Space Invaders Artificial Intelligence

By: Kenneth Vuong

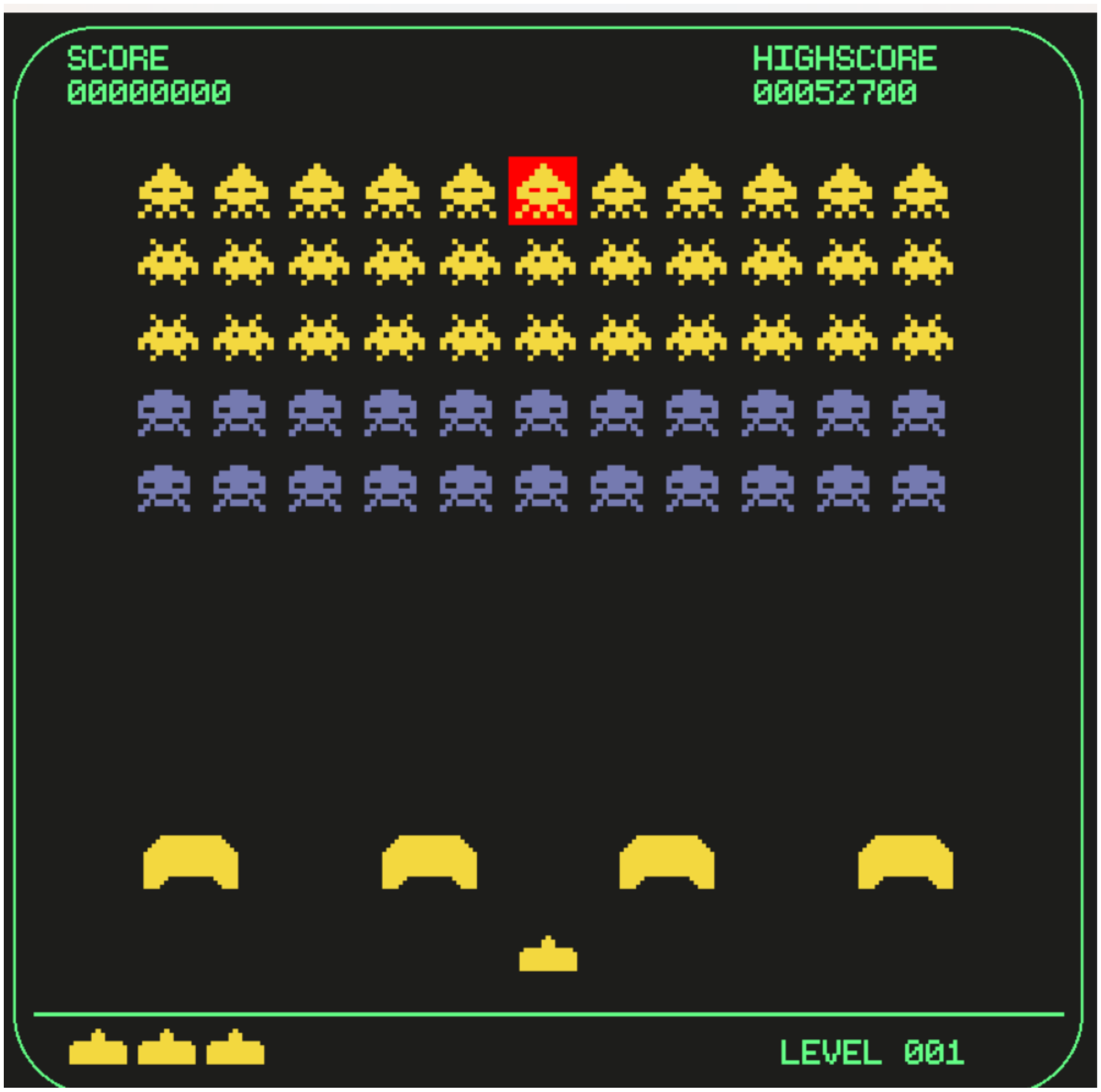


Figure 1 – Space Invaders Game

Introduction

Space Invaders is a popular arcade game where the player must control a spaceship to destroy oncoming aliens. The environment of the game consists of rows of aliens, obstacles, a mystery ship, and the player-controlled ship itself. The aliens will shoot lasers which damage the player. Obstacles provide a shield for the player to leverage to take cover from alien lasers. The mystery ship is a randomly generated ship that will provide additional points if hit. The objective of the game is to clear as many aliens as possible before losing all remaining lives.

