Snake AI Game Agent Implementation

Abstract -

The snake game has existed since 1976, and since has evolved with generations to bring more entertainment from a very simple game concept, still being a trend nowadays, in this project we're gonna understand the implementation of an AI Agent that will be able to play the snake game and learn through reinforcement learning how to get higher scores.

Introduction -

For the implementation of a snake game AI agent we need to understand some concepts first, for example, Reinforcement Learning is a part of machine learning, in the case of this project we're gonna use Deep Learning using Pytorch, reinforcement learning is about taking acceptable actions/behaviors to magnify the reward for a specific situation, compared to supervised learning were the training set has the correct answers/actions in reinforcement learning there is no training set and all the learning has to be done through experience.

One important detail about this project using reinforcement learning is about the reward and punishment system that it has for the snake game, it receives an input, being the position where the snake is and the output will be the variety of positions or movements that the snake can go into to avoid hitting a wall or eating itself, and also possible movements to reach the fruit and eat it too.The training portion of this project will be based on the input, depending if the snake did the correct action or a wrong move then it will receive either a reward or a punishment, thus making the model continuously learn and improve the variety of the output.

Since we're also using Deep Q Learning for this project through Pytorch, using this extension of reinforcement learning that uses neural network to predict the actions the snake is going to take, for another example to explain this, the Q value is the quality of the action, which improves the snake depending on the action it takes, this being that the Q value we start (doing random movements to move) when there has be no learning to train the model, but as the agent plays more and more and receives punishments and rewards, this helps the prediction of the actions improve and better train the model.

Reward System -

The system that the agent has to punish, reward or maintain the training of the model is the following:

- Reward: when the snake eats a fruit it gets rewarded with 10 point
- Punishment: when the snake hits a wall or bites itself it gets removed 10 points
- Maintain: with any other movement that does not provoke a game over or higher the score, it doesn't give or take away any point

Action System -

The way the model decides the next move the snake is gonna take can only be in three ways (straight, left and right), this is because the snake can't do a 180° since it would mean game over.

- Straight [1,0,0]: the snake will keep the same way its heading and won't change direction
- Right [0,1,0]: the snake will make a right turn depending on the original direction it was taking
- Left [0,0,1]: the snake will a left turn depending on the original direction it was taking

State Value -

This variables will help the model predict and understand its environment, this meaning we have a single, dynamic and complete but unknown environment, in this aspect we have 11 boolean variables to help the AI guide the snake:

- Danger: straight, right and left

- Direction: left, right, up and down
- Food: left, right, up and down

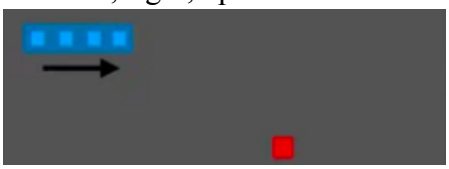

Fig. 1. Example of possible state values the snake would have in a specific position.

In this case, the state values we would receive to help the snake predict its next movement would be:

[0 , 0 , 0,
0 , 1 , 0, 0,
0 , 1 , 0 , 1]

This would mean that there is no immediate danger coming from either left, right or straight, the direction the snake is heading in the right direction inside the environment, and the fruit is to the right and down.

Model Training - For the training of the model we will be using a feedforward neural network, were we have the input (state values), with a hidden segment and an output (having three different values to take into consideration), for this we take the highest values out of the three and make that single value 1 and the rest become 0, indicating the direction the snake is going for example.

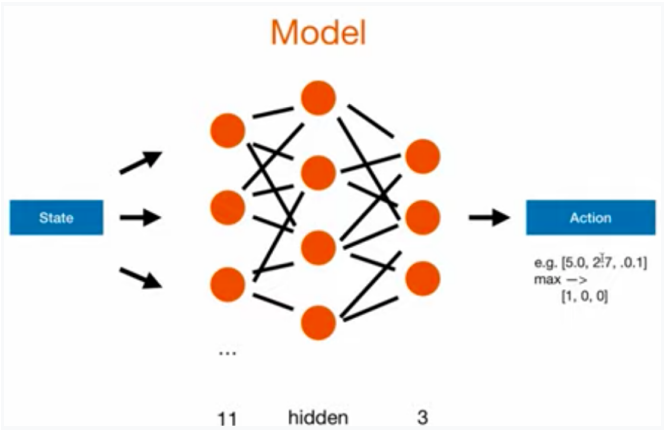

Fig. 2. Example of possible output value where the highest value becomes 1 and the other two become zero.

