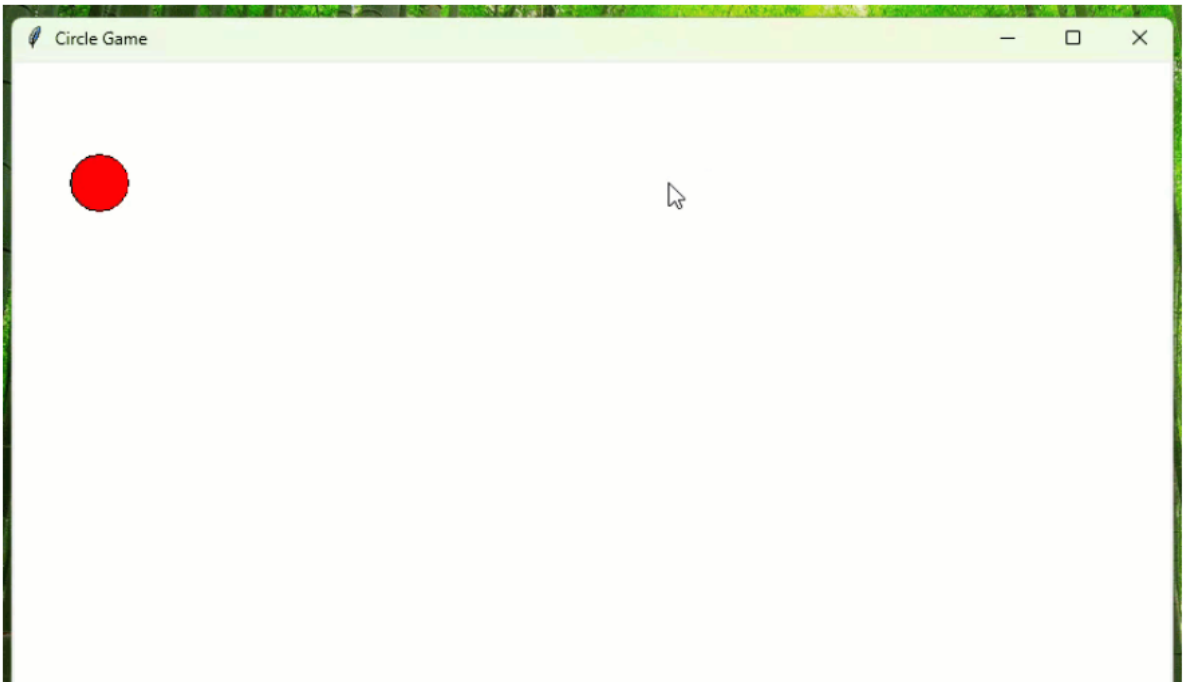
```
import mouse
from circlegame import CircleGame

def mouse_move_decision(target):
    absolute_x, absolute_y = target
    # Calculate the direction vector and normalize it
    mouse_x, mouse_y = mouse.get_position()

    # Move the mouse up to 5 pixels closer to the circle
    new_mouse_x = move_closer(absolute_x, mouse_x)
    new_mouse_y = move_closer(absolute_y, mouse_y)

    # Save the mouse position log
    current_position = (mouse_x, mouse_y)
    new_position = (new_mouse_x, new_mouse_y)

    speed_loss = calculate_loss(current_position, new_position, target)

    return current_position, new_position

def move_closer(target, input_number):
    # A strict move command that returns the target when close enough
    distance = abs(target - input_number)
    if distance <= 5:
        output = target
    # Or returns a number 5 closer to the target
    elif input_number < target:
        output = input_number + 5
    else:
        output = input_number - 5
    return output

```

```python
def calculate_loss(current_position, new_position, target):
    current_x, current_y = current_position
    new_x, new_y = new_position
    target_x, target_y = target

    # Calculate the loss in x direction
    x_loss = 0
    if abs(target_x - current_x) > 5:
        if (new_x - current_x) * (target_x - current_x) > 0:
            x_loss = 1 - abs(new_x - current_x) / 5
        else:
            x_loss = 1 + abs(new_x - current_x) / 5
    elif new_x != target_x:
        x_loss = abs(new_x - target_x) / 5

    # Calculate the loss in y direction
    y_loss = 0
    if abs(target_y - current_y) > 5:
        if (new_y - current_y) * (target_y - current_y) > 0:
            y_loss = 1 - abs(new_y - current_y) / 5
        else:
            y_loss = 1 + abs(new_y - current_y) / 5
    elif new_y != target_y:
        y_loss = abs(new_y - target_y) / 5

    # Calculate the overall loss
    loss = (x_loss + y_loss) / 2

    save_mouse_position_log(current_position, target, new_position, loss)

    return loss

def save_mouse_position_log(current_position, target_position, new_position, loss):
    with open("mouse_position_logs.csv", "a") as file:
        log_entry = f"{current_position[0]},{current_position[1]},{target_position[0]},{target_position[1]},
{new_position[0]},{new_position[1]},{loss}\n"
        file.write(log_entry)

if __name__ == '__main__':
    game = CircleGame(mouse_move_decision, train=False)
    game.after(1000, game.move_mouse_and_click)  # Start moving the mouse and clicking after a 1-second delay
    game.mainloop()
```

1]:

```python
# Play this cell to watch the arbitrary decision maker find the red cell

!python algoplayer.py
```

Or have a look at this recording



Data

The trainig data for the first NN was generated from a 27 minute long recording of the alogorithm playing the game. It resulted in 20,000 recordings of the current mouse position, the target position, the new chosen position and the loss calculation.