The programming language used for the project is Python. The game itself is programmed from the ground up in the Pygame API, while the artificial intelligence is programmed in the Pytorch framework.

Game Implementation

PyGame

To simulate the game in Pygame, several libraries were developed to implement the player ship, aliens, obstacles, the mystery ship, lasers, and logic. There was a library for each unique sprite class and were all integrated within the game.py and main.py files. The implementation is as follows:

- spaceship.py – class Spaceship(self,screen_width,screen_height, offset)
    - Initialize – spawns ship in at home position on screen, retrieves bounding box
    - Fire – Shoot laser
    - Reload – sets timer to shoot laser
    - Player Movement – retrieve player user input to move ship
    - AI Movement – retrieve passed array to move ship
    - Update – integrates fire, reload, and movement to be called by game
- laser.py – class Laser(self, position, speed, screen_height)
    - Initialize – spawns laser in at specified position, retrieves bounding box
    - Update – move laser across screen and destroy self when off screen
- alien.py – class Alien(self, type, x, y)
    - Initialize – spawns alien in at specified position, retrieves bounding box
    - Update – move alien along specified direction
- alien.py – class MysteryShip(self, screen_width,offset)
    - Initialize – spawn mystery ship in on one side of the screen, set velocity
    - Update – move mystery ship across screen and destroy self when off screen
- obstacle.py – class Block(self, x, y)
    - Initialize – spawn block at specified position, retrieves bounding box
- obstacle.py – class Obstacle(self, x, y)
    - Initialize – spawn group of blocks in at specified position, retrieves bounding box
- game_AI.py – class SpaceInvadersGameAI(self, ,screen_width,screen_height, offset, TICK)
    - Initialize – Create all sprites on screen, set game logic (lives, scores, game over)
    - Create Obstacles – generate obstacle group evenly spaced along screen
    - Create Aliens – generate alien group in evenly spaced grid
    - Move Aliens – update alien group with direction and speed
    - Fire Aliens – randomly select an alien to shoot a laser
    - Create Mystery Ship – generate a mystery ship sprite
    - Check for collisions – iterate through every sprite in game and update game parameters accordingly
        - Alien collision with player ship

        - Alien laser collision with player ship
        - Alien laser collision with obstacle block
        - Player laser collision with alien ship
        - Player laser collision with obstacle block
    - Game over – stops the game from running
    - Retrieve state – retrieves the state of the game to pass to game agent
    - Train movement – updates reward based off alien target position
    - Train timer – updates reward based off time since last alien shot
    - Train avoidance – updates reward based off alien laser sprite
    - Train obstacle safety – Updates reward based off obstacles destroyed by player
    - Reset – empty all sprite groups, create aliens, create obstacles, create spaceship, reset game logic
    - Check highscore – check score for current game, update high score if current score is greater than high score
    - Load highscore – update .txt file with high score, create .txt file if none preexisting
- main.py
    - Draw – update screen with sprite positions and User Interface (score, lives, highscore)
    - Train
        - While loop where the game is run
        - Set game timer
        - Trigger events for game sprites
            - Retrieve user input from keyboard OR from game AI
            - Randomly generate mystery ship
            - Randomly generate alien laser
        - Check if game over
- agent.py
    - Initialize – create the Linear Q Net and QTrainer models, initialize game tracker logic
    - Scale predictions – Scale output layer
    - Apply threshold – Apply a threshold to make decisions in output layer
    - Get state – retrieve state of game from game.py
    - Get action – exploration vs exploitation to choose a random move or predict a move from Linear Q Net model
    - Remember – append states to memory
    - Train short term memory – Run Q trainer on single past iteration
    - Train long term memory – Run Q trainer on multiple past iterations
    - Save model – Save neural net weights
    - Load model – load neural net weights

Most of the code logic is contained within the game.py file, with the events being handled within the main.py file. All sprites contain local data variables which store information about the sprites position, bounding box, and the image location of it's appearance if required. The object oriented approach towards game implementation makes the code modular and a large number of sprites to be handled within a few lines of code. The raw code can be found on my GitHub.

One difference between the original Space Invaders game and my implementation is that I allow the player to move the ship in the y-axis to some amount. This is to allow the player more freedom to play the game and make avoiding lasers more interesting.