As we explained in the introductory segment, this project is trained using Deep Q Learning, which means we want to improve the quality of action that the snake has, but first we start the model with random parameters so the model can start learning about its environment and receiving punishments/rewards, making the Q value improve more and more training the model.

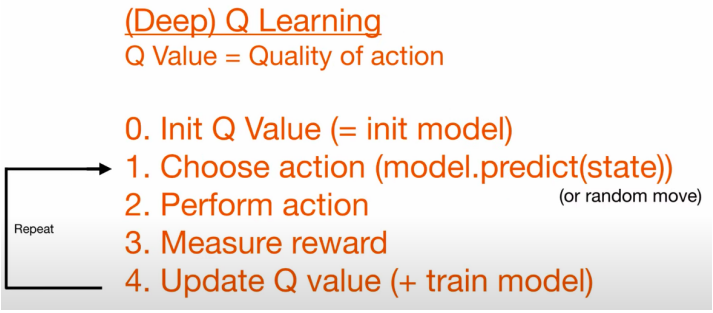

Fig. 3. Deep Q Learning modelization and demonstration of the actions the agent takes to improve the predictions of the snake.

Bellman Equation -
Showcasing in Figure 4, we can better understand how the new value for Q value is done through calculations. Where we take into account the current Q value, include the learning rate (and the reward/punishment) from the action just taken, then we include the discount rate and the maximum expected rewards (this being dependent on the new state and all possible actions that can be done in that state value).

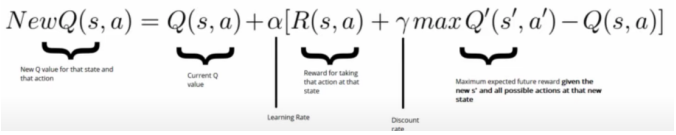

Fig. 4. Bellman Equation for this Deep Q Learning model.

But this equation can be simplified to better understand how this equation helps for our model:

$$Q = model.predict(state_0)$$

$$Q_{new} = R + \gamma \cdot max(Q(state_1))$$

Fig. 5. Bellman Equation simplified.

In which we establish that Q is used to predict with an old state and the new value for Q is the reward received, plus our discount rate with the max value of Q in the new state. Having these two formulas into consideration we also obtain the loss function, this being a simple mean squared error formula.

$$loss = (Q_{new} - Q)^2$$

Mean Squared Error

Fig. 6. Loss function.

Code Segmentation-

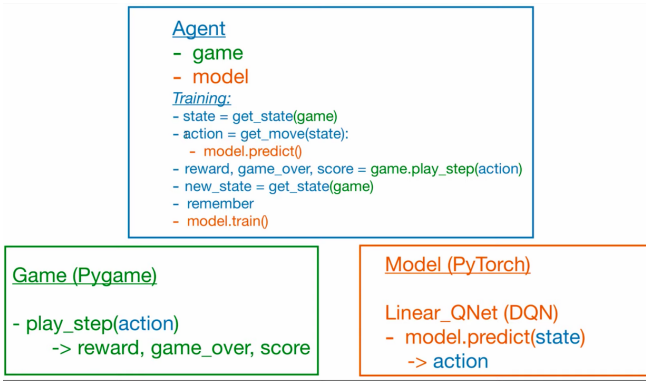

Fig. 7. The three segments needed for the project.

To implement the project correctly we first need to understand the implementation of the game, the model and the agent.

- Implementation of the game (Pygame)
  This segment need to be done in a loop, because every time game starts and finishes it doesn't stop the training, for this play_step function we get an action and with that the model works, then receives a reward (depending on the type of action our snake took), check if our game ended or not and update the score.

- Implementation of the model (PyTorch)
  The model we are using as defined before is a feedforward neural network based on Linear_QNet, in which we have segments where we store the old and new state values received from the agent that help train the model, by calling the model.predict to then execute the next action.

- Implementation of the agent
  The agent work as the segment that helps us put everything together combining the model and the game, here we implement the training loop and calculate the state value that we obtain from the game and with this state value we then obtain the next move using model.predict(), then with this we can play_step() where we get our reward, check if the game ended and score values to continue the cycle and get a new state value. With all this information we can train our model using the old and new state values, game_over and action information.

Code Implementation -

For this project, the first thing we will need to start working will be the correct environment, in this case, using a Windows system requires downloading the Anaconda Environment [6] to help run our model and game correctly with the agent, in addition to downloading the correct libraries for pytorch, pygame, ipython and matplotlib.