3]:
```
import csv
train_data = []
with open('train data.csv', 'r') as csvfile:
    csvreader = csv.reader(csvfile)
    for row in csvreader:
        train_data.append([float(x) for x in row])
```

3]:
```
# Training data samples
print('Start X, Start Y, Target X, Target Y, Next X, Next Y, Loss ')
train_data[:20]

# loss is 0 for all the recording because the algorithm chooses moves with no loss by definition
```

```
Start X, Start Y, Target X, Target Y, Next X, Next Y, Loss
```

3]:
```
[[1524.0, 19.0, 1054.0, 578.0, 1519.0, 24.0, 0.0],
 [1519.0, 24.0, 1054.0, 578.0, 1514.0, 29.0, 0.0],
 [1514.0, 29.0, 1054.0, 578.0, 1509.0, 34.0, 0.0],
 [1509.0, 34.0, 1054.0, 578.0, 1504.0, 39.0, 0.0],
 [1504.0, 39.0, 1054.0, 578.0, 1499.0, 44.0, 0.0],
 [1499.0, 44.0, 1054.0, 578.0, 1494.0, 49.0, 0.0],
 [1494.0, 49.0, 1054.0, 578.0, 1489.0, 54.0, 0.0],
 [1489.0, 54.0, 1054.0, 578.0, 1484.0, 59.0, 0.0],
 [1484.0, 59.0, 1054.0, 578.0, 1479.0, 64.0, 0.0],
 [1479.0, 64.0, 1054.0, 578.0, 1474.0, 69.0, 0.0],
 [1474.0, 69.0, 1054.0, 578.0, 1469.0, 74.0, 0.0],
 [1469.0, 74.0, 1054.0, 578.0, 1464.0, 79.0, 0.0],
 [1464.0, 79.0, 1054.0, 578.0, 1459.0, 84.0, 0.0],
 [1459.0, 84.0, 1054.0, 578.0, 1454.0, 89.0, 0.0],
 [1454.0, 89.0, 1054.0, 578.0, 1449.0, 94.0, 0.0],
 [1449.0, 94.0, 1054.0, 578.0, 1444.0, 99.0, 0.0],
 [1444.0, 99.0, 1054.0, 578.0, 1439.0, 104.0, 0.0],
 [1439.0, 104.0, 1054.0, 578.0, 1434.0, 109.0, 0.0],
 [1434.0, 109.0, 1054.0, 578.0, 1429.0, 114.0, 0.0],
 [1429.0, 114.0, 1054.0, 578.0, 1424.0, 119.0, 0.0]]
```

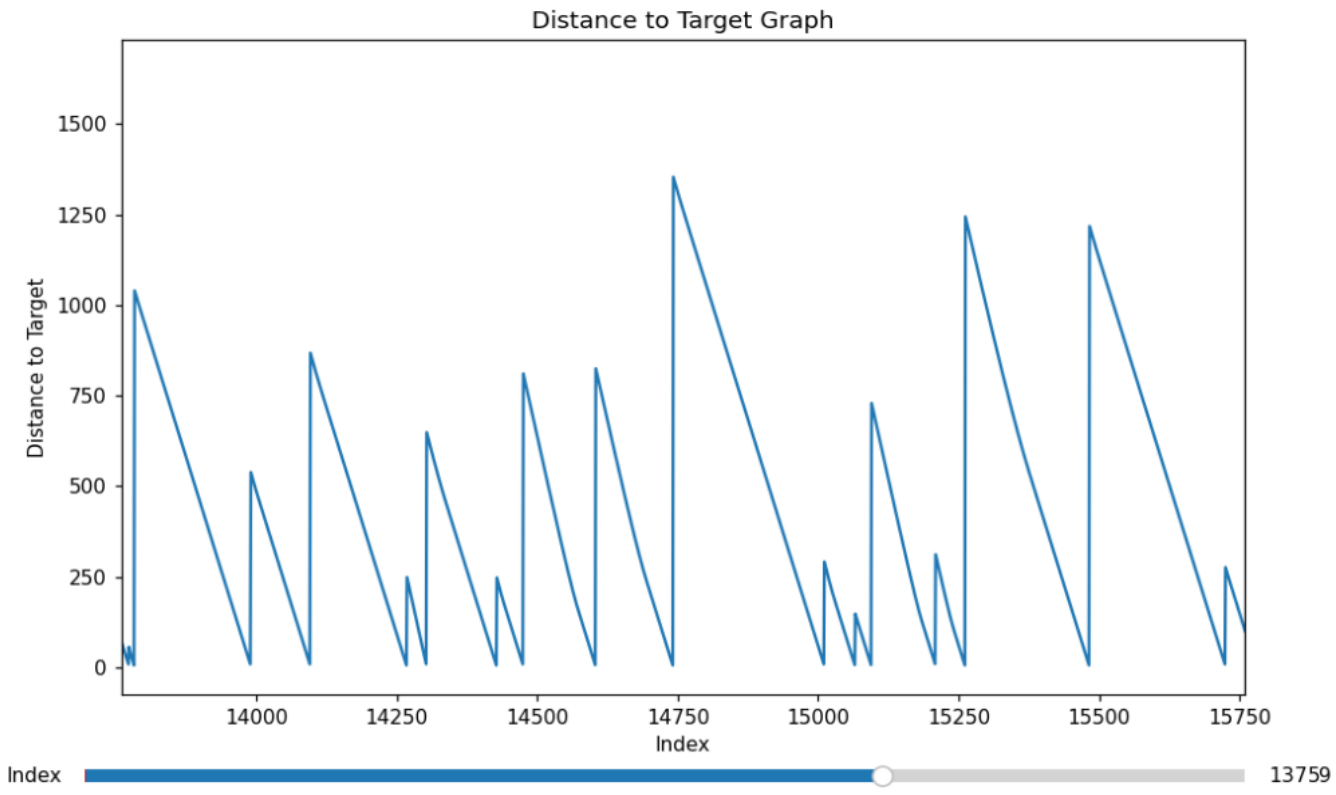
3]:
```
# Training data length

len(train_data)
```