AI Implementation

Deep-Q Learning Neural Network

Reinforcement Learning is a type of machine learning where an agent learns to make decisions by performing actions in an environment to achieve a goal. Q-learning is a model-free reinforcement learning algorithm where the algorithm evaluates the value of specific actions based off a particular state. Deep Q-learning is a technique which combines a neural network with Q-learning such that the neural network can assign values to actions given a state, and is widely used in complex environments.

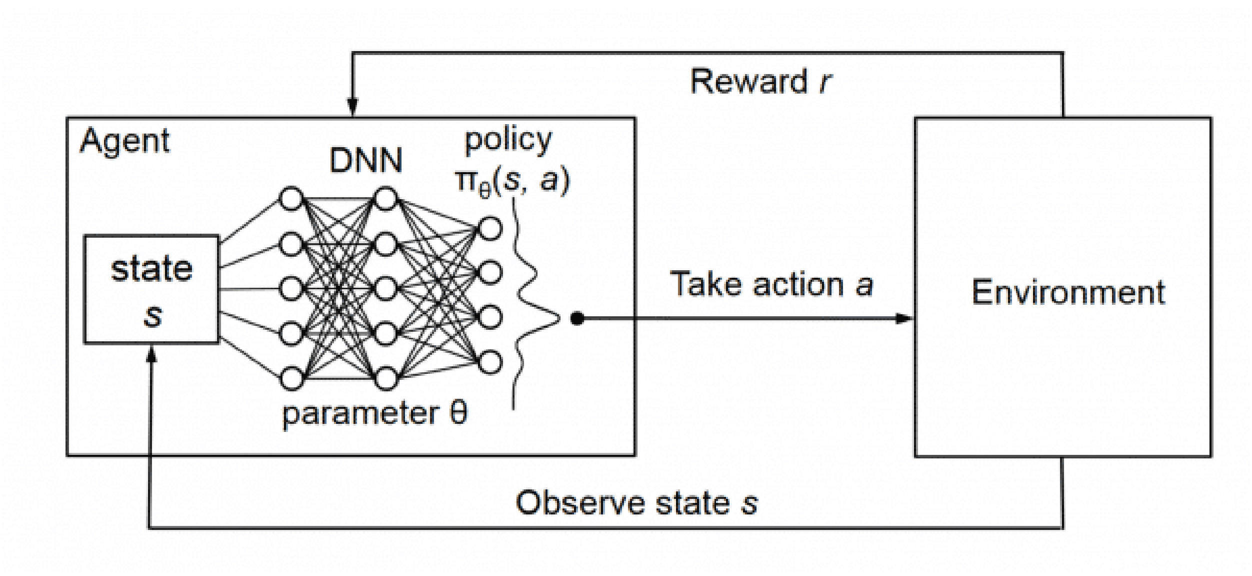
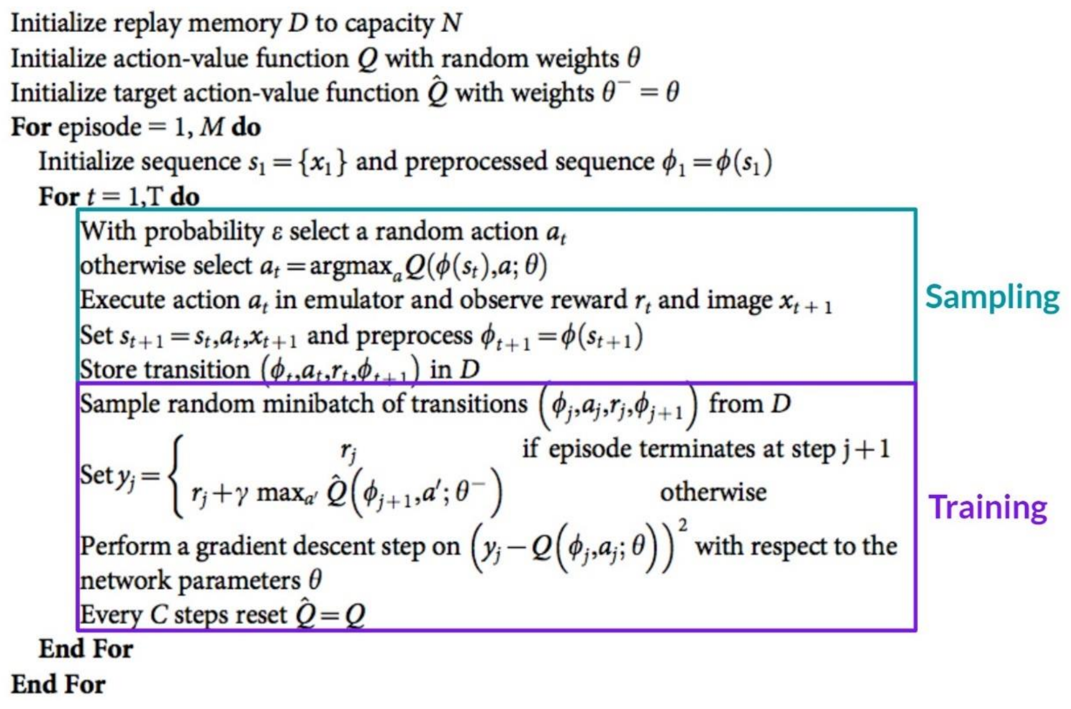