Then implementing the source code from the Github repository: https://github.com/patrickloeber/snake-ai-pytorch we're going to make the adequate modifications to be able to run the code with the agent without needing a manual input.

- Implementing the Game Code

The first aspects we need to modify is the name class for the Snake Game segment and add a reset function to the initializer:

```python
class AISnakeGame:

    #Intialize things need for the game
    def __init__(self, width=640, height=480):
        self.width = width
        self.height = height
        # init display
        self.display = pygame.display.set_mode((self.width, self.height))
        pygame.display.set_caption('Snake')
        self.clock = pygame.time.Clock()

        self.reset()

    def reset(self):
        # initial state of the snake will be UP
        self.direction = Direction.UP

        self.head = Point(self.width/2, self.height/2)
        self.snake = [self.head,
                      Point(self.head.x-BLOCK_SIZE, self.head.y),
                      Point(self.head.x-(2*BLOCK_SIZE), self.head.y)]

        self.score = 0
        self.food = None
        self.setFood()
        self.frame_iteration = 0
```

Fig. 8. Modifications to class name and addition of reset function

.

Then for the play_step function we will remove all the user input that was originally placed and replace it with the action parameter:

```python
def play_step(self, action):
    self.frame_iteration += 1
    # 1. collect user input
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                pygame.quit()
    # 2. move
    self.move(action) # update the head
    self.snake.insert(0, self.head)

    # 3. check if game over
    reward=0
    game_over = False
    if self.is_collision() or self.frame_iteration > 100*len(self.snake
        game_over = True
        reward= -10
        return reward, game_over, self.score

    # 4. place new food or just move
    if self.head == self.food:
        reward= 10
        self.score += 1
        self.setFood()
    else:
        self.snake.pop()

    # 5. update ui and clock
    self._update_ui()
    self.clock.tick(SPEED)
    # 6. return game over and score
    return reward, game_over, self.score
```

Fig. 9. Modifications to play_step function and remove user input details.

Also for this section of modifications we modify the move function replacing the self.direction with the action variable, then adding the reward inside the game_over action and the self.frame_iteration, this to prevent the snake from not doing anything for a long period of time. In addition inside the Is_collision function, the self.head value parameters are replaced with pt.

```python
def move(self, action):
    #Action: [straigt, right, left]
    def_direction = [Direction.UP, Direction.RIGHT, Direction.DOWN, Direction.LEFT]
    index_dir = def_direction.index(self.direction)

    #Move Straight
    if np.array_equal(action, [1,0,0]):
        direction = def_direction[index_dir]
    elif np.array_equal(action, [0,1,0]):
        new_index = (index_dir+1)%4
        direction= def_direction[new_index]
    else: #Action is equal to [0,0,1] Left
        new_index = (index_dir-1)%4
        direction= def_direction[new_index]

    self.direction = direction

    x = self.head.x
    y = self.head.y
    if direction == Direction.RIGHT:
        x += BLOCK_SIZE
    elif direction == Direction.LEFT:
        x -= BLOCK_SIZE
    elif direction == Direction.DOWN:
        y += BLOCK_SIZE
    elif direction == Direction.UP:
        y -= BLOCK_SIZE

    self.head = Point(x, y)
```

Fig. 10. Modifications to Is_collision function.

- Implementing the Model Code

First we have to import the libraries:

```python
import torch as T
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import os
```

Fig. 11. Libraries to import.

Then we create the DeepQNet class:

```
class DeepQNet(nn.Module):
    def __init__(self, input_dims, hidden_dims, output_dims):

        super().__init__()
        self.input_dims = input_dims
        self.hidden_dims = hidden_dims
        self.output_dims = output_dims
        #Create Layer 1 of the neural network
        self.lin1 = nn.Linear(input_dims, hidden_dims)
        #Create Layer 2 of the neural network
        self.lin2 = nn.Linear(hidden_dims, output_dims)

    def forward(self, state):

        x = F.relu(self.lin1(state))
        actions = self.lin2(x)
        return actions

    def save(self, file_name = 'model.pth'):

        #Create a path for a model folder
        path = './model'
        #Check if the directory folder already exists
        if not os.path.exists(path):
            #Create the model folder
            os.makedirs(path)
        file = os.path.join(path, file_name)
        #Save the file of the model
        T.save(self.state_dict(), file)
```

Fig. 12. Deep Q Neural Network class.