3]:
```
20000
```

7]:
```
%matplotlib notebook
import matplotlib.pyplot as plt
import matplotlib.widgets as widgets
import math

# Calculate the distance between two points
def distance(x1, y1, x2, y2):
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

# Defining the Data
start_x, start_y, target_x, target_y, new_x, new_y, speed_loss = zip(*train_data)

# Calculate the distance to the target
distances = [distance(sx, sy, tx, ty) for sx, sy, tx, ty in zip(start_x, start_y, target_x, target_y)]

fig, ax = plt.subplots(figsize=(10, 6))

```

```python
# Distance to the target
ax.plot(distances, label="Distance to Target")
ax.set_ylabel("Distance to Target")
ax.set_xlabel("Index")

ax.set_title("Distance to Target Graph")

# A slider to explore the graph
slider_ax = plt.axes([0.1, 0.0, 0.8, 0.03])
slider = widgets.Slider(slider_ax, 'Index', 0, len(distances) - 1, valinit=0, valstep=1)

# Update the graph based on the slider value
def update(val):
    index = int(slider.val)
    ax.set_xlim(index, index + 2000)  # zoom level (2000 data points visible)
    fig.canvas.draw_idle()

slider.on_changed(update)
update(0)  # Initialize the graph

# Show
plt.show()
```

```
<IPython.core.display.Javascript object>
```



The graph demonstrates that the algorithm was able to repeatedly close the gap on the target and click on it. There was no time limit so it was always successful. The difference in timing was related directly to distance from the target when it was moved.

Model Architecture

The next step is to bring in Neural Network models to learn to play this game

The first model attempted is a fully connected (dense) feedforward model. It has 4 hidden layers using ReLU. It outputs only 2 values, an x and y value based on the prediction of the next x and y coordinates from the input coordinates and the relation to the target.

Build Model

```python
def build_model():
    # Create the neural network model
    model = Sequential()
```

```
    model = Sequential()
    model.add(Dense(256, activation="relu", input_shape=(4,)))
    model.add(Dense(128, activation="relu"))
    model.add(Dense(64, activation="relu"))
    model.add(Dense(32, activation="relu"))
    model.add(Dense(2, activation="linear"))

    model.compile(optimizer="adam", loss="mse")

    return model
```

Mouse Movement decider

Mouse movement is determined by a prediction of what the NN thinks is the next best move based on the recordings of the best mouse mover model.

```
def mouse_move_decision(target):
    absolute_x, absolute_y = target
    current_position = mouse.get_position()
    mouse_x, mouse_y = current_position
    input_data = np.array([[mouse_x, mouse_y, absolute_x, absolute_y]])
    new_position = model(input_data, training=False).numpy().astype(int)[0]

    calculate_loss(current_position, new_position, target)

    return current_position, new_position
```