Figure 2 – Deep Q-Learning Reinforcement Learning Framework

The following pseudocode represents the Deep Q-learning method. Deep Q-learning balances "exploration" and "exploitation" such that the neural network is able to tune the weights of each node to exploitable strategies while also having the flexibility of trying new actions. This is useful in the context of a Space Invaders game, as the Deep Q Network (DQN) will be able to discover different strategies for playing the game. The Deep Q Network has an "experience replay" system which acts as memory which stores the performances of previous iterations. This is important to train both the short term memory (so that the algorithm improves within an iteration) and the long term memory (so the algorithm improves between iterations). To stabilize the training, the loss between the target Q value and the current Q value is calculated. This gives the algorithm an output objective such that it can update the weights within the neural network.


Sampling
Training

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t$ = 1,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

Figure 3 – Deep Q Network Pseudocode

A Deep Q Network is implemented in this project to act as the method for controlling the player ship. To update the weights of the DQN, rewards are introduced which correspond to different states in the environment of the game. The goal is for the DQN to be able to explore and exploit different strategies to maximize the reward received per a given state. The rewards are as follows.

Reward Algorithms

To teach a DQN to play Space Invaders, rewards need to either reinforce or punish specific actions taken at a state. Several algorithms were developed to reward the DQN to specific states. These algorithms correspond to different strategies, mainly in targeting alien ships, avoiding shooting obstacles, and in avoiding alien lasers.

Targeting Alien Ships:

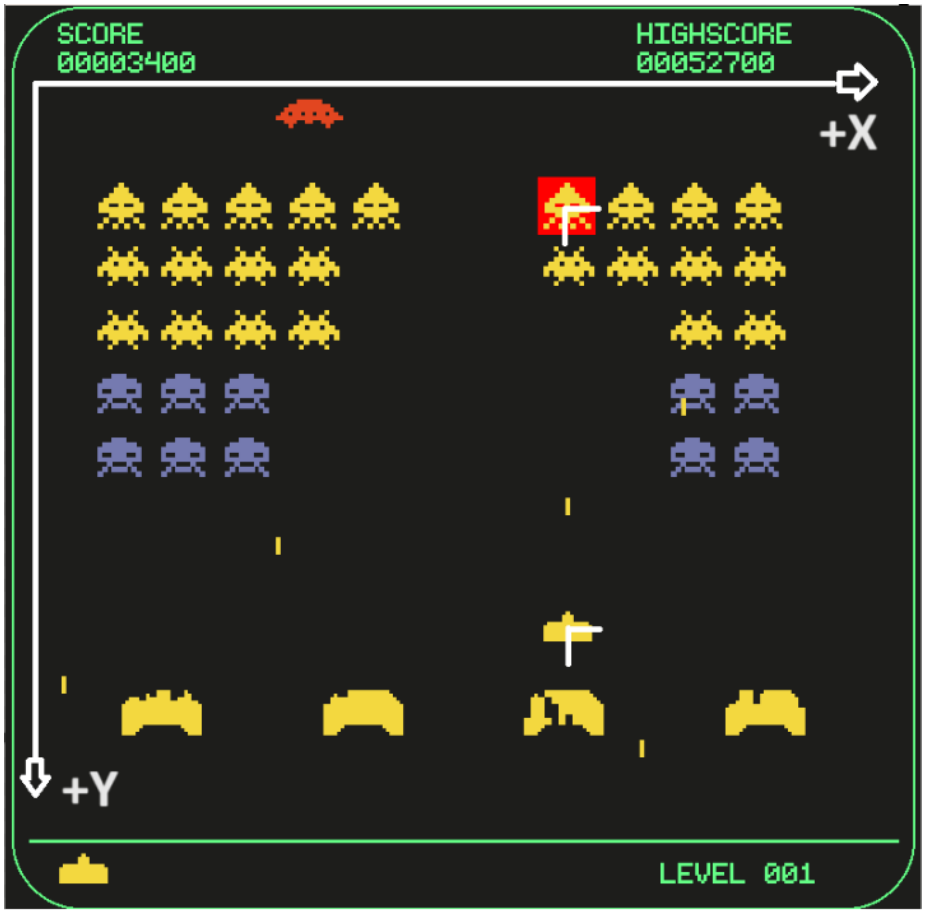


Figure 4 – Coordinate Frame Convention

The targeting ships algorithm is a depth-based approach towards destroying the alien fleet. The algorithm compares the center position of the ship to the center positions of every alien to determine which alien column is closest in the x direction. The target will be the highest alien of that column, the x and y coordinates of that alien is then stored. The y coordinate of the lowest alien in that column is also stored. There are two reward updates within this algorithm, one for the x distance and one for the y distance.

- The x distance reward function is a linear function where a greater reward is given the smaller the x distance is between the ship and the target alien.
- The y distance reward function is a linear function where a greater penalty is given the smaller the y distance is between the ship and the lowest alien in the target column. There is a boundary cutoff such that this reward function does not activate until the ship is a certain number of pixels away from the lowest alien.

Avoiding Shooting Obstacles:

The algorithm to avoid shooting obstacles checks whether a player ship laser sprite has collided with an obstacle block. If so, a penalty is introduced.

Avoid Alien Lasers:

The method to avoid alien lasers is split up into three parts: a left danger zone, right danger zone, and top danger zone. The algorithm will iterate through the positions of every alien laser sprite in the environment, and if the bounding box of the laser sprite is within some window of the player ship sprite, then a penalty/reward is given.