And finally we create the Trainer class:

```
class Trainer:
    """ Class for making the optimization of the model"""
    def __init__(self, model, gamma, alpha):

        self.alpha = alpha
        self.gamma = gamma
        self.model = model
        #Define the loss function
        self.loss = nn.MSELoss()
        #Create an optimizer
        self.opt  = optim.Adam(model.parameters(), lr=self.alpha)

    def train_step(self,state, next_state, action, reward, lose):

        #Create every argument except done, into a pytorch tensor so it can
        #be used in the model (n,x)
        state = T.tensor(state,dtype=T.float)
        next_state = T.tensor(next_state,dtype=T.float)
        action = T.tensor(action,dtype=T.long)
        reward = T.tensor(reward,dtype=T.float)
        #Handle multiple sizes or dimensions for the long term memory function
        #First check if state has only one dimension
        if len(state.shape)==1:
            #Re-shape the arguments so we have it like (1,x)
            #(n,x) where n is the number of batches and x is the value

            state = T.unsqueeze(state,0)
            next_state = T.unsqueeze(next_state,0)
            action= T.unsqueeze(action,0)
            reward = T.unsqueeze(reward,0)
            #make lose argument a tuple with one value
            lose = (lose, )

        #Calculate Bellman equation
        #obtain the prediction (Q) of current state and the of the next state
        predQ = self.model(state)
        next_predQ = self.model(next_state)
        #Create a clone of the prediction Q called target
        target = predQ.clone()
        for i in range (len(lose)):
            new_Q = reward[i]
            if not lose[i]:
                #Apply the Bellman equation if not lose
                #NewQ = r + gamma * max(next predicted Q value)
                new_Q=reward[i]+self.gamma*T.max(next_predQ[i])

            target[i][T.argmax(action).item()] = new_Q

        self.opt.zero_grad()
        loss = self.loss(target,predQ)
        #Create a backward propagation and update the gradients
        loss.backward()

        self.opt.step()
```

Fig. 13. Q Trainer class.

- Implementing the Agent Code

First we import the libraries and classes needed to combine the game, model and agent, and create the agent's parameters:

```
import numpy as np
import random
import torch
from SnakeGame import AISnakeGame, Direction, Point
from collections import deque
from DeepQNetModel import DeepQNet, Trainer
import matplotlib.pyplot as plt
from IPython import display

MAX_MEM= 100000 #Set a maximum capacity of memory
BATCH_SIZE = 1000 #number of samples for our memory
ALPHA = 0.001 #Learning rate
```

Fig. 14. Libraries to import and agent parameters

Then we create the Agent class that includes the Init function:

```
class Agent:

    def __init__(self):
        self.ep = 0  # coef of randomness
        self.gamma = 0.9 # discount rate
        self.num_games = 0
        #deque will automatically do popleft() if it exceeds the MAX_MEM
        self.memory = deque(maxlen=MAX_MEM)
        """To do: model trainer """
        self.model = DeepQNet(11,256,3)
        self.trainer = Trainer(self.model, gamma=self.gamma,alpha=ALPHA)
```

Fig. 15. Agent class with Init function.

getAction function:

```
def getAction(self, state):

    #Make random moves
    self.ep= 80 - self.num_games
    action = [0,0,0]
    if random.randint(0,200) < self.ep:
        #srl = straight (0), right(1), left(2)
        srl = random.randint(0,2)
        action[srl] = 1
    else:
        st = torch.tensor(state, dtype=torch.float)
        #get a prediction
        prediction = self.model(st)
        srl = torch.argmax(prediction).item()
        action[srl] = 1
```

Fig. 16. Get Action function.