Loading Data and fitting the model

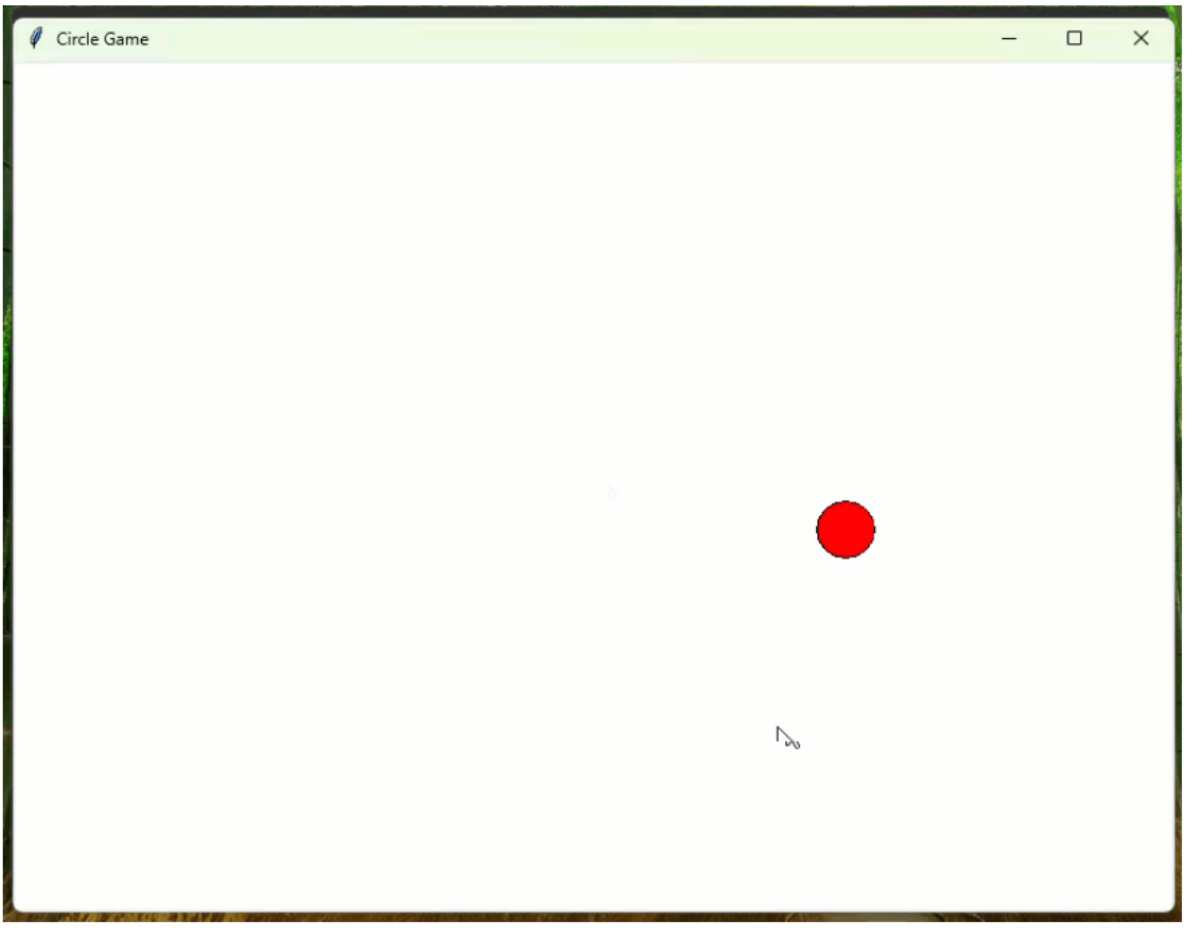
```
def load_data():
    data = pd.read_csv("train data.csv", header=None, names=["current_x", "current_y", "target_x", "target_y", "new_x", "new_y", "loss"])

    X = data[["current_x", "current_y", "target_x", "target_y"]].values
    y = data[["new_x", "new_y"]].values

    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)
    X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
    X_val = X_val.reshape(X_val.shape[0], X_val.shape[1], 1)

    return X_train, X_val, y_train, y_val

def fitting(model):
    # Define early stopping and model checkpoint callbacks
    early_stopping = EarlyStopping(monitor="val_loss", patience=5, verbose=1, restore_best_weights=True)
    model_checkpoint = ModelCheckpoint("best_model.h5", monitor="val_loss", save_best_only=True, verbose=1)

    # Train the model with callbacks
    model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=50, batch_size=8,
              callbacks=[early_stopping, model_checkpoint])

    return model
```

2]:
```
# Play this cell to watch the Neural Network decision maker train on the data and then attempt to find the red cell
# Training takes about 30 seconds to a minute. The output from the training will print after the game is closed.

!python nnplayer.py
```

```
 356/2000 [====>.........................] - ETA: 1s - loss: 88.8015
 423/2000 [=====>........................] - ETA: 1s - loss: 78.4522
 492/2000 [======>.......................] - ETA: 1s - loss: 71.7299
 562/2000 [=======>......................] - ETA: 1s - loss: 66.6082
 632/2000 [========>.....................] - ETA: 0s - loss: 68.5896
 703/2000 [=========>....................] - ETA: 0s - loss: 69.7226
 772/2000 [==========>...................] - ETA: 0s - loss: 65.5942
 839/2000 [===========>..................] - ETA: 0s - loss: 63.0202
 907/2000 [============>.................] - ETA: 0s - loss: 60.8735
 976/2000 [=============>................] - ETA: 0s - loss: 73.5826
1038/2000 [==============>...............] - ETA: 0s - loss: 82.5696
1103/2000 [===============>..............] - ETA: 0s - loss: 80.3255
1171/2000 [================>.............] - ETA: 0s - loss: 76.6884
1237/2000 [=================>............] - ETA: 0s - loss: 73.0527
1303/2000 [==================>...........] - ETA: 0s - loss: 69.8088
1366/2000 [===================>..........] - ETA: 0s - loss: 67.1492
1429/2000 [====================>.........] - ETA: 0s - loss: 64.6690
1500/2000 [=====================>........] - ETA: 0s - loss: 62.1818
```

```
1566/2000 [======================>.......] - ETA: 0s - loss: 61.0708
1631/2000 [======================>......] - ETA: 0s - loss: 60.3266
```

Or have a look at this recording



Feed Forward Neural Network performance

The NN didn't learn to succeed at the game, earning 0 points. But it did learn the general rule of move toward the circle. I would have been willing to run the experiment for longer, but it didn't look like it would ever reach the circle. I also knew it wasn't learning from the experience so running it for longer wouldn't have any good effect.