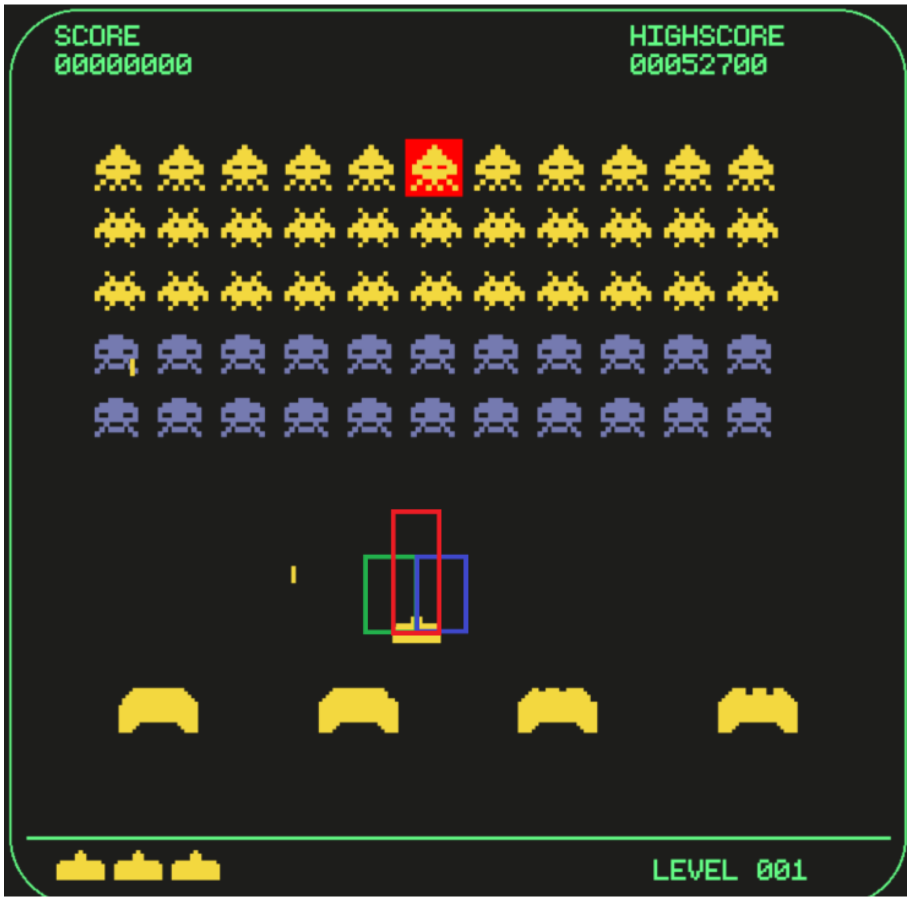


Figure 5 – Alien Laser Detection Window

The left danger detection and the right danger detection methods reward the algorithm when the a laser sprite is off to the side of the player ship. It uses a linear reward function where if the laser x coordinate is near the center x coordinate of the player ship, it punishes the algorithm while when the laser x coordinate is off the side it provides a positive reward.

The top danger detection method is a linear reward function which penalizes the algorithm the closer the y coordinate of a laser is to the y coordinate is to the top of the player ship.

Timer algorithm:

To incentivize the algorithm to destroy laser ships, a timer is introduced. This timer positive increments every frame of the game, and gradually penalizes the algorithm the higher the value is. The timer is reset when a player laser sprite collides with an alien sprite. As such, the ship is rewarded to take aggressive strategies in playing Space Invaders.

Reward Scaling

The reward ranges of the different algorithms are represented in the following table. The "/" means "between", so a "-8/8" would mean that the range goes from a -8 point penalty to a +8 point reward.

Reward	Top Danger	Left Danger	Right Danger	Relative Y	Relative X	Timer	Laser Fired?	Hit by Alien Laser	Successfully Shoot Alien
Range	-8/8	0/6	0/6	-10/0	-100/15	-100	1	-5	10

The scale of rewards was kept on the tens scale to avoid introducing large variability in the neural network weight updates.

Neural Network Implementation

PyTorch

To implement the Deep Q Network, the PyTorch framework is utilized. A PyTorch linear Q network is constructed of a input layer consisting of 11 states, three hidden linear layers with 256 nodes each, and an output layer of five nodes. The forward propagation activation function for the nodes in the hidden layers is a ReLu activation function.

- class Linear_QNet(self, input_size, hidden_size1, hidden_size2, hidden_size3, output_size):
    - Initialize – construct the input layer, hidden layers, and output layer
    - Forward – define the forward propagation activation function for the hidden layers

To implement the target Q value, a "QTrainer" class is implemented. To determine the target Q value, the temporal difference update equation is utilized. Q_new = r + (gamma * max(next_predicted Q value)). At each training step, the current Q prediction is compared to the calculated target Q prediction, and weights are adjusted within the neural network accordingly. Hyperparameters such as the learning rate and the gamma value within the TD equation are introduced here as a method to fine tune the training. The Adam optimizer, an algorithm for gradient-based optimization, is used to train the neural network through back propagation (updating parameters based off Q-values).

- class QTrainer(self, model, learning rate, gamma):
    - Initialize – construct the input layer, hidden layers, and output layer
    - Train step – calculate the predicted Q value from the neural network model and compare to the target Q value, update weights within neural network through backpropagation using the Adam optimizer

Agent.py

To integrate the game environment, DQN, and reward feedback an agent class is introduced. The functions of the agent class are to get the state of the game, predict an action tensor given the state, scale predictions, apply a threshold on scaled predictions, remember past states, train short term memory, train long term memory, and implement exploration vs exploitation of actions.

The states retrieved from the game_AI.py file are as follows:

1. Left danger
2. Right danger
3. Top danger
4. Alien ahead binary
5. Laser ready binary
6. Target relative x
7. Closest alien relative y
8. Obstacle ahead binary
9. Time since last successful shot
10. Ship center x-coordinate
11. Ship center y-coordinate

The states are then fed through the agent class to the DQN to make predictions and train the neural network to play the game.

Implementation

To train the AI model potentially millions of iterations will need to be run to update the weights properly and get a large enough dataset to improve the algorithms ability to generalize to different game states. A fast GPU or server is required to run this training efficiently, and will enable further evaluation of the Space Invaders AI.