getState function:

```
def getState(self, game):
    #get coordinates of head of the snake and the posible next points
    head = game.snake[0]
    head_x = head.x
    head_y = head.y
    p_up = Point(head.x, head.y - 20)
    p_right = Point(head.x + 20, head.y)
    p_down = Point(head.x, head.y + 20)
    p_left = Point(head.x - 20, head.y)
    #get coordinates of food
    food_x = game.food.x
    food_y = game.food.y
    #get the current direction the snake is pointing at
    dir_up = game.direction == Direction.UP
    dir_right = game.direction == Direction.RIGHT
    dir_down = game.direction == Direction.DOWN
    dir_left = game.direction == Direction.LEFT

    #Check if there is danger around the snake and where
    #Use the is_collision function from SnakeGame.py
    if dir_up:
        danger_straight = game.is_collision(p_up)
        danger_right = game.is_collision(p_right)
        danger_left = game.is_collision(p_left)
    elif dir_right:
        danger_straight = game.is_collision(p_right)
        danger_right = game.is_collision(p_down)
        danger_left = game.is_collision(p_up)
    elif dir_down:
        danger_straight = game.is_collision(p_down)
        danger_right = game.is_collision(p_left)
        danger_left = game.is_collision(p_right)
    elif dir_left:
        danger_straight = game.is_collision(p_left)
        danger_right = game.is_collision(p_up)
        danger_left = game.is_collision(p_down)

    #Check where the food is respect to the head of the snake
    food_right = head_x < food_x
    food_left = head_x > food_x
    food_up = head_y > food_y
    food_down = head_y < food_y

    #Make the state list
    state = [danger_straight, danger_right, danger_left,
             dir_left, dir_right, dir_up, dir_down,
             food_left, food_right, food_up, food_down]
    return np.array(state, dtype=int)
```

Fig. 17. Get State function.

The training function:

```
def train():
    scores = []
    mean_scores = []
    total_score = 0
    best_score = 0
    agent = Agent()
    game = AISnakeGame()

    #Game training loop
    while True:
        # get old state
        state = agent.getState(game)
        #get action
        action = agent.getAction(state)
        #Model predict
        reward, game_over, score = game.play_step(action)
        new_state = agent.getState(game)

        #Train the short term memory
        agent.shortMem(state, new_state, action, reward, game_over)
        #call remember function
        agent.remember(state, new_state, action, reward, game_over)

        if game_over:
            game.reset()
            agent.num_games += 1
            #Train long term memory
            agent.longMem()

            if score > best_score:
                best_score = score
                agent.model.save()

            print('Game:', agent.num_games, 'Score:', score, 'Best Score:', best_score)

            scores.append(score)
            total_score += score
            mean_score = total_score / agent.num_games
            mean_scores.append(mean_score)
            plot(scores, mean_scores)


if __name__ == '__main__':
    train()
```

Fig. 18. Training function.

remember, shortMem and longMem:

```
def remember(self, state, next_state, action, reward, lose):

    self.memory.append((state, next_state, action, reward, lose))

def shortMem(self,state, next_state, action, reward, lose):

    self.trainer.train_step(state, next_state, action, reward, lose)

def longMem(self):

    if len(self.memory) > BATCH_SIZE:
        mini_sample = random.sample(self.memory, BATCH_SIZE) #List of tuples
    else:
        mini_sample = self.memory

    #get every sate together, every next_sate together, etc.
    states, next_states, actions, rewards, loses = zip(*mini_sample)
    self.trainer.train_step(states, next_states, actions, rewards, loses)
```

Fig. 19. Training remember, short memory and long memory functions.

Results -

When we run the programs correctly after following all the steps we obtain three windows on display:

1. The first window is our main window, showcasing the interface of the snake game were we see the AI playing the game:

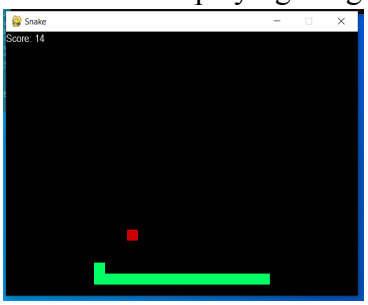

Fig. 20. Snake play screen.

2. A graph window showcasing our training data in a visual way to better understand the progress the model is doing using the score obtained compared to the number of games:

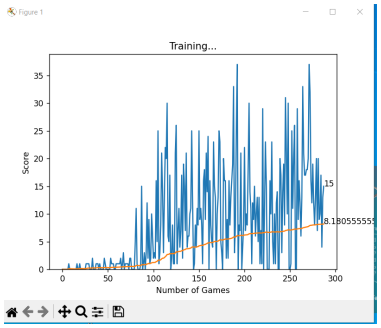

Fig. 21. Training analysis data graph.

3. Including the original window were we first called for the program with the command () it also displays the information of each run of the game, stating the score obtained and the highest score:

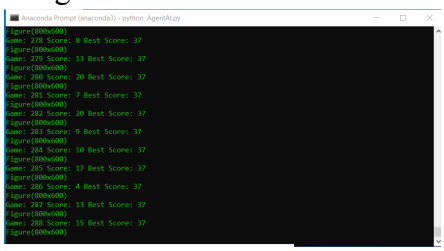

Fig. 22. Score data window.