Deep Q Learning

Deep Q-Learning is a built up reinformcement model that approaches the problem from the other side. Instead of teaching a NN on a recording of good or expected behavior, the Q model explores the feature space and learns how to navigate based on rewards or punishments when it's actions are beneficial or not.

The Q algorithm exists outside of Neural Networking and is an attempt to completely comprehend an actionable space. It might learn all the moves the checker based on all possible states of pieces. Deep Q-learning takes advantage of the theory that neural networks are a universal algorithm approximator and attempts to approximate the desired Q algorithm.

In this case, the learner is not given any historical data, but instead interacts with the feature space, making random decisions 10% of the time. (epsilon =0.1) Each decision is graded based on how close it gets to moving the mouse toward the target at a speed of 5 pixels. The model slowly learns through trial and error to move toward the target. It is not quick enough to beat the 5 second timer, but it does eventually get to the positions when given enough time.

The Q Learning model testing is just an adaptation of the same Dense fully connected network in the feed forward NN. Instead of training it on correct behavior, the model is fit as it plays the game. The one modification is to add leaky ReLU to prevent the death of neurons as the model learns.

The model must also be controlled differently, the Q learning does better if it is given a limited set of actions to perform which in this case are movement in the 8 cardinal directions from center. The reward is also the opposite of loss from the NN.

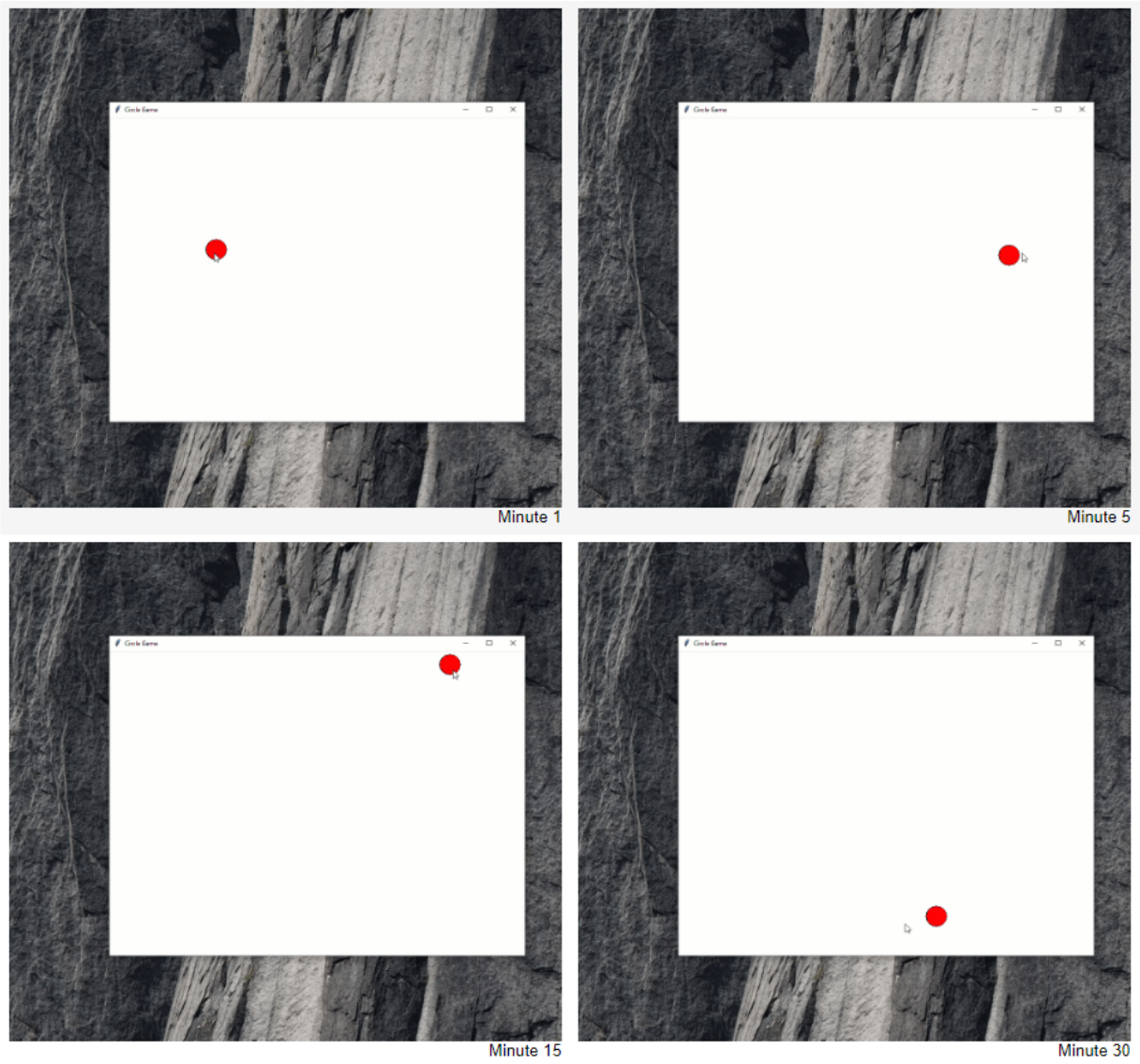
The reward is structured to expect a movement speed of 5 toward the target and to reduce rewards for any other action.

```
1 import numpy as np
```

```python
import random
import mouse
from keras.models import Sequential
from keras.layers import Dense, LeakyReLU
from circlegame import CircleGame

alpha = 0.1
gamma = 0.99
epsilon = 0.1


def mouse_move_decision(target):
    absolute_x, absolute_y = target
    current_position = mouse.get_position()
    mouse_x, mouse_y = current_position
    current_state = np.array([[mouse_x, mouse_y, absolute_x, absolute_y]])

    action = epsilon_greedy(current_state, epsilon)
    new_mouse_x, new_mouse_y = apply_action(action)
    new_position = new_mouse_x, new_mouse_y

    new_state = np.array([[new_mouse_x, new_mouse_y, absolute_x, absolute_y]])

    # Calculate reward
    reward = calculate_reward(current_state, new_state)

    # Update Q-values
    q_values = model.predict(current_state)
    next_q_values = model.predict(new_state)
    q_values[0, action] = q_values[0, action] + alpha * (reward + gamma * np.max(next_q_values) - q_values[0, action])

    # Train the model with the updated Q-values
    model.fit(current_state, q_values, epochs=1, verbose=0)

    return current_position, new_position

def apply_action(action):
    match action:
        case 0:
            mouse.move(5,0, absolute=False)
        case 1:
            mouse.move(5,5, absolute=False)
        case 2:
            mouse.move(0,5, absolute=False)
        case 3:
            mouse.move(-5,5, absolute=False)
        case 4:
            mouse.move(-5,0, absolute=False)
        case 5:
            mouse.move(-5,-5, absolute=False)
        case 6:
            mouse.move(0,-5, absolute=False)
        case 7:
            mouse.move(5,-5, absolute=False)
    new_position = mouse.get_position()
    return new_position


def build_model():
    # Create the neural network model
    model = Sequential()
    model.add(Dense(128, activation=LeakyReLU(alpha=0.01), input_shape=(4,)))
    model.add(Dense(64, activation=LeakyReLU(alpha=0.01)))
    model.add(Dense(32, activation=LeakyReLU(alpha=0.01)))
    model.add(Dense(8, activation="linear"))

    model.compile(optimizer="adam", loss="mse")

    return model


def epsilon_greedy(state, epsilon):
    if random.random() < epsilon:
        return random.randint(0, 7)  # Choose a random action from the action space
    else:
        return np.argmax(model.predict(state))  # Choose the action with the highest Q-value

def calculate_reward(current_state, new_state):
    current_x, current_y, target_x, target_y = current_state[0]
```

```
    current_x, current_y, target_x, target_y = current_state[0]
    new_x, new_y = new_state[0][:2]
    # Calculate the Euclidean distance between the current state and the target and then the new state and the target
    current_distance = np.sqrt((current_x - target_x)**2 + (current_y - target_y)**2)
    new_distance = np.sqrt((new_x - target_x)**2 + (new_y - target_y)**2)

    # Distance difference is another way of saying speed toward target
    distance_difference = current_distance - new_distance

    # Calculate the loss based on how well the new state got closer to the target
    reward = 10 - (distance_difference - 5) ** 2

    return reward
```