From the result graphs, we can observe from our graph we can observe that most of the times after 100 or 150 games the snakes starts obtaining higher scores meaning the reinforcement learning is working better and improving the prediction of movements that the snake needs to do.

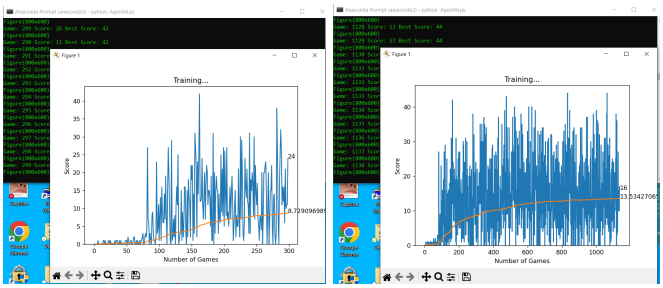

Fig. 23 & 24. First game run.

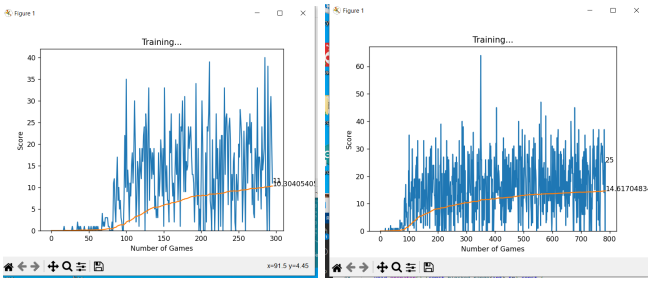

Fig. 25 & 26. Second game run.

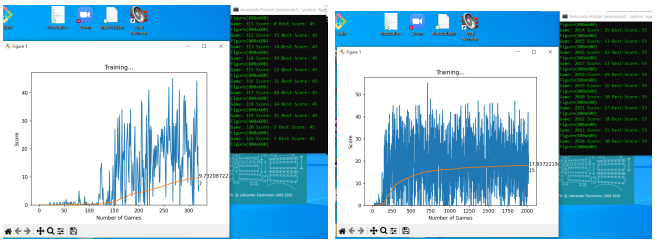

Fig. 27 & 28. Third game run.

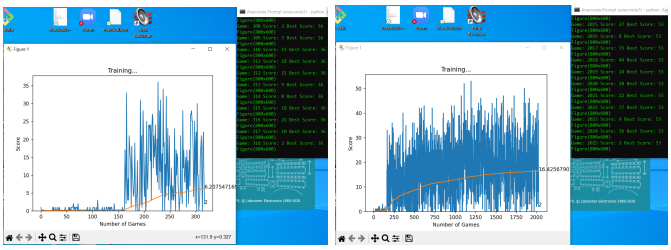

Fig. 29 & 30. Fourth game run.

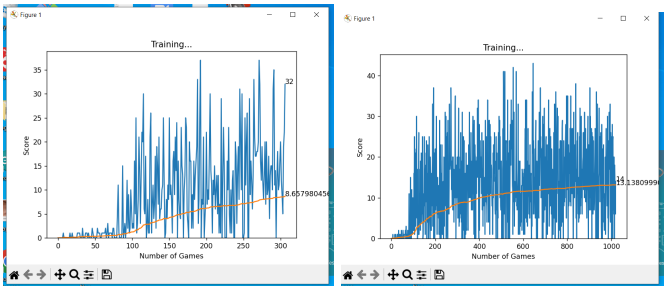

Fig. 31 & 32. Fifth game run.

After 5 different runs we can observe a pattern repeating that for certain periods of time the agent starts learning more about collisions against the edges of the screen but has more trouble learning about collisions with itself due to growing more and more in size so rapidly, but the learning curve shown in orange demonstrates how fast and then how it almost maintains the same after a certain period of time.

References.

[1]https://www.geeksforgeeks.org/what-is-reinforcement-learning/

[2]https://theprint.in/features/nokias-snake-the-mobile-game-that-became-an-entire-generations-obsession/462873/

[3]https://youtube.com/playlist?list=PLqnslRFeH2UrDh7vUmJ60YrmWd64mTTKV

[4]https://www.geeksforgeeks.org/types-of-environments-in-ai/

[5]https://github.com/patrickloeber/snake-ai-pytorch

[6]https://www.youtube.com/watch?v=9nEh-OXVaNI