Have a look at these recordings

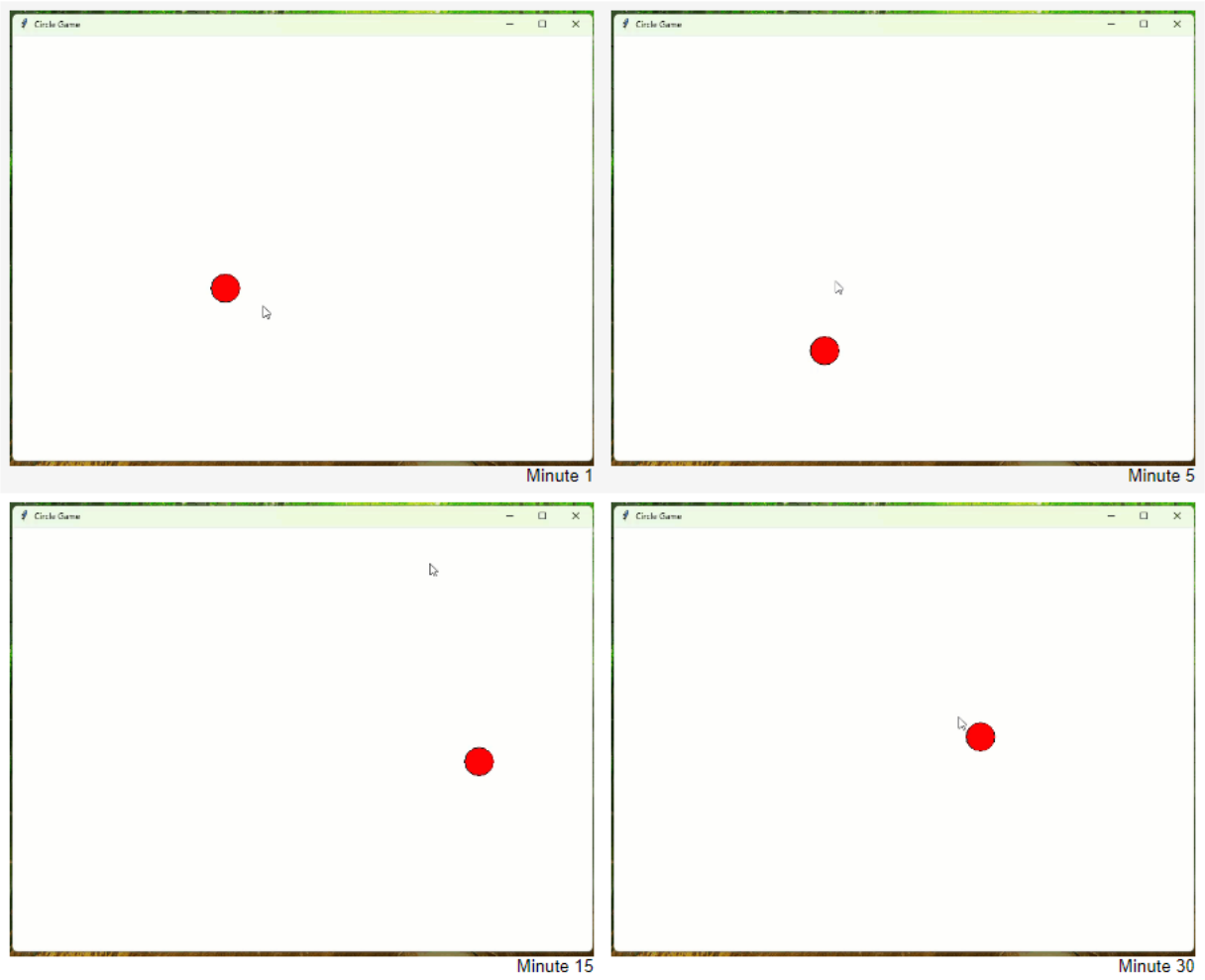


Reward Structure

I did attempt the Q-Learning model with a different reward structure. One that gave increasingly large rewards for moving toward the target and negative rewards (punishments) when moving away from the target. The results of that reward strucure showed some improvement, but not enough to call the model "good" at the game

```
def calculate_reward(current_state, new_state):
    current_x, current_y, target_x, target_y = current_state[0]
    new_x, new_y = new_state[0][:2]

    current_distance = np.sqrt((current_x - target_x) ** 2 + (current_y - target_y) ** 2)
    new_distance = np.sqrt((new_x - target_x)**2 + (new_y - target_y)**2)
```

```
    # Distance difference is another way of saying speed toward target
    distance_difference = current_distance - new_distance

    # Calculate the loss based on how it is getting closer to the target
    reward = (800 / new_distance) * distance_difference
    print(reward)

    return reward
```



Absolute Vs Relative

The last attempt I made was rewrite the model to look at the relative position of the mouse to the circle instead of using the absolute positions. This model was much better at finding the circles and I think it is because the model was trying to decide what to do at each point and had to encode the relative location of the circle to the mouse somewhere in the model. Using relative mouse positions helped simplify the information required down to just looking at the location of the mouse compared to where it needs to go since the Q learning actions are all relative anyway. Basically, the model can ignore the current position and just use the relative positions to make the action decisions.

```
def mouse_move_decision(target):
    absolute_x, absolute_y = target
    current_position = mouse.get_position()
    mouse_x, mouse_y = current_position

    relative_x, relative_y = absolute_x - mouse_x, absolute_y - mouse_y
    current_state = np.array([[mouse_x, mouse_y, relative_x, relative_y]])


    action = epsilon_greedy(current_state, epsilon)
    new_mouse_x, new_mouse_y = apply_action(action)
    new_position = new_mouse_x, new_mouse_y

    relative_x, relative_y = absolute_x - new_mouse_x, absolute_y - new_mouse_y
    new_state = np.array([[new_mouse_x, new_mouse_y, relative_x, relative_y]])

```

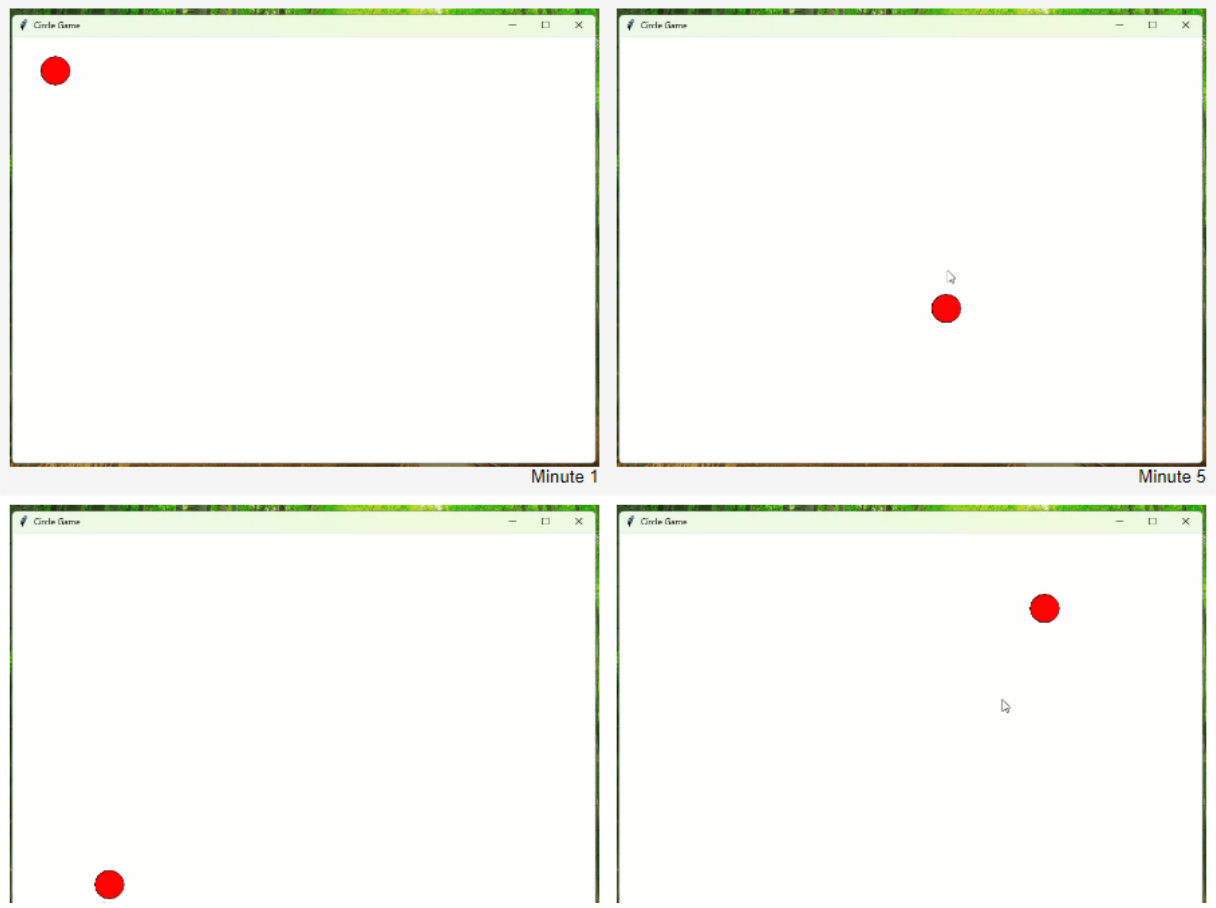
```
    # Calculate reward
    reward = calculate_reward(current_state, new_state)

    # Update Q-values
    q_values = model.predict(current_state)
    next_q_values = model.predict(new_state)
    q_values[0, action] = q_values[0, action] + alpha * (reward + gamma * np.max(next_q_values) - q_values[0, action])

    # Train the model with the updated Q-values
    model.fit(current_state, q_values, epochs=1, verbose=0)

    return current_position, new_position
```

3]:

```
# Play this cell to watch the Deep Q-learning decision start playing and attempt to find the red cell
# Since training occurs on the fly, there is no training time at the outset. Give it time and it will find a red circle.

!python qlearn.py
```

```
1/1 [==============================] - 0s 15ms/step
0.6690162011231552

1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 15ms/step

1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 14ms/step

1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 15ms/step
0.6722639471407941

1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 14ms/step

1/1 [==============================] - ETA: 0s
1/1 [==============================] - 0s 15ms/step
```

Or have a look at these recordings when I ran this



Conclusion

Includes all of the following: basic reiteration of result and discussion of learning and takeaways and discussion of why something didn't work, and suggestions for ways to improve.

The goal was to experiement with getting a Neural Network to learn to play a game. The game was arbitrarily easy and the network never got better than the ideal algorithm could. However, I learned a lot about training neural networks and how it might be handled so I think the goal was accomplished.

A basic NN can learn the general idea of the game, but lacks the specific knowledge of what to do with the mouse position other than kind of move toward the target.

The Q learning model did a much better job of learning how to reach the target and slowly got better at the task. However, it seemed to reach a plateau after only a few minutes of training and never got any better at the game. Improving the reward model helped some but the big breakthrough came when using relative target locations which helped simplify the model.

]:
```
1
```